Introduction to Complexity Theory – Lecture Notes

Oded Goldreich
Department of Computer Science and Applied Mathematics
Weizmann Institute of Science, Israel.
Email: oded@wisdom.weizmann.ac.il

July 31, 1999

Preface

Complexity Theory is a central field of Theoretical Computer Science, with a remarkable list of celebrated achievements as well as a very vibrant present research activity. The field is concerned with the study of the *intrinsic* complexity of computational tasks, and this study tend to aim at *generality*: It focuses on natural computational resources, and the effect of limiting those on the *class of problems* that can be solved.

These lecture notes were taken by students attending my year-long introductory course on Complexity Theory, given in 1998–99 at the Weizmann Institute of Science. The course was aimed at exposing the students to the basic results and research directions in the field. The focus was on concepts and ideas, and complex technical proofs were avoided. Specific topics included:

- Revisiting NP and NPC (with emphasis on search vs decision);
- Complexity classes defined by one resource-bound – hierarchies, gaps, etc;
- Non-deterministic Space complexity (with emphasis on NL);
- Randomized Computations (e.g., ZPP, RP and BPP);
- Non-uniform complexity (e.g., P/poly, and lower bounds on restricted circuit classes);
- The Polynomial-time Hierarchy;
- The counting class #P, approximate-#P and uniqueSAT;
- Probabilistic proof systems (i.e., IP, PCP and ZK);
- Pseudorandomness (generators and derandomization);
- Time versus Space (in Turing Machines);
- Circuit-depth versus TM-space (e.g., AC, NC, SC);
- Average-case complexity;

It was assumed that students have taken a course in computability, and hence are familiar with Turing Machines.

Most of the presented material is quite independent of the specific (reasonable) model of computation, but some (e.g., Lectures 5, 16, and 19–20) depends heavily on the locality of computation of Turing machines.

State of these notes

These notes are neither complete nor fully proofread, let alone being far from uniformly well-written (although the notes of some lectures are quite good). Still, I do believe that these notes suggest a good outline for an *introduction to complexity theory* course.

Using these notes

A total of 26 lectures were given, 13 in each semester. In general, the pace was rather slow, as most students were first year graduates and their background was quite mixed. In case the student body is *uniformly more advanced* one should be able to cover much more in one semester. Some concrete comments for the teacher follow

- Lectures 1 and 2 revisit the P vs NP question and NP-completeness. The emphasis is on presenting NP in terms of search problems, on the fact that the mere existence of NP-complete sets is interesting (and easily demonstratable), and on reductions applicable also in the domain of search problems (i.e., Levin reductions). A good undergraduate computability course should cover this material, but unfortunately this is often not the case. Thus, I suggest to give Lectures 1 and 2 if and only if the previous courses taken by the students failed to cover this material.

- There is something anal in much of Lectures 3 and 5. One may prefer to shortly discuss the material of these lectures (without providing proofs) rather than spend 4 hours on them. (Note that many statements in the course are given without proof, so this will not be an exception.)

- One should be able to merge Lectures 13 and 14 into a single lecture (or at most a lecture and a half). I failed to do so due to inessential reasons. Alternatively, may merge Lectures 13–15 into two lectures.

- Lectures 21–23 were devoted to communication complexity, and circuit depth lower bounds derived via communication complexity. Unfortunately, this sample fails to touch upon other important directions in circuit complexity (e.g., size lower bound for AC0 circuits). I would recommend to try to correct this deficiency.

- Lecture 25 was devoted to Computational Learning Theory. This area, traditionally associated with "algorithms", does have a clear "complexity" flavour.

- Lecture 26 was spent discussing the (limited in our opinion) meaningfulness of relativization results. The dilemma of whether to discuss something negative or just ignore it is never easy.

- Many interesting results were not covered. In many cases this is due to the trade-off between their conceptual importance as weighted against their technical difficulty.

Bibliographic Notes

There are several books which cover small parts of the material. These include:

1. Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

2. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998. Copies have been placed in the faculty's library.

3. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

4. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

However, the presentation of material in these lecture notes does not necessarily follow these sources.

Each lecture is planned to include bibliographic notes, but this intension has been only partially fulfilled so far.

Acknowledgments

I am most grateful to the students who have attended the course and partipiated in the project of preparing the lecture notes. So thanks to *Sergey Benditkis, Reshef Eilon, Michael Elkin, Amiel Ferman, Dana Fisman, Danny Harnik, Tzvika Hartman, Tal Hassner, Hillel Kugler, Oded Lachish, Moshe Lewenstein, Yehuda Lindell, Yoad Lustig, Ronen Mizrahi, Leia Passoni, Guy Peer, Nir Piterman, Ely Porate, Yoav Rodeh, Alon Rosen, Vered Rosen, Noam Sadot, Il'ya Safro, Tamar Seeman, Ekaterina Sedletsky, Reuben Sumner, Yael Tauman, Boris Temkin, Erez Waisbard,* and *Gera Weiss.*

I am grateful to Ran Raz and Dana Ron who gave guess lectures during the course: Ran gave Lectures 21–23 (on communication complexity and circuit complexity), and Dana gave Lecture 25 (on computational learning theory).

Thanks also to *Paul Beame*, *Ruediger Reischuk* and *Avi Wigderson* who have answered some questions I've had while preparing this course.

Lecture Summaries

Lecture 1: The P vs NP Question. We review the fundamental question of computer science, known as the $\mathcal{P}$ versus $\mathcal{NP}$ question: Given a problem whose solution can be verified efficiently (i.e., in polynomial time), is there necessarily an efficient method to actually *find* such a solution? Loosely speaking, the first condition (i.e., efficient verification) is captured in the definition of $\mathcal{NP}$, and the second in that of $\mathcal{P}$. The actual correspondence relies on the notion of *self-reducibility*, which relates the complexity of determining whether a solution exists to the complexity of actually finding one.

Notes taken by Eilon Reshef.

Lecture 2: NP-completeness and Self Reducibility. We prove that any relation defining an NP-complete language is self-reducible. This will be done using the SAT self-reducibility (proved in Lecture 1), and the fact that SAT is NP-Hard under Levin Reductions. The latter are Karp Reductions augmented by efficient transformations of NP-witnesses from the original instance to the reduced one, and vice versa. Along the way, we give a simple proof of the existence of NP-Complete languages (by proving that Bounded Halting is NP-Complete).

Notes taken by Nir Piterman and Dana Fisman.

Lecture 3: More on NP and some on DTIME. In the first part of this lecture we discuss two properties of the complexity classes P, NP and NPC: The property is that NP contains problems which are neither NP-complete nor in P (provided NP $\neq$ P), and the second one is that NP-relations have optimal search algorithms. In the second part we define new complexity classes based on exact time bounds, and consider some relations between them. We point out the sensitivity of these classes to the specific model of computation (e.g., one-tape versus two-tape Turing machines).

Notes taken by Michael Elkin and Ekaterina Sedletsky.

Lecture 4: Space Complexity. We define "nice" complexity bounds; these are bounds which can be computed within the resources they supposedly bound (e.g., we focus on time-constructible and space-constructible bounds). We define space complexity using an adequate model of computation in which one is not allowed to use the area occupied by the input for computation. Before dismissing sub-logarithmic space, we present two results regarding it (contrasting sub-loglog space with loglog space). We show that for "nice" complexity bounds, there is a hierarchy of complexity classes – the more resources one has the more tasks one can perform. One the other hand, we mention that this increase in power may not happen if the complexity bounds are not "nice".

Notes taken by Leia Passoni and Reuben Sumner.

Lecture 5: Non-Deterministic Space. We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced: the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. We then turn to investigate the relation between the non-deterministic and deterministic space complexity (i.e., Savitch's Theorem).

Notes taken by Yoad Lustig and Tal Hassner.

Lecture 6: Non-Deterministic Logarithmic Space We further discuss composition lemmas underlying previous lectures. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space): We show that directed graph connectivity is complete for $\mathcal{NL}$. Finally, we prove that $\mathcal{NL} = \mathrm{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

Notes taken by Amiel Ferman and Noam Sadot.

Lecture 7: Randomized Computations We extend the notion of efficient computation by allowing algorithms (Turing machines) to toss coins. We study the classes of languages that arise from various natural definitions of acceptance by such machines. We focus on probabilistic polynomial-time machines with one-sided, two-sided and zero error probability (defining the classes $\mathcal{RP}$ (and $\mathrm{co}\mathcal{RP}$), $\mathcal{BPP}$ and $\mathcal{ZPP}$). We also consider probabilistic machines that uses logarithmic spaces (i.e., the class $\mathcal{RL}$).

Notes taken by Erez Waisbard and Gera Weiss.

Lecture 8: Non-Uniform Polynomial Time ($\mathcal{P}$/Poly). We introduce the notion of non-uniform polynomial-time and the corresponding complexity class $\mathcal{P}$/poly. In this (somewhat fictitious) computational model, Turing machines are provided an external advice string to aid them in their computation (on strings of certain length). The *non-uniformity* is expressed in the fact that an arbitrary advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly "upper bounds" the notion of efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet this upper bound is not tight (as $\mathcal{P}$/poly contains non-recursive languages). The effect of introducing *uniformity* is discussed, and shown to collapse $\mathcal{P}$/poly to $\mathcal{P}$. Finally, we relate the $\mathcal{P}$/poly versus $\mathcal{NP}$ question to the question of whether NP-completeness via Cook-reductions is more powerful that NP-completeness via Karp-reductions. This is done by showing, on one hand, that $\mathcal{NP}$ is Cook-reducible to a sparse set iff $\mathcal{NP} \subset \mathcal{P}$/poly, and on the other hand that $\mathcal{NP}$ is Karp-reducible to a sparse set iff $\mathcal{NP} = \mathcal{P}$.

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman.

Lecture 9: The Polynomial Hierarchy (PH). We define a hierarchy of complexity classes extending $\mathcal{NP}$ and contained in PSPACE. This is done in two ways, shown equivalent: The first by generalizing the notion of Cook reductions, and the second by generalizing the definition of $\mathcal{NP}$. We then relate this hierarchy to complexity classes discussed in previous lectures such as $\mathcal{BPP}$ and $\mathcal{P}$/Poly: We show that $\mathcal{BPP}$ is in PH, and that if $\mathcal{NP} \subseteq \mathcal{P}$/poly then PH collapses to is second level.

Notes taken by Ronen Mizrahi.

Lecture 10: The counting class $\#\mathcal{P}$. The class $\mathcal{NP}$ captures the difficulty of determining whether a given input has a solution with respect to some (tractable) relation. A potentially harder question, captured by the class $\#\mathcal{P}$, refers to determining the number of such solutions. We first define the complexity class $\#\mathcal{P}$, and classify it with respect to other complexity classes. We then prove the existence of $\#\mathcal{P}$-complete problems, and mention some natural ones. Then we try to study the relation between $\#\mathcal{P}$ and $\mathcal{NP}$ more exactly, by showing we can probabilistically approximate $\#\mathcal{P}$ using an oracle in $\mathcal{NP}$. Finally, we refine this result by restricting the oracle to a weak form of SAT (called $uniqueSAT$).

Notes taken by Oded Lachish, Yoav Rodeh and Yael Tauman.

Lecture 11: Interactive Proof Systems. We introduce the notion of interactive proof systems and the complexity class IP, emphasizing the role of randomness and interaction in this model. The concept is demonstrated by giving an interactive proof system for Graph Non-Isomorphism. We discuss the power of the class IP, and prove that co$\mathcal{NP} \subseteq \mathcal{IP}$. We discuss issues regarding the number of rounds in a proof system, and variants of the model such as public-coin systems (a.k.a. Arthur-Merlin games).

Notes taken by Danny Harnik, Tzvika Hartman and Hillel Kugler.

Lecture 12: Probabilistically Checkable Proof (PCP). We introduce the notion of Probabilistically Checkable Proof (PCP) systems. We discuss some complexity measures involved, and describe the class of languages captured by corresponding PCP systems. We then demonstrate the alternative view of $\mathcal{NP}$ emerging from the PCP Characterization Theorem, and use it in order to prove non-approximability results for the problems $max3SAT$ and $maxCLIQUE$.

Notes taken by Alon Rosen and Vered Rosen.

Lecture 13: Pseudorandom Generators. Pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. The latter are indistiguishable from truely random sequences by any efficient observer. We show that, for efficiently sampleable distributions, computational indistiguishability is preserved under multiple samples. We related pseudorandom generators and one-way functions, and show how to increase the stretching of pseudorandom generators. The notes are augmented by an essay of Oded.

Notes taken by Sergey Benditkis, Il'ya Safro and Boris Temkin.

Lecture 14: Pseudorandomness and Computational Difficulty . We continue our discussion of pseudorandomness and show a connection between pseudorandomness and computational difficulty. Specifically, we show how the difficulty of inverting one-way functions may be utilized to obtain a pseudorandom generator. Finally, we state and prove that a hard-to-predict bit (called a hard-core) may be extracted from any one-way function. The hard-core is fundamental in our construction of a generator.

Notes taken by Moshe Lewenstein and Yehuda Lindell.

Lecture 15: Derandomization of BPP. We present an efficient deterministic simulation of randomized algorithms. This process, called derandomization, introduce new notions of pseudorandom generators. We extend the definition of pseudorandom generators and show how to construct a generator that can be used for derandomization. The new construction differ from the generator that constructed in the previous lecture in it's running time (it will run slower, but fast enough for the simulation). The benefit is that it is relying on a seemingly weaker assumption.

Notes taken by Erez Waisbard and Gera Weiss.

Lecture 16: Derandomizing Space-Bounded Computations. We consider derandomization of space-bounded computations. We show that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, namely, any bounded-probability Logspace algorithm can be deterministically emulated in $O(\log^2 n)$ space. We further show that $\mathcal{BPL} \subseteq \mathcal{SC}$, namely, any such algorithm can be deterministically emulated in $O(\log^2 n)$ space and (simultaneously) in polynomial time.

Notes taken by Eilon Reshef.

Lecture 17: Zero-Knowledge Proof Systems. We introduce the notion of zero-knowledge interactive proof system, and consider an example of such a system (Graph Isomorphism). We define perfect, statistical and computational zero-knowledge, and present a method for constructing zero-knowledge proofs for NP languages, which makes essential use of bit commitment schemes. We mention that zero-knowledge is preserved under sequential composition, but is not preserved under the parallel repetition.

Notes taken by Michael Elkin and Ekaterina Sedletsky.

Lecture 18: NP in PCP[poly,O(1)]. The main result in this lecture is $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. In the course of the proof we introduce an $\mathcal{NPC}$ language "Quadratic Equations", and show it to be in $\mathcal{PCP}(poly, O(1))$. The argument proceeds in two stages: First assuming properties of the proof (oracle), and then testing these properties. An intermediate result that of independent interest is an efficient probabilistic algorithm that distinguishes between linear and far-from-linear functions.

Notes taken by Yoad Lustig and Tal Hassner.

Lecture 19: Dtime(t) contained in Dspace(t/log t). We prove that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. That is, we show how to simulate any given deterministic multi-tape Turing Machine (TM) of time complexity t, using a deterministic TM of space complexity $t/\log t$. A main ingrediant in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

Notes taken by Tamar Seeman and Reuben Sumner.

Lecture 20: Circuit Depth and Space Complexity. We study some of the relations between Boolean circuits and Turing machines. We define the complexity classes $\mathcal{NC}$ and $\mathcal{AC}$, compare their computational power, and point out the possible connection between uniform-$\mathcal{NC}$ and "efficient" parallel computation. We conclude the discussion by establishing a strong connection between space complexity and depth of circuits with bounded fan-in.

Notes taken by Alon Rosen and Vered Rosen.

Lecture 21: Communication Complexity. We consider Communication Complexity – the analysis of the amount of information that needs to be communicated betwen two parties which wish to reach a common computational goal. We start with some basic definitions, considering both deterministic and probabilistic models for the problem, and annotating our discussion with a few examples. Next we present a couple of tools for proving lower bounds on the complexity of communication problems. We conclude by proving a linear lower bound on the communication complexity of probabilistic protocols for computing the inner product of two vectors, where initially each party holds one vector.

Notes taken by Amiel Ferman and Noam Sadot.

Lecture 22: Circuit Depth and Communication Complexity. The main result presented in this lecture is a (tight) nontrivial lower bound on the monotone circuit depth of s-t-Connectivity. This is proved via a series of reductions, the first of which is of significant importance: A connection between circuit depth and communication complexity. We then get a communication game and proceed to reduce it to other such games, until reaching a game called FORK. We conclude that a lower bound on the communication complexity of FORK, to be given in the next lecture, will yield an analogous lower bound on the monotone circuit depth of s-t-Connectivity.

Notes taken by Yoav Rodeh and Yael Tauman.

Lecture 23: Depth Lower Bound for Monotone Circuits (cont.). We analyze the FORK game, introduced in the previous lecture. We give tight lower and upper bounds on the communication needed in a protocol solving FORK. This completes the proof of the lower bound on the depth of monotone circuits computing the function st-Connectivity.

Notes taken by Dana Fisman and Nir Piterman.

Lecture 24: Average-Case Complexity. We introduce a theory of average-case complexity which refers to computational problems coupled with probability distributions. We start by defining and discussing the classes of P-computable and P-samplable distributions. We then define the class DistNP (which consists of NP problems coupled with P-computable distributions), and discuss the notion of average polynomial-time (which is unfortunately more subtle than it may seem). Finally, we define and discuss reductions between distributional problems. We conclude by proving the existence of a complete problem for DistNP.

Notes taken by Tzvika Hartman and Hillel Kugler.

Lecture 25: Computational Learning Theory. We define a model of automoatic learning called probably approximately correct (PAC) learning. We define efficient PAC learning, and present several efficient PAC learning algorithms. We prove the Occam's Razor Theorem, which reduces the PAC learning problem to the problem of finding a succinct representation for the values of a large number of given labeled examples.

Notes taken by Oded Lachish and Eli Porat.

Lecture 26: Relativization. In this lecture we deal with relativization of complexity classes. In particular, we discuss the role of relativization with respect to the $\mathcal{P}$ vs. $\mathcal{NP}$ question; that is, we shall see that for some oracle A, $\mathcal{P}^A = \mathcal{NP}^A$, whereas for another A (actually for almost all other A's) $\mathcal{P}^A \neq \mathcal{NP}^A$. However, it also holds that $\mathcal{IP}^A \neq \mathcal{PSPACE}^A$ for a random A, whereas $\mathcal{IP} = \mathcal{PSPACE}$

Notes taken by Leia Passoni.

Contents

Lecture 1

The P vs NP Question

Notes taken by Eilon Reshef

Summary: We review the fundamental question of computer science, known as $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$: given a problem whose solution can be verified efficiently (i.e., in polynomial time), is there necessarily an efficient method to actually *find* such a solution? First, we define the notion of $\mathcal{NP}$, i.e., the class of all problems whose solution can be verified in polynomial-time. Next, we discuss how to represent search problems in the above framework. We conclude with the notion of *self-reducibility*, relating the hardness of determining whether a feasible solution exists to the hardness of actually finding one.

1.1 Introduction

Whereas the research in complexity theory is still in its infancy, and many more questions are open than closed, many of the concepts and results in the field have an extreme conceptual importance and represent significant intellectual achievements.

Of the more fundamental questions in this area is the relation between different flavors of a problem: the *search* problem, i.e., finding a feasible solution, the *decision* problem, i.e., determining whether a feasible solution exists, and the *verification* problem, i.e., deciding whether a given solution is correct.

To initiate a formal discussion, we assume basic knowledge of elementary notions of computability, such as Turning machines, reductions, polynomial-time computability, and so on.

1.2 The Complexity Class $\mathcal{NP}$

In this section we recall the definition of the complexity class $\mathcal{NP}$ and overview some of its basic properties. Recall that the complexity class $\mathcal{P}$ is the collection of all languages L that can be recognized "efficiently", i.e., by a deterministic polynomial-time Turing machine. Whereas the traditional definition of $\mathcal{NP}$ associates the class $\mathcal{NP}$ with the collection of languages that can be efficiently recognized by a *non-deterministic* Turning machine, we provide an alternative definition, that in our view better captures the conceptual contents of the class.

Informally, we view $\mathcal{NP}$ as the class of all languages that admit a short "certificate" for membership in the language. Given this certificate, called a *witness*, membership in the language can be verified efficiently, i.e., in polynomial time.

For the sake of self-containment, we recall that a (binary) relation R is *polynomial-time decidable* if there exists a polynomial-time Turing machine that accepts the language $\{E(x,y) \mid (x,y) \in R\}$, where $E(x,y)$ is a unique encoding of the pair (x,y). An example of such an encoding is $E(\sigma_1 \cdots \sigma_n, \tau_1 \cdots \tau_m) \triangleq \sigma_1\sigma_1 \cdots \sigma_n\sigma_n 01 \tau_1\tau_1 \cdots \tau_n\tau_n$.

We are now ready to introduce a definition of $\mathcal{NP}$.

Definition 1.1 *The complexity class $\mathcal{NP}$ is the class of all languages L for which there exists a relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$, such that*

- *R_L is polynomial-time decidable.*
- *There exists a polynomial b_L such that $x \in L$ if and only if there exists a* witness *w, $|w| \leq b_L(|x|)$ for which $(x,w) \in R_L$.*

Note that the polynomial bound in the second condition is required despite the fact that R_L is polynomial-time decidable, since the polynomiality of R_L is measured with respect to the length of the pair (x,y), and not with respect to $|x|$ only.

It is important to note that if x is not in L, there is no polynomial-size witness w for which $(x,w) \in R_L$. Also, the fact that $(x,y) \notin R_L$ does *not* imply that $x \notin L$, but rather that y is not a proper witness for x.

A slightly different definition may sometimes be convenient. This definition allows only *polynomially-bounded* relations, i.e.,

Definition 1.2 *A relation R is* polynomially bounded *if there exists a polynomial $p(\cdot)$, such that for every $(x,y) \in R$, $|y| \leq p(|x|)$.*

Since a composition of two polynomials is also a polynomial, any polynomial in $p(|x|)$, where p is a polynomial, is also polynomial in $|x|$. Thus, if a polynomially-bounded relation R can be decided in polynomial-time, it can also be decided in time polynomial in the size of first element in the pair $(x,y) \in R$.

Now, definition 1.1 of $\mathcal{NP}$ can be also formulated as:

Definition 1.3 *The complexity class $\mathcal{NP}$ is the class of all languages L for which there exists a* polynomially-bounded *relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$, such that*

- *R_L is polynomial-time decidable.*
- *$x \in L$ if and only if there exists a* witness *w, for which $(x,w) \in R_L$.*

In this view, the fundamental question of computer science, i.e., $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be formulated as the question whether the *existence* of a short witness (as implied by membership in $\mathcal{NP}$) necessarily brings about an efficient algorithm for finding such a witness (as required for membership in $\mathcal{P}$).

To relate our definitions to the traditional definition of $\mathcal{NP}$ in terms of a non-deterministic Turning machine, we show that the definitions above indeed represent the same complexity class.

Proposition 1.2.1 *$\mathcal{NP}$ (as in definition 1.1) = $\mathcal{NP}$ (as in the traditional definition).*

Proof: First, we show that if a language L is in $\mathcal{NP}$ according to the traditional definition, then it is also in $\mathcal{NP}$ according to definition 1.1.

Consider a non-deterministic Turing machine $\tilde{M}_L$ that decides on L after at most $p_L(|x|)$ steps, where p_L is some polynomial depending on L, and x is the input to $\tilde{M}_L$. The idea is that one can encode the non-deterministic choices of $\tilde{M}_L$, and to use this encoding as a witness for membership in L. Namely, $\tilde{M}_L$ can always be assumed to first make all its non-deterministic choices (e.g., by writing them on a separate tape), and then execute deterministically, branching according to the choices that had been made in the first step. Thus, $\tilde{M}_L$ is equivalent to a deterministic Turning machine M_L accepting as input the pair (x, y) and executing exactly as $\tilde{M}_L$ on x with a pre-determined sequence of non-deterministic choices encoded by y. An input x is accepted by $\tilde{M}_L$ if and only if there exists a y for which (x, y) is accepted by M_L.

The relation R_L is defined to be the set of all pairs (x, y) accepted by M_L.

Thus, $x \in L$ if and only if there exists a y such that $(x, y) \in R_L$, namely if there exists an accepting computation of M_L. It remains to see that R_L is indeed polynomial-time decidable and polynomially bounded. For the first part, observe that R_L can be decided in polynomial time simply by simulating the Turing machine M_L on (x, y). For the second part, observe that $\tilde{M}_L$ is guaranteed to terminate in polynomial time, i.e., after at most $p_L(|x|)$ steps, and therefore the number of non-deterministic choices is also bounded by a polynomial, i.e., $|y| \leq p_L(|x|)$. Hence, the relation R_L is polynomially bounded.

For the converse, examine the witness relation R_L as in definition 1.1. Consider the polynomial-time deterministic Turing machine M_L that decides on R_L, i.e., accepts the pair (x, y) if and only if $(x, y) \in R_L$. Construct a non-deterministic Turning machine $\tilde{M}_L$ that given an input x, guesses, non-deterministically, a witness y of size $b_L(|x|)$, and then executes M_L on (x, y). If $x \in L$, there exists a polynomial-size witness y for which $(x, y) \in R_L$, and thus there exists a polynomial-time computation of $\tilde{M}_L$ that accepts x. If $x \notin L$, then for every polynomial-size witness y, $(x, y) \notin R_L$ and therefore $\tilde{M}_L$ always rejects x. ■

1.3 Search Problems

Whereas the definition of computational power in terms of languages may be mathematically convenient, the main computational goal of computer science is to solve "problems". We abstract a computation problem Π by a *search problem* over some binary relation R_Π: the input of the problem at hand is some x and the task is to find a y such that $(x, y) \in R_\Pi$ (we ignore the case where no such y exists).

A particularly interesting subclass of these relations is the collection of *polynomially verifiable* relations R for which

- R is polynomially bounded. Otherwise, the mere writing of the solution cannot be carried out efficiently.

- R is polynomial-time recognizable. This captures the intuitive notion that once a solution to the problem is given, one should be able to verify its correctness efficiently (i.e., in polynomial time). The lack of such an ability implies that even if a solution is provided "by magic", one cannot efficiently determine its validness.

Given a polynomially-verifiable relation R, one can define the corresponding language $L(R)$ as the set of all words x for which there *exists* a solution y, such that $(x, y) \in R$, i.e.,

$$L(R) \triangleq \{x \mid \exists y \ (x, y) \in R\}. \tag{1.1}$$

By the above definition, $\mathcal{NP}$ is exactly the collection of the languages $L(R)$ that correspond to search problems over polynomially verifiable relations, i.e.,

$$\mathcal{NP} \triangleq \{L(R) \mid R \text{ is polynomially verifiable}\}$$

Thus, the question $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ can be rephrased as the question whether for every polynomially verifiable relation R, its corresponding language $L(R)$ can be decided in polynomial time.

Following is an example of a computational problem and its formulation as a search problem.

PROBLEM: 3-Coloring Graphs
INPUT: An undirected graph $G = (V, E)$.
TASK: Find a 3-coloring of G, namely a mapping $\varphi : V \to \{1, 2, 3\}$ such that no adjacent vertices have the same color, i.e., for every $(u, v) \in E$, $\varphi(u) \neq \varphi(v)$.

The natural relation R_{3COL} that corresponds to 3-Coloring is defined over the set of pairs (G, φ), such that $(G, \varphi) \in R_{3COL}$ if

- φ is indeed a mapping $\varphi : V \to \{1, 2, 3\}$.
- For every $(u, v) \in E$, $\varphi(u) \neq \varphi(v)$.

Clearly, with any reasonable representation of φ, its size is polynomial in the size of G. Further, it is easy to determine in polynomial time whether a pair (G, φ) is indeed in R_{3COL}.

The corresponding language $L(R_{3COL})$ is the set of all 3-colorable graphs, i.e., all graphs G that have a legal 3-coloring.

Jumping ahead, it is $\mathcal{NP}$-hard to determine whether such a coloring exists, and hence, unless $\mathcal{P} = \mathcal{NP}$, no efficient algorithm for this problem exists.

1.4 Self Reducibility

Search problems as defined above are "harder" than the corresponding decision problem in the sense that if the former can be carried out efficiently, so can the latter. Given a polynomial-time search algorithm $\mathcal{A}$ for a polynomially-verifiable relation R, one can construct a polynomial-time decision algorithm for $L(R)$ by simulating $\mathcal{A}$ for polynomially many steps, and answering "yes" if and only if $\mathcal{A}$ has terminated and produced a proper y for which $(x, y) \in R$.

Since much of the research in complexity theory evolves around decision problems, a fundamental question that naturally arises is whether an efficient procedure for solving the *decision* problem guarantees an efficient procedure for solving the *search* problem. As will be seen below, this is not known to be true in general, but can be shown to be true for any $\mathcal{NP}$-complete problem.

We begin with a definition that captures this notion:

Definition 1.4 *A relation R is* self-reducible *if solving the search problem for R is Cook-reducible to deciding the corresponding language $L(R) \stackrel{\triangle}{=} \{x \mid \exists y\ (x, y) \in R\}$.*

Recall that a Cook reduction from a problem Π_1 to Π_2 allows a Turing machine for Π_1 to use Π_2 as an oracle (polynomially many times).

Thus, if a relation R is self-reducible, then there exists a polynomial-time Turing machine that solves the search problem (i.e., for each input x finds a y such that $(x, y) \in R$), except that the Turning machine is allowed to access an oracle that decides $L(R)$, i.e., for each input x' outputs whether there exists a y' such that $(x', y') \in R$. For example, in the case of 3-colorability, the search algorithm is required to *find* a 3-coloring for an input graph G, given as an oracle a procedure that tells *whether* a given graph G' is 3-colorable. The search algorithm is not limited to ask the oracle only about G, but rather may query the oracle on a (polynomially long) sequence of graphs G', where the sequence itself may depend upon answers to previous invocations of the oracle.

We consider the example of SAT.

PROBLEM: SAT
INPUT: A CNF formula φ over $\{x_1, \ldots, x_n\}$.
TASK: Find a satisfying assignment σ, i.e., a mapping $\sigma : \{1, \ldots, n\} \to \{T, F\}$, such that $\varphi(\sigma(1), \ldots, \sigma(n))$ is true.

The relation R_{SAT} corresponding to SAT is the set of all pairs (φ, σ) such that σ is a satisfying assignment for φ. It can be easily verified that the length of σ is indeed polynomial in n and that the relation can be recognized in polynomial time.

Proposition 1.4.1 *R_{SAT} is self-reducible.*

Proof: We show that R_{SAT} is self-reducible by showing an algorithm that solves the search problem over R_{SAT} using an oracle $\mathcal{A}$ for deciding $SAT \stackrel{\triangle}{=} L(R_{SAT})$. The algorithm incrementally constructs a solution by building partial assignments. At each step, the invariant guarantees that the partial assignment can be completed into a full satisfying assignment, and hence when the algorithm terminates, the assignment satisfies φ. The algorithm proceeds as follows.

- Query whether $\varphi \in$ SAT. If the answer is "no", the input formula φ has no satisfying assignment.
- For i ranging from 1 to n, let $\varphi_i(x_{i+1}, \ldots, x_n) \stackrel{\triangle}{=} \varphi(\sigma_1, \ldots, \sigma_{i-1}, 1, x_{i+1}, \ldots, x_n)$. Using the oracle, test whether $\varphi_i \in SAT$. If the answer is "yes", assign $\sigma_i \leftarrow 1$. Otherwise, assign $\sigma_i \leftarrow 0$. Clearly, the partial assignment $\sigma(1) = \sigma_1, \ldots, \sigma(i) = \sigma_i$ can still be completed into a satisfying assignment, and hence the algorithm terminates with a true assignment.

■

Consequently, one may deduce that if SAT is decidable in polynomial time, then there exists an efficient algorithm that solves the search problem for R_{SAT}. On the other hand, if SAT is not decidable in polynomial time (which is the more likely case), there is no efficient algorithm for solving the search problem. Therefore, research on the complexity of deciding SAT relates directly to the complexity of searching R_{SAT}.

In the next lecture we show that every $\mathcal{NP}$-complete language has a self-reducible relation. However, let us first discuss the problem of graph isomorphism, which can be easily shown to be in $\mathcal{NP}$, but is not known to be $\mathcal{NP}$-hard. We show that nevertheless, graph isomorphism has a self-reducible relation.

PROBLEM: Graph Isomorphism
INPUT: Two simple[1] graphs $G_1 = (V, E_1)$, $G_2 = (V, E_2)$. We may assume, without loss of generality, that none of the input graphs has any isolated vertices
TASK: Find an isomorphism between the graphs, i.e. a permutation $\varphi : V \to V$, such that $(u, v) \in E_1$ if and only if $(\varphi(u), \varphi(v)) \in E_2$.

The relation R_{GI} corresponding to the graph isomorphism problem is the set of all pairs $((G_1, G_2), \varphi)$ for which φ is an isomorphism between G_1 and G_2.

Proposition 1.4.2 *R_{GI} is self-reducible.*

Proof: To see that graph isomoprphism is self-reducible, consider an algorithm that uses a graph-isomorphism membership oracle along the lines of the algorithm for SAT. Again, the algorithm fixes the mapping $\varphi(\cdot)$ vertex by vertex.

[1]Such graphs have no self-loops and no parallel edges, and so each vertex has degree at most $|V| - 1$.

At each step, the algorithm fixes a single vertex u in G_1, and finds a vertex v such that the mapping $\varphi(u) = v$ can be completed into a graph isomorphism. To find such a vertex v, the algorithm tries all candidate mappings $\varphi(u) = v$ for all unmapped $v \in V$, using the oracle to tell whether the mapping can still be completed into a complete isomorphism. If there exists an isomorphism to begin with, such a mapping must exist, and hence the algorithm terminates with a complete isomorphism.

We now show how a partial assignment can be decided by the oracle. The trick here is that in order to check if u can be mapped to v, one can "mark" both vertices by a unique pattern, say by rooting a star of $|V|$ leaves at both u and v, resulting in new graphs G'_1, G'_2. Next, query the oracle whether there is an isomorphism φ between G'_1 and G'_2. Since the degrees of u and v are strictly larger than the degrees of other vertices in G'_1 and G'_2, an isomorphism φ' between G'_1 and G'_2 would exist if and only if there exists an isomorphism φ between G_1 and G_2 that maps u to v.

After the mapping of u is determined, proceed by incrementally marking vertices in V with stars of $2|V|$ leaves, $3|V|$ leaves, and so on, until the complete mapping is determined. ■

A point worth mentioning is that the definition of self-reducibility applies to relations and not to languages. A particular language $L \in \mathcal{NP}$ may be associated with more than one search problem, and the self-reducibility of a given relation R (or the lack thereof) does not immediately imply self-reducibility (or the lack thereof) for a different relation R' associated with the same language L.

It is believed that not every language in $\mathcal{NP}$ admits a self-reducible relation. Below we present an example of a language in $\mathcal{NP}$ for which the "natural" search problem is believed not to be self-reducible. Consider the language of composite numbers, i.e.,

$$L_{COMP} \triangleq \{N \mid N = n_1 \cdot n_2 \quad n_1, n_2 > 1\}.$$

The language L_{COMP} is known to be decidable in polynomial time by a randomized algorithm. A natural relation R_{COMP} corresponding to L_{COMP} is the set of all pairs $(N, (n_1, n_2))$ such that $N = n_1 \cdot n_2$, where $n_1, n_2 > 1$. Clearly, the length of (n_1, n_2) is polynomial in the length of N, and since R_{COMP} can easily be decided in polynomial time, L_{COMP} is in $\mathcal{NP}$.

However, the search problem over R_{COMP} requires finding a pair (n_1, n_2) for which $N = n_1 \cdot n_2$. This problem is computationally equivalent to factoring, which is believed not to admit any (probabilistic) polynomial-time algorithm. Thus, it is very unlikely that R_{COMP} is (random) self-reducible.

Another language whose natural relation is believed not to be self-reducible is L_{QR}, the set of all quadratic residues. The language L_{QR} contains all pairs (N, x) in which x is a quadratic residue modulo N, namely, there exists a y for which $y^2 \equiv x \pmod{N}$. The natural search problem associated with L_{QR} is R_{QR}, the set of all pairs $((N, x), y)$ such that $y^2 \equiv x \pmod{N}$. It is well-known that the search problem over R_{QR} is equivalent to factoring under randomized reductions. Thus, under the assumption that factoring is "harder" than deciding L_{QR}, the natural relation R_{QR} is not (random) self-reducible.

Bibliographic Notes

For a historical account of the "P vs NP Question" see [2].

The discussion regarding Quadratic Residiousity is taken from [1]. This paper contains also further evidence to the existence of NP-relations which are not self-reducible.

1. M. Bellare and S. Goldwasser, "The Complexity of Decision versus Search", *SIAM Journal on Computing*, Vol. 23, pages 97–119, 1994.

2. Sipser, M., "The history and status of the P versus NP problem", *Proc. 24th ACM Symp. on Theory of Computing*, pages 603–618, 1992.

Lecture 2

NP-completeness and Self Reducibility

Notes taken by Nir Piterman and Dana Fisman

Summary: It will be proven that the relation R of any $\mathcal{NP}-$complete language is Self-reducible. This will be done using the $\mathcal{SAT}$ self reducibility proved previously and the fact that $\mathcal{SAT}$ is $\mathcal{NP}-$hard (under Levin reduction). Prior to that, a simpler proof of the existence of $\mathcal{NP}-$complete languages will be given.

2.1 Reductions

The notions of self reducibility and $\mathcal{NP}-$completeness require a definition of the term reduction. The idea behind reducing problem Π_1 to problem Π_2, is that if Π_2 is known to be easy, so is Π_1 or vice versa, if Π_1 is known to be hard so is Π_2

Definition 2.1 (*Cook Reduction*):
A Cook reduction *from problem Π_1 to problem Π_2 is a polynomial oracle machine that solves problem Π_1 on input x while getting oracle answers for problem Π_2.*

For example:
Let Π_1 and Π_2 be decision problems of languages L_1 and L_2 respectively and χ_L the characteristic function of L defined to be $\chi_L(x) = \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$
Then Π_1 will be Cook reducible to Π_2 if exists an oracle machine that on input x asks query q, gets answer $\chi_{L_2}(q)$ and gives as output $\chi_{L_1}(x)$ (May ask multiple queries).

Definition 2.2 (*Karp Reduction*):
A Karp reduction (many to one reduction) *of language L_1 to language L_2 is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that $x \in L_1$ if and only if $f(x) \in L_2$.*

Claim 2.1.1 *A Karp reduction is a special case of a Cook reduction.*

Proof: Given a Karp reduction $f(\cdot)$ from L_1 to L_2 and an input x to be decided whether x belongs to L_1, define the following oracle machine:

1. On input x compute the value $f(x)$.

2. Present $f(x)$ to the oracle of L_2.
3. The oracle's answer is the desired decision.

The machine runs polynomial time since Step 1 is polynomial as promised by Karp reduction and both Steps 2 and 3 require constant time.
Obviously M accepts x if and only if x is in L_1. ■

Hence a Karp reduction can be viewed as a Cook reduction.

Definition 2.3 (*Levin Reduction*):
A Levin reduction *from relation R_1 to relation R_2 is a triplet of polynomial time computable functions f, g and h such that:*

1. $x \in L(R_1) \Longleftrightarrow f(x) \in L(R_2)$
2. $\forall (x, y) \in R_1$, $(f(x), g(x, y)) \in R_2$
3. $\forall x, z \ \ (f(x), z) \in R_2 \Longrightarrow (x, h(x, z)) \in R_1$

Note: A Levin reduction from R_1 to R_2 implies a Karp reduction of the decision problem (using condition 1) and a Cook reduction of the search problem (using conditions 1 and 3).

Claim 2.1.2 *Karp reduction is transitive.*

Proof: Let $f_1 : \Sigma^* \longrightarrow \Sigma^*$ be a Karp reduction from L_a to L_b and $f_2 : \Sigma^* \longrightarrow \Sigma^*$ be a Karp reduction from L_b to L_c
The function $f_1 \circ f_2(\cdot)$ is a Karp reduction from L_a to L_c:

- $x \in L_a \iff f_1(x) \in L_b \iff f_2(f_1(x)) \in L_c$.
- f_1 and f_2 are polynomial time computable, so the composition of the functions is again polynomial time computable.

■

Claim 2.1.3 *Levin reduction is transitive.*

Proof: Let (f_1, g_1, h_1) be a Levin reduction from R_a to R_b and (f_2, g_2, h_2) be a Levin reduction from R_b to R_c. Define:

- $f_3(x) \stackrel{\triangle}{=} f_2(f_1(x))$
- $g_3(x, y) \stackrel{\triangle}{=} g_2(f_1(x), g_1(x, y))$
- $h_3(x, y) \stackrel{\triangle}{=} h_1(x, h_2(f_1(x), y))$

We show that the triplet (f_3, g_3, h_3) is a Levin reduction from R_a to R_c:

- $x \in L(R_a) \iff f_3(x) \in L(R_c)$
  Since:
  $x \in L(R_a) \iff f_1(x) \in L(R_b) \iff f_2(f_1(x)) \in L(R_c) \iff f_3(x) \in L(R_c)$

- $\forall (x,y) \in R_a$, $(f_3(x), g_3(x,y)) \in R_c$
  Since:
  $(x,y) \in R_a \Longrightarrow (f_1(x), g_1(x,y)) \in R_b \Longrightarrow (f_2(f_1(x)), g_2(f_1(x), g_1(x,y))) \in R_c \Longrightarrow$
  $(f_3(x), g_3(x,y)) \in R_c$

- $\forall x, z \ (f_3(x), z) \in R_c \Longrightarrow (x, h_3(x,z)) \in R_a$
  Since:
  $(f_3(x), z) \in R_c \Longrightarrow (f_2(f_1(x)), z) \in R_c \Longrightarrow (f_1(x), h_2(f_1(x), z)) \in R_b \Longrightarrow$
  $(x, h_1(x, h_2(f_1(x), z))) \in R_a \Longrightarrow (x, h_3(x,z)) \in R_a$

■

Theorem 2.4 *If* Π_1 *Cook reduces to* Π_2 *and* $\Pi_2 \in \mathcal{P}$ *then* $\Pi_1 \in \mathcal{P}$.

Here class $\mathcal{P}$ denotes not only languages but also any problem that can be solved in polynomial time.
Proof: We shall build a deterministic polynomial time Turing machine that recognizes Π_1:
As Π_1 Cook reduces to Π_2, there exists a polynomial oracle machine M_1 that recognizes Π_1 while asking queries to an oracle of Π_2.
As $\Pi_2 \in \mathcal{P}$, there exists a deterministic polynomial time Turing machine M_2 that recognizes Π_2.
Now build a machine M, recognizer for Π_1 that works as following:

- On input x, emulate M_1 until it poses a query to the oracle.
- Present the query to the machine M_2 and return the answer to M_1.
- Proceed until no more queries are presented to the oracle.
- The output of M_1 is the required answer.

Since the oracle and M_2 give the same answers to the queries, correctness is obvious.
Considering the fact that M_1 is polynomial, the number of queries and the length of each query are polynomial in $|x|$. Hence the delay caused by introducing the machine M_2 is polynomial in $|x|$. Therefore the total run time of M is polynomial. ■

2.2 All $\mathcal{NP}$-complete relations are Self-reducible

Definition 2.5 ($\mathcal{NP}-$complete *language*):
A language L is $\mathcal{NP}$-complete *if:*

1. $L \in \mathcal{NP}$
2. *For every language L' in $\mathcal{NP}$, L' Karp reduces to L.*

These languages are the hardest problems in $\mathcal{NP}$, in the sense that if we knew how to solve an $\mathcal{NP}-$complete problem efficiently we could have efficiently solved any problem in $\mathcal{NP}$. $\mathcal{NP}-$completeness can be defined in a broader sense by Cook reductions. There are not many known $\mathcal{NP}-$complete problems by Cook reductions that are not $\mathcal{NP}-$complete by Karp reductions.

Definition 2.6 *1. R is a $\mathcal{NP}$* relation *if $L(R) \in \mathcal{NP}$*

*2. A relation R is $\mathcal{NP}$-*hard **under Levin reduction** *if any $\mathcal{NP}$ relation R' is Levin reducible to R.*

Theorem 2.7 *For every $\mathcal{NP}$ relation R, if $L(R)$ is $\mathcal{NP}-$*complete *then R is Self-reducible.*

Proof: To prove the theorem we shall use two facts:

1. $\mathcal{SAT}$ is Self-reducible (was proved last lecture).

2. $R_{\mathcal{SAT}}$ is $\mathcal{NP}-$hard under Levin reduction (will be proven later).

Given an $\mathcal{NP}$ relation R of an $\mathcal{NP}-$complete language, a Levin reduction (f, g, h) from R to $R_{\mathcal{SAT}}$, a Karp reduction k from $\mathcal{SAT}$ to $L(R)$ and x, the following algorithm will find y such that $(x, y) \in R$ (provided that $x \in L(R)$).

The idea behind the proof is very similar to the self reducibility of $R_{\mathcal{SAT}}$:

1. Ask $L(R)$'s oracle whether $x \in L(R)$.

2. On answer $'no'$ declare: $x \notin L(R)$ and abort.

3. On answer $'yes'$ use the function f, that preserves the property of belonging to the language, to translate the input x for $L(R)$ into a satisfiable CNF formula $\varphi = f(x)$.

4. Compute $(\sigma_1, ..., \sigma_n)$ a satisfying assignment for φ as follows:

   (a) Given a partial assignment $\sigma_1, ..., \sigma_i$ such that $\varphi_i(x_{i+1}, ..., x_n) = \varphi(\sigma_1, ..., \sigma_i, x_{i+1}, x_{i+2}, ..., x_n) \in \mathcal{SAT}$, where $x_{i+1}, ..., x_n$ are variables and $\sigma_1, ..., \sigma_i$ are constants.

   (b) Assign $x_{i+1} = 1$ and compute $\varphi_i(1, x_{i+2}, ..., x_n) = \varphi(\sigma_1, ..., \sigma_i, 1, x_{i+2}, ..., x_n)$

   (c) Use the function k to translate the CNF formula $\varphi_i(1, x_{i+2}, ..., x_n)$ into an input to the language $L(R)$. Ask $L(R)$'s oracle wheather $k(\varphi_i(1, x_{i+2}, ..., x_n)) \in L(R)$.

   (d) On answer $'yes'$ assign $\sigma_{i+1} = 1$, otherwise assign $\sigma_{i+1} = 0$.

   (e) Iterate until $i = n - 1$.

5. Use the function h that translates a pair x and a satisfying assignment $\sigma_1, ..., \sigma_n$ to $\varphi = f(x)$ into a witness $y = h(x, (\sigma_1, ..., \sigma_n))$ such that $(x, y) \in R$.

Clearly $(x, y) \in R$. ∎

Note: The above argument uses a Karp reduction of $\mathcal{SAT}$ to $L(R)$ (guaranteed by the NP-completeness of the latter). One may extend the argument to hold also for the case one is only given a Cook reduction of $\mathcal{SAT}$ to $L(R)$. Specifically in stage 4(c) instead of getting the answer to whether $\varphi_i(1, x_{i+2}, ..., x_n)$ is in $\mathcal{SAT}$ by quering on whether $k(\varphi_i)$ is in $L(R)$, we can get the answer by running the oracle machine given in the Cook reduction (which makes queries to $L(R)$).

2.3 $B_{ounded}H_{alting}$ is $\mathcal{NP}-$complete

In order to show that indeed exist problems in $\mathcal{NP}-$complete (i.e. the class $\mathcal{NP}-$complete is not empty) the language BH will be introduced and proved to be $\mathcal{NP}-$complete.

Definition 2.8 (Bounded Halting):

1. $BH \triangleq \left\{ (\langle M \rangle, x, 1^t) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a non-deterministic machine} \\ \textit{that accepts input } x \textit{ within } t \textit{ steps.} \end{array} \right\}$

2. $BH \triangleq \left\{ (\langle M \rangle, x, 1^t) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a deterministic machine and exists } y \textit{ whose} \\ \textit{length is polynomial in } |x| \textit{ such that } M \textit{ accepts } (x,y) \textit{ within } t \textit{ steps.} \end{array} \right\}$

The two definitions are equivalent if we consider the y wanted in the second as the sequence of non deterministic choices of the first. The computation is bounded by t hence so is y's length.

Definition 2.9 $R_{BH} \triangleq \left\{ ((\langle M \rangle, x, 1^t), y) \;\middle|\; \begin{array}{l} \langle M \rangle \textit{ is the description of a deterministic machine} \\ \textit{that accepts input } (x,y) \textit{ within } t \textit{ steps.} \end{array} \right\}$

Once again the length of the witness y is bounded by t, hence it is polynomial in the length of the input $(\langle M \rangle, x, 1^t)$.

Directly from $\mathcal{NP}$'s definition: $BH \in \mathcal{NP}$.

Claim 2.3.1 *Any language L in $\mathcal{NP}$, Karp reduces to BH*

Proof:
Given a language L in $\mathcal{NP}$, implies the following:

- A witness relation R_L exists and has a polynomial bound $b_L(\cdot)$ such that:
  $\forall\ (x,y) \in R_L,\quad |y| \leq b_L(|x|)$
- A recognizer machine M_L for R_L exists and its time is bounded by another polynomial $p_L(\cdot)$.

The reduction maps x to $f(x) \triangleq (\langle M_L \rangle, x, 1^{p_L(|x|+b_L(|x|))})$, which is an instance to BH by Version 2 of Definition 2.8 above.

Notice that the reduction is indeed polynomial since $\langle M_L \rangle$ is a constant string for the reduction from L to BH. All the reduction does is print this constant string, concatenate the input x to it and then concatenate a polynomial number of ones.
We will show now that $x \in L$ if and only if $f(x) \in BH$:

$x \in L \Longleftrightarrow$
Exists a witness y whose length is bounded by $b_L(|x|)$ such that $(x,y) \in R_L \Longleftrightarrow$
Exists a computation of M_L with $t \triangleq P_L(|x| + b_L(|x|))$ steps accepting $(x,y) \Longleftrightarrow$
$(\langle M \rangle, x, 1^t) \in BH$
■

Note: The reduction can be easily transformed into Levin reduction of R_L to R_{BH} with the identity function supplying the two missing functions.
Thus $BH \in \mathcal{NP}-$complete.

Corollary 2.10 *There exist $\mathcal{NP}-$complete sets.*

2.4 $C_{ircuit}S_{atisfiability}$ is $\mathcal{NP}-$complete

Definition 2.11 (Circuit Satisfiability):

1. *A* **Circuit** *is a directed a-cyclic graph $G = (V, E)$ with vertices labeled:*
   $output, \bigwedge, \bigvee, \neg, x_1, ..., x_m, 0, 1$
   With the following restrictions:
   - *a vertex labeled by $\neg$ has in-degree 1.*
   - *a vertex labeled by x_i has in-degree 0 (i.e. is a source).*
   - *a vertex labeled by 0 (or 1) has in-degree 0.*
   - *there is a single sink (vertex of out-degree 0), it has in-degree 1 and is labeled 'output'.*
   - *in-degree of vertices labeled $\bigwedge, \bigvee$ can be restricted to 2.*

   Given an assignment $\sigma \in \{0,1\}^m$ to the variables $x_1, ..., x_m$, $C(\sigma)$ will denote the value of the circuit's output. The value is defined in the natural manner, by setting the value of each vertex according to the boolean operation it is labeled with. For example, if a vertex is labelled $\bigwedge$ and the vertices with a directed edge to it have values a and b, then the vertex has valu $a \bigwedge b$.

2. **Circuit Satisfiability**
   $CS \stackrel{\triangle}{=} \{C : C \text{ is a circuit and exists } \sigma, \text{ an input to circuit } C \text{ such that } C(\sigma) = 1\}$

3. $R_{CS} \stackrel{\triangle}{=} \{(C, \sigma) : C(\sigma) = 1\}$

The relation defined above is indeed an $\mathcal{NP}$ relation since:

1. σ contains assignment for all variables $x_1, x_2, ..., x_m$ appearing in C and hence its length is polynomial in $|C|$.
2. Given a couple (C, σ) evaluating one gate takes $O(1)$ (since in-degree is restricted to 2) and in view that the number of gates is at most $|C|$, total evaluation time is polynomial in $|C|$.

Hence $CS \in \mathcal{NP}$.

Claim 2.4.1 $C_{ircuit}S_{atisfiability}$ *is* $\mathcal{NP}-$complete

Proof: As mentioned previously $CS \in \mathcal{NP}$.
We will show a Karp reduction from BH to CS, and since Karp reductions are transitive and BH is $\mathcal{NP}-$complete, the proof will be completed. In this reduction we shall use the second definition of BH as given in Definition 2.8.
Thus we are given a triplet $(\langle M \rangle, x, 1^t)$. This triplet is in BH if exists a y such that the deterministic machine M accepts (x, y) within t steps. The reduction maps such a triplet into an instance of CS.
The idea is building a circuit that will simulate the run of M on (x, y), for the given x and a generic y (which will be given as an input to the circuit). If M does not accept (x, y) within the first t steps of the run, we are ensured that $(\langle M \rangle, x, 1^t)$ is not in BH. Hence it suffices to simulate only the first t steps of the run.
Each one of these first t configurations is completely described by the letters written in the first t tape cells, the head's location and the machine's state.

Hence the whole computation can be encoded in a matrix of size $t \times t$. The entry (i, j) of the matrix will consist of the contents of cell j at time i, an indicator whether the head is on this cell at time i and in case the head is indeed there the state of the machine is also encoded. So every matrix entry will hold the following information:

- $a_{i,j}$ the letter written in the cell
- $h_{i,j}$ an indicator to head's presence in the cell
- $q_{i,j}$ the machine's state in case the head indicator is 1 (0 otherwise)

The contents of matrix entry (i, j) is determined only by the three matrix entries $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. If the head indicator of these three entries is off, entry (i, j) will be equal to entry $(i-1, j)$.
The following constructs a circuit that implements the idea of the matrix and this way emulates the run of machine M on input x. The circuit consists of t levels of t triplets $(a_{i,j}, h_{i,j}, q_{i,j})$ where $0 \le i \le t,\ 1 \le j \le t$. Level i of gates will encode the configuration of the machine at time i. The wiring will make sure that if level i represents the correct configuration, so will level $i+1$.
The (i, j)-th triplet, $(a_{i,j}, h_{i,j}, q_{i,j})$, in the circuit is a function of the three triplets $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$.
Every triplet consists of $O(log\ n)$ bits, where $n \triangleq |(\langle M\rangle, x, 1^t)|$:

- Let G denote the size of M's alphabet. Representing one letter requires $log\ G$ many bits: $log\ G = O(log\ n)$ bits.
- The head indicator requires one bit.
- Let K denote the number of states of M. Representing a state requires $log\ K$ many bits: $log\ K = O(log\ n)$ bits.

Note that the machine's description is given as input. Hence the number of states and the size of the alphabet are smaller than input's size and can be represented in binary by $O(log\ n)$ many bits (Albeit doing the reduction directly from any $\mathcal{NP}$ language to CS, the machine M_L that accepts the language L wouldn't have been given as a parameter but rather as a constant, hence a state or an alphabet letter would have required a constant number of bits).
Every bit in the description of a triplet is a boolean function of the bits in the description of three other triplets, hence it is a boolean function of $O(log\ n)$ bits.

Claim 2.4.2 *Any boolean function on m variables can be computed by a circuit of size $m2^m$*

Proof: Every boolean function on m variables can be represented by a $(m+1) \times 2^m$ matrix. The first m columns will denote a certain input and the last column will denote the value of the function. The 2^m rows are required to describe all different inputs.
Now the circuit that will calculate the function is:
For line l in the matrix in which the function value is 1 ($f(l) = 1$), build the following circuit:
$C_l = (\bigwedge_{\text{input } y_i=1} y_i) \wedge (\bigwedge_{\text{input } y_i=0} \neg y_i)$
Now take the OR of all lines (value 1):
$C = \bigvee_{f(l)=1} C_l$
The circuit of each line is of size m and since there are at most 2^m lines of value 1, the size of the whole circuit is at most $m2^m$. ■

So far the circuit emulates a generic computation of M. Yet the computation we care about refers to one specific input. Similarly the initial state should be q_0 and the head should be located at time 0 in the first location. This will be done by setting all triplets $(0, j)$ as following:
Let $x = x_1 x_2 x_3 ... x_m$ and $n \stackrel{\triangle}{=} |(\langle M \rangle, x, 1^t)|$ the length of the input.

- $a_{0,j} = \begin{cases} x_j & 1 \leq j \leq m \quad \text{constants set by input } x \\ y_{j-m} & m < j \leq t \quad \text{these are the inputs to the circuit} \end{cases}$
- $h_{0,j} = \begin{cases} 1 & j = 1 \\ 0 & j \neq 1 \end{cases}$
- $q_{0,j} = \begin{cases} q_0 & j = 1 \quad \text{where } q_0 \text{ is the initial state of } M \\ 0 & j \neq 1 \end{cases}$

The y elements are the variables of the circuit. The circuit belongs to CS if and only if there exists an assignment σ for y such that $C(\sigma) = 1$. Note that y, the input to the circuit plays the same role as the short witness y to the fact that $(\langle M \rangle, x, 1^t)$ is a member of BH. Note that (by padding y with zeros), we may assume without loss of generality that $|y| = t - |x|$.

So far (on input y) the circuit emulates a running of M on input (x, y), it is left to ensure that M accepts (x, y). The output of the circuit will be determined by checking whether at any time the machine entered the $'accept'$ state. This can be done by checking whether in any of the $t \times t$ triplets in the circuit the state is $'accept'$.

Since every triplet (i, j) consists of $O(log\ n)$ bits we have $O(log\ n)$ functions associated with each triplet. Every function can be computed by a circuit of size $O(n\ log\ n)$, so the circuit attached to triplet (i, j) is of size $O(n\ log^2\ n)$.
There are $t \times t$ such triplets so the size of the circuit is $O(n^3\ log^2\ n)$.
Checking for a triplet (i, j) whether $q_{i,j}$ is $'accept'$ requires a circuit of size $O(log\ n)$. This check is implemented for $t \times t$ triplets, hence the overall size of the output check is $O(n^2\ log\ n)$ gates.
The overall size of the circuit will be $O(n^3\ log^2\ n)$.

Since the input level of the circuit was set to represent the right configuration of machine M when operated on input (x, y) at time 0, and the circuit correctly emulates with its i^{th} level the configuration of the machine at time i, the value of the circuit on input y indicates whether or not M accepts (x, y) within t steps. Thus, the circuit is satisfiable if and only if there exists a y so that M accepts (x, y) within t steps, i.e. $(\langle M \rangle, x, 1^t)$ is in BH.

For a detailed description of the circuit and full proof of correction see Appendix.

The above description can be viewed as instructions for constructing the circuit. Assuming that building one gate takes constant time, constructing the circuit following these instructions will be linear to the size of the circuit. Hence, construction time is polynomial to the size of the input $(\langle M \rangle, x, 1^t)$.
■

Once again the missing functions for Levin reduction of R_{BH} to R_{CS} are the identity functions.

2.5 $R_{\mathcal{SAT}}$ *is* $\mathcal{NP}-$complete

Claim 2.5.1 *$R_{\mathcal{SAT}}$ is $\mathcal{NP}-$*hard *under Levin reduction.*

Proof: Since Levin reduction is transitive it suffices to show a reduction from R_{CS} to $R_{\mathcal{SAT}}$:
The reduction will map a circuit C to an CNF expression φ_{C} and an input y for the circuit to an assignment y' to the expression and vice versa.
We begin by describing how to construct the expression φ_{C} from C.
Given a circuit C we allocate a variable to every vertex of the graph. Now for every one of the vertices v build the CNF expression φ_v that will force the variables to comply to the gate's function:

1. For a $\neg$ vertex v with edge entering from vertex u:
   - Write $\varphi_v(v,u) = ((v \bigvee u) \bigwedge(\neg u \bigvee \neg v))$
   - It follows that $\varphi_v(v,u) = 1$ if and only if $v = \neg u$
2. For a $\bigvee$ vertex v with edges entering from vertices u, w:
   - Write $\varphi_v(v,u,w) = ((u \bigvee w \bigvee \neg v) \bigwedge(u \bigvee \neg w \bigvee v) \bigwedge(\neg u \bigvee w \bigvee v) \bigwedge(\neg u \bigvee \neg w \bigvee v))$
   - It follows that $\varphi_v(v,u,w) = 1$ if and only if $v = u \bigvee w$
3. For a $\bigwedge$ vertex v with edges entering from vertices u, w:
   - Similarly write $\varphi_v(v,u,w) = ((u \bigvee w \bigvee \neg v) \bigwedge(u \bigvee \neg w \bigvee \neg v) \bigwedge(\neg u \bigvee w \bigvee \neg v) \bigwedge(\neg u \bigvee \neg w \bigvee v))$
   - It follows that $\varphi_v(v,u,w) = 1$ if and only if $v = u \bigwedge w$
4. For the vertex marked *output* with edge entering from vertex u:
   Write $\varphi_{output}(u) = u$

We are ready now to define $\varphi_{\mathrm{C}} = \bigwedge_{v \in V} \varphi_v$, where V is the set of all vertices of in-degree at least one (i.e. the constant inputs and variable inputs to the circuit are not included).
The length of φ_{C} is linear to the size of the circuit. Once again the instructions give a way to build the expression in linear time to the circuit's size.

We next show that $C \in CS$ if and only if $\varphi_{\mathrm{C}} \in \mathcal{SAT}$. Actually, to show that the reduction is a Levin-reduction, we will show how to efficiently transform witnesses for one problem into witnesses for the other. That is, we describe how to construct the assignment y' to φ_{C} from an input y to the circuit C (and vice versa):
Let C be a circuit with m input vertices labeled $x_1, ..., x_m$ and d vertices labeled $\bigvee, \bigwedge$ and $\neg$ namely, $v_1, ..., v_d$. An assignment $y = y_1, ..., y_m$ to the circuit's input vertices will propagate into the circuit and set the value of all the vertices. Considering that the expression φ_{C} has a variable for every vertex of the circuit C, the assignment y' to the expression should consist of a value for every one of the circuit vertices. We will build $y' = y'_{x_1}, ..., y'_{x_m}, y'_{v_1}, y'_{v_2}, ..., y'_{v_d}$ as following:

- The variables of the expression that correspond to input vertices of the circuit will have the same assignment: $y'_{x_h} = y_h,\ 1 \le h \le m$.
- The assignment y'_{v_l} to every other expression variable v_l will have the value set to the corresponding vertex in the circuit, $1 \le l \le d$.

Similarly given an assignment to the expression, an assignment to the circuit can be built. This will be done by using only the values assigned to the variables corresponding to the input vertices of the circuit. It is easy to see that:
$C \in CS \iff$ exists y such that $C(y) = 1 \iff \varphi_{\mathrm{C}}(y') = 1 \iff \varphi_{\mathrm{C}} \in \mathcal{SAT}$ ■

Corollary 2.12 $\mathcal{SAT}$ *is* $\mathcal{NP}-$complete

Bibliographic Notes

The initiation of the theory NP-completeness is attributed to Cook [1], Levin [4] and Karp [3]: Cook has initiated this theory in the West by showing that SAT is NP-complete, and Karp demonstrated the wide scope of the phenumena (of NP-completeness) by demonstrating a wide variety of NP-complete problems (about 20 in number). Independently, in the East, Levin has shown that half a dozen different problems (including SAT) are NP-complete. The three types of reductions discussed in the lecture are indeed the corresponding reductions used in these papers. Whereas the Cook–Karp exposition is in terms of decision problems, Levin's exposition is in terms of search problems – which explains why he uses the stronger notion of reduction.

Currently thousands of problems, from an even wider range of sciences and technologies, are known to be NP-complete. A partial (out-dated) list is provided in Garey and Johnson's book [2].

Interestingly, almost all reductions used to establish NP-completeness are much more restricted than allowed in the definition (even according to Levin). In particular, they are all computable in logarithmic space (see next lectures for definition of space).

1. Cook, S.A., "The Complexity of Theorem Proving Procedures", *Proc. 3rd ACM Symp. on Theory of Computing*, pp. 151–158, 1971.

2. Garey, M.R., and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

3. Karp, R.M., "Reducibility among Combinatorial Problems", *Complexity of Computer Computations*, R.E. Miller and J.W. Thatcher (eds.), Plenum Press, pp. 85–103, 1972.

4. Levin, L.A., "Universal Search Problems", *Problemy Peredaci Informacii 9*, pp. 115–116, 1973. Translated in *problems of Information Transmission 9*, pp. 265–266.

Appendix: Details for the reduction of BH to CS

We present now the details of the reduction from BH to CS. The circuit that will emulate the run of machine M on input x can be constructed in the following way:
Let $(\langle M\rangle, x, t)$ be the input to be determined whether is in BH, where $x = x_1x_2...x_m$ and $n \triangleq |(\langle M\rangle, x, t)|$ the length of the input.
We will use the fact that every gate of in-degree r can be replaced by r gates of in-degree 2. This can be done by building a balanced binary tree of depth $log\ r$. In the construction $'and','or'$ gates of varying in-degree will be used. When analyzing complexity, every such gate will be weighed as its in-degree.
The number of states of machine M is at most n, hence $log\ n$ bits can represent a state. Similarly the size of alphabet of machine M is at most n, and therfore $log\ n$ bits can represent a letter.

1. Input Level
   y is the witness to be entered at a later time (assume y is padded by zeros to complete length t as explained earlier).

   - $a_{0,j} = \begin{cases} x_j & 1 \le j \le m \quad \text{constants set by input } x \\ y_{j-m} & m < j \le t \quad \text{these are the inputs to the circuit} \end{cases}$
   - $h_{0,j} = \begin{cases} 1 & j = 1 \\ 0 & j \ne 1 \end{cases}$
   - $q_{0,j} = \begin{cases} q_0 & j = 1 \quad \text{where } q_0 \text{ is the initial state of } M \\ 0 & j \ne 1 \end{cases}$

   As said before this represents the configuration at time 0 of the run of M on (x, y).
   This stage sets $O(n \; log \; n)$ wires.

2. For $0 < i < t$, $h_{i+1,j}$ will be wired as shown in figure 1:

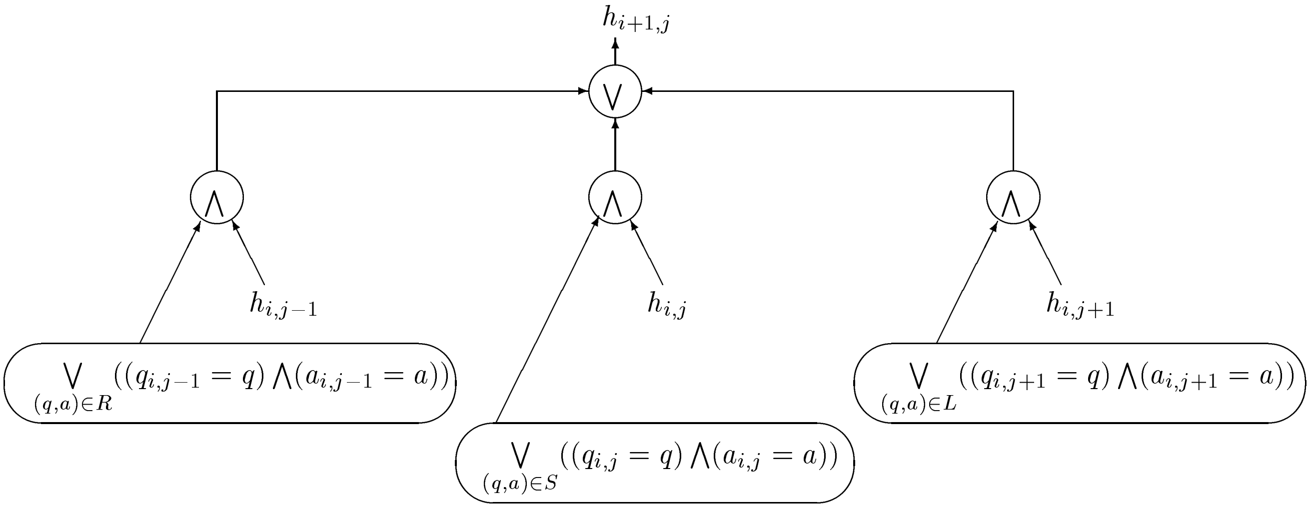


figure 1

The definition of groups R, S, L is:
$R \triangleq \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,R)\}$
$S \triangleq \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,S)\}$
$L \triangleq \{(q,a) : q \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (*,*,L)\}$
The equations are easily wired using an $'and'$ gate for every equation.
The size of this component:
The last item on every entry in the relation δ is either R, L or S. For every one of these entries there is one comparison above. Since δ is bounded by n there are at most n such comparisons. A comparison of the state requires $O(log\ n)$ gates. Similarly a comparison of the letter requires $O(log\ n)$ gates. Hence the total number of gates in figure 1 is $O(n\ log\ n)$

3. For $0 < i < t$, $q_{i+1,j}$ will be wired as shown in figure 2:

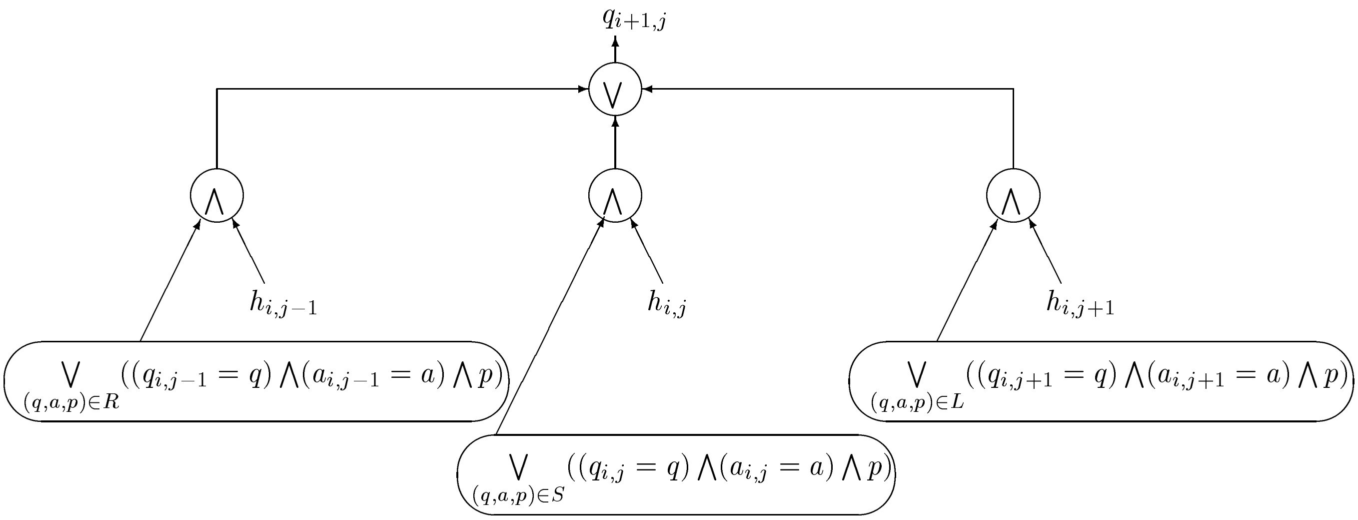


figure 2

The definition of groups R, S, L in:
$R \triangleq \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,R)\}$
$S \triangleq \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,S)\}$
$L \triangleq \{(q,a,p) : q,p \in K \bigwedge a \in \{0,1\} \bigwedge \delta(q,a) = (p,*,L)\}$
Once again every comparison requires $O(log\ n)$ gates. Every state is represented by $log(n)$ bits so the figure has to be multiplied for every bit.
Overall complexity of the component in figure 2 is $O(n\ log^2\ n)$.

4. For $0 < i < t$, $a_{i+1,j}$ will be wired as shown in figure 3:

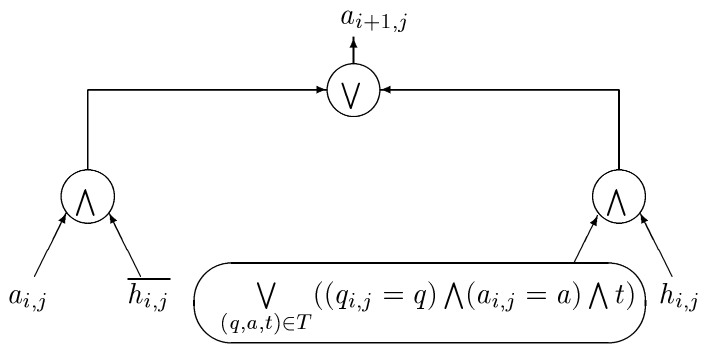


figure 3

The definition of T is:
$T \stackrel{\triangle}{=} \{(q,a,t) : q \in K \bigwedge a,t \in \{0,1\} \bigwedge \delta(q,a) = (*,t,*)\}$
Once again all entries of the relation δ have to be checked, hence there are $O(n)$ comparisons of size $O(log\ n)$.
Since the letter is represented by $O(log\ n)$ bits, the overall complexity of the component in figure 3 is $O(n\ log^2\ n)$.

5. Finally the output gate of the circuit will be a check whether at any level of the circuit the state was *accept*. This will be done by comparing $q_{i,j},\ 1 \le j \le t,\ 0 \le i \le t$ to $'accept'$. There are $t \times t$ such comparisons, each of them takes $O(log\ n)$ gates. Taking an OR on all these comparisons costs $O(n^2\ log\ n)$ gates.

For every cell in the $t \times t$ matrix we used $O(n^3)$ gates. The whole circuit can be built with $O(n^5)$ gates. With this description, building the circuit is linear to circuit's size. Hence, this can be done in polynomial time.

Correctness: We will show now that $(\langle M\rangle, x, 1^t) \in BH$ if and only if $C_{(\langle M\rangle,x,t)} \in CS$

Claim 2.5.2 *Gates at level i of the circuit represent the exact configuration of M at time i on input (x,y).*

Proof: By induction on time i.

- $i = 0$, stage 1 of the construction ensures correctness.
- Assume C's gates on level i correctly represent M's configuration at time i and prove for $i+1$:
  Set j as the position of the head at time i ($h_{i,j} = 1$).
  - The letter contents of all cells $(i+1,k), k \neq j$ does not change. Same happens in the circuit since $(a_{i,k} \bigwedge(\neg h_{i,k})) = a_{i,k}$.
  - Likewise the head can not reach cells $(i+1,k)$ where $k < (j-1)$ or $k > (j+1)$. Respectively $h_{i,k} = 0$ since $h_{i,k-1} = h_{i,k} = h_{i,k+1} = 0$.

- The same argument shows that state bits for all gates of similar $k's$ will be reset to zero.

Let $\delta(q_{i,j}, a_{i,j}) = (q, a, m)$
We shall look into what happens when machine's head stays in place, i.e. $m = S$. The other two posibilities for movement of the head are similar.

- Cell (i,j) on the tape will change into a. Since $h_{i,j} = 1$ and correspondingly $(q_{i,j} = q_{i,j} \bigwedge a_{i,j} = a_{i,j} \bigwedge a)$ will return a
- The head stays in place and respectively:
  1. $h_{i+1,j-1} = 0$ since $h_{i,j} = 1$ but $\delta(q_{i,j}, a_{i,j})$ is not $(*, *, L)$.
  2. $h_{i+1,j} = 1$ since $h_{i,j} = 1$ and one $\delta(q_{i,j}, a_{i,j}) = (*, *, S)$ returns 1.
  3. $h_{i+1,j+1} = 0$ since $h_{i,j} = 1$ but $\delta(q_{i,j}, a_{i,j})$ is not $(*, *, R)$.
- The machine's next state is q, and respectively:
  1. Similarly $q_{i+1,j-1}$ and $q_{i+1,j+1}$ will be reset to zero.
  2. $q_{i+1,j}$ will change into q since $h_{i,j} = 1$ and $(q_{i,j} = q_{i,j} \bigwedge a_{i,j} = a_{i,j} \bigwedge q)$ will return q.

So at any time $0 \leq i \leq t$, gate level i correctly represents M's configuration at time i.
■

Lecture 3

More on NP and some on DTIME

Notes taken by Michael Elkin and Ekaterina Sedletsky

Summary: In this lecture we discuss some properties of the complexity classes P, NP and NPC (theorems of Ladner and Levin). We also define new complexity classes (DTime_i), and consider some relations between them.

3.1 Non-complete languages in NP

In this lecture we consider several items, that describe more closely the picture of the complexity world. We already know that $P \subseteq NP$ and we conjecture that it's a strict inequality, although we can't prove it. Another important class that we have considered is $NP-complete$ (NPC) problems, and as we have already seen, if there is a gap between P and NP then the class of NPC problems is contained in this gap ($NPC \subseteq NP \backslash P$).

The following theorem of Ladner tells us additional information about this gap $NP \backslash P$, namely that NPC is strictly contained in $NP \backslash P$.

Formally,

Theorem 3.1 *If $P \neq NP$ then there exists a language $L \in (NP \backslash P) \backslash NPC$.*

That is, NP (or say SAT) is not (Karp) reducible to L. Actually, one can show that SAT is not even Cook-reducible to L.

Oded's Note: *Following is a proof sketch.*

We start with any $B \in NP \backslash P$, and modify it to $B' = B \cap S$, where $S \in P$, so that B' is neither NP-complete nor in P. The fact that S is in P implies that B' is in NP. The "seiving" set S will be constructed to to foil both all possible polynomial-time algorithms for B' and all possible reductions. At the extreme, setting $S = \{0,1\}^*$, foil all algorithms since in this case $B' = B \notin P$. On the other hand, setting $S = \emptyset$, foils all possible reductions since in this case $B' = \emptyset$ and so (under $P \neq NP$) cannot be NP-complete (as reducing to it gives nothing). Note that the above argument extends to the case S (resp., $\overline{S}$) is a finite set.

The "seiving" set S is constructed iteratively so that in odd iterations is fails machines from a standard enumeration of polynomial-time machines (so that in iteration $2i-1$ we fail the i^{th} machine). (Here we don't need to emulate these machines in polynomial-time *in length of their inputs.) In even iterations we fail oracle-machines from a standard enumeration of such machines (which correspond to Cook-reductions of B to B') so that in iteration $2i$ we fail the i^{th} oracle-machine.*

The iteration number is determined by a deciding algorithm for S which is operates as follows. For simplicity, the algorithm puts z in S iff it puts $1^{|z|}$ in S. The decision whether to put 1^n in S is taken as follows. Starting with the first iteration, and using a time-out mechanism with a fixed polynomial bound $b(n)$ (e.g., $b(n) = n^2$ or $b(n) = \log n$ will do), we try to find a input $z \in \{0,1\}^*$ so that the first polynomial-time algorithm, A_1, fails on z (i.e., $A_1(z) \neq \chi_B(z)$). In order to decide $\chi_B(z)$ we run the obvious exponential-time algorithm, but z is expected to be much shorter than n (or else we halt by time-out before). We scan all possible z's in lexicographic order until reaching time-out. Once we reach time-out while not finding such bad z, we let $1^n \in S$. Eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \notin S$ and continue to the second iteration, where we consider the first polynomial-time oracle-machine, M_1. Now we try to find an input z for B on which M_1 with oracle S. Note that we know the value $n' < n$ so that $1^{n'}$ was the first string not put in S. So currently, S is thought of as containing only strings of length smaller than n'. We emulate M_1 while answering its queries to $B' = B \cap S$ accordingly, using the exponential-time algorithm for deciding B (and our knowledge of the portion of S determined so far). We also use the exponential-time algorithm for B to check if the recuction of z to B' is correct for each z. Once we reach time-out while not finding such bad z, we let $1^n \notin S$. Again, eventually, for a sufficiently large n we will find a bad z within the time-out. In such a case we let $1^n \in S$ and continue to the third iteration, where we consider the second polynomial-time algorithm, and so on.

Some implementation details are provided below. Specifically, the algorithm T below computes the number of iterations completed wrt input $x \in \{0,1\}^n$.

Proof: Let $B \in NP \backslash P$. Let $A_0, A_1, \ldots$ be the enumeration of all polynomial time bounded Turing machines, that solve decision problems and $M_0, M_1, \ldots$ be the enumeration of polynomial time bounded oracle Turing machines. Let $L(A_i)$ denote the set recognized by A_i and for every set S let $L(M_i^S)$ to be the set recognized by machine M_i with oracle S.

We construct a polynomial time bounded Turing machine T with range $\{1\}^*$ in such a way that, for $B' = \{x \in B : |T(x)| \text{ is even}\}$, both $B' \notin P$ and $B \not\propto_C B'$ (i.e., B is not Cook reducible to B'). It follows that $B' \notin P \cup NPC$.

We show that for any such T, $B' \propto_K B$ (B' is Karp reducible to B) and $B' \in NP$ follows, since diciding B' can be done by deciding B and $B \in NP$.The Karp-reduction (of B' to B), denoted f, is defined as follows. Let $x_0 \notin B$. (We assume that $B \neq \Sigma^*$, because otherwise $B \in P$). The function f will compute $|T(x)|$ and if $|T(x)|$ is even it will return x and otherwise it will return x_0. Now if $x \in B'$, then $x \in B$ and $|T(x)|$ is even, hence $f(x) = x \in B$. Otherwise $x \notin B'$ and then there are two possibilities.

1. If $x \notin B$, then for even $|T(x)|$ holds $f(x) = x$, and for odd $|T(x)|$ holds $f(x) = x_0$, and so for any x holds $f(x) \notin B$.
2. If $x \in B$ and $|T(x)|$ is odd, then $f(x) = x_0 \notin B$.

(Recall that $x \notin B'$ rules out "$x \in B$ and even $|T(x)|$")

To complete the construction we have to build a Turing machine T such that

(1) $B' \stackrel{\text{def}}{=} \{x \in B : |T(x)| \text{ is even}\} \neq L(A_i)$, for any $i = 0, 1, 2, \ldots$ (and so $B' \notin P$)

(2) $L(M_i^{B'}) \neq B$, for any $i = 0, 1, 2, \ldots$ (and so $B \not\propto_C B'$).

A machine T that satisfies these conditions may be constructed in the following way:

On input λ (empty string), T prints λ. On an input x of length n where $x \neq 0^n$ (unary), T prints $T(0^n)$. It remains to say what T does on inputs of the form 0^n where $n \geq 1$.

On input 0^n where $n \geq 1$, T does the following:

1. For n moves, try to reconstruct the sequence $T(\lambda), T(0), T(0^2), \ldots$. Let $T(0^m)$ be the last number of this sequence that is computed.

2. We consider two cases depending on the parity of $|T(0^m)|$. We associate B with an exponential-time algorithm decoding B (by scanning all possible NP-witnesses).

   Case(i): $|T(0^m)|$ is even. Let $i = |T(0^m)|/2$. For n moves, try to find a z such that $B'(z) \neq A_i(z)$. This is done by successively simulating B and T (to see what B' is) and A_i, on inputs $\lambda, 0, 1, 00, 01, \ldots$. If no such z is found, print 1^{2*i}; otherwise, print 1^{2*i+1}.

   Case(ii): $|T(0^m)|$ is odd. Let $i = (|T(0^m)| - 1)/2$. For n moves, try to find a z such that $B(z) \neq M_i^{B'}(z)$. This is done by simulating an algorithm B and the procedure M_i successively on inputs $\lambda, 0, 1, 00, 01, \ldots$. In simulating M_i on some input, whenever M_i makes a query of length at most m, we answer it according to B' determined by B and the values of T computed in Step (1). In case the query has lengths exceeding m, we behave as if we have already completed n steps. (The moves in this side calculation are counted among the n steps allowed.) If no such z is found, print 1^{2*i+1}; otherwise, print 1^{2*i+2}.

Such a machine can be programmed in a Turing machine that runs in polynomial time. For this specific machine T we obtain that $B' = \{x \in B : |T(x)| \text{ is even}\}$, satisfies: $B' \in NP \backslash P$ and $B \not\propto_C B' \Longrightarrow B' \notin NPC \Longrightarrow B' \in (NP \backslash P) \backslash NPC$. ■

The set B' constructed in the above proof is certainly not a natural one (even in case B is). We mention that there are some natural problems conjectured to be in $(NP\backslash P)\backslash NPC$: For example, Graph Isomorphism (the problem of whether two graphs are isomorphic).

3.2 Optimal algorithms for NP

The following theorem, due to Levin, states an interesting property of the class NP. Informally, Levin's Theorem tells us that exists optimal algorithm for any NP search problem.

Theorem 3.2 *For any NP-relation R there exist a polynomial $P_R(\cdot)$ and an algorithm $A_R(\cdot)$ which finds solutions whenever they exist, so that for every algorithm A which finds solutions and for any input x*

$$time_{A_R}(x) \leq O(time_A(x) + P_R(|x|)), \tag{3.1}$$

where $time_A(x)$ is the running time of the algorithm A on input x.

This means that for every algorithm A exists a constant c such that for any sufficiently long x

$$time_{A_R}(x) \leq c * (time_A(x) + P_R(|x|)).$$

This c is not a universal constant, but an algorithm-specific one (i.e., depends on A). The algorithm A_R is optimal in the following sense: for any other algorithm A there exists a constant c such that for any sufficiently long x

$$time_A(x) \geq \frac{1}{c} * time_{A_R}(x) - Q_R(|x|),$$

where $Q_R(|x|) = \frac{1}{c} * P_R(|x|)$.

The algorithms we are talking about are not TM's. For proving the theorem, we should define exactly the computational model we are working with. Either it will be a one-tape machine or two-tape one and etc. Depending on the exact model the constant c may be replaced by some low-n function like $logn$. A constant may be achieved only in more powerful/flexible models of computation that are not discussed here.

We observe also that although the proof of Levin's Theorem is a constructive one, it's completely impractical, since as we'll see it incorporates a huge constant in its running time. On the other hand, it illustrates an important asymptotic property of the NP class.

Proof: The basic idea of the proof is to have an enumeration of all possible algorithms. This set is countable, since the set of all possible deterministic TM's is countable. Using this enumeration we would like to run all the algorithms in parallel. It's, of course, impossible since we can't run a countable set of TM's in parallel, but the solution to the problem is to run them in different rates.

There are several possible implementations of this idea. One of the possibilities is as following. Let us divide the execution of the simulating machine into rounds. The simulating machine runs machine i at round r if and only if $r \equiv 0 \pmod{i^2}$. That is, we let i'th machine to make t steps during $i^2 * t$ rounds of the simulating machine. Also the number of steps made in these $r = i^2 * t$ rounds is $\sum_{j \geq 1} |\frac{r}{j^2}| < r$.

Such a construction would fail if some of these machines would provide wrong answers. To solve this difficulty we "force" the machines to verify the validity of their outputs. So, without loss of generality, we assume that each machine is augmented by a verifying component that checks the validity of machine's output. Since the problem is in NP, verifying the output takes polynomial amount of time. When we estimate the running time of the algorithm A_R, we take into account this polynomial amount of time and it is the reason that P_R arises in Eq. (3.1). So, without loss of generality, the outputs of the algorithms are correct.

Another difficulty, is that some of these machines could not have sufficient amount of time to halt. On the other hand, since each of the machines solves the problem, it is sufficient for us that one of them will halt.

Levin's optimal algorithm A_R is this construction running interchangeably all these machines.

We'll claim the following property of the construction.

Claim 3.2.1 *Consider A that solves the problem. Let i be the position of A in the enumeration of all the machines. Let $time_A(\cdot)$ be the running time of A. It is run in A_R in $\frac{1}{f(i)}$ rate. Then A_R runs at most $f(i) * time_A(\cdot)$.*

We took $f(i) = (i+1)^2$, but that is not really important: we observe that i is a constant (and thus, so is $f(i)$). Of course, it's a huge constant, because each machine needs millions of bits to be encoded (even the simplest deterministic TM that does nothing needs several hundreds of bits to be encoded) and the index of machine M in the enumeration (i.e. i) is $i \approx 2^{|M|}$, where $|M|$ is the number of bits needed to encode machine M and then $f(i)$ will be $(i+1)^2 = (2^{|M|}+1)^2$. (This constant makes the algorithm completely impractical.) ■

3.3 General Time complexity classes

The class P was introduced to capture our notion of efficient computation. This class has some "robustness" properties which make it convenient for investigation.

1. P is not model-dependent: Remain the same if we consider one tape TM or two tape TM. This remains valid for any "reasonable" and "general" enough model of computation.
2. P is robust under "Reasonable" changes to an algorithm: Closed classes under "reasonable" changes of the algorithm, like flipping the output and things like that. This holds for P, but probably not for NP and NPC. The same applies also for (3) and (4).
3. Closed under serial composition: Concatenation of two (efficient) algorithms from a class will produce another (efficient) algorithm from the same class.
4. Closed under subroutine operation: Using one algorithm from a class as a subroutine of another algorithm from the same class provides an algorithm from the class (the class is set of problems and when we are talking about an algorithm from the class we mean an algorithm for solving a problem from the class and the existence of this algorithm is evidence that the problem indeed belongs to this class).

None of these nice properties holds for classes that we will now define.

3.3.1 The DTime classes

Oded's Note: *DTime denotes the class of languages which are decideable within a specific time-bound. Since this class specifies one time-bound rather than a family of such bounds (like polinomial-time), we need to be soecific with respect to the model of computation.*

Definition 3.3 *$DTime_i(t(\cdot))$ is the class of languages decidable by a deterministic i-tape Turing Machine within $t(\cdot)$ steps. That is, $L \in DTime_i(t(\cdot))$ if there exists a deterministic i-tape Turing Machine M which accepts L so that for any $x \in \{0,1\}^*$, on input x machine M makes at most $t(|x|)$ steps.*

Usually, we consider $i = 1$ or $i = 2$ talking about one- and two-tape TM's respectively. When we'll consider space complexity we'll find it very natural to deal with 3-tape TM. If there is no index in DTime, then the default is $i = 1$.

Using this new notation we present the following theorem:

Theorem 3.4 *For every function $t(\cdot)$ that is at least linear*

$$DTime_2(t(\cdot)) \subseteq DTime_1(t(\cdot)^2)$$

The theorem is important in the sense that it enables us sometimes to skip the index, since with respect to polynomial-time computations both models (one-tape and two-tape) coincide. The proof is by simulating the two-tape TM on a one-tape one.

Proof: Consider a language $L \in DTime_2(t(\cdot))$. Therefore, there exists a two-tape TM M_1 which accepts L in $O(t(\cdot))$. We can imagine that the tape of a one-tape TM as divided into 4 tracks. We can construct M_2, a one-tape TM with 4 tracks, 2 tracks for each of M_1's tapes. One track records the contents of the corresponding tape of M_1 and the other is blank, except for a marker in the cell that holds the symbol scanned by the corresponding head of M_1. The finite control of M_2 stores the state of M_1, along with a count of the number of head markers to the right of M_2's tape head.

Each move of M_1 is simulated by a sweep from left to right and then from right to left by the tape head of M_2, which takes $O(t(\cdot))$ time. Initially, M_2's head is at the leftmost cell containing a head marker. To simulate a move of M_1, M_2 sweeps right, visiting each of the cells with head markers and recording the symbol scanned by both heads of M_1. When M_2 crosses a head marker, it must update the count of head markers to the right. When no more head markers are to the right, M_2 has seen the symbols scanned by both of M_1's heads, so M_2 has enough information to determine the move of M_1. Now M_2 makes a pass left, until it reaches the leftmost head marker. The count of markers to the right enables M_2 to tell when it has gone far enough. As M_2 passes each head marker on the leftward pass, it updates the tape symbol of M_1 "scanned" by that head marker, and it moves the head marker one symbol left or right to simulate the move of M_1. Finally, M_2 changes the state of M_1 recorded in M_2's control to complete the simulation of one move of M_1. If the new state of M_1 is accepting, then M_2 accepts.

Finding the mark costs $O(t(\cdot))$, and as there are not more than $t(\cdot)$ moves, totally the execution costs at most $O(t^2(\cdot))$. ■

The next theorem is less important and brought from elegancy considerations, and says that in general one cannot do a better simulation than one in Theorem 3.
We recall the definition of "O", "Ω" and "o" notations:

- $f(n) = O(g(n))$ means that exists c such that for any sufficiently large n $f(n) \leq c * g(n)$.
- $f(n) = \Omega(g(n))$ means that exists $c > 0$ such that for any sufficiently large n $f(n) \geq c * g(n)$.
- $f(n) = o(g(n))$ means that for any c exists N such that for all $n > N$ it holds $f(n) < c * g(n)$.

Theorem 3.5 *$DTime_2(O(n))$ is not contained in $DTime_1(o(n^2))$.*

We note that it's much harder to prove that some things are "impossible" or "can not be done", than the opposite, because for the latter, constructive proofs can be used.

There are several possible ways to prove the theorem. The following one uses the notion of communication complexity.

Proof: Define language $L = \{xx : x \in \{0,1\}^*\}$. This language L is clearly in $DTime_2(O(n))$. We will show that $L \notin DTime_1(o(n^2))$ by "reduction" to a communication complexity problem.

Introduce, for the sake of proof, computational complexity model: two parties A and B have two strings, A has $\alpha \in \{0,1\}^n$ and B has $\beta \in \{0,1\}^n$, respectively. Their goal is to calculate $f(\alpha, \beta)$, where f is a function from $\{0,1\}^n * \{0,1\}^n$ to $\{0,1\}$. At the end of computation both parties should know $f(\alpha, \beta)$.

Let us also introduce a notation $R_0(f)$ to be the minimum expected cost of a randomized protocol, that computes f with zero error.

In our case it is sufficient to consider Equality (EQ) function defined by

$$EQ(\alpha, \beta) \stackrel{\text{def}}{=} 1 \text{ if } \alpha = \beta \text{ and } 0 \text{ otherwise.}$$

We state without proof a lower bound on the randomized zero-error communication complexity of EQ.

$$R_0(EQ) = \Omega(n), \tag{3.2}$$

The lower bound can be proven by invoking the lower bound on the "nondeterministic communication complexity" of EQ, and from the observation that nondeterminism is generally stronger than (zero-error!) randomization: Intuitively, if a randomized algorithm reaches the solution with some non-zero probability, then there is a sequence of values of flipped coins that causes the randomized algorithm to reach the solution. The nondeterministic version of the same algorithm could just guess correctly all these coins' values and to reach the solution as well.

We now get back to the proof. Suppose for contradiction, that there exists a one-tape Turing machine M which decides L in $o(n^2)$ time. Then we will build a zero-error randomized protocol Π that solves EQ in expected complexity $o(n)$, contradicting Eq. (3.2).

This protocol Π, on input α and β each of length n simulates the Turing machine on input $\alpha 0^n \beta 0^n$ in the following way. They output 1 or 0 depending on whether the machine accepts or rejects this input. They first choose together, uniformly at random, a location at the first 0-region of the tape. The party A simulates the machine whenever the head is to the left of this location, and the party B whenever the head is to the right of this location. Each time the head crosses this location only the state of the finite control ($O(1)$ bits) need to be sent. If the total running time of the machine is $o(n^2)$, then the expected number of times it crosses this location (which has been chosen at random among n different possibilities) is at most $o(n)$, contradicting Eq. (3.2).

Therefore, we have proved that a one-tape Turing machine which decides L runs $\Omega(n^2)$ time. ■

An alternative way of proving the theorem, a direct proof, based on the notion of a crossing sequence, is given in the Appendix.

3.3.2 Time-constructibility and two theorems

Definition 3.6 *A function $f : N \longrightarrow N$ is time-constructible if there exists an algorithm A st. on input 1^n, A runs at most $f(n)$ steps and outputs $f(n)$ (in binary).*

One motivation to the definition of time-constructible function is the following: if the machine's running time is this specific function of input length, then we can calculate the running time within the time required to perform the whole calculation. This notion is important for simulating results, when we want to "efficiently" run all machines which have time bound $t(\cdot)$. We cannot enumerate these machines. Instead we enumerate all machines and run each with time bound $t(\cdot)$. Thus, we need to be able to compute $t(\cdot)$ within time $t(\cdot)$. Otherwise, just computing $t(\cdot)$ will take too much time.

For example, n^2, 2^n, n^n are all time-constructible functions.

Time Hierarchy: A special case of the Time Hierarchy Theorem asserts that

for every constant $c \in N$, for any $i \in \{1,2\}$

$$DTime_i(n^c) \subset DTime_i(n^{c+1})$$

(where $A \subset B$ denotes that A is strictly contained in B)

That is, in this case there is no "complexity gaps" and the set of problems that can be solves grows whewn allowing more time: There are computational tasks that can be done in $O(n^{c+1})$, but can not be done in $O(n^c)$. The function n^{c+1} (above) can be replaced even by $n^{c+\frac{1}{2}}$ etc. The general case of the Time Hierarchy Theorem is

Theorem 3.7 (Time Hierarchy): *For every time-constructible functions $t_1, t_2 : N \longrightarrow N$ such that*

$$\lim_{n\to\infty} \frac{t_1(n) * \log t_2(n)}{t_2(n)} = 0$$

then

$$DTime_i(t_1(n)) \subset DTime_i(t_2(n)).$$

The proof for an analogous space hierarchy is easier, and therefore we'll present it first, but in following lectures.

Linear Speed-up: The following *Linear Speed-up Theorem* allows us to discard constant factors in running-time. Intuitively, there is no point in holding such an accurate account when one does not specify other components of the algorithm (like the size of its finite control and work-tape alphabet).

Theorem 3.8 (Linear Speed-up): *For every function $t : N \longrightarrow N$ and for every i*

$$DTime_i(t(n)) \subseteq DTime_i(\frac{t(n)}{2} + O(n)).$$

The proof idea is as following: let Γ be the original work alphabet. We reduce the time complexity by a constant factor by working with larger alphabet $\Gamma_k = \underbrace{\Gamma * \Gamma * ... * \Gamma}_{k \text{ times}}$, which enables us to process adjacent symbols simultaneously. Then we construct a new machine with alphabet Γ_k. Using this alphabet, any k adjacent cells of the original tape are replaced by one cell of the new tape.

So the new input will be processed almost k times faster, but dealing with the input will produce $O(n)$ overhead.

Let M_1 be an i-tape $t(n)$ time-bounded Turing machine. Let L be a language accepted by M_1. Then L is accepted by a i-tape $(\frac{t(n)}{2} + O(n))$ time-bounded TM M_2.

Proof: A Turing machine M_2 can be constructed to simulate M_1 in the following manner. First M_2 copies the input onto a storage tape, encoding 16 symbols into one. From this point on, M_2 uses this storage tape as the input tape and uses the old input tape as a storage tape. M_2 will encode the contents of M_1's storage tape by combining 16 symbols into one. During the course of the simulation, M_2 simulates a large number of moves of M_1 in one basic step consisting of eight moves of M_2. Call the cells currently scanned by each of M_2's heads the home cells. The finite control of M_2 records, for each tape, which of the 16 symbols of M_1 represented by each home cell is scanned by the corresponding head of M_2.

To begin a basic step, M_2 moves each head to the left once, to the right twice, and to the left once, recording the symbols to the left and right of the home cells in its finite control. Four moves of M_2 are required, after which M_2 has returned to its home cells.

Next, M_2 determines the contents of all of M_1's tape cells represented by the home cells and their left and right neighbors at the time when some tape head of M_1 first leaves the region represented by the home cell and its left and right neighbors. (Note that this calculation by M_2 takes no time. It is built into the transition rules of M_2.) If M_1 accepts before some tape head leaves the represented region, M_2 accepts. If M_1 halts, M_2 halts. Otherwise M_2 then visits, on each, the two neighbors of the home cell, changing these symbols and that of the home cell if necessary. M_2 positions each of it heads at the cell that represents the symbol that M_1's corresponding head is scanning at the end of the moves simulated. At most four moves of M_2 are needed.

It takes at least 16 moves for M_1 to move a head out of the region represented by a home cell and its neighbors. Thus in eight moves, M_2 has simulated at least 16 moves of M_1. ■

Bibliographic Notes

For a detailed proof of Ladner's Theorem, the reader is referred to the original paper [3]. The existence of an optimal algorithm for any NP-problem (referred to above as Levin's Theorem) was proven in [4].

The separtion of one-tape Turing machines from two-tape ones (i.e., $\mathrm{DTime}_2(O(n))$ is NOT contained in $\mathrm{DTime}_1(o(n^2))$) can be proved in various ways. Our presentation follows the two-step proof in [2], while omitting the second step (i.e., proving a lower bound on the error-free randomized communication complexity of `equality`). The alternative (direct) proof, presented in the appendix to this lecture, is adapted from Exercise 12.2 (for which a solution is given) in the textbook [1].

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

2. E. Kushilevitz and N. Nisan. *Communication Complexity*, Cambridge University Press, 1996.

3. R.E. Ladner, "On the Structure of Polynomial Time Reducibility", *Jour. of the ACM*, 22, 1975, pp. 155–171.

4. Levin, L.A., "Universal Search Problems", *Problemy Peredaci Informacii 9*, pp. 115–116, 1973. Translated in *problems of Information Transmission 9*, pp. 265–266.

Appendix: Proof of Theorem 3.5, via crossing sequences

Consider one-tape TM M with transition function δ, input w of length m, such that M accepts w and an integer i, $0 < i < m$.

Denote by w_j, $0 \leq j \leq m$, the j'th symbol of the word w.

Consider the computation of the machine M on input w. This computation is uniquely determined by the machine description and the input, since the machine is deterministic.

The computation is a sequence of ID's (instantaneous descriptions), starting with $q_0 w_1 ... w_n$ and ending with $w_1 ... w_n p$ for some $p \in F$ (accepting or rejecting state).

Denote the elements of the computation sequence by $(ID_j)_{j=1}^{r}$ for some finite r.

Consider a sequence $(L_j)_{j=1}^{r-1}$ of pairs $L_j = (ID_j, ID_{j+1})$. For some $0 \leq i \leq m$ let $(L_{j_l})_{l=1}^{t}$ $t \leq r-1$ be the subsequence of $(L_j)_{j=1}^{r-1}$ of elements (ID_{j_l}, ID_{j_l+1}) of the form:

either

$ID_{j_l} = w_1 ... w_{i-1} p w_i ... w_n$ and $ID_{j_l+1} = w_1 ... w_i q w_{i+1} ... w_n$, for some $p, q \in Q$ and this specific i,

or

$ID_{j_l} = w_1 ... w_i p w_{i+1} ... w_n$ and $ID_{j_l+1} = w_1 ... w_{i-1} q w_i ... w_n$, for some $p, q \in Q$ and this specific i.

By definition of Turing machine computation $ID_{j_l} \vdash^M ID_{j_l+1}$ and therefore in the first case $\delta(p, w_i) = (q, w_{i+1}, R)$ and in the second case $\delta(p, w_{i+1}) = (q, w_i, L)$.

For every $1 \leq l \leq t$, let q_l be the state recorded in $ID_{j_{l+1}}$, where $ID_{j_{l+1}}$ are as above. Then the sequence $(q_l)_{l=1}^t$ is defined to be the **crossing sequence** of the triple (M, w, i) (machine M on input w in the boundary i).

Consider

$$L = \{wcw : w \in \{0,1\}^*\},$$

where c is a special symbol.

Clearly, a 2-tape TM will decide the language in $O(n)$ just by copying all the symbols before c to another tape and comparing symbol by symbol.

Let's prove that 1-tape TM will need $\Omega(n^2)$ steps to decide this language.

For $w \in \{0,1\}^m$, and $1 \leq i \leq m-1$, let $l_{w,i}$ be the length of the crossing sequence of (M, wcw, i). Denote by s the number of states of M.

Denote the average of $l_{w,i}$ over all m-long words w by $p(i)$.

Then from counting considerations, at least for half of w's it holds $l_{w,i} \leq 2 * p(i)$.

Let $N = 2^m$.So there are at least 2^{m-1} words w for which holds $l_{w,i} \leq 2 * p(i)$.

The number of possible crossing sequences of length $\leq 2 * p(i)$ is

$$\sum_{j=0}^{2*p(i)} s^j < s^{2*p(i)+1},$$

where s is the number of states of M.

So there is at least $\frac{2^{m-1}}{s^{2*p(i)+1}}$ words w of length m with the same crossing sequence for boundary $(i, i+1)$ (by pigeonhole principle). We are interested in such words w with the same suffix ($i+1, ..., m$ symbols). The number of such different suffixes is 2^{m-i}. Therefore if for some i holds

$$\frac{2^{m-1}}{s^{2*p(i)+1}} / 2^{m-i} > 1, \tag{3.3}$$

then by pigeonhole principle there are two different w's with the same suffix and the same crossing sequence between i and $i+1$ positions. We'll show that this leads to contradiction.

Denote the differing i-prefixes by α_1 and α_2 and the common $(m-i)$-bit suffix by β. Consider the input word $\alpha_1\beta c\alpha_1\beta$ and the input word $\alpha_2\beta c\alpha_1\beta$. Since the crossing sequence between α_1 and β is the same as the one between α_2 and β, the machine will not be able to distinguish between the two cases, and will accept the second word too, contradiction.

So Eq. (3.3) can not hold for any i and therefore for every i it holds

$$\frac{2^{m-1}}{2^{2*p(i)+1}2^{m-i}} \leq 1$$

and so

$$2^{i-1} \leq 2^{2*p(i)+1},$$

implying

$$i - 1 \leq 2 * p(i) + 1,$$

and

$$p(i) \geq \frac{i-2}{2} \tag{3.4}$$

follows.

Denote by $T_m(w)$ the time needed to machine M to accept the word wcw. Let us compute the average time $Av_M(w)$ needed for M to accept wcw for an m-long word w.

$$Av_M(w) = \frac{1}{N}\sum_w T_M(w) \geq \frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i},$$

because the running time of a TM on input w is the sum of lengths of the crossing sequences of (M, w, i) for $0 < i < m$. We have inequality here since there are crossing sequences for $i > m$ in the word wcw. Now, we have

$$\frac{1}{N}\sum_w \sum_{i=1}^{m} l_{w,i} = \sum_{i=1}^{m}\sum_w \frac{l_{w,i}}{N} = \sum_{i=1}^{m} p(i)$$

and so, by Eq. (3.4),

$$Av_M(w) \geq \sum_{i=1}^{m} p(i) \geq \sum_{i=1}^{m} \frac{i-2}{2} = \Omega(m^2).$$

So the average running time of M on wcw is $\Omega(m^2)$ implying that there exists an input wcw of length $2 * m + 1$ on which M runs $\Omega(m^2)$ steps.

Therefore, we have proved a lower bound of the worst case complexity of the language-decision problem for $L = \{wcw : w \in \{0,1\}^*\}$ on one-tape Turing machine. This lower bound is $\Omega(m^2)$ for $O(m)$-long input. On the other hand, this problem is decidable in $O(m)$ time by two-tape TM. Therefore

$$DTime_2(O(n)) \not\subseteq DTime_1(o(n^2)).$$

And so Theorem 3 is tight in the sense that there are functions $t(\cdot)$ such that

$$DTime_2(O(t(\cdot))) \not\subseteq DTime_1(o(t^2(\cdot))).$$

Lecture 4

Space Complexity

Notes taken by Leia Passoni and Reuben Sumner

Summary: In this lecture we introduce space complexity and discuss how a proper complexity function should behave. We see that properly choosing complexity functions yields as a result well-behaved hierarchies of complexity classes. We also discuss space complexity below logarithm.

4.1 On Defining Complexity Classes

So far two main complexity classes have been considered: $\mathcal{NP}$ and $\mathcal{P}$. We now consider general complexity measures. In order to specify a complexity class, we first have to set the *model of computation* we are going to use, the specific *resource* we want to bound – time or space – and finally the bound itself, that is the *function* with respect to which we want complexity to be measured.

What kind of functions $f : \mathsf{N} \mapsto \mathsf{N}$ should be considered appropriate in order to define "adequate" complexity classes? Such functions should be computable within a certain amount of the resource they bound, and *that* amount has to be a value of the function itself. In fact, choosing a too complicated function as a complexity function could give as a result that the function itself is not computable within the amount of time or space it permits. These functions are not good in order to understand and classify usual computational problems: even though we can use *any* such function in order do formally define its complexity class, strange things can happen between complexity classes if we don't choose these functions properly. This is the reason why we have defined *time constructible functions* when dealing with time complexity. For the same reason we will here define *space constructible functions*.

4.2 Space Complexity

In space complexity we are concerned with the amount of space that is needed for a computation. The model of computation we will use is a 3-*tape Turing Machine*. We use this model because it is easier to deal with it. We remind that any multi-tape TM can be simulated by an ordinary TM with a loss of efficiency that is only polynomial. For the reminder of this lecture notes, " Turing Machine" will refer to a 3-tape Turing Machine. The 3 tapes are:

1. *input tape*. Read-only

2. *output tape.* Write-only. Usually considered unidirectional: this assumption is not essential but useful. For decision problems, as considered below, one can omit the output-tape altogether and have the decision in the machine's state.

3. *work tape.* Read and write. Space complexity is measured by the bounds on the machine's position on this tape.

Writing is not allowed on the input tape: this way space is measured only on the worktape. If we allowed writing on the input tape then the length of the input itself should be taken into account when measuring space. Thus we could only measure space complexities which are at least linear. In order to consider also sublinear space bounds we restrict the input tape to be read-only. Define $W_M(x)$ to be the index of the rightmost cell on the worktape scanned by M on input x. Define $S_M(n) = max_{|x|=n} W_M(x)$. For any language L define $\chi_L(x)$ so that if $x \in L$ then $\chi_L(x) = 1$ otherwise $\chi_L(x) = 0$

Definition 4.1 (Dspace):

$$\text{Dspace}(s(n)) = \{L \,|\exists \text{a Turing machine } M, M(x) = \chi_L(x) \text{and } \forall n \; S_M(n) \leq s(n)\,\}$$

We may multiply $s(\cdot)$ by $\log_2 |\Gamma_M|$ where Γ_M is the alphabet used by M. Otherwise, we could always linearly compress the number of space cells using a bigger alphabet. We may also add $\log_2(|x|)$ to $s(\cdot)$, where x is the input. (However, this convention disallow treatment of sub-logarithmic space, and therefore will not be done when discussing such space bounds.) This is done in order to have a correspondence to the number of configurations.

Definition 4.2 (Configuration) : *A* configuration *of M is an instantaneous representation of the computation carried on by M on a given input x. Therefore if $|x| = n$ a configuration gives information about the following:*

- *state of M* (O(1) bits)
- *contents of the work tape* ($s(n)$ bits)
- *head position in the input tape* ($\log(n)$ bits)
- *head position in the work tape* ($\log(s(n))$ bits)

4.3 Sub-Logarithmic Space Complexity

Working with sublogarithmic space is not so useful. One may be tempted to think that whatever can be done in $o(\log(n))$ space can also be done in constant space. Formally this would mean

$$\text{Dspace}(o(\log(n))) \subseteq \text{Dspace}(O(1))$$

and since obviously $\text{Dspace}(O(1)) \subseteq \text{Dspace}(o(\log(n)))$, we may also (incorrectly) argue that in fact

$$\text{Dspace}(o(\log(n))) = \text{Dspace}(O(1))$$

This intuition comes from the following imprecise observation: if space is not constant, machine M must determine how much space to use. Determining how much space to use seems to require the machine counting up to at least $|x| = n$ which needs $O(\log(n))$ space. Therefore any M that uses less than $O(\log(n))$ cells, is forced to use constant space. It turns out that this intuition is wrong and the reason is that the language itself can help in deciding how much space to use.

Oded's Note: *This should serve as warning against making statements based on vague intuitions on how a "reasonable" algorithm should behave. In general, trying to make claims about "reasonable" algorithms is a very dangerous attitude to proving lower bounds and impossibility results. It is rarely useful and quite often misleading.*

Note: It is known that Dspace($O(1)$) equal the set of regular languages. This fact will be used to prove the following

Theorem 4.3 Dspace($o(\log(n))$) *is a proper superset of* Dspace($O(1)$).

Proof: We will show that Dspace($o(\log(n))$) $\supseteq$ Dspace($\log\log(n)$) is not contained in Dspace($O(1)$). In fact there is a language L so that $L \in$ Dspace($\log\log(n)$) but $L \notin$ Dspace($O(1)$). For simplicity, we define a language, L, over the alphabet $\{0, 1, \$\}$:

$$L = \left\{ w = 0\cdots0\$0\cdots01\$0\cdots010\$\cdots\$1\cdots1\$ \;\middle|\; \forall k \in \mathsf{N} \begin{array}{l}\text{the } l\text{-th substring of } w \text{ delimited by \$ has}\\ \text{length } k \text{ and is the binary representation}\\ \text{of the number } l-1, \text{ where } 0 \le l < 2^k\end{array} \right\}$$

It can be easily shown that L is not regular using standard pumping lemma techniques. We then prove that $L \in$ Dspace($\log\log(n)$). Note that $L = \{x_k : k \in \mathsf{N}\}$, where

$$x_k = 0^{k-2}\$0^{k-2}01\$0^{k-2}10\$0^{k-2}11\$\ldots\$1^k\$$$

First consider a simplified case where we only measure space when in fact $x = x_k \in L$, $|x_k| = (k+1)2^k$, but we need to check if $x \in L$. We have to

1. Check the first block is all 0's and the last block is all 1's

2. For any two consecutive intermediate blocks in x_k, check that the second is the binary increment by 1 of the first one.

Step (1) can be done in constant space. In Step (2) we count the number of 1's in the first block, starting from the right delimiter \$ and going left until we reach the first 0. If the number of 1's in the first block is i, we then check that in the second block there are exactly i 0's followed by 1. Then we check the remaining $k - i - 1$ digits in the two consecutive blocks are the same. On input x_k, step 2 can be done in $O(\log(k))$ space, which in terms of $n = |x_k| = (k+1)2^k$, means $O(\log\log(n))$ space.

Handling the case where $x \notin L$ while still using space $O(\log\log(n))$ is slightly trickier. If we only proceeded as above then we might be tricked by an input of the form "$0^n\$$" into using space $O(\log(n))$. We think of x being "parsed" into blocks separated by \$, doing this requires only constant space. We avoid using too much space by making k passes on the input. On the first pass we make sure that the last bit of every block is 0 then 1 then 0 and so on. On the second pass we make sure that the last two bits of every block are 00 then 01 then 10 then 11 and then back to 00 and so on. In general on the i^{th} pass we check that the last i bits of each block form an increasing sequence modulo 2^i. If we ever detect consecutive blocks of different length then we reject. Otherwise, we accept if in some (i.e., i^{th}) pass, the first block is of length i, and the entire sequence is increasing mod 2^i. This multi-pass approach, while requiring more time, is guaranteed never to use too much space. Specifically, on any input x, we use space $O(1 + \log i)$, where $i = O(\log|x|)$ is the index of the last pass performed before termination. ■

Going further on, we can consider Dspace($o(\log\log(n))$ and Dspace($O(1)$). We will show that these two complexity classes are equivalent. The kind of argument used to prove their equivalence extends the one used to prove the following simpler fact.

Theorem 4.4 *For any* $s(n) : s(n) \geq \log(n)$ $\quad$ $\text{Dspace}(s(n)) \subseteq \text{Dtime}(2^{o(s(n))})$

Proof: Fix an input $x : |x| = n$ and a deterministic machine M that accepts x in space $s(n)$. Let be $\mathcal{C}$ the number of possible configurations of M on input x. Then an upper bound for $\mathcal{C}$ is:

$$\mathcal{C} \leq |Q_M| \cdot n \cdot s(n) \cdot 2^{o(s(n))}$$

where $Q_M$is the set of states of M, n is the number of possible locations of the head on the input tape, $s(n)$ is the number of possible locations of the head on the worktape and $2^{o(s)}$ is the number of possible contents in the work tape – the exponent is $o(s)$ because the alphabet is not necessarily binary. We can write $s(n) \cdot 2^{o(s(n))} = 2^{o(s(n))}$ and since s is at least logarithmic , $n \leq 2^{o(s(n))}$. Therefore

$$\mathcal{C} \leq 2^{o(s(n))}$$

M cannot run on input x for a time $t(n) > 2^{s(n)}$. Otherwise, M will go through the same configuration at least twice, entering an infinite loop and never stop. Then necessarily M has to run in time $t(n) \leq 2^{o(s)}$. ■

Theorem 4.5 $\text{Dspace}(o(\log_2 \log_2(n)) = \text{Dspace}(O(1))$

Proof: Consider a $s(\cdot)$-bounded machine M on the alphabet $\{0,1\}$.

Claim: given input $x : |x| = n$ such that M accepts x, then M can be on every cell on the input tape at most $k = 2^{s(n)} \cdot s(n) \cdot |Q_M| = O\left(2^{s(n)}\right)$ times. The reason being that if M were to be on the cell more than k times then it would be in the same configuration twice, and thus never terminate.

We define a semi-configuration as a configuration with the position on the input tape replaced by the symbol at the current input tape position. For every location i on the input tape, we consider all possible semi-configurations of M when passing location i. If the sequence of such configurations is $\mathcal{C}^i = \mathcal{C}^i_1, \ldots, \mathcal{C}^i_r$ then by the above claim its length is bounded: $r \leq O\left(2^{s(n)}\right)$. The number of possible different sequences of semi-configurations of M, associated with any position on the input tape, is bounded by

$$\left(2^{s(n)}\right)^{\left(2^{s(n)}\right)} = 2^{2^{O(s(n))}}$$

Since $s(n) = o(\log_2 \log_2 n)$ then $2^{2^{O(s(n))}} = o(n)$ and therefore there exists $n_0 \in \mathsf{N}$ such that $\forall n \geq n_0,\ 2^{2^{O(s(n))}} < \frac{n}{3}$. We then show that $\forall n \geq n_0,\ s(n) = s(n_0)$. Thus $L \in \text{Dspace}(s(n_0)) = \text{Dspace}(O(1))$ proving the theorem.

Assume to the contrary that there exists an n' such that $s(n') > s(n_0)$. Let $n_1 = \min_{|x|>n_0} \{W_M(x) > s(n_0)\}$ and let $x_1 \in \{0,1\}^{n_1}$ be such that $W_M(x_1) > s(n_0)$. That is, x_1 is the shortest input on which M uses space more than $s(n_0)$.

The number of sequences of semi-configurations at any position in the input tape is $< \frac{n_1}{3}$. So labelling n_1 positions on the input tape by at most $\frac{n_1}{3}$ sequences means that there must be at least three positions with the same sequence of semi-configurations. Say $x_1 = \alpha a \beta a \gamma a \sigma$. Where each of the positions with symbol a has the same sequence of semi-configurations attached to it.

Claim: The machine produces the same final semi-configuration with either βa or γa eliminated from the input. For the sake of argument consider cutting β leaving $x_1' = \alpha a \gamma a \sigma$. On x_1' the machine proceeds on the input exactly as with x_1 until it first reaches the a. This is the first entry in our sequence of semi-configurations. Locally, M will make the same decision to go left or right on x_1' as it did on x_1 since all information stored in the machine at at the current read head position is identical. If the machine goes left then its computation will proceed identically on x_1' as on x_1 because it still hasn't seen any difference in input and will either terminate or once again come to the first a. On the other hand consider the case of the machine going right. Say this is the ith time at the first a. We now compare the computation of M to what it did following the ith time going past the second a (after the now nonexistent β). Since the semi-configuration is the same in both cases then on input x_1 the machine M also went right on the ith time seeing the second a. The machine proceeded and either terminated or came back for the $i+1$st time to the second a. In either case on input x_1' the machine M is going to do the same thing but now on the first a. Continuing this argument as we proceed through the sequence of semi-configurations (arguing each time that on x_1' we will have the same sequence of semi-configurations) we see that the final semi-configuration on x_1' will be same as for x_1. The case in which γa is eliminated is identical.

Now consider the space usage of M on x_1. Let $x_2 = \alpha a \beta a \sigma$ and $x_3 = \alpha a \gamma a \sigma$. If peak space usage processing x_1 was in αa or σ then $W_M(x_2) = W_M(x_3) = W_M(x_1)$. If peak space usage was in βa then $W_M(x_3) \leq W_M(x_2) = W_M(x_1)$. If peak space usage was in γa then $W_M(x_2) \leq W_M(x_3) = W_M(x_1)$. Choose $x_1' \in \{x_2, x_3\}$ to maximize $W_M(x_1')$. Then $W_M(x_1') = W_M(x_1)$ and $|x_1'| < |x_1|$. This contradicts our assumption that x_1 was a minimal length string that used more than $s(n_0)$ space. Therefore no such x_1 exists. ■

Discussion: Note that the proof of Theorem 4.4 actually establishes $\text{Dspace}(O(\log\log n)) \neq \text{Dspace}(O(1))$. Thus, combined with Theorem 4.5 we have a separation between $\text{Dspace}(O(\log\log n))$ and $\text{Dspace}(o(\log\log n))$.

The rest of our treatment focuses on space complexity classes with space bound which is at least logarithmic. Theorem 4.5 says that we can really dismiss space bounds below double-logarithmic (alas Theorem 4.4 says there are some things beyond finite-automata that one may do with sub-logarithmic space).

4.4 Hierarchy Theorems

As we did for time, we give now

Definition 4.6 (Space Constructible Function): *A* space constructible function *is a function* $s : \mathsf{N} \mapsto \mathsf{N}$ *for which there exists a machine* M *of space complexity at most* $s(\cdot)$ *such that* $\forall n \;\; M(1^n) = s(n)$

For sake of simplicity, we consider only machines which halt on every input. Little generality is lost by this –

Lemma 4.4.1 *For any space bounded Turing Machine* M *using space* $s(n)$*, where* $s(n)$ *is at least* $\log(n)$ *and space constructible we can construct* $M' \in \text{Dspace}(O(s(n)))$ *such that* $L(M') = L(M)$ *and machine* M' *halts on all inputs.*

Proof: Machine M' first calculates a time bound equal to the number of possible configurations of M which is $2^{s(n)} \cdot s(n) \cdot n \cdot |Q_M|$. This takes space $s(n)$, and same holds for the counter to be maintained in the sequel. Now we simulate the computation of M on input x and check at every step that we have not exceeded the calculated time bound. If the simulated machine halts before reaching its time bound we accept or reject reflecting the decision of the simulated machine. If we reach the time bound before the simulated machine terminates that we are assured that the simulated machine will never terminate, in particular never accept, and we reject the input. ■

Theorem 4.7 (Space Hierarchy Theorem): *For any space-constructible $s_2 : \mathsf{N} \mapsto \mathsf{N}$ and every at least logarithmic function $s_1 : \mathsf{N} \mapsto \mathsf{N}$ so that $s_1(n) = o(s_2(n))$, the class* $\mathrm{Dspace}(s_1(n))$ *is strictly contained in* $\mathrm{Dspace}(s_2(n))$.

We prove the theorem only for machines which halt on every input. By the above lemma, this does not restrict the result in case s_1 is space-constructible. Alternatively, the argument can be extended to deal also with non-halting machines.
Proof: The idea is to construct a language L in $\mathrm{Dspace}(s_2(n))$ such that any machine M using space s_1 will fail recognizing L. We will enumerate all machines running in space s_1 and we will use a diagonalization technique.

- Compute the allowed bound on input x: for instance let it be $\frac{1}{10}s_2(|x|)$.
- Write the language:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M \rangle 01^* \text{ and such that} \\ \text{- } |\langle M \rangle| < \frac{1}{10}s_2(|x|) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most space } \frac{1}{10}s_2(|x|) \end{array} \right\}$$

  Here $\langle M \rangle$ is a binary encoding of the machine M, so we can see $x \in L$ as a description of M itself.

- Show that $L \in \mathrm{Dspace}(s_2(n))$ and $L \notin \mathrm{Dspace}(s_1(n))$.

To see that $L \in \mathrm{Dspace}(s_2(n))$, we write an algorithm that recognizes L:

<u>On input x:</u>

1. Check if x is of the right form
2. Compute the space bound $S \leftarrow \frac{1}{10}s_2(|x|)$
3. Check the length of $\langle M \rangle$ is correct: $|\langle M \rangle| < \frac{1}{10}s_2(|x|)$.
4. Emulate the computation of machine M on input x. If M exceeds the space bound then $x \notin L$ so we reject.
5. If M rejects x then accept. Else, reject.

The computation in Step (1) can be done in $O(1)$ space. The computation of S in Step (2) can be done in space $s_2(|x|)$ because s_2 is space constructible. Step (3) needs $\log(S)$ space. In Step(4) we have to make sure that (# cells M scans) $\times$ ($\log_2 |\Gamma_M|$) $< S$. Checking that M does not exceed the space bound needs space S. As for the implementation of Step(4), on the work tape we first copy the description $\langle M \rangle$ and then mark a specific area in which we are allowed to operate. Then it is possible to emulate the behavior of M going back and forth on the work tape from $\langle M \rangle$ to the

simulated machine's work area, stopping when we are out of space. The algorithm then is running in Dspace($s_2(n)$).
Note: Since we want to measure space, we are not concerned on how much time is "wasted" going back and forth on the worktape from the description of M to the operative area.

Now we have to see that $L \notin$ Dspace($s_1(n)$).
We will show that for every machine M of space complexity s_1, $L(M) \neq L$.

There exists $n : s_1(n) < \frac{1}{10}s_2(n)$ since $s_1(n) = o(s_2(n))$. We then consider $M : |\langle M\rangle| < \frac{1}{10}s_2(n)$ and see how M acts on input x of the form $x = \langle M\rangle 01^{n-(|\langle M\rangle|+1)}$ – note that it is always possible to find inputs of the above form for any sufficiently large n. There are two cases:

1. if M accepts x, then (by definition of L) $x \notin L$.
2. if M rejects x, then since $|\langle M\rangle| < \frac{1}{10}s_2(n)$ and $M(x)$ uses at most $s_1(|x|) < \frac{1}{10}s_2(|x|)$ space, $x \in L$.

In either case $L(M) \neq L$. Therefore any M using space s_1 cannot recognize L. ■

Theorem 4.8 (Time Hierarchy Theorem): *For any time-constructible $t_2 : \mathsf{N} \mapsto \mathsf{N}$ and every at least linear function $t_1 : \mathsf{N} \mapsto \mathsf{N}$ so that $\lim_{n\to\infty} \frac{t_1(n)\log(t_2(n))}{t_2(n)} = 0$, the class* Dtime($t_1$) *is strictly contained in* Dtime(t_2).

Proof: It is analogous to the previous one used for space. The only difference is in the definition of the language L:

$$L = \left\{ x \in \{0,1\}^* \;\middle|\; \begin{array}{l} x \text{ is of the form } \langle M\rangle 01^* \text{ and such that} \\ \text{- } |\langle M\rangle| < \frac{1}{10}\log(t_2(|x|)) \\ \text{- on input } x, M \text{ rejects } x \text{ while using at most time } \frac{1}{10\log t_2(|x|)}t_2(|x|) \end{array} \right\}$$

Dealing with time, we require $|\langle M\rangle| < \log(t_2(|x|))$. The reason for requiring a small description for M is that we cannot implement Step (4) of the algorithm as it has been done with space: scanning $\langle M\rangle$ and going back and forth from $\langle M\rangle$ to the operative area would blow up time. In order to save time we copy $\langle M\rangle$ in the operative area on the worktape, shifting $\langle M\rangle$ while moving on the worktape to the right. If $|\langle M\rangle| < \log(t_2(|x|))$ it takes then time $\log t_2(|x|)$ to copy $\langle M\rangle$ when needed and time $\log(t_2(|x|))$ to scan it. In Step (4) each step of the simulated machine takes time $O(\log(t_2(|x|)))$ so the total execution time will be

$$\log(t_2(|x|)) \cdot \frac{t_2(|x|)}{10 \cdot \log(t_2(|x|))} = O(t_2(|x|))$$

The logarithmic factor we have to introduce in Step (4) for the simulation of M is thus the reason why in Time Hierarchy Theorem we have to increase the time bound by a logarithmic factor in order to get a bigger complexity class. ■

The Hierarchy Theorems show that increasing the time-space bounding functions by any small amount, gives as a result bigger time-space complexity classes – which is what we intuitively would expect: given more resources, we should be able to recognize more languages.

However, it is also clear that the complexity classes hierarchy is strict only if we use proper time/space bounding functions, namely *time and space constructible* functions. This is not the case if we allow any recursive function for defining complexity classes, as it can be seen in the following theorems.

4.5 Odd Phenumena (The Gap and Speed-Up Theorems)

The following theorems are given without proofs, which can be found in [1].

Theorem 4.9 (Borodin's Gap Theorem): *For any recursive function $g : \mathsf{N} \mapsto \mathsf{N}$ with $g(n) \geq n$, there exists a recursive function $s_1 : \mathsf{N} \mapsto \mathsf{N}$ so that for $s_2(n) = g(s_1(n))$, the class* Dspace$(s_1(n))$ *equals* Dspace$(s_2(n))$.

Theorem 4.9 is in a sense the opposite of the Space Hierarchy Theorem: between space bounds $s_1(n)$ and $g(s_1(n))$ there is no increase in computational power. For instance, with $g(n) = n^2$ one gets $g(s_1(n)) = s_1(n)^2$. The idea is to choose $s_1(n)$ that grows very fast and such that even if $g(s_1(n))$ grows faster, no language can be recognized using a space complexity in between.

> Oded's Note: *The proof can be extended to the case where $g : \mathsf{N} \times \mathsf{N} \mapsto \mathsf{N}$ and $s_2(n) = g(n, s_1(n))$. Thus, one can have $s_2(n) = n \cdot s_1(n)$, answering a question raised in class.*

Theorem 4.10 (Blum's Speed-up Theorem): *For any recursive function $g : \mathsf{N} \mapsto \mathsf{N}$ with $g(n) \geq n$, there exists a recursive language L so that for any machine M deciding L in space $s : \mathsf{N} \mapsto \mathsf{N}$ there exists a machine M' deciding L in space $s' : \mathsf{N} \mapsto \mathsf{N}$ with $s'(n) = g^{-1}(s(n))$.*

So there exist languages for which we can always choose a better machine M recognizing them.

> Oded's Note: *Note that an analogous theorem for time-complexity (which holds too), stands in some contrast to the optimal algorithm for solving NP-search problems presented in the previous lecture.*

Bibliographic Notes

Our presentation is based mostly on the textbook [1]. A proof of the hierarchy theorem can also be found in [4]. The proofs of Theorems 4.9 and 4.10 can be found in [1]. Theorem 4.3 and 4.5 are due to [2] and [3] respectively.

1. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

2. Lewis, Stearns, Hartmanis, "Memory bounds for recognition of context free and context sentitive languages", in proceedings of *IEEE Switching Circuit Theory and Logical Design* (old FOCS), 1965, pages 191–202.

3. Stearns, Lewis, Hartmanis, "Hierarchies of memory limited computations", in proceedings of *IEEE Switching Circuit Theory and Logical Design* (old FOCS), 1965, pages 179–190.

4. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

Lecture 5

Non-Deterministic Space

Notes taken by Yoad Lustig and Tal Hassner

Summary: We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced: the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. After the relationships between the non-deterministic models are presented we turn to investigate the relation between the non-deterministic and deterministic space complexity. Savitch's Theorem is presented and we conclude with a translation lemma.

5.1 Preliminaries

During the last lectures we have introduced the notion of space complexity, and in order to be able to measure sub-linear space complexity, a variant model of a Turing machine was introduced. In this model in addition to the work tape(s) and the finite state control, the machine contains two special tapes : an input tape and an output tape. These dedicated tapes are restricted each in it's own way. The input tape is read only and the output tape is write only and unidirectional (i.e. the head can only move in one direction).
In order to deal with non-deterministic space complexity we will have to change the model again, but before embarking on that task, two basic facts regarding the relations between time and space complexity classes should be reminded.

To simplify the description of asymptotic behaviour of functions we define :

Definition 5.1 *Given two functions $f : \mathsf{N} \rightarrow \mathsf{N}$ and $g : \mathsf{N} \rightarrow \mathsf{R}$*
f is at least g *if there exists an $n_0 \in \mathsf{N}$ s.t. for all $n \geq n_0$ $f(n) \geq \lceil g(n) \rceil$.*
f is at least linear *if there exists a linear function g s.t. f is at least g (there exists a constant $c > 0$ s.t. f is at least cn).*

Fact 5.1.1 *For every function $S(\cdot)$ which is at least $log(\cdot)$ $DSPACE(S) \subseteq DTIME(2^{O(S)})$.*

Proof: Given a Turing machine M, a complete description of it's computational state on a fixed input at time t can be given by specifying :

- The contents of the work tape(s).
- The location of the head(s) on work tape(s).
- The location of the head on the input tape.
- The state of the machine.

Denote such a description a *configuration of M*. (Such a configuration may be encoded in many ways, however in the rest of the discussion we will assume a standard encoding was fixed, and would not differentiate between a configuration and it's encoding. For example we might refer to the space needed to hold such a configuration. This is of course the space needed to hold the representation of the configuration and therefore this is a property of the encoding method, however from an asymptotic point of view the minor differences between reasonable encoding methods make little difference). A complete description of an entire computation can be made simply by specifying the configuration at every time t of the computation.

If during a computation at time t, machine M reached a configuration in which it has already been in at time $t_1 < t$, (i.e. the configurations of M at times t_1 and t are identical), then there is a cycle in which the machine moves from one configuration to the next ultimately returning to the original configuration after $t - t_1$ steps. Since M is deterministic such a cycle cannot be broken and therefore M's computation will never end.

The last observation shows that during a computation in which M stops, there are no such cycles and therefore no configuration is ever reached twice. It follows that the running time of such a machine is bounded by the number of possible configurations, so in order to bound the time it is enough to bound the number of possible configurations.

If a machine M never uses more than s cells, then on a given input x, the number of configurations is bounded by the number of possible contents of s cells (i.e. $|\Gamma_M|^s$, where Γ_M is the tape alphabet of machine M),times the number of possible locations of the work head (i.e. s), times the number of possible locations of the input head (i.e. $|x|$), times the number the possible states (i.e. $|S_M|$). If the number of cells used by a machine is a function of the input's length the same analysis holds and gives us a bound on the number of configurations as a function of the input's length.

For a given machine M and input x, denote by $\#conf(M,x)$ the number of possible configurations of machine M on input x. We have seen that for a machine M that works in space $S(\cdot)$ on input x, $\#conf(M,x) = |\Gamma_M|^{S(|x|)} \cdot S(|x|) \cdot |x| \cdot |S_M| = 2^{O(S(|x|))} \cdot |x|$

Therefore in the context of the theorem (i.e. $S(|x|) = \Omega(log(|x|))$) we get that on input x the time of M's computation is bounded by : $\#conf(M,x) = 2^{O(S(|x|))}$ ■

Fact 5.1.2 *For every function $T(\cdot)$ $DTIME(T) \subseteq DSPACE(T)$.*

Proof: Clearly no more then $T(|x|)$ cells can be reached by the machine's head in $T(|x|)$ steps. ■

Note : In the (far) future we will show a better bound (i.e. $DTIME(T) \subseteq DSPACE(\frac{T}{log(T)})$) which is non-trivial.

5.2 Non-Deterministic space complexity

In this section we define and relate three different models of non-deterministic space complexity.

5.2.1 Definition of models (online vs offline)

During our discussion on $\mathcal{NP}$ we noticed that the idea of a non-deterministic Turing machine can be formalized in two approaches, the first approach is that the transition function of the machine is non-deterministic (i.e. the transition function is a multi-valued function), in the second approach the transition function is deterministic but in addition to the input the machine gets an extra string (viewed as a guess): the machine is said to accept input x iff there exists a guess y s.t. the machine's computation on (x,y) ends in an accepting state. (In such a case y is called a witness for x).

In this section we shall try to generalize these approaches and construct a model suitable for measuring non-deterministic space complexity. The first approach can be applied to our standard turing machine model.
Put formally, the definition of a non-deterministic Turing machine under the first approach is as follows :

Definition 5.2 (non-deterministic Turing machine): *A* non deterministic Turing machine *is a Turing machine with a non-deterministic transition function, having a work tape, a read-only input tape, and a unidirectional write-only output tape. The machine is said to accept input x if there exists a computation ending in an accepting state.*

Trying to apply the second approach in the context of space complexity a natural question arises : should the memory used to hold the guess be metered?
It seems reasonable not to meter that memory as the machine does not "really" use it for computation. (Just as the machine does not "really" use the memory that holds the input). Therefore a special kind of memory (another tape) must be dedicated to the guess and that memory would not be metered. However if we do not meter the machine for the guess memory, we must restrict the access to the guess tape, just as we did in the case of the input tape. (surely if we allow the machine to write on the guess tape without being metered and that way get "free" auxiliary memory that would be cheating).
It is clear that the access to the guess tape should be read only.

Definition 5.3 (offline non-deterministic Turing machine): *An* offline non-deterministic Turing machine *is a Turing machine with a work tape, a read-only input tape, a two-way read-only guess tape, and a unidirectional write-only output tape, where the contents of the guess tape is selected non-deterministically. The machine is said to accept input x if there exists contents to the guess tape (a guess string y) s.t. when the machine starts working with x in the input tape and y in the guess tape it eventually enters an accepting state.*

As was made explicit in the definition, there is another natural way in which access to the guess tape can be farther limited: the tape can be made unidirectional (i.e. allow the head to move only in one direction).

Definition 5.4 (online non-deterministic Turing machine): *An* online non-deterministic Turing machine *is a Turing machine with a work tape,a read-only input tape, a unidirectional read-only guess tape (whos contents are selected non-deterministicly), and a unidirectional write-only output tape. Again, the machine is said to accept x if there exists a guess y s.t. the machine working on (x,y)* *will eventually enter an accepting state.*

An approach that limits the guess tape to be unidirectional seems to correspond to an online guessing process – a non-deterministic machine works and whenever there are two (or more) possible ways to continue the machine guesses (online) which way to choose. If such a machine "wants" to "know" which way it guessed in the past, it must record it's guesses (use memory). On the other hand, the approach that allows the guess tape to be two-way corresponds to an offline guessing process i.e. all the guesses are given before hand (as a string) and whenever the machine wants to check what was guessed at any stage of the computation, it can look at the guesses list.

It turns out that the first non-deterministic model and the online model are equivalent. (Although the next claim is phrased for language decision problems, it holds with the same proof for other kinds of problems).

Claim 5.2.1 *For every language L there exists a non-deterministic Turing machine M_N that identifies L in time $O(T)$ and space $O(S)$ iff there exists an online Turing machine M_{on} that identifies L in time $O(T)$ and space $O(S)$.*

Proof: Given M_N it can be easily transformed to an online machine M_{on} in the following way: M_{on} simulates M_N and whenever M_N has several options for a next move (it must choose non-deterministicaly which option to take), M_{on} decides which option to take according to the content of the cell scanned at the guess tape, then move the guess tape head one cell to the right.

In some cases we may want to restrict the alphabet of the guess (for example to $\{0,1\}$). In those cases there is a minor flaw in the above construction as the number of options for M_N's next move may be bigger than the guess alphabet thus the decision which option to take cannot be made according to the content of a single guess tape cell. This is only an apparent flaw since we can assume with out loss of generality that M_N has at most two options to choose from. Such an assumption can be made since a choice from any number of options can be transformed to a sequence of choices from two options at a time by building a binary tree with the original options as leaves. This kind of transformation can be easily implemented on M_N by adding states that correspond to the inner nodes of the tree. The time of the transformed machine has increased at most by a factor of the height of the tree which is constant in the input size.

The transformation from an online machine M_{on} to a non-deterministic machine is equally easy: If we would have demanded that the guess head of M_{on} must advance every time, the construction would have been trivial i.e. at every time M_{on} moves according to it's state and the contents of the cells scanned by the input-tape, work-tape and guess-tape heads, if the contents of the guess cell scanned are not known there may be several moves possible (one for each possible guess symbol), M_N could have simply choose non-deterministically between those. However as we defined it, the guess tape head may stay in place, in such a case the non-deterministic moves of the machine are dependendent (are fixed by the same symbol) untill the guess head moves again. This is not a real problem, all we have to do is remember the current guess symbol, i.e. M_N states would be $S_{M_{on}} \times \Sigma$ where $S_{M_{on}}$ is $M'_{on}s$ states and Σ is the guess alphabet, (M_N being in state (s,a) corresponds to M_{on} being in state s while it's guess head scans a). The transition function of M_N is defined in the natural way. Suppose M_N is in state (s,a) and scans symbols b and c in it's work and input tapes, this correspond to M_{on} being in state s while scanning a,b and c. In this case M_{on} transition function is well defined, (denote the new state by s'), M_N will move the work and input heads as M_{on} moves it's heads, if the guess head of M_{on} stays fixed then the new state of M_N is (s',a), otherwise M_{on} reads a new guess symbol, so M_N chooses non-deterministically a new state of the form (s', a') (i.e. guesses what is read from the new guess tape cell). ∎

These models define complexity classes in a natural way. In the following definitions $M(x, y)$ should be read as "the machine M with input x and guess y".

Definition 5.5 ($NSPACE_{on}$): *For any function* $T : \mathsf{N} \to \mathsf{N}$

$$NSPACE_{on}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an online Turing machine } M_{on} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{on}(x, y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{on} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

Definition 5.6 ($NSPACE_{off}$): *For any function* $T : \mathsf{N} \to \mathsf{N}$

$$NSPACE_{off}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an offline Turing machine } M_{off} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{off}(x, y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{off} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

5.2.2 Relations between $NSPACE_{on}$ and $NSPACE_{off}$

In this section the exponential relation between $NSPACE_{on}$ and $NSPACE_{off}$ will be established.

Theorem 5.7 *For any function* $S : \mathsf{N} \to \mathsf{N}$ *so that* S *is at least logarithmic and* $\log S$ *is space constructible,*
$NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$.

Given an online machine M_{on} that works in space bounded by S we shall construct an offline machine M_{off} which recognizes the same language as M_{on} and works in space bounded by $O(log(S))$. We will see later (Theroem 8) the opposite relation i.e. given an offline machine M_{off} that works in space S, one can construct an online machine M_{on} that recognizes the same language and works in space $2^{O(S)}$.

The general idea of the proof is that if we had a full description of the computation of M_{on} on input x, we can just look at the end of the computation and copy the result (many of us are familiar with the general framework from our school days). The problem is that M_{off} does not have a computation of M_{on} however it can use the power of non-determinism to guess it. This is not the same as having a computation, since M_{off} cannot be sure that what was guessed is really a computation of M_{on} on x. This has to be checked before copying the result. (The absence of the last stage caused many of us great troubles in our school days).

To prove the theorem all we have to show is that checking that a guess is indeed a computation of a space $S(\cdot)$-online machine can be done in $log(S(|x|))$ space. To do that we will first need a technical result concerning the length of computations of such a machine M_{on}, this result is obtained using a similar argument to the one used in the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$).

Proof: *(Theorem 5.7: $NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$):*
Given an online machine M_{on} that works in space bounded by S we shall construct an offline machine M_{off} which recognize the same language as M_{on} and works in space bounded by $O(log(S))$. Using claim 2.1, there exists a non-deterministic machine M_N equivalent to M_{on}, so it is enough to construct M_{off} to be equivalent to M_N.

As in the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$) we would like to describe the state of the computation by a configuration. (As M_N uses a different model of computation we must redefine configuration to capture the full description of the computation at a given moment, however after re-examination we discover that the state of the computation in the non-deterministic model is fully captured by the same components i.e. the contents of the work tape, the location of the work and input tape heads and the state of the machine, so the definition of a configuration can remain the same).

Claim 5.2.2 *If there exists an accepting computation of M_N on input x then there exists such a computation in which no configuration appears more than once.*

Proof: Suppose that $c_0, c_1, \ldots, c_n$ is a description of an accepting computation as a sequence configurations in which some configuration appear more than once. We can assume, without loss of generality that both c_0 and c_n appear only once. Assume for $0 < k < l < n$, $c_k = c_l$. We claim that $c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is also a description of an accepting computation. To prove that, one has to understand when is a sequence of configurations a description of an accepting computation, This is the case if the following hold :

1. The first configuration (i.e. c_0) describes a situation in which M_N starts a computation with input x (initial state, the work tape empty).

2. Every configuration c_j is followed by a configuration (i.e. c_{j+1}) that is possible in the sense that, M_N may move in one step from c_j to c_{j+1}.

3. The last configuration (i.e. c_n) describes a situation in which the M_N accepts.

When $c_{k+1}, \ldots, c_l$ (the cycle) is removed properties 1 and 3 do not change as c_0 and c_n remain the same. Property 2 still holds since c_{l+1} is possible after c_l and therefore after c_k.
$c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is a computation with a smaller number of identical configurations and clearly one can iterate the process to get a sequence with no identical configurations at all. ■

Remark : The proof of the last claim follows a very similar reasoning to the proof of Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), but with an important difference. In the context of non-determinism it is possible that a computation of a given machine is arbitrarily long (the machine can enter a loop and leave it non-deterministicaly). The best that can be done is to prove that short computations exist.

We saw that also arbitrarily long computations may happen, these computations do not add power to the model since the same languages can be recognized if we forbid long computations. A similar question may rise regarding infinite computations. A machine may reject either by halting in a rejecting (non-accepting) state, or by entering an infinite computation, it is known that by demanding that all rejecting computations of a turing machine will halt, one reduces the power of the model (the class R as opposed to RE), the question is is the same true for space bounded machines ? It turns out that this is not the case (i.e. we may demand with out loss of generality that every computation of a space bounded machine halts). By Claim 5.2.2 machine that works in space S works in time $2^{O(S)}$, we can transform such a machine to a machine that always halts by adding a time counter that counts untill the time limit has passed and then halts in a rejecting state (time out). Such a counter would only cost $log(2^{O(S)}) = O(S)$ so adding it does not change the space bound significantly.

Now we have all we need to present the idea of the proof.

Given input x machine M_{off} will guess a sequence of at most $\#conf(M, x)$ of configurations of M_N, and then check that it is indeed an accepting computation by verifying properties 1–3 (in the proof of Claim 5.2.2). If the guess turns out to be an accepting computation, M_{off} will accept otherwise reject.

How much space does M_{off} need to do the task?

The key point is that in order to verify these properties M_{off} need only look at 2 consecutive configurations at a time and even those are already on the guess tape, so the work tape only keeps a fixed number of counters (pointing to the interesting cell numbers on the guess and input tapes).

M_{off} treats it's guess as if it is composed of blocks, each contains a configuration of M_{on}.
To verify property 1, all M_{off} has to do is check that the first block (configuration) describes an initial computational state i.e. check that M_N is in the initial state and that the work tape is empty. That can be done using $O(1)$ memory.
To verify property 2 for a specific couple of consecutive configurations M_{off} has to check that the contents of the work tape in those configurations is the same except perhaps the cell on which M_N's work head was, that the content of the cell the head was on, the state of the machine and the new location of the work head are the result of a possible move of M_N. To do that M_{off} checks that these properties hold for every two consecutive blocks on the guess tape. This can be done using a fixed number of counters (each capable of holding integers upto the length of a single block) + $O(1)$ memory.
To verify property 3 all M has to do is to verify the last block (configuration) describes an accepting configuration. That can be done using $O(1)$ memory.
All that is left is to calculate the space needed to hold a counter. This is the maximum between log the size of a configuration and $log(|x|)$. A configuration is composed of the following parts :

- The contents of the work-tape – $O(S(|x|))$ cells
- The location of the work head – $log(O(S(|x|)))$ cells
- The state of the machine M_N – $O(1)$ cells
- The location of the input head – $O(log(|x|))$ cells

Since S is at least logaithmic, the length of a configuration is $O(S(|x|))$, and the size of a counter which points to location in a configuration is $O(1) + \log(S(|x|)))$.

Comment: Two details which were omitted are (1) the low-level implementation of the verification of property 2, and (2) dealing with the case that the guess is not of the right form (i.e., does not consists of a sequence of configurations of M_{on}). ■

Theorem 5.8 *For any space constractable function $S : \mathsf{N} \to \mathsf{N}$ which is at least logarithmic. $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.*

As in the last theorem, given a machine of one model we would like to find a machine of the other model accepting the same language. This time an offline machine M_{off} is given and we would like to construct an online machine M_{on}.

In such a case the naive approach is simulation, i.e. trying to build a machine M_{on} that simulates M_{off}. This approach would not give us the space bound we are looking for, however, trying to follow that approach will be instructive, so that is what we will do.

The basic approach is to try and simulate M_{off} by an online machine M_{on} (in the previous theorem we did even better than that by guessing the computation and only verifying it's correctness (that way the memory used to hold the computation was free). This kind of trick will not help us here because the process of verification involves comparing two configurations and in an online machine that would force us to copy a configuration to the work tape. Since holding a configuration on the work tape costs $O(S(|x|))$ space we might as well try to simulate M_{off} in a normal way).

Since we only have an online machine which cannot go back and forth on the guess tape, the straightforward approach would seem to be : guess the content of a guess tape for M_{off} then copy it to the work tape of the online machine M_{on}. That gives M_{on} two way access to the guess and now M_{on} can simulate M_{off} in a straight forward way. The only question remains how much space would be needed ? (clearly at least as long as the guess)

The length of the guess can be bounded using a similar analysis to the one we saw at Fact 5.1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), only this time things are a bit more complicated.

If we look on M_{off}'s guess head during a computation it moves back and forth thus it's movement forms a "snake like path" over the guess tape.

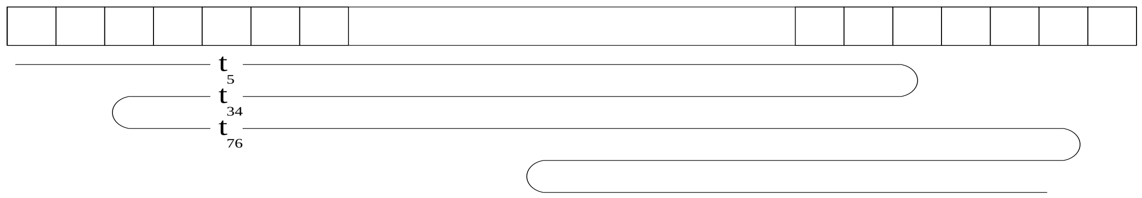


Figure 5.1: The guess head movement

The guess head can visit a cell on the guess tape many times, but we claim the number of times a cell is visited by the head can be bounded. The idea is, as in Fact 5.1.1, that a machine cannot be in the exact same situation twice without entering an infinite loop.

To formalize the last intuition we would need a notion of configuration (a machine's exact situation) this time for an offline machine. To describe in full the computational state of an offline machine one would have to describe all we described in a deterministic model (contents of work tape, location of work and input head and the machine state) and in addition the contents of the guess tape and the location of the guess head. However we intend to use the configuration notion for a very specific purpose, in our case we are dealing with a specific cell on the guess tape while the guess is fixed. Therefore denote by CWG (configuration without guess) of M_{off} its configuration without the the guess tape contents and the guess head location. (exactly the same components as in the non-deterministic configuration). Once again the combinatorial analysis shows us that the number of possible CWGs is $|\Gamma|^{S(|x|)} S(|x|)|S_M|log(|x|)$ which is equal to $\#conf(M,x)$.

Claim 5.2.3 *The number of times during an accepting computation of M_{off} in which the guess tape head visits a specified cell is lesser or equal to $\#conf(M,x)_M = 2^{O(S)}$.*

Proof: If M_{off} visits a single cell twice while all the parameters in the CWG (contents of work tape, location of work and input head and state of the machine) are the same then the entire computation state is the same, because the contents of the guess tape and the input remains fixed throughout the computation. Since M_{off}'s transition function is deterministic this means that M_{off} is in an infinite loop and the computation never stops.

Since M_{off} uses only $S(|x|)$ space there are only $\#conf(M,x)$ possible CWGs and therefore $\#conf(M,x)$ bounds the number of times the guess head may return to a specified cell. ■

Now we can (almost) bound the size of the guess.

Claim 5.2.4 *If for input x there exists a guess y s.t. the machine M_{off} stops on x with guess y, then there exists such a guess y satisfying $|y| < |\Gamma| \cdot \#conf(M,x)^{\#conf(M,x)} = 2^{2^{O(S(|x|))}}$.*

Proof: Denote the guess tape cells $c_0 c_1 \dots c_{|y|}$ and their content $y = g_o \dots g_{|y|}$. Given a computation of M_{off} and a specified cell c_i the guess head may have visited c_i several times during the computation, each time M_{off} was in another CWG. We can associate with every cell c_i the sequence of CWGs M_{off} was in when it visited c_i, denote such a sequence by *visiting sequence of* c_i. (Thus the first CWG in the visiting sequence of c_i is the CWG M_{off} was in the first time the guess head visited c_i, the second CWG in the visiting sequence is the CWG M_{off} was in the second time the guess head visited c_i and so on). By the last claim we get that the length of a visiting sequence is between 0 and $\#conf(M,x)$.

Suppose that for $k < l$, c_k and c_l both have the same visiting sequence and the same content i.e. $g_k = g_l$. Then the guess $g_0, \dots, g_k, g_{l+1}, \dots, g_{|y|}$ is also a guess that will cause M_{off} to accept input x. The idea is the same as we saw in the proof of Claim 5.2.2, i.e. if there are two points in the computation in which the machine is in the exact same situation, then the part of the computation between these two points can be cut off and the result would still be a computation of the machine. To see that this is the case here, we need just follow the computation, when the machine first tries to move from cell c_k to cell c_{k+1} (denote this time t_1^k) it's CWG is the same CWG that describes the machine's state when first moving from cell c_l to c_{l+1} (denote this time t_1^l) therefore we can "skip" the part of the computation between t_1^k and t_1^l and just put the guess head on c_{l+1} and still have a "computation" (the reason for the quatation marks is that normal computations do not have guess head teleportations). By similar reasoning whenever the machine tries to move from c_{l+1} to c_l (or from c_k to c_{k+1}) we can just put the guess head on c_k (respectively c_{l+1}) and "cut off" the part of the computation between the time it moved from c_{l+1} to the correponding time it arrived at c_k (respectively c_k and c_{l+1}). If we would have done exactly that i.e. always "teleporting" the head and cutting the middle part of the computation, we would get a "computation" in which the guess head never entered the part of the guess tape between c_k and c_{l+1} so actually we would have a real computation (this time with out the quotation marks) on the guess $g_0 g_1 \dots g_k g_{l+1} g_{l+2} \dots g_{|y|}$.

Since we can iterate cut and paste process until we get a guess with no two cells with identical visiting sequences and content, we can assume the guess contains no two such cells.

There are $\#conf(M,x)$ possible CWGs therefore $\#conf(M,x)^n$ sequences of n CWGs. Each visiting sequence is a sequence of CWGs of length at most $\#conf(M,x)$ so over all there are $\sum_{i=1}^{\#conf(M,x)} \#conf(M,x)^i \leq \#conf(M,x) \cdot \#conf(M,x)^{\#conf(M,x)} = \#conf(M,x)^{\#conf(M,x)+1} = 2^{2^{O(S(|x|))}}$ possibilities for a visiting sequence. Multiplied by the $|\Gamma|$ possibilities for the guess itself at each guess tape cell, this bounds the length of our short guess. ■

We have succeeded in bounding the length of the guess and therefore the space needed to simulate M_{off} in an online machine using a straightforward approach. Unfortunately the bound is a double exponential bound and we want better. The good news is that during the analysis of the naive approach to the problem we have seen almost all that is necessary to prove Theorem 5.8.

Proof: *(Theorem 5.8: $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.):*
Given an offline machine M_{off} we shall construct an online machine M_{on} that accepts the same language.

In the proof of the last claim (bounding the length of the guess) we saw another way to describe the computation. If we knew the guess, instead of a configuration sequence (with time as an index), one can look at a sequence of visiting sequences (with the guess tape cells as index). Therfore if we add the contents of the guess cell to each visiting sequence, the sequence of the augumented visiting sequences would describe the computation.

Our online machine M_{on} will guess an M_{off} computation described in the visiting sequences form and check whether indeed the guess is an accepting computation of M_{off} (accept if so, reject otherwise). The strategy is very similar to what was done in the proof of Theorem 5.7 (where an offline machine guessed a computation of an online machine and verified it).

To follow this strategy we need to slightly augment the definition of a visiting sequence. Given a computation of M_{off} and a guess tape cell c_i denote by *directed visiting sequence (DVS) of* c_i :

- The content of the guess cell c_i
- The visiting sequence of c_i
- For every CWG in the visiting sequence, the direction from which the guess head arrived to the cell (either R, L or S standing for Right, Left or Stay)

We shall now try to characterize when a string of symbols represents an accepting computation in this representation.

A DVS has the *reasonable returning direction property* if : whenever according to a CWG and cell content the guess head should move right, then the direction associated with the next CWG (returning direction) is left. (respectively the returning direction from a left head movement is right, and from staying is stay).

An ordered pair of DVSs is called *locally consistent* if they appear as if they may be consecutive in a computation i.e. whenever according to the CWG and the guess symbol in one of the DVSs the guess head should move to the cell that the other DVS represents then the CWG in the other DVS that corresponds to the consecutive move of M_{off} is indeed the CWG M_{off} would be in according to the transition function. (The corresponding CWG is well defined because we can count how many times did the head leave the cell of the first DVS in the direction of the cell of other DVS and the corresponding CWG can be found by counting how many time sthe head arrived from that direction). In addition to that, both DVSs must be first entered from the left, and both must have the reasonable returning property.

What must be checked in order to verify a candidate string is indeed an encoded computation of M_{off} on input x ?

1. The CWG in the first DVS is describing an initial configuration of M_{off}.
2. Every two consecutive DVSs are locally consistent.
3. In some DVS the last CWG is describing an accepting configuration.
4. In the last (most right) DVS, there is no CWG that according to it and the symbol on the guess tape the guess head should move to the right.

M_{on} guesses a sequence of DVSs and checks the properties 1–4. To do that, M_{on} never has to hold more then two consecutive DVSs + $O(1)$ memory. Since by Claim 5.2.4 the space needed for a DVS is $log(2^{2^{O(S(|x|))}}) = 2^{O(S(|x|))}$, M_{on} works in space $2^{O(S(|x|))}$. ■

The online model is considered more natural for measuring space complexity (and is equivalent to the first formulation of a non-deterministic Turing machine), therefore it is considered the standard model. In the future when we say "non-deterministic space" we mean as measured in the online model. Thus, we shorthand $NSPACE_{on}$ by $NSPACE$. That is, for any function $S : \mathsf{N} \to \mathsf{N}$, we let $NSPACE(S) \stackrel{\text{def}}{=} NSPACE_{on}(S)$.

5.3 Relations between Deterministic and Non-Deterministic space

The main thing in this section is Savitch's Theorem asserting that non-deterministic space is at most quadratically stronger than deterministic space.

5.3.1 Savitch's Theorem

In this section we present the basic result regarding the relations between deterministic and non deterministic space complexity classes. It is easy to see that for any function $S : \mathsf{N} \to \mathsf{N}$, $DSPACE(S) \subseteq NSPACE(S)$ as deterministic machines are in particular degenerated non-deterministic machines. The question is how much can be "gained" by allowing non-determinism.

Theorem 5.9 (Savitch): *For every space constractable function $S(\cdot)$ which is at least logarithmic $NSPACE(S) \subseteq DSPACE(S^2)$.*

For any non-deterministic machine M_N that accepts L in space S, we will show a deterministic machine M that accepts L in space S^2.

Definition 5.10 (M's configuration graph over x) : *Given a machine M which works in space S and an input string x, M's* configuration graph over *x, G_M^x, is the directed graph in which the set of vertices is all the possible configurations of M (with input x) and there exists a directed edge from s_1 to s_2 iff it is possible for M, being in configuration s_1, to change to configuration s_2.*

Using this terminology, M is deterministic iff the out degree of all the vertices in G_M^x is one.

Since we can assume without loss of generality that M accepts only in one specific configuration (assume M clears the work tape and move the head to the initial position before accepting), denote that configuration by $accept_M$ and the initial configuration by $start_M$. The question whether there exists a computation of M that accepts x can now be phrased in the graph terminology as "is there a directed path from $start_M$ to $accept_M$ in G_M^x".

Another use of this terminology may be in formulating the argument we have repeatedly used during the previous discussions : if there exists a computation that accept x then there exists such a computation in which no configuration appears more than once. Phrased in the configuration graph terminology this reduces to the obvious statement that if there exists a path between two nodes in a graph then there exists a simple path between them. If M works in space $S(|x|)$ then the number of nodes in G_M^x is $|V_M^x| = \#conf(M, x)$ therefore if there exists a path from $start_M$ to $accept_M$ then there is one of length at most $|V_M^x|$.

We reduced the problem of whether M accepts x to a graph problem of the sort "is there a directed path in G from s to t which is at most l long ?". This kind of problem can be solved in $O(log(|l|) \cdot log(|G|))$ space. (The latter is true assuming that the graph is given in a way that enables the machine to find the vertices and the vertices neighbors in a space efficient way, this is the case in G_M^x).

Claim 5.3.1 *Given a graph $G = (V, E)$, two vertices $s, t \in V$ and a number l, in a way that solving the question of whether there exists an edge between two vertices can be done in $O(S)$ space, the question "is there a path of length at most l from s to t" can be answered in space $O(S \cdot log(l))$.*

Proof: If there is a path from s to t of length at most l either there is an edge from s to t or there is a vertex u s.t. there is a path from s to u of length at most $\lceil l/2 \rceil$ and a path from u to t

of length at most $\lfloor l/2 \rfloor$. It is easy to implement a recursive procedure PATH(a, b, l) to answer the question.

```
boolean PATH(a, b, l)
  if there is an edge from a to b then return TRUE
  (otherwise continue as follows :)
    for every vertex v
      if PATH(a, v, ⌈l/2⌉) and PATH(v, b, ⌊l/2⌋)
        then return TRUE
    otherwise return FALSE
```

How much space does PATH(a, b, l) use?

When we call PATH with parameter l it uses $O(S)$ space to store a, b and l, check whether there is an edge from s to t, and handle the for-loop control variable (i.e. v). In addition it invokes PATH twice with parameter $l/2$, but the key point is that both invocations use the same space (or in other words, the second invocations re-uses the space used by the first). Letting $W(l)$ denote the space used in invoking PATH with parameter l, we get the recursion $W(l) = O(S) + W(l/2)$, with end-condition $W(1) = O(S)$. The solution of this relation is $W(l) = O(S \cdot log(l))$.

(The solution is obvious because we add $O(S)$, $log(l)$ times (halving l at every iteration, it will take $log(l)$ iterations to get to 1). The solution is also easily verified by induction, denote by c_1 the constant from the $O(S)$ and $c_2 = 2c_1$, the induction step : $W(l) \leq c_1 S + c_2 S \cdot log(l/2) = c_1 S + c_2 Slog(l) - c_2 S = c_2 Slog(l) + (c_1 - c_2)S$ and for $c_2 > c_1$ we get $W(L) \leq c_2 Slog(l/2)$). ■

Now the proof of Savitch's theorem is trivial.

Proof: *(Theorem 5.9 (Savitch's theorem): $NSPACE(S) \subseteq DSPACE(S^2)$) :*
The idea is to apply Claim 5.3.1 by asking "is there a path from $start_M$ to $accept_M$ in G_M^x ?" (we saw that this is equivalent to "does M accept x"). It may seem that we cannot apply Claim 5.3.1 in this case since G_M^x is not given explicitly as an input, however since the deterministic machine M get x as the input, it can build G_M^x so G_M^x is given implicitly. Our troubles are not over since storing all G_M^x is too space consuming, but there is no need for that, our deterministic machine can build G_M^x on the fly i.e. build and keep in memory only the parts it needs for the operation it performs now then reuse the space to hold other parts of the graph that may be needed for the next operations. This can be done since the vertices of G_M^x are configurations of M_N and there is an edge from v to u iff it is possible for M_N being in configuration v to change for configuration u, and that can easily be checked by looking at the transition function of M_N. Therefore If M works in $O(S)$ space then in G_M^x we need $O(S)$ space to store a vertex (i.e. a configuration), and $log(O(S))$ space to check if there is an edge between two stored vertices, all that is left is to apply the Claim 5.3.1. ■

5.3.2 A translation lemma

Definition 5.11 (NL) *The complexity class* Non-Deterministic logarithmic space, *denoted $\mathcal{NL}$, is defined as $NSPACE(O(log(n)))$.*

Sometimes Savitch's theorem can be found phrased as:

$$\mathcal{NL} \subseteq DSPACE(log(n)^2).$$

This looks like a special case of the theorem as we phrased it, but is actually equivalent to it. What we miss in order to see the full equivalence is a proof that containment of complexity classes "translates upwards".

Lemma 5.3.2 (Translation lemma): *Given S_1, S_2, f space constractable functions s.t. $S_2(f)$ is also space constractible and $S_2(n) \geq log(n), f(n) \geq n$ then if $NSPACE(S_1(n)) \subseteq DSPACE(S_2(n))$ then $NSPACE(S_1(f(n))) \subseteq DSPACE(S_2(f(n)))$.*

Using the translation lemma, it is easy to derive the general Savitch's therorem from the restricted case of NL: Given that $\mathcal{NL} \subseteq DSPACE(log(n)^2)$, given a function $S(\cdot)$ choose $S_1(\cdot) = log(\cdot), S_2(\cdot) = log(\cdot)^2$ and $f(\cdot) = 2^{S(\cdot)}$ (f would be constractible if S was) now, applying the translation lemma, we get that $NSPACE(log(2^S)) \subseteq DSPACE(log(2^S)^2)$ which is equivalent to $NSPACE(S) \subseteq DSPACE(S^2)$.

Proof: Given $L \in NSPACE(S_1(f(n)))$ we must prove the existence of a machine M that works in space $S_2(f(n))$ and accepts L.

The idea is simple, transform our language L of non-deterministic space complexity $S_1(f)$ to a language L^{pad} of non-deterministic space complexity S_1 by enlarging the input, this can be done by padding. Now we know that L^{pad} is also of deterministic space complexity S_2. Since the words of L^{pad} are only the words of L padded, we can think of a machine that given an input pads it and then checks if it is in L^{pad}. The rest of the proof is just carrying out this program carefully while checking that we do not step out of the space bounds for any input.

There exists M_1 which works in space $S_1(f(n))$ and accepts L. Denote by L^{pad} the language $L^{pad} \stackrel{\text{def}}{=} \{x\$^i | x \in L$ and M_1 accpets x in $S_1(|x| + i)$ space.$\}$ where \$ is a new symbol.

We claim now that L^{pad} is of non-deterministic space complexity S_1. To check whether a candidate string s is in L^{pad} we have to check that it is of form $x\$^j$ for some j (that can be done using $O(1)$ space). If so (i.e. $s = x\$^j$), we have to check that M_1 accepts x in $S_1(f(|x|+j))$ space and do that without stepping out of the S_1 space bound on the original input (i.e. $S_1(|s|) = S_1(|x|+j)$). This can be done easily by simulating M_1 on x while checking that M_1 does not step over the space bound (the space bound $S_1(|x|+j)$ can be calculated since S_1 is space constructable). (The resulting machine is referred to as M_2.)

Since L^{pad} is in $NSPACE(S_1)$ it is also in $DSPACE(S_2)$; i.e., there exists a deterministic machine M_3 that recognizes L^{pad} in S_2 space.

Given the deterministic machine M_3 we will construct a deterministic machine M_4 that accepts the original L in space $S_2(f)$ in the following way:

On input x, we simulate M_3 on $x\$^j$ for $j = 1, 2, \ldots$ as long as our space permits (i.e., using space at most $S_2(f(|x|))$, including all our overheads). This can be done as follows: If the head of M_3 is within x, M_4's input head will be on the corresponding point in the input tape, whenever the head of M_3 leaves the x part of the input, M_4 keeps a counter of M_3's input head position (and supplies the simualted M_3 with either \$ or black as appropriate). Recall that we also keep track that M_3 does not use more that $S_2(f(|x|))$ (for that reason we need $S_2(f)$ to be constractible), and if M_3 tries to step out of this bound we will treat it as if M_3 rejected. If during our simulations M_3 accept so does M_4 otherwise M_4 rejects.

Basicly M_4 is trying to find a right j that will cause M_3 to accept, if x is not in L then neither is $x\$^j$ in L^{pad} (for any j) and therefore M_3 will not accept any such string untill M_4 will eventually reject x (which will happen when j is sufficiently large so that $\log j$ superseeds $S_2(f(|x|))$ which is our own space bound). If on the other hand x is in L than M_1 accepts it in $S_1(f(|x|))$ space therefore M_3 accepts $x\$^j$ for some $j \leq f(|x|) - |x|$ (since to hold $f(|x|) - |x|$ one needs only a counter of size $log(f(|x|)$ and S_2 is bigger then log this counter can be kept within the space bound of $S_2(f(|x|))$ and M_4 will get to try the right $x\$^i$ and will eventually accept x). ■

Remark: In the last proof thre was no essential use of the model deterministic or non-deterministic, so by similar argument we can prove analogous results (for example, $DSPACE(S_1) \subseteq DSPACE(S_2)$ implies $DSPACE(S_1(f)) \subseteq DSPACE(S_2(f))$).

By a similar argument we may also prove analogous results regarding time complexity classes. In this case we cannot use our method of searching for the correct padding since this method (while being space efficient) is time consuming. On the other hand, under suitable hypothesis, we can can compute f directly and so do not need to search for the righ padding. We define $L_2^{pad} = \{x\$^{f(|x|)-|x|} : x \in L\}$ and now M_4 can compute $f(|x|)$ and run M_3 on $x\$^{f(|x|)-|x|}$ in one try. There are two minor modifications that have to be done. Firstly, we assume all the functions involved $S_1, S_2, f \geq n$ (this is a reasonable assumption when dealing with time-complexity classes). Secondly, M_2 has to check whether the input $x\$^j$ is indeed $x\$^{f(|x|)-|x|}$; this is easy if it can compute $f(|x|)$ within it's time bounds (i.e., $S_1(|x\$^j|)$), but may not be the case if the input $x\$^j$ is much shorter than $f(|x|)$. To solve that, M_2 only has to time itself while computing $f(|x|)$ and if it fails to compute $f(|x|)$ within the time bound it rejects.

Bibliographic Notes

> Oded's Note: *To be done – find the references for the relationship between the two definitions of non-deterministic space.*

Savitch's Theorem is due to [1]; its proof can be found in any standard textbook (e.g., see textbooks referred to in the previous lecture).

1. W.J. Savitch, "Relationships between nondeterministic and deterministic tape complexities", *JCSS*, Vol. 4 (2), pages 177-192, 1970.

Lecture 6

Inside Non-Deterministic Logarithmic Space

Notes taken by Amiel Ferman and Noam Sadot

Summary: We start by considering space complexity of (decision and search) problems solved by using oracles with known space complexities. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space); We show a problem which is complete for $\mathcal{NL}$, namely the Connectivity problem (Deciding for a given directed graph $G = (V, E)$ and two vertices $u, v \in V$ whether there is a directed path from u to v). Then we prove the somewhat surprising result: $\mathcal{NL} = \text{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

6.1 The composition lemma

The following lemma was used implicitly in the proof of Savitch's Theorem:

Lemma 6.1.1 (composition lemma – decision version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_t$. *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

Proof: Let us fix a certain input x of length n for the machine M. First, it is clear from the definition of M that a space of length at most $s(n)$ is used on M's work-tape for the computation to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Let M_i be (a determinstic) Turing Machine, computing decision task Π_i. Since at each step of its computation, M may query some oracle M_i, it is clear that the contents of each such query depends on the different configurations that M went through until it reached the configuration in which it invoked the oracle. In this sense, the input to M_i is a query that M "decided on". We may deduce that an input to an oracle is bounded by the size of the set of all configurations of machine M on the (fixed) input x (this is the maximal length of a such a query). Let us bound the maximal size of such a query: It is the number of all different configurations of M on input of size n: $|\Sigma_M|^{s(n)} \times s(n) \times n$, where we multiply the number of all possible contents of the work-tape (whose length is bounded by $s(n)$) with the number of possible positions (of the head) on the work-tape and with the number of possible positions on the input-tape (whose length is n) respectively (Σ_M is the work alphabet defined for the machine M).

Since the number of configurations of the machine M on input of length n is $exp(s(n))$, it is clear that the simulation of M_i would require no more than $s_i(exp(s(n)))$. Since we do not need to store the contents of the work-tape after each such simulation, but rather invoke each M_i whenever we need it and erase all contents of the work-tape related to that simulation, it is clear that in addition to the space $s(n)$ of work-tape mentioned above, we need to consider the maximum space that a certain M_i would need during its simulation, hence the result. ■

We stress that the above lemma refers to decision problems, where the output is a single bit. Thus, in the simulation of the M_i's the issue of storing parts of the output of M_i does not arise. Things are different if we compose search problems. (Recall that above and below we refer to deterministic space-bounded computations)

Lemma 6.1.2 (composition lemma – search version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* search tasks $\Pi_1, ..., \Pi_t$. (As we shall see, it does not matter if machine M has a one-way or two-way access to the oracle-reply tape.) *Further suppose that all queries of machine M have length bounded by $exp(s(n))$ and that the answers are also so bounded.*[1] *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

The emulation of the oracles here is more complex than before since these oracles may return strings rather than single bits. Furthermore, the replies to different oracles Π_i's may be read concurrently. In the emulation we cannot afford to run M_i on the required query and store the answer, since storing the answer would use too much space. In order to avoid this, every time we need any bit in the answer to $\Pi_i(q)$, (where q is a query) we need to run M_i again on q and fetch the required bit from the on-line generated output, scanning (and omitting) all other bits; i.e., the answer that would be received from the oracle would be written on the output one bit at a time and by using a counter, the machine could tell when did it reach the desired bit of the answer (this process would halt since the length of the answer is bounded). Note that this procedure is applicable regardless if M has one-way or two-way access to the oracle-reply tape. Note that unlike in Lemma 6.1.1, here we cannot bound the length of the query by the number of possible configurations since this number is too large (as it includes the number of possible oracle answers). Instead, we use the hypothesis in the lemma.

Anallagous, but much simpler, result holds for the time complexity:

Lemma 6.1.3 (composition lemma – time version): *Suppose that machine M solves problem Π while using time $t(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_k$. *Further suppose that for every i, the task Π_i can be solved within time $t_i(\cdot)$. Then, Π can be solved within time $t'(\cdot)$, where $t'(n) \stackrel{\text{def}}{=} t(n) \times \max_i\{t_i(t(n))\}$.*

Proof: Similarly to the proof regarding Space, we shall fix a certain input x of length n for the machine M. First, it is clear from the definition of M that time $t(n)$ is suffices for the computation of M on x to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Here at each step of the computation M could invoke an oracle M_i and so it is clear that the time complexity of the computation would be $t(n)$ multiplied by the

[1] Without this assumption, we cannot bound the number of configurations of machine M on a fixed input, as the configuration depends on the location of M's head on the oracle-reply tape.

maximal time complexity of an oracle. In order to find the time complexity of some oracle M_i, we have to consider the possible length of a query to M_i; since there are $t(n)$ time units during the computation of M on x, the size of the query to M_i could be at most $t(n)$.

We deduce that the time complexity of some oracle M_i which is invoked at some point of the computation of M on x, could be at most $t_i(t(n))$. According to what was said above, the time complexity of M on x would be the number of time units of its computation - $t(n)$ - multiplied by the maximal time complexity of some oracle M_i $(1 \leq i \leq k)$, hence the result. ■

6.2 A complete problem for $\mathcal{NL}$

The complexity class $\mathcal{NL}$ is defined to be simply $NSPACE(O(log(n)))$. More formally we have:

Definition 6.1 *$\mathcal{NL}$: A language L belongs to $\mathcal{NL}$ if there is a nondeterminstic Turing machine M that accepts L and a function $f(n) = O(log(n))$ such that for every input x and for every computation of M at most $f(|x|)$ different work-tape cells are used.*

Our goal in this section and the following one would be to study some properties of the class $\mathcal{NL}$. To that end we define the following:

Definition 6.2 *A log-space reduction of L_1 to L_2 is a log-space computable function f such that $\forall x,\ x \in L_1 \Leftrightarrow f(x) \in L_2$*

Note that a log-space reduction is analagous to a Karp-reduction (where space corresponds to time and the logarithmic number of cells correspond to polynomial number of steps). Actually, since each function that can be computed in space $s(\cdot)$, can also be computed in time $exp(s(\cdot))$, we have that a log-space reduction is a special case of a polynomial time reduction. The next definition would define a notion analagous to $\mathcal{NP}$-completeness (as we will see, this would prove useful in proving a proposition about $\mathcal{NL}$ which is analagous to a proposition in $\mathcal{NP}$):

Definition 6.3 *L is $\mathcal{NL}$-Complete if:*

(1) *$L \in \mathcal{NL}$; and*

(2) *$\forall L' \in \mathcal{NL}$, L' is log-space reducible to L.*

6.2.1 Discussion of Reducibility

As implied from the definitions above, our goal would be to find a problem which is complete for the class $\mathcal{NL}$. Prior to that, we must make sure that the concept of completness is indeed meaningful for the class $\mathcal{NL}$. The following propositions ensure exactly that.

Proposition 6.2.1 *If L is log-space reducible to L' and L' is solvable in log-space then L is solvable in log-space.*

Proof: Since L' is solvable in logarithmic space, there exists a machine M' which decides L' using logarithmic space. Furthermore, since L is log-space reducible to L', there exists a function $f(\cdot)$ computable in log-space, such that $x \in L \Leftrightarrow f(x) \in L'$ and so there exists a machine M such that for every input x would first compute $f(x)$ and then would simulate M' in order to decide on $f(x)$, both actions demanding log-space (as $lg(|f(x)|) \leq lg(exp(lg(|x|))) = O(lg(|x|)))$ and ensuring that M would accept x iff $x \in L$. ■

Interestingly, such reduction also preserve non-deterministic space:

Proposition 6.2.2 *If L is log-space reducible to L' and $L' \in \mathcal{NL}$ than $L \in \mathcal{NL}$*

Instead of proving the last proposition, we will prove a related proposition regarding Non-Deterministic Time:

Proposition 6.2.3 *If L if Karp-Reducible to L' and $L' \in \mathcal{NP}$ then $L \in \mathcal{NP}$*

Proof: Since L is Karp-Reducible to L', there is a many-to-one function $f(\cdot)$, computable in polynomial time, such that: $x \in L \Leftrightarrow f(x) \in L'$. Furthermore, since L' is in $\mathcal{NP}$, there is a Non-Deterministic Turing machine M' that can guess a witness y for an input z (the length of y is a polynomial in the size of z) in polynomial time such that $R_{L'}(z, y)$ holds (where $R_{L'}$ is the relation that defines the language L' in $\mathcal{NP}$). We will construct a Non-Determinstic machine M for deciding L in the following way: For a given input $x \in L$, M will compute $f(x)$ (deterministically in polynomial time) and then would just simulate M' (mentioned above) on input $f(x)$ to find a witness y (non-deterministically in polynomial time) such that $R_{L'}(f(x), y)$ would hold. Thus, M defines a relation R_L such that for every input $x \in L$, it guesses a witness y (non-determinstically in polynomial time) such that $R_L(x, y)$ holds (i.e., $R_L(x, y) = R_{L'}(f(x), y)$). So by definition, L is in $\mathcal{NP}$. ■

We can use the proof of Proposition 6.2.3 to prove Proposition 6.2.2: Instead of a function $f(\cdot)$ computable in polynomial time, we are guarnteed to have a function $f(\cdot)$ which is computed in logarithmic space. Furthermore we may presume the existence of a machine M' deciding the language L' in logarithmic space (instead of non-determinstic polynomial time). It is now clear, that one may construct a non-deterministic machine M which may decide the language L in logarithmic space (which is analagous to the machine M which decided L in non-deterministic polynomial time).

Note that requiring the existence of a Cook-Reduction instead of a Karp-Reduction in Proposition 6.2.3 would probably make this proposition false: This stems from the fact that if a language L is Cook-Reducible to a language $L' \in \mathcal{NP}$ it does not necessarily mean that $L \in \mathcal{NP}$. In particular, any co$\mathcal{NP}$ language is Cook-Reducible to its complement. Still, if $\mathcal{NP} \neq$ co$\mathcal{NP}$ we have that $\overline{SAT} \notin \mathcal{NP}$ (and yet $\overline{SAT}$ is reducible to SAT). We conclude that if $\mathcal{NP} \neq$ co$\mathcal{NP}$ then Cook reductions are strictly more powerful than Karp reductions (since the class of languages which are Cook-reducible to $\mathcal{NP}$ contains co$\mathcal{NP}$, whereas the languages which are Karp-reducible to $\mathcal{NP}$ are exactly $\mathcal{NP}$). A more trivial example of this difference in power is the fact that any language in $\mathcal{P}$ is Cook-reducible to the empty set, whereas only the empty set is Karp-reducible to the empty set.

However, in the next proposition, as well as in Proposition 6.2.1, a Cook-reduction would do:

Proposition 6.2.4 *If L is polynomial-time reducible to L' and $L' \in P$ then $L \in P$*

In this last proposition, if L would be Cook-Reducible to L' then it is clear that the machine that emulates the oracle machine and answeres the queries by simulating the macine that decides L' (and runs in polynomial time), would be a polynomial-time machine that decides L (here the use of the oracle on L' and its actual simulation didn't make a difference in the running time). An analouge argument applies to Proposition 6.2.1. That is, if there exists a log-space oracle machine which deecides L by making polynomially-bounded queries to L', and L' is solvable in log-space then so is L (actually this follows from Lemma 6.1.1).

6.2.2 The complete problem: directed-graph connectivity

The problem that we will study would be graph connectivity (denoted as CONN) which we define next:

Definition 6.4 (directed connectivity – CONN): *$CONN$ is the is defined as a set of triples, (G, v, u), where $G = (V, E)$ is a directed graph, $v, u \in V$ are two vertices in the graph so that there is is a directed path from v to u in G*

As we shall see, the problem CONN is a natural problem to study in the context of space complexity. Intuitively, a computation of a Turing machine (deterministic or not) on some fixed input, could allways be pictured as a graph with nodes realting to the machine configurations and edges relating to transitions between configurations. Thus the question of whether there exists a certain accepting computation in the machine reduces to the question of the existence of a certain directed path in a graph: that path which connects the node which corresponds to the initial configuration and the node which corresponds to an accepting configuration (on a certain input). We note that in a deterministic machine the out-degree of each node in the graph would be exactly one, while in a non-determinstic machine the out-degree of each node could be any non-negative number (because of the possibilty of the non-determinstic machine to move to any one of a certain configurations), however in both cases the out-degree of each node is constant (depending only on the machine). Continuing this line of thought, it's not hard to see that CONN could be proved to be complete for the class $\mathcal{NL}$, i.e. it is itself in $\mathcal{NL}$ and every machine in $\mathcal{NL}$ could be reduced to it. The details are proved in the following:

Theorem 6.5 *$CONN$ is $\mathcal{NL}$-Complete.*

> Oded's Note: *The following proof is far too detailed to my taste. The basic ideas are very simple. Firstly, it is easy to design a non-deterministic log-space machine which accepts $CONN$ by just guessing an adequate directed path. Secondly, it is easy to reduce any language $L \in \mathcal{NL}$ to $CONN$ by just considering the directed graph of configurations of a log-space machine (accepting L) on the given input, denoted x. Each such configuration consists of a location on the input-tape, a location of the work-tape, the contents of the work-tape and the state of the machine. A directed edge leads from one configurations to another iff they are possible consequetive configuration of a computation on input x. The key point is that the edge relation can be determined easily by examining the two configurations and the relevant bit of x (pointed to in the first configuration).*

Proof: First we show that $CONN \in \mathcal{NL}$ (see Definition 6.3). We will build a machine M that would decide for the input $G = (V, E)$ and $v, u \in V$ whether there exists a path in G from v to u. Of course, M would do so non-determinstically in $O(log(n))$ space where n is the size of the input. The outline of the algorithm is as follows: We start with the node v (given at the input) and a counter which is initialized to the number of nodes in G. At each step we decrement the counter and guess a node which is adjacent to the current node we have (initially v). If the node we have guessed is not adjacent to the node we hold, we just reject. No harm is done since this is a non-deterministic computation: it suffices that there will be some computation that would accept. This procedure concludes when either the counter reaches 0 (the path shold not be longer than the number of nodes) or the last node we have guessed is u (the other node specified in the input). The actual guessing of a node could be done in several ways, one would be to implement a small procedure that would non-deterministically write on the work tape symbols that would encode some node (by scanning the list of edges or adjacency matrix which are part of the input). Then, it will just check wheather the node it had written is adjacent to the current node we hold. Correctness and complexity analysis follow the formal specification of the algorithm:

Input: $G = (V, E)$, $v, u \in V$

Task: Find whether there exists a directed path from v to u in G.

1. $x \leftarrow v$
2. counter $\leftarrow |V|$
3. repeat
4. decrement counter by 1
5. guess a node $y \in V$ s.t. $(x, y) \in E$
6. if $y \neq u$ then $x \leftarrow y$
7. until $y = u$ or counter $= 0$
8. if $y = u$ then accept, else reject

First we will prove the correctness of this algorithm: On the one hand, suppose that the algorithm accepts the input $G = (V, E), v, u \in V$. This implies that during the repeat-until loop, the algorithm has guessed a sequence of nodes such that each one of them had a directed edge to its successor and that the final node in this sequence is u and the initial node is v (from the first line and the check that is made in the last line). Clearly, this implies a directed path from v to u in G (Note that from the existence of the counter, the number of steps in this computation is bounded by $O(n)$). On the other hand, suppose that there is a directed path in G from v to u. This path is a sequence $\{v, x_1, \ldots, x_k, u\}$ where $k \leq (n-2)$ and there is a directed edge from each node in this sequence to its successor. In this case it is clear that the computation of the algorithm above in which it guesses each one of the nodes in the sequence starting from v (from the first line of the algorithm) and ending with u (from the last line of the computation) is an accepting computation, and thus the algorithm would accept the input $G = (V, E), v, u \in V$.

We conclude that there is an accepting computation of the algorithm above on the input $G = (V, E), v, u \in V$ iff there is a directed path in G from v to u.

All that is left to be shown is that the implementation of such an algorithm in a non-determinstic machine would require no more than a logarithmic space in the size of the input: First, it is clear that each one of the variables required to represent a node in the graph need not be represented in more than logarithmic number of cells in the size of the input (for example, in order to represent a number n in binary notation, we need no more than $lg(n)$ bits). The same argument applies to the counter which has to count a number which is bounded by the size of the input. Secondly, all other data besides the variables may be kept at a constant number of cells of the work-tape (for example a bit that would indicate whether $y = u$ etc.). As was specified above regarding the implementation of step 5 (the guessing of the node), the actuall guessing procedure, which would be done non-determinstically, uses number of cells which is equal exactly to the length required to the representation of a node (which is again logarithmic in the size of the input). We conclude that the implementation of the algorithm above on a non-determinstic machine M requires a logarithmic space in the size of the input, and so we may conclude that the machine M decides CONN in non-deterministic logrithmic space, i.e., $CONN \in \mathcal{NL}$.

Now we need to show that every language $L \in \mathcal{NL}$ is log-space reducible to CONN. Let L be a language in $\mathcal{NL}$, then there is a non-deterministic logarithmic space machine M that decides L. We will show that for every input x, we can build in nondeterministic logarithmic space an input $(G = (V, E), start \in V, end \in V)$ (which is a function of machine M and input x) such that there is a path in G from *start* to *end* if and only if M accepts input x. The graph G we will construct would simply be the graph of all possible configurations of M given x as an input. That is, the nodes denote different configurations of M while computing on input x, and the arcs denote possible immediate transitions between configurations.

The graph is constructed (deterministic) log-space as follows;

Input: An input string x (the machine M is fixed)

Task: Output a graph $G = (V, E)$ and two nodes $v, u \in V$ such that there is a path from v to u in the graph iff x is accepted by M.

1. compute n, the number of different configurations of M while computing input x
2. for $i = 1$ to n
3.   for $j = 1$ to n
4.   if there is a transition (by a single step of M) from configuration number i to configuration number j ouput 1 otherwise output 0
5. output 1 and n

First we will show that this procedure indeed outputs the representation of a graph and two nodes in that graph such that there exists a directed path between those two nodes iff the input x is accepted by machine M. In the first line we compute the number of all possible configurations of machine M while computing on input x. Then, we consider every ordered pair of configurations (represented by numbers between 1 and n) and output 1 iff there is indeed a directed transition between those two configurations in the computation of M on x. Our underlying assumption is that 1 represents the initial configuration and n represents the (only) accepting configuration (if there were several accepting configurations we define a new one and draw edges from the previous accepting configurations to the new one). Thus, the output of the above procedure is simply an adjacency matrix of a graph in which each of its nodes correspond to a unique configuration of M while computing on input x , and a directed edge exists between two nodes i and j iff there is a (direct) transition in M between the configuration represented by x and the configuration represented by y. It is now clear that a directed path from the first node (according to our enumeration) to the last node in the graph would correspond to an accepting computation of machine M on input x, and that such a path would not exists should there be no such accepting computation.

Next, we must show that the above procedure could indeed be carried out using no more than a logarithmic space in the length of the input (i.e., the input x). In order to do that, we will show that the number of different configurations of M while computing x is polynomial in the length of x. This would imply that in order to count these configurations we need no more than a logarithmic space in the length of x. So, we will count the number of possible configurations of M while computing on a given input x. That number would be the number of possible states (a constant determined by M) multiplied by the number of possible contents of the work-tape which is $|\Sigma_M|^{O(log(n))}$ where Σ_M is the alphabet of M and is also a constant determined by M (let us remember that since M is log-space bounded, the number of used work-tape squares could not surpass $O(log(n))$), multiplied by the number of different positions of the reading head on the input tape which is n and finally, mutiplied by the number of different possible positions of the reading head on the work-tape which is $O(log(n))$. All in all, the number of different configurations of M while computing input x is: $|States_M| \times |\Sigma_M|^{O(log(n))} \times n \times O(log(n)) = O(n^k)$ (where k is a constant). That is, the number of different configurations of M while computing input x is polynomial in the length of x.

We may conclude now that the initial action in the procedure above, that of counting the number of different configurations of M while computing on input x could be carried out in logarithmic space in the length of x.

Secondly, we show that the procedure of checking whether there exists a (direct) transition between two configurations represented by two integers can be implemented as well in logarithmic space: We show that there is a machine M'' that receives as inputs two integers and would return a positive answer if and only if there is a (direct) transition of M between the two configurations which are represented by those integers. Note first that integers correspond to strings over some convinient alphabet and that such strings correspond to configurations. (The correspondance is by trivial computations.) Thus, all we need to determine is whether the fixed machine M can pass in one (non-deterministic) step, on input x, between a given pair of configurations. This depends only on the transition function of M, which M'' has hard-wired, and on a single bit in the input x; that is, the bit the location of which is indicated in the first of the two given configurations. That is, suppose that the first configuration is of the form (i, j, w, s), where i is a location on the input-tape, j a location on the work-tape, w the contents of the work-tape and s the machine's state. Same for the second configuration, denoted (i', j', w', s'). Then we check if M when reading symbol x_i (the i^{th} bit of x) from its input tape, and w_j (the j^{th} symbol of w) from its work-tape can make a single transition resulting in the configuration (i', j', w', s'). In particular, it must hold that $i' \in \{i-1, i, i+1\}$, $j' \in \{j-1, j, j+1\}$, and w' differs from w at most on location j. Furthermore, these small changes must depend on the transition function of M. Since there is a constant number of possible transitions (in M's transition function), we may just check all of them.

We have shown that the above procedure outputs a representation of a graph and two nodes in that graph for a machine M and input x, such that there is a directed path between the nodes iff there is an accepting computation of M on x. Furthermore we have shown that this procedure may be implemented requiring no more than a log-space in the size of the input (the input string x) which concludes the proof. ■

6.3 Complements of complexity classes

Definition 6.6 (complement of a language): *Let $L \subseteq \{0,1\}^*$ be a language. The* complement of a language *L, denoted $\overline{L}$ is the language $\{0,1\}^* \backslash L$.*

To make this definition more accurate, we assume that every word in $\{0,1\}^*$ represents an instance of the problem.

Example 6.7 *: $\overline{CONN}$ is the following set: $\{(G, u, v) : G$ is a directed graph, $u, v \in V(G)$, there is no directed path from u to v in G $\}$.*

Definition 6.8 (complement of class): *Let $\mathcal{C}$ be a complexity class. The* complement of the class $\mathcal{C}$ *is denoted* co$\mathcal{C}$ *and is defined to be* $\{\overline{L} : L \in \mathcal{C}\}$

It is immediately obvious that if $\mathcal{C}$ is a deterministic time or space complexity class, then co$\mathcal{C} = \mathcal{C}$, in particular, $\mathcal{P} = \mathrm{co}\mathcal{P}$. This is true, since we can change the result of the derministic machine from 'yes' to 'no' and vice versa.

However, in non-deterministic complexity classes, this method does not work. Let M be a Turing Machine that accepts a language L non-deterministicly. If $x \in L$, then there is at least one successful computation of M on x (i.e., there is a succinct verification that $x \in L$). We denote by $\overline{M}$ the non-deterministic Turing Machine that does the same as M, but replaces the output from "yes" to "no" and vice versa. Hence, if the new machine $\overline{M}$ accepts an input z, there is one accepting computation for $\overline{M}$ on z, i.e. non-accepting computation in M (by definition). In other words, $\overline{M}$ will accept z, if M has some unsuccessful guesses to prove that $z \in L$. This, however, does not mean that $\overline{M}$ accepts $\overline{L}$, since z could possibly be in L by other guesses of the machine M. For example we don't know whether co$\mathcal{NP}$ is equal to $\mathcal{NP}$. The conjecture is that $\mathcal{NP} \neq \mathrm{co}\mathcal{NP}$

Yet, in the particular case of nondeterministic space equality does hold. It can be proven that any non-deterministic space $NSPACE(s(n))$ for $s(n) \geq log(n)$ is closed under complementation. This result which was proven by Neil Immerman in 1988, is going to be proven here for the case $\mathcal{NL}$. By the following proposition, it suffices to show that $CONN \in \mathrm{co}\mathcal{NL}$ (or equivalently $\overline{CONN} \in \mathcal{NL}$.

Proposition 6.3.1 *: If for an $\mathcal{NL}$-complete language L it holds that $L \in \mathrm{co}\mathcal{NL}$ then $\mathcal{NL} = \mathrm{co}\mathcal{NL}$.*

Proof: Let L' be a language in $\mathcal{NL}$. Since $L \in \mathcal{NL}$-complete, we have a log-space reduction f from L' to L (see Definition 6.2). The function f satisfies:

$$x \in L' \Leftrightarrow f(x) \in L$$

Taking the opposite direction, we get:

$$x \in \overline{L'} \Leftrightarrow f(x) \in \overline{L}$$

By the definition of the reduction, f is also a reduction from $\overline{L'}$ to $\overline{L}$. By proposition 6.2.2 we know that since $\overline{L} \in \mathcal{NL}$ (because by hypothesis $L \in \mathrm{co}\mathcal{NL}$) then $\overline{L'} \in \mathcal{NL}$ (i.e. $L' \in \mathrm{co}\mathcal{NL}$). We conclude that for every $L' \in \mathcal{NL}$, $\overline{L'} \in \mathcal{NL}$, thus $\mathcal{NL} = \mathrm{co}\mathcal{NL}$. ■

6.4 Immerman Theorem: $\mathcal{NL} = \mathrm{co}\mathcal{NL}$

In this section, we are going to prove a surprising theorem, which claims that non-deterministic log-space is closed under complementation. Due to the proof in Theorem 6.5 that $CONN \in \mathcal{NL}$-complete, and using proposition 6.3.1, we only need to prove that $\overline{CONN} \in \mathcal{NL}$, where $\overline{CONN}$ is the complementary problem of $CONN$ as defined in Example 6.7. Formally, The decision problem of the language $\overline{CONN}$, is obtained as the following:

Input: a directed graph $G = (V, E)$ and two nodes $u, v \in V(G)$.

Question: Is there no directed path from v to u ?

In order to show that $\overline{CONN} \in \mathcal{NL}$, we use the following theorem, which will be proven later:

Theorem 6.9 *Given a directed graph $G = (V, E)$ and a node $v \in V(G)$, the number of nodes reachable from v in G can be computed by a non-deterministic Turing Machine within log-space.*

A non-deterministic Turing Machine that computes a function f of the input, as in the theorem above, is defined as follows:

Definition 6.10 (non-deterministic computation of functions): *A non-deterministic Turing Machine M is said to compute a function f if on any input x, the following two conditions hold:*

1. *either M halts with the right answer $f(x)$, or M halts with output "failure"; and*
2. *at least one of the machine computations halts with the right answer.*

6.4.1 Theorem 6.9 implies $\mathcal{NL} = \text{co}\mathcal{NL}$

Lemma 6.4.1 *Assuming Theorem 6.9, $\overline{CONN} \in \mathcal{NL}$*

Assuming Theorem 6.9, we have a non-deterministic Turing Machine denoted CR ($CR \stackrel{\text{def}}{=}$ Count Reachable) that counts all the nodes that are reachable in a directed graph G from a single node v in non-deterministic log-space. The idea of the proof is that once we know to compute using CR the number of nodes reachable from v in G, we can also non-deterministically scan all the vertices reachable from v, using this value. This is done non-deterministically by guessing connected paths to each of the reachable nodes from v. Once the machine discovered all reachable nodes from v (i.e. the number of reachable nodes it found equals the output of CR) and the node u isn't reachable, it can decide that there is no connected path between v and u in G.

Proof: Let $x = (G, u, v)$ be an input for the problem $\overline{CONN}$. We fix (G, v) and give it as an input to the machine CR, which is the non-deterministic machine that enumerates all the nodes in the graph G reachable from the node v as was assumed before to work in non-deterministic log-space. In other words, we use CR, as a "black box".

We construct a non-deterministic machine here that uses the following simulation that solves this problem:

- Firstly, it simulates CR on the input (G, v). If the run fails, the machine rejects. Otherwise, we denote the answer by N.
- For each vertex w in the graph, it guesses whether w is reachable from v and if yes, it guesses non-deterministicly a directed path from v to w, by guessing at most $n-1$ vertices (we know by a simple combinatorial fact, that each two connected nodes in a grpah $G = (V, E)$ are connected within path of length less than or equal to $n-1$) and verifies that it is a valid path. For each correct path, it increments a counter.
- If $w = u$, and it founds a valid path then it rejects.
- If $counter \neq N$ then the machine rejects the input, otherwise the machine accepts the input. ($counter \neq N$ means that not all reachable vertices were found and verified, whereas $counter = N$ means that all were examined. If none of these equals u then we should indeed accept).

Formally, we have the following algorithm:

Input: $G = (V, E)$, $v, u \in V(G)$
Task: Find whether there is no connected path between u and v.

1. **Simulating CR. If CR fails, the algorithm rejects, else $N \leftarrow CR((G,v))$.**

2. *counter* $\leftarrow 0$

3. **for** $w = 1$ **to** n **do** *(w is a candidate reachable vertex)*

4. **guess if w is reachable from v. If not, proceed to next iteration of step 3.**
(we continue in steps 5-17 only if we guessed that w is reachable from v)

5. $p \leftarrow 0$ *(counter for path length)*

6. $v_1 \leftarrow v$ *(v_1 is initially v)*

7. **repeat** *(guess and verify a path from v to w)*

8. $p \leftarrow p + 1$

9. **guess a node** v_2 *(v_1 and v_2 are the last and the current nodes)*

10. **if** $(v_1, v_2) \notin E$ **then reject**

11. **if** $v_2 \neq w$ **then** $v_1 \leftarrow v_2$

12. **until** $(v_2 = w)$ **or** $(p = n - 1)$

13. **if** $(v_2 = w)$ **then** *(counting all reachable $w \neq u$)*

14. **begin**

15. *counter* $\leftarrow$ *counter* $+ 1$

16. **if** $w = u$ **then reject**

17. **end**

18. **if** $N \neq$ *counter* **then reject, else accept.**

We know that CR works in $O(log(|G|))$. In each step of the simulation, our algorithm uses only 6 variables in addition of those of CR, namely the counters $counter, w, p$, the representations of the nodes v_2, v_1 and N. the counters, and N are bounded by the number of vertices, n. Every new change of one of this variables will be written again on the work tape by reusing space. Therefore, they can be implemented in $O(log(n))$ space. The nodes, clearly, are represented in $O(log(|G|))$ space. Thus, we use no more then $O(log(|G|))$ space in the work tape in this machine (where x is the input). The correctness is proved next:

To show correctness we need to show that it has a computation that accepts the input if and only if there is no direct path from v to u in G.

Consider first the case that the machine accepts. A necessary condition (for this event) is that $counter = N$ (line 18); that is, the number of vertices that were found to be reachable is exactly the correct one (i.e., N). This means that every possible vertex that is reachable from v was counted. But, if u was found to be one of them the machine should have rejected before (in line 16). Therefore, u cannot be reachable from v by a directed path.

Suppose, on the other hand, that there is no directed path between u and v in G. Then if all guesses made are correct then the machine will necessarily accept. Specifically, we look at a computation in which (1) the machine correctly guesses (in line 4) for each vertex w whether it is reachable from v; (2) for each reachable vertex w it guesses well a directed path from v; and (3) machine CR did not fail (and thus N equals the number of vertices reachable from v). In this case $N = counter$, and since u is not connected to v the machine accepts.

Therefore, we proves the lemma. ■

Using this result (under the assumption that Theorem 6.9 is valid), we obtain $\mathcal{NL} = \text{co}\mathcal{NL}$.

Theorem 6.11 (Immerman 88'): $\mathcal{NL} = \text{co}\mathcal{NL}$

Proof: We proved in Theorem 6.5 that $CONN \in \mathcal{NL}$-complete. In Lemma 6.4.1, we proved that $\overline{CONN} \in NL$ (or $CONN \in \text{co}\mathcal{NL}$). Using Proposition 6.3.1, we get, that $\mathcal{NL} = \text{co}\mathcal{NL}$. ■

An extension of this theorem can show that for any $s(n) \geq log(n)$, $NSPACE(s(n)) = coNSPACE(s(n))$.

6.4.2 Proof of Theorem 6.9

To conclude this proof we are only left with the proof of Theorem 6.9, i.e. the existness of a machine CR, that computes the number of nodes reachable from a vertex v in a directed graph $G = (V, E)$.

We use the following notations for a fixed point v in a fixed directed graph $G = (V, E)$:

Definition 6.12 *R_j is the set of vertices which are reachable from v by a path of length less than or equal to j. In addition, N_j is defined to be the number of nodes in R_j, namely $|R_j|$.*

It can be seen that,

$$\{v\} = R_0 \subseteq R_1 \subseteq \cdots \subseteq R_{n-1} = R$$

where n denotes the number of nodes in G, and R denotes the set of vertices reachable from v.

There is a strong connection between R_j and R_{j-1} for $j \geq 1$, since any path of length j is a path of length $j - 1$ with an additional edge. The following claim will be used later in the development of the machine CR:

Claim 6.4.2 *The following equation holds:*

$$R_j = \begin{cases} R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} & \textit{if } j \geq 1 \\ \{v\} & \textit{if } j = 0 \end{cases}$$

Proof: For $j = 0$: Clear from definition.
For $j \geq 1$: $R_{j-1} \subseteq R_j$ by definition. $\{u : w \in R_{j-1}, (w,u) \in E(G)\}$ represents all the nodes which are adjacent to R_{j-1}, i.e. have length at most $j-1+1 = j$. This set is also contained in R_j. Thus, $R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} \subseteq R_j$.

In the opposite direction, every node $u \in R_j$, which is not v, is reachable from v along a path with length less or equal to j. Thus, its predecessor in this path has length less or equal to $j - 1$. Thus, $R_j \subseteq \{u : w \in R_{j-1}, (w,u) \in E(G)\} \cup \{v\} \subseteq R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\}$ (since $\{v\} \subseteq R_{j-1}$ for any $j \geq 1$).

Therefore, the claim follows. ■

Corollary 6.13 *For any $j \geq 1$, a node $w \in R_j$, if and only if there is a node $r \in R_{j-1}$ such that $r = w$ or $(r, w) \in E(G)$.* ■

We now construct a non-deterministic Turing Machine, CR, that counts the number of nodes in a directed graph G, reachable from a node in the graph v.

Our purpuse in this algorithm is to compute N_{n-1} where n is the number of nodes in G, to find the number of all reachable nodes from v. This recursive idea in Claim 6.4.2 is the main idea behind the following algorithm, which is build iteratively. In each stage, the algorithm computes N_j by using N_{j-1}. It has an initial value N_0, which we know to be $|\{v\}| = 1$. The itrations use the non-deterministic power of the machine.

The high-level description of the algorithm is as follows:

- For each j from 1 to $n-1$, it tries to calculate recursively N_j from N_{j-1}. This is done from N_0 to N_{n-1}, which is the desired output. Here is how N_j is computed.
  - For each node w in the graph,
    - For each node r in the graph, it guesses if $r \in R_{j-1}$ and if the answer is yes, it guesses a path with length less than or equal to $j-1$, from v to r. It verifies that the path is valid. If it is, it knows that r is a node in R_{j-1}. Otherwise, it rejects. It counts each node r, such that $r \in R_{j-1}$, by $counter_{j-1}$. The machine checks whether $w = r$ or $(r, w) \in E(G)$. If it is, (by using Corrolary 6.13), $w \in R_j$, and then it indicates by a flag, $flag_w$ that w is in R_j. ($flag_w$ is initially 0).
  - It counts the number of vertices r in R_{j-1} we found, and verifies that it is equal to N_{j-1}. Otherwise, it rejects. If the machine does not reject, we know that every node $r \in R_{j-1}$ was found. Therfore using Corrolary 6.13, the membership of w in R_j is decided properly (i.e. $flag_w$ has the right value).
  - At the end of this process it sums up the flags, $flag_w$'s into a counter, $counter_j$ (i.e. counts the number of nodes that were found to be in R_j).
- It stores the value of $counter_j$ for N_j to begin a new iteration (or to give the result in case we reach $j = n-1$).

We stress that all counters are implemented using the same space. That is, the only thing which passes from iteration $j-1$ to iteration j is N_{j-1}.

The detailed code follows:

Input: $G = (V, E)$, $v \in V(G)$

Task: Find the number of reachable nodes from v in G.

1. **Computing** $n = |V(G)|$

2. $N_0 \leftarrow 1 = |R_0|$

3. **for** $j = 1$ **to** $n-1$ **do**

4. $\quad counter_j \leftarrow 0$

5. $\quad$ **for** $w = 1$ **to** n **do** $\qquad$ *(lines 5-24 compute N_j)*

6. $\qquad counter_{j-1} \leftarrow 0$ $\qquad$ *(w is a potential member in R_j)*

7. $\qquad flag_w = 0$

8. **for** $r = 1$ **to** n **do** *(We try to enumerate R_{j-1} using N_{j-1})*

9. **guess if** $r \in R_{j-1}$**. If not, proceed to next iteration of step 8.**
(we continue in steps 10-21 only if we guessed that $r \in R_{j-1}$)

10. $v_1 \leftarrow v$ *(v_1 is initially v)*

11. $p \leftarrow 0$

12. **repeat** *(guess and verify a path from v to r, such that $r \in R_{j-1}$)*

13. $p \leftarrow p + 1$

14. **guess a node** v_2 *(v_1 and v_2 are the last and current nodes)*

15. **if** $(v_1, v_2) \notin E$ **then halt(failure)**

16. **if** $v_2 \neq r$ **then** $v_1 \leftarrow v_2$

17. **until** $(v_2 = r)$ **or** $(p = j - 1)$

18. **if** $v_2 \neq r$ **then halt(failure)**

19. $counter_{j-1} \leftarrow counter_{j-1} + 1$

20. **if** $(r = w)$ **or** $((r, w) \in E(G))$ **then** *(check that $w \in R_j$)*

21. $flag_w = 1$

22. **if** $counter_{j-1} \neq N_{j-1}$ **then halt(failure)**

23. $counter_j \leftarrow counter_j + flag_w$

24. $N_j = counter_j$

25. **output** N_{n-1}

Lemma 6.4.3 *The machine CR uses $O(log(n))$ space.*

Proof: Computing the number of nodes (line 1) can be made in at most $O(log(n))$, by simply counting the number of nodes with a counter on the input tape. In each other step of the running of the machine, it only needs to know at most ten variables, i.e. the counters for the 'for' loops: j, w, r, p, the value of N_{j-1} for the current j, the two counters for the size of the sets $counter_j, counter_{j-1}$, two nodes of the guessing part v_2, v_1, and the indicating flag $flag_w$.

> Oded's Note: *The proof might be more convincing if the code was modified so that N_{PREV} is used instead of N_{j-1}, $counter_{CURR}$ instead of $counter_j$, and $counter_{PREV}$ instead of $counter_{j-1}$. In such a case, line 24 will prepare for the next iteration by setting $N_{PREV} \leftarrow counter_{CURR}$. Such a description will better emphasise the fact that we use only a constant number of variables, and that their storage space is re-used in the iterations.*

Whenever is needed to change information, like increasing a counter, or changing a variable, it reuses the space it needs for this goal. Every counter we use counts no more than the number of nodes in the graph, hence we can implement each one of them in $O(log(n))$ space. Each node is assumed to take $O(log(n))$ to store, i.e. its number in the list of nodes. And the $flag_w$ clearly takes only 1 bit. Therefore, to store these variables, it is enough to use $O(log(n))$ space.

Except for these variables, we don't use any additional space. All that is done is comparing nodes with the input tape, and checking whether two nodes are adjacent in the adjecancy matrix that represents the graph. These operations can be done only by scanning the input tape, and take no more then $O(log(n))$ space, for counters that scan the matrix.

Therefore, this non-deterministic Turing Machine uses only $O(log(n))$ or $O(log(|x|))$ where $x = (G, v)$ is the input to the machine. ■

Lemma 6.4.4 *If the machine CR outputs an integer, then it correctly gives the result of N_{n-1}.*

Proof: We'll prove it by induction on the iteration of computing N_j:
For $j = 0$: It is obviously correct.
If it computes correctly N_{j-1}, and it did not halt while computing N_j, then it computes correctly N_j as well: By the assumption of the induction we have a computation that computes N_{j-1}, that is stored correctly. All we have to prove is that $counter_j$ is incremented if and only if the current w is indeed in R_j (line 23), since then N_j will have correctly the number of nodes in R_j. Since the machine didn't failed till now, $counter_{j-1}$ has to be equal to N_{j-1} (line 22), by the assumption of the induction. This means that the machine indeed found all $r \in R_{j-1}$, since $counter_{j-1}$ is incremented for each node that is found to be in R_{j-1} (line 19). Therefore, using Corrolary 6.13, we know that the machine changes the flag, $flag_w$, of a node w if only if $w \in R_j$. And this flag is the value that is added to $counter_j$ (line 23). Therefore, the counter is incremented if and only if $w \in R_j$. ■

Corollary 6.14 *Machine CR satisfies Theorem 6.9.*

Proof: We have shown in Lemma 6.4.4 that if the machine doesn't fail, it gives the right result. It is left to prove that there exists a computation in which the machine doesn't fail.

The correct computation is done as follows. For each node $r \notin R_{j-1}$, the machine guesses well in line 9 that indeed $r \notin R_{j-1}$ and stops working on this node. For each node $r \in R_{j-1}$, the machine guesses in line 9 so, and in addition it guesses correctly the nodes that form the directed path from v to r in line 14. In this computation, the machine will not fail. In lines 15 and 18, there is no failure, since only $r \in R_{j-1}$ nodes get to these lines, and in these lines it guesses corectly the connected path from v. Therefore, in line 22, all nodes $r \in R_{j-1}$ were counted, since the machine guesses them corectly, and the machine will not halt either. Thus, the machine doesn't fail on the above computation.

Using Lemma 6.4.3, we know that the machine uses $O(log(n))$ space. Therefore, CR is a non-deterministic machine that satisfies Theorem 6.9. ■

Bibliographic Notes

The proofs of both theorems (i.e., NL-completeness of CONN and NL=coNL) can be found in [2]. The latter result was proved independently by Immerman [1] and Szelepcsenyi [3].

1. N. Immerman. Nondeterministic Space is Closed Under Complementation. *SIAM Jour. on Computing*, Vol. 17, pages 760–778, 1988.

2. M. Sipser. *Introduction to the Theory of Computation*, PWS Publishing Company, 1997.

3. R. Szelepcsenyi. A Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica*, Vol. 26, pages 279–284, 1988.

Lecture 7

Randomized Computations

Notes taken by Erez Waisbard and Gera Weiss

Summary: In this lecture we extend the notion of efficient computation by allowing algorithms (Turing machines) to toss coins. We study the classes of languages that arise from various natural definitions of acceptance by such machines. We will focus on polynomial running time machines of the following types:

1. One-sided error machines ($RP, coRP$).
2. Two-sided error machines (BPP).
3. Zero error machines (ZPP)

We will also consider probabilistic machines that uses logarithmic spaces (RL).

7.1 Probabilistic computations

The basic thought underlying our discussion is the association of efficient computation with probabilistic polynomial time Turing machines. We will consider efficient only algorithms that run in time that is no more than a fixed polynomial in the length of the input.
There are two ways to define randomized computation. One, that we will call *online* is to enter randomized steps, and the second that we will call *offline* is to use an additional randomizing input and evaluate the output on such random input.

In the fictitious model of non-deterministic machines, one accepting computation was enough to include an input in the language accepted by the machine. In the randomized model we will consider the probability of acceptance rather than just asking if the machine has an accepting computation.

> *Then he said, "May the Lord not be angry, but let me speak just once more. What if only ten can be found there?" He answered, "For the sake of ten, I will not destroy it."* [Genesis 18:32].

As God didn't agree to save Sedom for the sake of less then ten peoples, we will not consider an input to be in the accepted language unless it has a noticeable probability to be accepted.

> Oded's Note: *The above illustration is certainly not my initiative. Besides some reservations regarding this specific part of the bible (and more so the interpretations given*

to it during the centuries), I fear that 10 may not strick the reader as "many" but rather as closer to "existence". In fact, standard interpretations of this passage stress the minimalistic nature of the challenge – barely above unique existence...

The online approach: One way to look at a randomized computation is to allow the Turing machine to make random moves. Formally this can be modeled as letting the machine to choose randomly among the possible moves that arise from a nondeterministic transition table. If the transition table maps one ($< state >, < symbol >$) pair to two different ($< state >, < move >, < symbol >$) triples then the machine will choose each transition with equal probabilities.

Syntactically, the online probabilistic Turing machine will look the same as the nondeterministic machine. The difference is at the definition of the accepted language. The criterion of an input to be accepted by a regular nondeterministic machine is that the machine will have at least one accepting computation when it is invoked with this input. In the probabilistic case, we will consider the probability of acceptance. We would be interested in *how many* accepting computation the machine has (or rather what is the probability of such computation). We postulate that the machine choose every step with equal probability, and so get a probability space on possible computations. We look at a computation as a tree, where a node is a configuration and it's children are all the possible configurations that the machine can pass to in a single step. The tree is describing the possible computations of the machine when running on a given input. The output of a probabilistic Turing machine on an input x is not a string but a random variable. Without loss of generality we can consider only binary tree because if the machine has more than two possible steps, it is possible to build another machine that will simulate the given machine with two step transition table. This is possible even if the original machine had steps with probability that has infinite binary expansion. Let say, for example, that the machine has a probability of $\frac{1}{3}$ to get from step A to step B. Then we have a problem when trying to simulate it by unbiased binary coins, because there is the binary expansion of $\frac{1}{3}$ is infinite. But we can still get as close as we want to the original machine, and this is good enough for our purposes.

The offline approach: Another way to consider nondeterministic machines is, as we did before, to use an additional input as a guess. For NP machines we gave an additional input that was used as a witness. The analogous idea is to view the outcome of the internal coin tosses as an auxiliary input. The machine will receive two inputs, the real input, x, and the guess input, r. Imagine that the machine receives this second input from an external 'coin tossing device' rather than toss coins internally.

Notation: We will use the following notation to discuss various properties of probabilistic machines:

$$Prob_r[M(x,r) = z]$$

Sometimes, we will drop the r and keep it implicitly like in the following notation:

$$Prob[M(x) = z]$$

By this notations we mean the probability that the machine M with real input x and guess input r, distributed uniformly, will give an output z. The probability space is that of all possible r taken with uniform distribution. This statement is more confusing than it seems to be because the machine may use different number of guesses for different inputs. It may also use different number of guesses on the same input, if the computation depends on the outcome of previous guesses.

Oded's Note: *Actually, the problem is with the latter case. That is, if on each input all computations use the same number of coin tosses (or "guesses"), denoted l, then each such computation occurs with probability 2^{-l}. However, in the general case, where the number of coin tosses may depend on the outcome of previous tosses, we may just observe that a halting computation with coin outcome sequence r occurs with probability exactly $2^{-|r|}$.*

Oded's Note: *An alternative approach is to modify the randomized machine so that it does use the same number of coin tosses in each computation on the same input.*

7.2 The classes RP and $coRP$ – One-Sided Error

The first two classes of languages that arise from probabilistic computations that we consider are the one-sided error (polynomial running time) computable languages. If there exist a machine that can decide the language with good probability in polynomial time it is reasonable to consider the problem as relatively easy. Good probability here means that the machine will be sure only in one case and will give the right answer in the other case but only with good probability (the cases are when $x \in L$ and when $x \notin L$).
From here on, a polynomial probabilistic Turing machine means a probabilistic machine that always (no matter what coin tosses it gets) halts after a polynomial (in the length of the input) number of steps.

Definition 7.1 (Random Polynomial-time – RP): *The complexity class RP is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2}.$$

$$x \notin L \Rightarrow Prob[M(x) = 1] = 0.$$

Definition 7.2 (Complementary Random Polynomial-time – $coRP$): *The complexity class $coRP$ is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$x \in L \Rightarrow Prob[M(x) = 1] = 1.$$

$$x \notin L \Rightarrow Prob[M(x) = 0] \geq \frac{1}{2}.$$

One can see from the definitions that these two classes complement each other. If you have a machine that decides a language L with good probability (in one of the above senses), you can use the same machine to decide the complementary language in the complementary sense.
That is, an alternative (and equivalent) way to define $coRP$ is:

$$coRP = \{\overline{L} : L \in RP\}$$

Comparing NP to RP: It is instructive to compare the definitions of RP and NP. In both classes we had the offline definition that used an external witness (in NP) or randomization (in RP).

Given an RP machine, M, since the machine run in polynomial-time, the size of the guesses that it can use is bounded by a polynomial in the size of x. For every given integer $n \in \mathsf{N}$ we consider the relation:

$$R_n \stackrel{\text{def}}{=} \left\{(x,r) \in \{0,1\}^n \times \{0,1\}^{p(n)} : M(x,r) = 1\right\}$$

which consists of all accepted inputs of length n and their accepting coin tosses (i.e r).

The same is also applicable for NP machines, which run also in polynomial-time and can only use witnesses that are bounded by a polynomial in the length of the input. So, for NP machine M, we consider the relation:

$$R_n \stackrel{\text{def}}{=} \left\{(x,y) \in \{0,1\}^n \times \{0,1\}^{p(n)} : M(x,y) = 1\right\}$$

which consist of all accepted inputs of length n and their witnesses (i.e y).

In both cases we will use the relation:

$$R = \bigcup_{n=1}^{\infty} R_n$$

which consists of all the accepted inputs and their witness/coin-tosses.

Using this relation we can compare Definition 7.1 to the definition of NP in the following table:

NP	RP
$x \in L \Rightarrow \exists y,\, (x,y) \in R$	$x \in L \Rightarrow \text{Prob}_r\, [(x,r) \in R] \geq \frac{1}{2}$
$x \notin L \Rightarrow \forall y,\, (x,y) \notin R$	$x \notin L \Rightarrow \forall r,\, (x,r) \notin R$

From this table, it is seems that these two classes are close. The witness in the nondeterministic model is replaced by the coin-tosses and the criteria for acceptance has changed. The difference is that, in the nondeterministic model, one witness was enough for us to say that an input is accepted, and in the probabilistic model we are asking for many coin-tosses. Clearly,

Proposition 7.2.1 $NP \supseteq RP$

Proof: Let L be an arbitrary language in RP. If $x \in L$ then there exist a Turing machine M and a coin-tosses y such that $M(x,y) = 1$ (more than $\frac{1}{2}$ of the coin-tosses are such). So we can use this y as a witness (considering the same machine as a nondeterministic machine with the coin-tosses as witnesses). If $x \notin L$ then $Prob_r[M(x,r) = 1] = 0$ so there is no witness. ■

Notice that there is a big difference between nondeterministic Turing machines and probabilistic Turing machines. The first is a fictitious concept that is invented to explore the properties of search problems, while the second is a realistic model that describe machines that one can really build.We use the nondeterministic model to describe problems like a search problem with an efficient verification, while the probabilistic model is used as an efficient computation.

It is fair to ask if a computer can toss-coins as an elementary operation. We answer this question positively based on our notion of randomness and the ability of computers to use random-generating instrumentation like reading unstable electric circuits. The question is whether this random operation gives us more power than we had with the regular deterministic machines.

RP is one-sided error: The definition of RP does not ask for the same behavior on inputs that are in the language as it asks for inputs that are in the language.

- If $x \notin L$ then the answer of the machine must be correct no matter what guesses we make. In this case, the probability to get a wrong answer is zero so the answer of the machine is right for every r.
- But, if $x \in L$, the machine is allowed to make mistakes. In this case, we have a non-zero probability that the answer of the machine will be wrong (still this probability is not "too big").

The definition favors one type of mistake while in practice we don't find very good reason to favor it. We will see later that there are different families of languages that do not favor any type of error. We will call these languages two-sided error languages.

It was reasonable to discuss one-sided errors when we where developing NP, because verification is one-sided by nature, but it is less useful for exploring the notion of efficient computation.

Invariance of the constant and beyond: Recall that for $L \in RP$

$$x \in L \Rightarrow Prob_r[M(x,r) = 1] \geq \frac{1}{2}$$

The constant $\frac{1}{2}$ in the definition of RP is arbitrary. We could choose every constant strictly threshold between zero and one, and get the same complexity class. Our choice of $\frac{1}{2}$ is somewhat appealing because it says that at least half of the witnesses are good.
If you have, for example, a machine that can decide some language L with a greater probability than $\frac{1}{3}$ to say "YES" for an input that is in the language, you can build another machine that will invoke the first machine three times on every input and return the "YES" if one of them answered "YES". Obviously this machine will answer correctly on inputs that are not in the language (because the first machine will always say "NO"), and it will say "YES" on inputs that are in the language with higher probability than before. The original probability of not getting the correct answer when the input is in the language was smaller than $\frac{2}{3}$, when repeating the computation for three time this probability falls down to less than $\left(\frac{2}{3}\right)^3 = \frac{8}{27}$ meaning that we now get the correct answer with probability greater than $\frac{19}{27}$ (which is greater than $\frac{1}{2}$).

So we could use $\frac{1}{3}$ instead of $\frac{1}{2}$ without changing the class of languages. This procedure of amplification can be used to show the same result for every constant, but we will prove further that one can even use thresholds that depend on the length of the input.

We are looking at two probability spaces: one when $x \notin L$ and one when $x \in L$, and defined a random variable (representing the decision of the machine) on each of this spaces. In case $x \notin L$ the latter random variable is identically zero (i.e., "reject"), whereas in case $x \in L$ the random variable may be non-trivial (i.e., is 1 with probability above some given threshold and 0 otherwise).

Moving from one threshold to a higher one amount to the following: In case $x \in L$, the fraction of points in the probability space assigned the value 1 is lower bounded by the first threshold. Our aim is to hit such a point with probability lower bounded by a higher threshold. This is done by merely making repeated independent samples into the space, where the number of the trials is easily determined by the relation between the two thresholds. We stress that in case $x \notin L$ all points in the probability space are identically assigned (the value 0) and so it does not matter how many times we try (we'll always see zeros).

We will show that one can even replace the constant $\frac{1}{2}$ by either $\frac{1}{p(|x|)}$ or $1-2^{-p(|x|)}$, where $p(\cdot)$ is any fixed polynomial, and get the same family of languages. We take these two margins, because once we will show the equivalence of these two thresholds, it will follow that every threshold that one might think of in between will do. Consider the following definitions:

Definition 7.3 ($RP1$): *L is in $RP1$ if there exist a polynomial running-time Turing machine M and a polynomial $p(\cdot)$ such that* $\begin{array}{l} x \in L \Rightarrow Prob_r[M(x,r)=1] \geq \frac{1}{p(|x|)} \\ x \notin L \Rightarrow Prob_r[M(x,r)=0]=1 \end{array}$

Definition 7.4 ($RP2$): *L is in $RP2$ if there exist a polynomial running-time Turing machine M and a polynomial $p(\cdot)$ such that* $\begin{array}{l} x \in L \Rightarrow Prob_r[M(x,r)=1] \geq 1-2^{-p(|x|)} \\ x \notin L \Rightarrow Prob_r[M(x,r)=0]=1 \end{array}$

These definitions seems very far from each other, because in $RP1$ we ask for a probabilistic algorithm (Turing machine) that answer correctly with a very small probability (but not negligible), while in $R2$ we ask for an efficient algorithm (Turing machine) that we can almost ignore the probability of it's mistake. However, these two definition actually define the same class (as we will prove in the next paragraph). This implies that having an algorithm with a noticeable probability of success implies existence of and efficient algorithm with negligible probability of error.

Proposition 7.2.2 *RP1=RP2*

Proof:

$RP1 \supseteq RP2$
This direction is trivial because if $|x|$ is big enough then the bound in Definition 7.3 (i.e $\frac{1}{p(|x|)}$) is smaller than the bound in Definition 7.4 (i.e $1-2^{-p(|x|)}$) so being in $RP2$ implies being in $RP1$ for almost all inputs. The finitely many inputs for which this does not hold can be incorporated in the machine of Definition 7.3. Thus $RP1 \supseteq RP2$.

$RP1 \subseteq RP2$
We will use a method known as *amplification*:
We will try the weaker machine (of $RP1$) enough times so that the probability of giving a wrong answer will be small enough. Assume that we have a machine M_1 such that

$$\forall x \in L : Prob_r[M_1(x,r)=1] \geq \frac{1}{p(|x|)}$$

We will define a new machine M_2, up to a function $t(|x|)$ that we will determine later, as follows:

$$M_2(x) \stackrel{\text{def}}{=} \begin{cases} invoke\ M_1(x)\ t(|x|)\ times\ with\ different\ randomly\ selected\ r's \\ if\ some\ of\ these\ invocations\ returned\ 'YES'\ return\ 'YES' \\ else\ return\ 'NO' \end{cases}$$

Let $t = t(|x|)$. Then for $x \in L$

$$Prob[M_2(x)=0] = (Prob[M_1(x)=0])^{t(|x|)} \leq \left(1-\frac{1}{p(|x|)}\right)^{t(|x|)}$$

To find the desired $t(|x|)$ we can solve the equation:

$$\left(1-\frac{1}{p(|x|)}\right)^{t(|x|)} \leq 2^{-p(|x|)}$$

And obtain

$$t(|x|) \geq p(|x|) \cdot \log_2 \left(1-\frac{1}{p(|x|)}\right)^{-1} = \frac{p(|x|)^2}{\log_2 e}$$

Where $e \approx 2.7182818...$ is the natural logarithm base.

So by letting $t(|x|) = p(|x|)^2$ in the definition of M_2 we get a machine that run in polynomial time and decides L with probability greater than $1 - 2^{-p(|x|)}$ to give right answer for $x \in L$ (and always correct on $x \notin L$). ■

7.3 The class BPP – Two-Sided Error

One may argue that RP is too strict because it ask that the machine has to give 100% correct answer for inputs that are not in the language.

We derived the definition of RP from the definition of NP, but NP didn't reflect an actual computational model for search problems but rather a model for verification. One may find that looking at a two-sided error is more appealing as a model for search problem computations.

We want a machine that will recognize the language with high probability, where probability refers to the event "The machine answers correctly on an input x regardless if $x \in L$ or $x \notin L$". This will lead us to two-sided error version of the randomized computation. First recall the notation:

$$\chi_L(x) \stackrel{\text{def}}{=} \begin{cases} 1 & x \in L \\ 0 & x \notin L \end{cases}$$

Definition 7.5 (Bounded-Probability Polynomial-time – BPP): *The complexity class BPP is the class of all languages L for which there exist a probabilistic polynomial-time Turing machine M, such that*

$$\forall x : Prob[M(x) = \chi_L(x)] \geq \frac{2}{3}.$$

That means that:

$$If\ x \in L\ then\ Prob[M(x) = 1] \geq \frac{2}{3}.$$

$$If\ x \notin L\ then\ Prob[M(x) = 1] < \frac{1}{3}.$$

The phrase "bounded-probability" means that the success probability is bounded away from failure probability.

The BPP machine is a machine that makes mistakes but returns the correct answer most of the time. By running the machine a large number of times and returning the majority of the answers we are guaranteed by the law of large numbers that our mistake will be very small.

The idea behind the BPP class is that M accept by majority with a noticeable gap between the probability to accept inputs that are in language and the probability to accept inputs that are not in the language, and it's running time is bounded by a polynomial.

Invariance of constant and beyond: The $\frac{2}{3}$ is, again, an arbitrary constant. Replacing the $\frac{2}{3}$ in the definition by any other constant greater than $\frac{1}{2}$ does not change the class defined. If, for example we had a machine, M that recognize some language L with probability $p > \frac{1}{2}$, meaning that $Prob[M(x) = \chi_L(x)] \geq p$, we could easily build a machine that will recognize L with any given probability $q > p$ by invoking this machine sufficiently many times and returning the majority of the answers. This will clearly increase the probability of giving correct answer to the wanted threshold, and run in polynomial time.

In the RP case we had two probability spaces that we could distinguish easily because we had a guarantee that if $x \notin L$ then the probability to get one is zero, hence if you get $M(x) = 1$ for some input x, you could say *for sure* that $x \in L$.
In the BPP case, the amplification is less trivial because we have zeroes and ones in both probability spaces (the probability space is not constant when $x \in L$ nor when $x \notin L$).
The reason that we can apply amplification in the BPP case (despite the above difference) is that invoking the machine many times and counting how many times it returns one gives us an estimation on the fraction of ones in the whole probability space. It is useful to get an estimator for the fraction of the ones in the probability space because when this fraction is greater than $\frac{2}{3}$ we have that $x \in L$, and when this fraction is less than $\frac{1}{3}$ we have that $x \notin L$ (this fraction tells us in which probability space we are in).

If we rewrite the condition in Definition 7.5 as:

$$If \ x \in L \ then \ Prob[M(x) = 1] \geq \frac{1}{2} + \frac{1}{6}.$$

$$If \ x \notin L \ then \ Prob[M(x) = 1] < \frac{1}{2} - \frac{1}{6}.$$

We could consider the following change of constants:

$$If \ x \in L \ then \ Prob[M(x) = 1] \geq p + \epsilon.$$

$$If \ x \notin L \ then \ Prob[M(x) = 1] < p - \epsilon.$$

for any given $p \in (0,1)$ and $0 < \epsilon < \min\{p, 1-p\}$.

If we had such a machine, we could invoke the machine many times and get increasing probability to have the fraction of ones in our innovations to be in an ϵ neighborhood of the real fraction of ones in the whole space (by the law of large numbers). After some fixed number of iterations (that does not depend on x), we can get that probability to be larger than $\frac{2}{3}$.
This means that if we had such a machine (with p and ϵ instead of $\frac{1}{2}$ and $\frac{1}{6}$), we could build another machine that will invoke it some fixed number of times and will decide the same language with probability greater than $\frac{2}{3}$.

The conclusion is that the $\frac{1}{2} \pm \frac{1}{6}$ is arbitrary in Definition 7.5, and can be replaced by any $p \pm \epsilon$ such that $p \in (0,1)$ and $0 < \epsilon < \min\{p, 1-p\}$. But we can do more than that and use threshold that depend on the length of the input as we will prove in the following claims:

The weakest possible BPP definition: Using the above framework, we'll show that for every polynomial-time computable threshold, denoted f below, and any "noticeable" margin (represented by 1/poly), we can recover the "standard" threshold (of 1/2) and the "safe" margin of 1/6.

Claim 7.3.1 *$L \in BPP$ if and only if there exist a polynomial-time computable function $f : \mathbb{N} \mapsto [0,1]$, a positive polynomial $p(\cdot)$ and a probabilistic polynomial-time Turing machine M, such that:*

$$\forall x \in L : Prob[M(x) = 1] \geq f(|x|) + \frac{1}{p(|x|)}$$

$$\forall x \notin L : Prob[M(x) = 1] < f(|x|) - \frac{1}{p(|x|)}$$

Proof:
It is easy to see that by choosing $f(|x|) \equiv \frac{1}{2}$ and $p(|x|) \equiv 6$ we get the original definition of BPP (see Definition 7.5), hence every BPP language satisfies the above condition.

Assume that we have a probabilistic Turing machine, M, with these bounds on the probability to get 1. Then, for any given input x, we look at the random variable $M(x)$, which is a Bernoulli random variable with unknown parameter $p = Exp[M(x)]$. Using a well known fact that the expectation of a Bernoulli random variable is exactly the probability to get one we get that $p = Prob[M(x) = 1]$.
So by estimating p we can say something about whether $x \in L$ or $x \notin L$. The most natural estimator is to take the mean of n samples of the random variable (i.e the answers of n independent invocations of $M(x)$).
Then we will use the known statistical method of confidence intervals on the parameter p. The confidence interval method gives a bound within which a parameter is expected to lie with a certain probability. Interval estimation of a parameter is often useful in observing the accuracy of an estimator as well as in making statistical inferences about the parameter in question.
In our case we want to know with probability higher than $\frac{2}{3}$ if $p \in \left[0, f(|x|) - \frac{1}{p(|x|)}\right]$ or $p \in \left[f(|x|) + \frac{1}{p(|x|)}, 1\right]$. This is enough because $p \in \left[0, f(|x|) - \frac{1}{p(|x|)}\right] \Rightarrow x \notin L$ and $p \in \left[f(|x|) + \frac{1}{p(|x|)}, 1\right] \Rightarrow x \in L$ (note that $p \in \left(f(|x|) - \frac{1}{p(|x|)}, f(|x|) + \frac{1}{p(|x|)}\right)$ is impossible). So if we can get a bound of size $\frac{1}{p(|x|)}$ within which p is expected to lie within a probability greater than $\frac{2}{3}$, we can decide $L(M)$ with this probability (and hence $L(M) \in BPP$ by Definition 7.5).
We define the following Turing machine (up to an unknown number n that we will compute later) :

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} \textit{Invoke } M(x) \ n \textit{ times (call the result of the } i'\textit{th invocation } t_i). \\ \textit{Compute } \hat{p} \leftarrow \frac{1}{n} \cdot \Sigma_{i=1}^{n} t_i \\ \textit{if } \hat{p} \ > \ f(|x|) \textit{ say } 'YES' \textit{ else say } 'NO' \end{cases}$$

Note that $\hat{p}$ is exactly the mean of a sample of size n taken from the random variable $M(x)$. This machine do the normal statistical process of estimating a random variable by taking samples and using the mean as an estimator for the expectation. If we will be able to show that with an appropriate n the estimator will not fall too far from the real value with a good probability, it will follow that this machine answers correctly with the same probability.
To resolve n we will use Chernoff's inequality which states that for any set of n independent Bernoulli variables $\{X_1, X_2, ..., X_n\}$ with the same expectations $p \leq \frac{1}{2}$ and for every $\delta, 0 < \delta \leq p(p-1)$, we have

$$Prob\left[\left|\frac{\Sigma_{i=1}^{n} X_i}{n} - p\right| > \delta\right] < 2 \cdot e^{-\frac{\delta^2}{2 \cdot p(1-p)} \cdot n} \leq 2 \cdot e^{-\frac{\delta^2}{2 \cdot \frac{1}{4}} \cdot n} = 2 \cdot e^{-2n \cdot \delta^2}$$

So by taking $\delta = \frac{1}{p(|x|)}$ and $n = -\frac{\ln\frac{1}{6}}{2\cdot\delta^2}$ we get that our Turing machine M' will decide $L(M)$ with probability greater than $\frac{2}{3}$ suggesting that $L(M) \in BPP$. ■

The strongest possible BPP definition: On the other hand, one can reduce the error probability of BPP machines to an exponentially vanishing amount.

Claim 7.3.2 *For every $L \in BPP$ and every positive polynomial $p(\cdot)$ there exist a probabilistic polynomial-time Turing machine M, such that:*

$$\forall x : Prob[M(x) = \chi_L(x)] \geq 1 - 2^{-p(|x|)}$$

Proof:
If this condition is true for every polynomial, we can choose $p(|x|) \equiv 2$ and get M such that:

$$\begin{array}{l} \forall x : Prob[M(x) = \chi_L(x)] \geq 1 - 2^{-2} = \frac{3}{4} \\ \Rightarrow \forall x : Prob[M(x) = \chi_L(x)] \geq \frac{2}{3} \\ \Rightarrow L \in BPP \end{array}$$

Let L be a language in BPP and let M be the machine guaranteed in Definition 7.5. We can amplify the probability of right answer by invoking M many times and taking the majority of it's answers. Define the following machine (again up to the number n that we will find later):

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} \textit{Invoke } M(x)\ n \textit{ times (call the result of the } i'\textit{th invocation } t_i). \\ \textit{Compute } \hat{p} \leftarrow \frac{1}{n} \cdot \Sigma_{i=1}^{n} t_i \\ \textit{if } \hat{p} > \frac{1}{2} \textit{ say 'YES' else say 'NO'} \end{cases}$$

From Definition 7.5, we get that if we know that $Exp[M(x)]$ is greater than half it follows that $x \in L$ and if we know that $Exp[M(x)]$ is smaller than half it follows that $x \notin L$ (because $Exp[M(x)] = Prob[M(x) = 1]$)
But Definition 7.5 gives us more. It says that the expectation of $M(x)$ is bounded away from $\frac{1}{2}$ so we can use the confidence interval method.

From Chernoff's inequality we get that

$$Prob\left[\left|M'(x) - Exp[M(x)]\right| \leq \frac{1}{6}\right] \geq 1 - 2 \cdot e^{-\frac{n}{18}}$$

But if $|M'(x) - Exp[M(x)]|$ is smaller than $\frac{1}{6}$ we get from Definition 7.5 that the answer of M' is correct, because it is close enough to the the expectation of $M(x)$ which is guaranteed to be above $\frac{2}{3}$ when $x \in L$ and bellow $\frac{1}{3}$ when $x \notin L$. So we get that:

$$Prob\left[M'(x) = \chi_L(x)\right] \geq 1 - 2 \cdot e^{-\frac{n}{18}}$$

Thus, for every polynomial $p(\cdot)$, we can choose n, such that

$$2^{p(|x|)} \geq 2 \cdot e^{-\frac{n}{18}}$$

and get that:

$$Prob\left[M'(x) = \chi_L(x)\right] \geq 1 - 2^{p(|x|)}$$

So M' satisfies the claimed condition. ■

Conclusion: We see that a gap of $\frac{1}{p(|x|)}$ and a gap of $1 - 2^{-p(|x|)}$ which look like "weak" and "strong" versions of BPP are the same. As shown above the "weak" version is actually equivalent to the "strong" version, and both are equivalent to the original definition of BPP.

Some comments about BPP:

1. $RP \subseteq BPP$
   It is obvious that one-sided error is a special case of two-sided error.

2. We don't know if $BPP \subseteq NP$. It might be so but we don't get it from the definition like we did in RP.

3. If we define $coBPP \stackrel{\text{def}}{=} \{\overline{L} : L \in BPP\}$ we get, from the symmetry of the definition of BPP, that $coBPP = BPP$.

7.4 The class PP

The class PP is wider than what we have seen so far. In the BPP case we had a gap between the number of accepting computations and non-accepting computations. This gap enabled us to determine with good probability (using confidence intervals) if $x \in L$ or $x \notin L$.
The gap was wide enough so we could invoke the machine polynomially many times and notice the difference between inputs that are in the language and inputs that are not in the language. The PP class don't put the gap restriction, hence the gap may be very small (even one guess can make a difference).
Running the machine polynomially many times may not help. If we have a machine that answers correctly with probability more than $\frac{1}{2}$, and we want to get another machine that answers correctly with probability greater than $\frac{1}{2} + \epsilon$ (for a given $0 < \epsilon < \frac{1}{2}$) we can't always do it in polynomial time because we might not have the gap that we had in Definition 7.5.

Definition 7.6 $PP \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exist a polynomial time} \\ \textit{Turing machine } M \textit{ s.t} \\ \forall x, Prob[M(x) = \chi_L(x)] > \frac{1}{2} \end{array} \right\}$

Note that it is important that we define $>$ and not $\geq$, since otherwise we can simply "flip a coin" and completely ignore the input (we can decide to say $'YES'$ if we get head and $'NO'$ if we get tail and this will satisfy the definition of the machine) and there is no use for a machine that runs a lot of time and gives no more knowledge than what we already have (assuming one knows how to flip a coin). However the actual definition of PP gives very little as well: The difference between what happens in case $x \in L$ and in case $x \notin L$ is negligible (rather than "noticeable" as in the definition of BPP). We abuse this weak requirement in the proof of Item 3 (below).

From the definition of PP we get a few interesting facts:

1. $PP \subseteq PSPACE$
   Let L be a language in PP, let M be the probabilistic Turing machine that exists according to Definition 7.6. Let $p(\cdot)$ be the polynomial bounding it's running time. We will build a new machine M' that decides L in a polynomial space. Given an input x, the new machine will run M on x using all possible coin tosses with length $p(|x|)$ and decides by majority (i.e if M

accepted the majority of it's invocations then M' accepts x, else it rejects x).
Every invocation of M on x requires a polynomial space. And, because we can use the same space for all invocations, we see that M' uses polynomial space (the fact that we run it exponentially many times does not matter). The answer of M' is correct because M is a PP machine that answers correctly for more than half of the guesses.

2. Small Variants
We mentioned that, in Definition 7.6, we can't take $\geq$ instead of of $>$ because this will give us no information. But what about asking for $\geq$ when $x \notin L$ and $>$ when $x \in L$ (or the other way around) ? We will show, in the next claim, that this will not change the class of languages. A language has such a machine if and only if it has a PP machine.

Consider the following definition:

Definition 7.7 $$PP1 \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exist a polynomial time} \\ \textit{Turing machine } M \textit{ s.t} \\ x \in L \Rightarrow Prob[M(x) = 1] > \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x) = 0] \geq \frac{1}{2} \end{array} \right\}$$

The next claim will show that this relaxation will not change the class defined:

Claim 7.4.1 $PP1 = PP$

Proof:
$PP \subseteq PP1$:
If we have a machine that satisfies Definition 7.6 it also satisfies Definition 7.7, so clearly $L \in PP \Rightarrow L \in PP1$.

$PP \supseteq PP1$:
Let L be any language in $PP1$. If M is the machine guaranteed by Definition 7.7, and $p(\cdot)$ is the polynomial bounding it's running time (and thus the number of coins that it uses), we can define another machine M' as follows:

$$M'\left(x, \left(a_1, a_2, ..., a_{p(|x|)+1}, b_1, b_2, ..., b_{p(|x|)}\right)\right) \stackrel{\text{def}}{=} \begin{cases} if\ a_1 = a_2 = ... = a_{p(|x|)+1} = 0\ then\ return\ 'NO' \\ else\ return\ M\left(x, \left(b_1, b_2, ..., b_{p(|x|)}\right)\right) \end{cases}$$

M' chooses one of two moves. One move, which happens with probability $2^{-(p(|x|)+1)}$, will return $'NO'$. The second move, which happens with probability $1 - 2^{-(p(|x|)+1)}$ will invoke M with independent coin tosses.
This gives us that

$$Prob[M'(x) = 1] = Prob[M(x) = 1] \cdot \left(1 - 2^{-(p(|x|)+1)}\right)$$

and

$$Prob[M'(x) = 0] = Prob[M(x) = 0] \cdot \left(1 - 2^{-(p(|x|)+1)}\right) + 2^{-(p(|x|)+1)}$$

The trick is to shift the answer of M towards the $'NO'$ direction with a very small probability. This shift is smaller than the smallest probability difference that M could have. So if $M(x)$ is biased towards the $'YES'$, our shift will keep the direction of the bias (it will only lower it). But if there is no bias (or bias towards NO), our shift will give us a bias towards the $'NO'$ answer.

If $x \in L$ then $Prob[M(x) = 1] > \frac{1}{2}$, hence $Prob[M(x) = 1] \geq \frac{1}{2} + 2^{-p(|x|)}$ (because the difference is at least one computation which happens with probability $2^{-p(|x|)}$), so:

$$\begin{aligned} Prob[M'(x) = 1] &\geq \left(\frac{1}{2} + 2^{-p(|x|)}\right) \cdot \left(1 - 2^{-(p(|x|)+1)}\right) \\ &= \frac{1}{2} + 2^{-p(|x|)} - 2^{-(p(|x|)+2)} - 2^{-(2p(|x|)+1)} > \frac{1}{2} \end{aligned}$$

If $x \notin L$ then $Prob[M(x) = 0] \geq \frac{1}{2}$, hence

$$\begin{aligned} Prob[M'(x) = 0] &\geq \frac{1}{2} \cdot \left(1 - 2^{-(p(|x|)+1)}\right) + 2^{-(p(|x|)+1)} \\ &= \frac{1}{2} - 2^{-(p(|x|)+2)} + 2^{-(p(|x|)+1)} > \frac{1}{2} \end{aligned}$$

And, as a conclusion, we get that in any case

$$Prob[M'(x) = \chi_L(x)] > \frac{1}{2}$$

So M' satisfies Definition 7.6, and thus $L \in PP$. ■

3. $NP \subseteq PP$
Suppose that $L \in NP$ is decided by a nondeterministic machine M with a running-time that is bounded by the polynomial $p(|x|)$. The following machine M' then will decide L by means of Definition 7.6:

$$M'\left(x, \left(b_1, b_2, ..., b_{p(|x|)+1}\right)\right) \stackrel{\text{def}}{=} \begin{cases} if\ b_1 = 1\ then\ return\ M\left(x, \left(b_2, b_3, ..., b_{p(|x|)+1}\right)\right) \\ else\ return\ 'YES' \end{cases}$$

M' uses it's random coin-tosses as a witness to M with only one toss that it does not pass to M'. This toss is used to choose it's move. One of the two possible moves gets it to the ordinary computation of M with the same input (and the witness is the random input). The other choice gets it to a computation that always accepts.
Consider a string x.

If M doesn't have an accepting computation then the probability that M' will answer 1 is exactly $\frac{1}{2}$ (it is the probability that the first coin will fall on one). On the other hand, if M has at least one accepting computation then the probability that M' will answer correctly is greater than $\frac{1}{2}$.

So we get that:

$x \in L \Rightarrow \quad Prob[M'(x) = 1] > \frac{1}{2}$
$x \notin L \Rightarrow \quad Prob[M'(x) = 0] \geq \frac{1}{2}$

By Definition 7.7, we conclude that $L \in PP1$, and by the previous claim ($PP = PP1$), we get that $L \in PP$.

4. $coNP \subseteq PP$
   Easily seen from the symmetry in the definition of PP.

7.5 The class ZPP – Zero error probability.

RP definition is asymmetric and we can't say whether $RP = coRP$. It would be interesting to examine the properties of $RP \bigcap coRP$ which is clearly symmetric. It seems that problems which are in $RP \bigcap coRP$ can benefit from the accurate result of RP deciding Turing machine (if $x \notin L$) and of $coRP$ deciding Turing machine (if $x \in L$).
Another interesting thing to consider is to let the machine say "I don't know" for some inputs. We will discuss machines that can return this answer but answer correctly otherwise.
We will prove that these two ideas give rise to the same class of languages.

Definition 7.8 (ZPP): *$L \in ZPP$ if there exist a probabilistic polynomial-time Turing machine M, such that:*

$$\begin{array}{rc} \forall x, & Prob[M(x) = \perp] \leq \frac{1}{2} \\ \forall x, & Prob[M(x) = \chi_L(x) \ or \ M(x) = \perp] = 1 \end{array}$$

Where we denote the unknown answer sign as $\perp$.

Again the value $\frac{1}{2}$ is arbitrary and can be replaced like we did before to be anything between $2^{-p(|x|)}$ to $1 - \frac{1}{p(|x|)}$. If we have a ZPP machine that doesn't know the answer with probability half, we can run it $p(|x|)$ times and get a machine that doesn't know the answer with probability $2^{-p(|x|)}$ because this is the probability that none of our invocation know the answer (the other way is obvious because $2^{-p(|x|)}$ is smaller than $\frac{1}{2}$ for all but final inputs). If we have a machine that know the answer with probability $\frac{1}{p(|x|)}$, we can use it to build a machine that know the answer with probability $\frac{1}{2}$ by invoking it $p(|x|)$ times (the other way is, again, trivial).

Proposition 7.5.1 $ZPP = RP \bigcap coRP$

Proof: Take $L \in ZPP$. Let M be the machine guaranteed in Definition 7.8. We will show how to build a new machine M' which decides L according to Definition 7.1 (this will imply that $ZPP \subseteq RP$).

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} b \leftarrow M(x) \\ if \ b = \perp \ then \ output \ 0 \\ else \ output \ b \ itself \end{cases}$$

By doing so, if $x \notin L$ then by returning 0 when $M(x) = \perp$ we will always answer correctly (because in this case $M(x) \neq \perp \Rightarrow M'(x) = \chi_L(x) \Rightarrow M'(x) = 0$).
If $x \in L$, the probability of getting the right answer with M' is greater than $\frac{1}{2}$ because M will return a definite answer ($M(x) \neq \perp$) with probability greater than $\frac{1}{2}$ and M's definite answers are

always correct (it never return a wrong answer because it returns $\perp$ when it is uncertain).
In the same way it can be seen that $ZPP \subseteq coRP$ (the machine that we will build will return 1 when M is uncertain), hence we get that $ZPP \subseteq RP \bigcap coRP$.

Assume now that $L \in RP \bigcap coRP$. Let M_{RP} be the RP machine and M_{coRP} the $coRP$ machine that decides L (according to Definition 7.1 and Definition 7.2). We define $M'(x)$ using M_{RP} and M_{coRP} as follows:

$$M'(x) \stackrel{\text{def}}{=} \begin{cases} run\ M_{RP}(x),\ if\ it\ says\ 'YES'\ then\ return\ 1 \\ run\ M_{coRP}(x),\ if\ it\ says\ 'NO'\ then\ return\ 0 \\ otherwise\ return\ \ \perp \end{cases}$$

If M_{RP} says $'YES'$ then, by Definition 7.1, we are guaranteed that $x \in L$. Notice that it can happen that $x \in L$ and $M_{RP}(x) = 0$ but not the other way around (There are 1's in the probability space $M(x)$ when $x \in L$, but the probability space $M(x)$ when $x \notin L$ is all zeroes. So if $M(x)$ returns $'YES'$, we know that the first probability space is the case).
In a similar way, if M_{coRP} says $'NO'$ then, by Definition 7.2, we are guaranteed that $x \notin L$. Thus we never get a wrong answer.

If $x \in L$ then, by Definition 7.1, we will get a $'YES'$ answer form M_{RP} and hence from M' with probability greater than $\frac{1}{2}$. If $x \notin L$ than, by Definition 7.2, we will get a $'NO'$ answer form M_{coRP} and hence from M' with probability greater than $\frac{1}{2}$.
So in either cases we can be sure that M' returns a definite (not $\perp$) and correct answer with probability greater than $\frac{1}{2}$.

The conclusion is that M' is indeed a ZPP machine so $RP \bigcap coRP \subseteq ZPP$ and, together with the previous part, we conclude that $RP \bigcap coRP = ZPP$. $\blacksquare$

Summing what we have seen so far we can write the following relations

$$P \subseteq ZPP \subseteq RP \subseteq BPP$$

It is believed that $BPP = P$ so there is no real help that randomized computations can contribute when trying to solve search problems. Also if the belief is true then all the distinctions between the above classes are of no use.

7.6 Randomized space complexity

Like we did with NL, we also define randomized space classes. Here also, it is possible to consider both the online and off-line models and we will work with the online model.

7.6.1 The definition

Definition 7.9 *For any function $S : \mathsf{N} \to \mathsf{N}$*

$$RSPACE(S) \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} There\ exists\ a\ randomized\ Turing\ machine\ M \\ s.t.\ for\ any\ input\ x \in \{0,1\}^* \\ x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x) = 0] = 0 \\ and\ M\ uses\ at\ most\ S(|x|)\ space \\ and\ exp(S(|x|))\ time. \end{array} \right\}$$

We are interested in the case where the space is logarithmic. The class which put the logarithmic space restriction is RL.

Definition 7.10 $RL \stackrel{\text{def}}{=} RSPACE(log)$

The time restriction is very important. Let us see what happens if we don't put the time restriction in Definition 7.9.

Definition 7.11 *For any function* $S : \mathsf{N} \to \mathsf{N}$

$$badRSPACE(S) \stackrel{\text{def}}{=} \left\{ L \subseteq \{0,1\}^* \;\middle|\; \begin{array}{l} \textit{There exists a randomized Turing machine } M \\ \textit{s.t. for any input } x \in \{0,1\}^* \\ x \in L \Rightarrow Prob[M(x) = 1] \geq \frac{1}{2} \\ x \notin L \Rightarrow Prob[M(x) = 0] = 0 \\ \textit{and } M \textit{ uses at most } S(|x|) \textit{ space} \\ \text{(no time restrictions!)} \end{array} \right\}$$

Proposition 7.6.1 badRSPACE(S) = NSPACE(S)

Proof: We start with the easy direction. Let $L \in badRSPACE(S)$. If $x \in L$ then there are many witnesses but one is enough. On the other hand for $x \notin L$ there are no witness.

The other direction is the interesting one. Suppose $L \in NSPACE(S)$. Let M be the Non-deterministic Turing machine which decides L in space $S(|x|)$. Recall that for every $x \in L$ there exists an accepting computation of M on input x which halts within $exp(S(|x|))$ steps (see previous lectures!). Then if $x \in L$ there exist r of length $exp(S(|x|))$, so that $M(x,r) = 1$ (here r denotes the offline non-deterministic guesses used by M). Thus, selecting r uniformly among the strings of length $exp(S(|x|))$, the probability that $M(x,r) = 1$ is at least $2^{-exp(S(|x|))}$. So if we repeatedly invoke $M(x,.)$ on random r's, we can expect that after $2^{exp(S(|x|))}$ tries we will see an accepting computation (assuming all the time that $x \in L$).

> Oded's Note: *Note that the above intuitive suggestion already abuses the fact that badRSPACE has no time bounds. We plan to run in expected time which is* double exponential *in the space bound; whereas the good definition of RSPACE allows only time exponential in the space bound.*

So we want to run M on x and a newly randomly selected r (of length $exp(S(|x|))$) for about $2^{exp(S(|x|))}$ times and accept iff M accepts in one of these tries. A naive implementation is just to do so. But this requires holding a counter capable of counting upto $t \stackrel{\text{def}}{=} 2^{exp(S(|x|))}$, which means using space $exp(S(|x|))$ (which is much more than we are allowed). So we have the basic idea which is good but still have a problem how to count. The solution will be to use a "randomized counter" that will only use $S(|x|)$ space.

The randomized counter is implemented as follows. We "flip" $k = \log_2 t$ coins. If all are heads then we will stop otherwise we go on. The expected number of tries is $2^{-k} = t$, exactly the number of tries we wanted to have. But this randomized counter requires only a real counter capable of counting upto k, and so can be implemented in space $\log_2 k = \log_2 \log_2 t = S(|x|)$. ■

Clearly,

Claim 7.6.2 $L \subseteq RL \subseteq NL$

7.6.2 Undirected Graph Connectivity is in RL

In the previous lecture we saw that directed connectivity is NL-Complete. We will now show in brief that undirected connectivity is in RL. The problem is defined as follows.

Input: An undirected graph G and two vertices s and t.

Task: Find if there is a path between s and t in G.

Claim 7.6.3 *Let n denote the number of vertices in the graph. Then, with probability at least $\frac{1}{2}$, a random walk of length $8n^3$ starting from s visits all vertices in the connected component of s.*

By a random walk, we mean a walk which iteratively selects at random a neighbour of the current vertex and moves to it.

Proof sketch: In the following, we consider the connected component of vertex s, denoted $G' = (V', E')$. For any edge, (u, v) (in E'), we let $T_{u,v}$ be a random variable representing the number of steps taken in a random walk starting at u until v is first encountered. It is easy to see that $\mathrm{E}[T_{u,v}] \le 2|E'|$. Also, letting $\mathrm{cover}(G')$ be the expected number of steps in a random walk starting at s and ending when the last of the vertices of V' is encountered, and C be any directed cycle which visits all vertices in G', we have

$$\begin{aligned}\mathrm{cover}(G') &\le \sum_{(u,v)\in C} \mathrm{E}[T_{u,v}] \\ &\le |C| \cdot 2|E'|\end{aligned}$$

Letting C be a traversal of some spanning tree of G', we conclude that $\mathrm{cover}(G') < 4 \cdot |E'| \cdot |V'|$. Thus, with probability at least $1/2$, a random walk of length $8 \cdot |E'| \cdot |V'|$ starting at s visits all vertices of G'. ■

The algorithm for deciding undirected connectivity is now obvious: Just take a "random walk" of length $8n^3$ starting from vertex s and see if t is encountered. The space requirement is merely a register to hold the current vertex (i.e., $\log n$ space) and a counter to count upto $8n^3$ (again $(\log n)$ space). Furthermore, the use of a counter guarantees that the running time of the algorithm is exponential in its (logarithmic) space bound. The implementation is straightforward

1. Set $counter = 0$ and $v = s$. Compute n (the number of vertices in the graph).
2. Uniformly select a neighbour u of v.
3. If $u = t$ then halt and accept, else set $v = u$ and $counter = counter + 1$.
4. If $counter = 8n^3$ then halt and reject, else goto Step (2).

Cleraly, if s is connected to t then, by the above claim, the algorithm accepts with probability at least $1/2$. On the other hand, the algorithm alwas rejects if s is not connected to t. Thus, UNdirected graph CONNectivity (UNCONN) is in RL.

Note that the straightforward adaptation of the above algorithm to the directed case (i.e., directed graph connectivity considered in previous lecture) fails: Consider, for example, a directed graph consisting of a directed path $1 \to 2 \to \cdots \to n$ augmented by directed edges going from every vertex $i > 1$ to vertex 1. An algorithm which tries to take a directed random walk starting from vertex 1 is highly unlikely to reach vertex n in poly(n) many steps. Loosely speaking, this is the case since in each step from a vertex $i > 1$, we move towards vertex n with probability 1/2, but otherwise return to vertex 1. The fact that the above algorithm fails should not come as a great surprise, as the directed connectivity problem is NL-complete and so placing it in RL will imply $NL = RL$.

> Oded's Note: *$NL = RL$ is not considred as unlikely as $NP = RP$, but even if $NL = RL$ proving this seems very hard.*

Bibliographic Notes

Probabilistic Turing Machines and corresponding complexity classes (including $\mathcal{BPP}, \mathcal{RP}, \mathcal{ZPP}$ and $\mathcal{PP}$) were first defined by Gill [2]. The proof that NSPACE equals badRSPACE (called RSPACE in [2]), as well as the techinque of a randomized counter is from [2].

The robusteness of the various classes under various thresholds was established above using straightforward amplifications (i.e., running the algorithm several times with independent random choices). Randomness-efficient amplification methods have been extensively studied since the mid 1980's. See Section 3.6 in [3].

The random-walk algorithm for deciding undirected connectivity is due to Aleliunas *et. al.* [1]. Other examples of randomized algorithms can be found in Appendix B.1 of [3]. We specifically recommend the following examples

- Testing primality (B.1.5): This BPP algorithm is different from the famous $coRP$ algorithm for recognizing the set of primes.
- Finding a perfect matching (B.1.2): This algorithm is arguably simpler than known deterministic polynomial-time algorithms.
- Finding minimum cuts in graphs (B.1.7): This algorithm is arguably simpler than known deterministic polynomial-time algorithms.

A much more extensive treatment of randomized algorithm is given in [4].

1. R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovász and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th FOCS*, pages 218–223, 1979.

2. J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, Vol. 6(4), pages 675–695, 1977.

3. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998. Copies have been placed in the faculty's library.

4. R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

Lecture 8

Non-Uniform Polynomial Time ($\mathcal{P}$/Poly)

Notes taken by Moshe Lewenstein, Yehuda Lindell and Tamar Seeman

Summary: In this lecture we introduce the notion of non-uniform polynomial time and the corresponding complexity class $\mathcal{P}$/poly. In this computational model, Turing machines are provided an external advice string to aid them in their computation. The *non-uniformity* is expressed in the fact that a different advice string may be defined for every different length of input. We show that $\mathcal{P}$/poly upper bounds efficient computation (as $\mathcal{BPP} \subseteq \mathcal{P}$/poly), yet even contains some non-recursive languages. The effect of introducing *uniformity* is discussed (as an attempt to rid $\mathcal{P}$/poly of its absurd intractable languages) and shown to reduce the class to be exactly $\mathcal{P}$. Finally, we show that, among other things, $\mathcal{P}$/poly may help us separate $\mathcal{P}$ from $\mathcal{NP}$. We do this by showing that trivially $\mathcal{P} \subset \mathcal{P}$/poly, and that under a reasonable conjecture $\mathcal{NP} \not\subseteq \mathcal{P}$/poly.

8.1 Introduction

The class of $\mathcal{P}$/poly, or non-uniform polynomial time, is the class of Turing machines which receive external advice to aid computation. More specifically for all inputs of length n a Turing machine is supplemented with a single advice string a_n of polynomial length . Alternatively we may view a non-uniform machine as an infinite series of Turing machines $\{M_n\}$, where M_i computes for inputs of length i. In this case the advice is "hardwired" into the machine.

The class of $\mathcal{P}$/poly provides an upper bound on what is considered to be efficient computation. This upper bound is not tight; for example, as we shall show later, $\mathcal{P}$/poly contains non-recursive languages. However, the upper bound ensures that every efficiently computable language is contained in $\mathcal{P}$/poly.

An additional motivation in creating the class of $\mathcal{P}$/poly is to help separate the classes of $\mathcal{P}$ and $\mathcal{NP}$. This idea is explained in further detail below.

8.1.1 The Actual Definition

We now define the class of $\mathcal{P}$/poly according to two different definitions, and then show that these two definitions are in fact equivalent. Recall that:

$$\chi_L(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 8.1 (standard): *$L \in \mathcal{P}$/poly if there exists a sequence of circuits $\{C_n\}$, where for each n, C_n has n inputs and one output, and there exists a polynomial $p(.)$ such that for all n, size(C_n) $\leq p(n)$ and $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

A series of polynomial circuits $\{C_n\}$ as defined above is called a *non-uniform family of circuits.* The non-uniformity is expressed in the fact that there is not necessarily any connection between a circuit of size n and $n+1$. In fact for every n we may define a completely different "algorithm".

Note that the circuits in the above definition can be simulated in time linear to their size. Thus although time is not explicitly mentioned in the definition, it is implicit.

Definition 8.2 (alternative): *$L \in \mathcal{P}$/poly if there exists a polynomial-time two-input machine M, a polynomial $p(.)$, and a sequence $\{a_n\}$ of advice strings, where length(a_n) $\leq p(n)$, such that for all n and for all $x \in \{0,1\}^n$, $M(a_n, x) = \chi_L(x)$.*

If exponentially long advice were allowed in the above definition, then a_n could be a look-up table containing $\chi_L(x)$ for any language L and every input x of length n. Thus every language would trivially be in such a class. However, this is not the case as a_n is polynomially bounded. Restricting the length of the advice defines a more meaningful class, but as we have mentioned, some intractable problems still remain "solvable".

Proposition 8.1.1 *The two definitions of $\mathcal{P}$/poly are equivalent.*

Proof:
($\Rightarrow$): Assume $L \in \mathcal{P}$/poly by Definition 1, i.e. there exists a family $\{C_n\}$ of circuits deciding L, such that size(C_n) is polynomial in n. Let desc(C_n) be the description of C_n according to a standard encoding of circuits. Consider the universal Turing machine M such that for all n, and all x of length n, M(desc(C_n),x) simulates $C_n(x)$. Then define the sequence $\{a_n\}$ of advice strings such that for every n, a_n = desc(C_n). Thus $L \in \mathcal{P}$/poly by Definition 2.

($\Leftarrow$): Assume L is in $\mathcal{P}$/poly by Definition 2, i.e. there exist a Turing machine M and a sequence of advice $\{a_n\}$ deciding L. We look at all possible computations of $M(a_n, \cdot)$ for n-bit inputs. $M(a_n, \cdot)$ is a polynomial time-bounded deterministic Turing machine working on n-length inputs. In the proof of Cook's Theorem, in Lecture 2, we showed that Bounded Halting is Levin-reducible to Circuit Satisfiability. Given an instance of Bounded Halting $(< M(\cdot,\cdot) >, x, 1^t)$ the reduction is comprised of constructing a circuit C which on input y outputs $M(x, y)$. The situation here is identical since for $M(a_n, \cdot)$ a circuit may be constructed which on input x outputs $M(a_n, x)$. In other words we build a sequence $\{C_n\}$ of circuits, where for each n, C_n is an encoding of $M(a_n, \cdot)$. Thus L is in $\mathcal{P}$/poly by Definition 1. ■

It should be noted that in Definition 2, M is a finite object, whereas $\{a_n\}$ may be an infinite sequence (as is the sequence $\{C_n\}$ of circuits according to Definition 1). Thus $\mathcal{P}$/poly is an unrealistic mode of computation, since such machines cannot actually be constructed.

8.1.2 $\mathcal{P}$/poly and the $\mathcal{P}$ versus $\mathcal{NP}$ Question

As mentioned above, one of the motivations in defining the class of $\mathcal{P}$/poly is to separate $\mathcal{P}$ from $\mathcal{NP}$. The idea is to show that there is a language which is in $\mathcal{NP}$ but is not in $\mathcal{P}$/poly, and thus not in $\mathcal{P}$. In this way, we would like to show that $\mathcal{P} \neq \mathcal{NP}$. To do so, though, we must first understand the relationship of $\mathcal{P}$/poly to the classes $\mathcal{P}$ and $\mathcal{NP}$. Trivially, $\mathcal{P} \subseteq \mathcal{P}$/poly because the class $\mathcal{P}$ may be viewed as the set of $\mathcal{P}$/poly machines with empty advice, i.e. $a_n = \lambda$ for all n.

At first glance, Definition 2 of $\mathcal{P}$/poly appears to resemble that of $\mathcal{NP}$. In $\mathcal{NP}$, $x \in L$ iff there exists a witness w_x such that $M(x, w_x) = 1$. The witness is somewhat analogous to the advice in $\mathcal{P}$/poly. However, the definition of $\mathcal{P}$/poly differs from that of $\mathcal{NP}$ in two ways:

1. For a given n, $\mathcal{P}$/poly has a universal witness a_n as opposed to $\mathcal{NP}$ where every x of length n may have a different witness.
2. In the definition of $\mathcal{NP}$, for every $x \notin L$, for every witness w, $M(x, w) = 0$. In other words, there do not exist false witnesses. However, this is not true for $\mathcal{P}$/poly. We do not claim that there are no bad advice strings for Definition 2 of $\mathcal{P}$/poly; we merely claim that there exists a good advice string.

We therefore see that the definitions of $\mathcal{NP}$ and $\mathcal{P}$/poly differ from each other; this raises the possibility that there may be a language which is in $\mathcal{NP}$ but not in $\mathcal{P}$/poly. As we shall show later this seems to be likely since a sufficient condition for the existence of such a language is based upon a reasonable conjecture. Since $\mathcal{P}$ is contained in $\mathcal{P}$/poly, finding such a language is sufficient to fulfill our goal. In fact, the original motivation for $\mathcal{P}$/poly was the belief that one may be able to prove lower bounds on sizes of circuits computing certain functions (e.g., the characteriztic function of an NP-complete language). So far, no such bounds are known (except if one restricts the circuits in various ways; as we'll discuss in next semester).

8.2 The Power of $\mathcal{P}$/poly

As we have mentioned, $\mathcal{P}$/poly is not a realistic mode of computation. Rather, it provides an *upper bound* on what we consider efficient computation (that is, any language not in $\mathcal{P}$/poly should definitely not be efficiently computable). In the last lecture we defined probabilistic computation and reevaluated our view of efficient computation to be $\mathcal{BPP}$, rather than $\mathcal{P}$. We now show that $\mathcal{BPP} \subseteq \mathcal{P}$/poly and therefore that $\mathcal{P}$/poly also upper bounds our "new" view of efficient computation.

However, we will also show that $\mathcal{P}$/poly contains far more than $\mathcal{BPP}$. This actually yields a very high upper bound. In fact $\mathcal{P}$/poly even contains non-recursive languages. This containment should convince anyone that $\mathcal{P}$/poly does not reflect any level of realistic computation.

Theorem 8.3 : *$\mathcal{P}$/poly contains non-recursive languages.*

Proof: This theorem is clearly implied from the following two facts:

1. There exist unary languages which are non-recursive, and
2. For every unary language L, $L \in \mathcal{P}$/poly.

We remind the reader that L is a unary language if $L \subseteq \{1\}^*$.

Proof of Claim 1:

Let L be any non-recursive language. Define $L' = \{1^{index(x)} \mid x \in L\}$ where index(x) is the position of x in the standard enumeration of binary strings (i.e. we view the string as a binary number). Clearly L' is unary and non-recursive (any Turing machine recognizing L' can trivially be used to recognize L).

Proof of Claim 2:

For every unary language L, define

$$a_n = \begin{cases} 1, & \text{if } 1^n \in L; \\ 0, & \text{otherwise.} \end{cases}$$

A Turing machine can trivially decide L in polynomial (even linear) time given x and $a_{|x|}$, by simply accepting iff x is unary and $a_{|x|} = 1$. Therefore, $L \in \mathcal{P}$/poly. ■

The ability to decide intractable languages is a result of the non-uniformity inherent in $\mathcal{P}$/poly. There is no requirement that the series $\{a_n\}$ is even computable.

Note that this method of reducing a language to its unary equivalent cannot help us with polynomial classes as the reduction itself is exponential. However, for recursive languages we are interested in computability only.

Theorem 8.4 : *$\mathcal{BPP} \subseteq \mathcal{P}$/poly.*

Proof: Let $L \in \mathcal{BPP}$. By means of amplification, there exists a probabilistic Turing machine M such that for every $x \in \{0,1\}^n$: $Prob_{r\in\{0,1\}^{poly(n)}}[M(x,r) = \chi_L(x)] > 1 - 2^{-n}$, (the probabilities are taken over all possible choices of random strings).
Equivalently, M is such that $Prob_r[M(x,r) \neq \chi_L(x)] < 2^{-n}$. We therefore have:

$$Prob_r[\exists x \in \{0,1\}^n : M(x,r) \neq \chi_L(x)] \leq \sum_{x\in\{0,1\}^n} Prob_r[M(x,r) \neq \chi_L(x)] < 2^n \cdot 2^{-n} = 1.$$

The first inequality comes from the Union Bound, that is, for every series of sets $\{A_i\}$ and every random variable X:

$$Prob(X \in \bigcup_{i=1}^{n} A_i) \leq \sum_{i=1}^{n} Prob(X \in A_i).$$

and the second inequality is based on the error probability of the machine.

Note that if for every random string r, there is at least one x such that $M(x,r) \neq \chi_L(x)$, then the above probability would equal 1. We can therefore conclude that there is *at least* one string r such that for every x, $M(x,r) = \chi_L(x)$. We therefore set $a_n = r$ (note that r is different for different lengths of input n, but this is fine according to the definition of $\mathcal{P}$/poly). Our $\mathcal{P}$/poly machine simulates M, using a_n as its random choices. ■

This method of proof is called *the probabilistic method*. We do not know how to find these advice strings and the proof of their existence is implicit. We merely argue that the probability that a random string is not an adequate advice is strictly smaller than 1. This is enough to obtain the theorem.

8.3 Uniform Families of Circuits

As we have mentioned earlier, circuits of different sizes belonging to a non-uniform family may have no relation to each other. This results in the absurd situation of having families of circuits deciding non-recursive languages.

This leads us to the following definition which attempts to define families of circuits which *do* match our expectations of realistic computation.

Definition 8.5 (uniform circuits): *A family of circuits $\{C_n\}$ is called* uniform *if there exists a deterministic polynomial time Turing machine M such that for every n, $M(1^n) = desc(C_n)$, where $desc(C_n)$ is a standard encoding of circuits.*

Thus a uniform family of circuits has a succinct (finite) description (or equivalently for a series of advice strings). Clearly, a uniform family of circuits cannot recognize non-recursive languages. Actually, the restriction of uniformity is far greater than just this.

Theorem 8.6 : *A language L has a uniform family of circuits $\{C_n\}$ such that for all n and for all $x \in \{0,1\}^n$ $C_n(x) = \chi_L(x)$ if and only if $L \in \mathcal{P}$.*

Proof:
($\Rightarrow$) Let $\{C_n\}$ be a uniform family of circuits deciding L, and M the polynomial time Turing machine which generates the family. The following is a polynomial time algorithm for deciding L:

On input x:

- $C_{|x|} \leftarrow M(1^{|x|})$
- Simulate $C_{|x|}(x)$ and return the result.

Since M is polynomial-time bounded and the circuits are of polynomial size, the algorithm clearly runs in polynomial time. Therefore $L \in \mathcal{P}$.

($\Leftarrow$) $L \in \mathcal{P}$. Therefore, there exists a polynomial time Turing machine M deciding L. As in the proof of Cook's Theorem, a polynomial size circuit deciding L on strings of length n may be built from M in time polynomial in n. The Turing machine M' that constructs the circuits may then be taken as M in the definition of uniform circuits. That is, given x, M' calculates $|x|$ and builds the appropriate circuit.

Alternatively, by Definition 2, no advice is necessary here and we may therefore take $a_n = \lambda$ for every n. ■

8.4 Sparse Languages and the $\mathcal{P}$ versus $\mathcal{NP}$ Question

In this section we will see why $\mathcal{P}$/poly may help us separate between $\mathcal{P}$ and $\mathcal{NP}$. We will first define sparse languages.

Definition 8.7 (sparse languages): *A language S is* sparse *if there exists a polynomial $p(\cdot)$ such that for every n $|S \cap \{0,1\}^n| \leq p(n)$.*

Example: Trivially, every unary language is sparse (take $p(n) = 1$).

Theorem 8.8 : *$\mathcal{NP} \subseteq \mathcal{P}/poly$ if and only if for every $L \in \mathcal{NP}$, the language L is Cook-reducible to a sparse language.*

As we conjecture that no $\mathcal{NP}$-Complete language can be sparse, we have that $\mathcal{NP}$ contains languages not found in $\mathcal{P}$/poly.

Proof: It is enough for us to prove that SAT $\in \mathcal{P}$/poly if and only if SAT is Cook-reducible to some sparse language.

($\Rightarrow$) Suppose that SAT $\in \mathcal{P}$/poly. Therefore there exists a series of advice strings $\{a_n\}$ and a Turing machine M as in Definition 2, where $\forall n\ |a_n| \leq q(n)$ for some polynomial $q(\cdot)$.
Define $s_i^n = 0^{i-1}10^{q(n)-i}$ and define $S = \{1^n 0 s_i^n : \text{for } n \geq 0 \text{ where bit } i \text{ of } a_n \text{ is } 1\}$.
Clearly S is sparse since for every n $|S \cap \{0,1\}^{n+q(n)+1}| \leq |a_n| \leq q(n)$.

We now show a Cook-reduction of SAT to S:
Input: φ of length n

1. Reconstruct a_n by $q(n)$ queries to S. Specifically, the queries are: $1^n 0 s_1^n, 1^n 0 s_2^n, ..., 1^n 0 s_{q(n)}^n$.
2. Run $M(a_n, \varphi)$ thereby solving SAT in (standard) polynomial time.

We therefore solve SAT with a polynomial number of queries to an S-oracle, i.e. SAT Cook-reduces to S.

($\Leftarrow$) Suppose that SAT Cook-reduces to some sparse language S. Therefore, there exists a polynomial time bounded oracle machine M^S which solves SAT. Let $t(\cdot)$ be M's (polynomial) time-bound. Then, on input x, machine M makes queries of length at most $t(|x|)$.
Construct a_n in the following way: concatenate all strings of length *at most* $t(n)$ in S. Since S is sparse, there exists some polynomial $p(\cdot)$ such that $\forall n\ |S \cap \{0,1\}^n| \leq p(n)$. The length of the list of strings of lengths exactly i in a_n is then less than or equal to $i \cdot p(i)$ (i.e. at most $p(i)$ different strings of length i each). Therefore:

$$|a_n| \leq \sum_{i=1}^{t(n)} i \cdot p(i) < t(n)^2 \cdot p(t(n))$$

So, a_n is polynomial in length. Now, given a_n, every oracle query to S can be "answered" in polynomial time. For a given string x, we check if $x \in S$ by simply scanning a_n and seeing if x appears or not. Therefore, M^S may by simulated by a deterministic machine with access to a_n. This machine takes at most $t(n) \cdot |a_n|$ time (each lookup may take as long as scanning the advice). Therefore SAT $\in \mathcal{P}$/poly. ■

As we have mentioned, we conjecture that there are no sparse $\mathcal{NP}$-Complete languages. This conjecture holds for both Karp and Cook reductions. However for Karp-reductions, the ramifications of the existence of a sparse $\mathcal{NP}$-Complete language would be extreme, and would show that $\mathcal{P} = \mathcal{NP}$. This is formally stated and proved in the next theorem. It is interesting to note that our belief that $\mathcal{NP} \not\subseteq \mathcal{P}$/poly is somewhat parallel to our belief that $\mathcal{P} \neq \mathcal{NP}$ when looked at in the context of sparse languages.

Theorem 8.9 *$\mathcal{P} = \mathcal{NP}$ if and only if for every language $L \in \mathcal{NP}$, the language L is Karp-reducible to a sparse language.*

Proof:
($\Rightarrow$): Let $L \in \mathcal{NP}$. We define the following trivial function as a Karp-reduction of L to $\{1\}$:

$$f(x) = \begin{cases} 1, & \text{if } x \in L; \\ 0, & \text{otherwise.} \end{cases}$$

If $\mathcal{P} = \mathcal{NP}$ then L is polynomial-time decidable and it follows that f is polynomial-time computable. Therefore, L Karp-reduces to the language $\{1\}$, which is obviously sparse.

($\Leftarrow$): For sake of simplicity we prove a weaker result for this direction. However the claim is true as stated. Beforehand we need the following definition:

Definition 8.10 (guarded sparse languages): *A sparse language S is called* guarded *if there exists a sparse language G in $\mathcal{P}$ such that $S \subseteq G$.*

The language that we considered in the proof of theorem 8: $S = \{1^n 0 s_i^n : \text{for } n \geq 0, \text{ where bit } i$ of a_n is $1\}$ is an example of a sparse guarded language. It is obviously sparse and it is guarded by $G = \{1^n 0 s_i^n : \forall n \geq 0 \text{ and } 1 \leq i \leq q(n)\}$. Note that any unary language is also a guarded sparse language since $\{1^n : n \geq 0\}$ is sparse and trivially in $\mathcal{P}$.

The slightly weaker result that we prove for this direction is as follows.

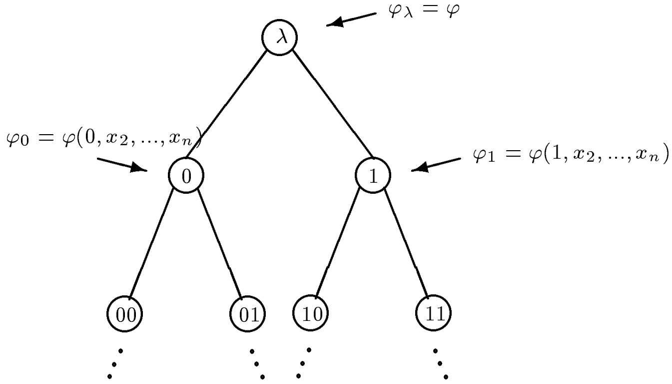
Proposition 8.4.1 *If SAT is Karp-reducible to a guarded sparse language then SAT $\in \mathcal{P}$.*

Proof: Assume that SAT is Karp-reducible to a sparse language S that is guarded by G. Let f be the Karp-reduction of SAT to S. We will show a polynomial-time algorithm for SAT.

Input: A Boolean formula $\varphi = \varphi(x_1, ..., x_n)$.

Envision the binary tree of all possible assignments. Each node is labelled $\alpha = \alpha_1\alpha_2...\alpha_i \in \{0,1\}^i$ which corresponds to an assignment of φ's first i variables. Let $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$ be the CNF formula corresponding to α. We denote $x_\alpha = f(\varphi_\alpha)$ (recall that $\varphi_\alpha \in \text{SAT} \Leftrightarrow x_\alpha \in S$).

The root is labelled λ, the empty string, where $\varphi_\lambda = \varphi$. Each node labelled α has two sons, one labelled $\alpha 0$ and the other labelled $\alpha 1$ (note that the sons have one variable less in their corresponding formulae). The leaves are labelled with n-bit strings corresponding to full assignments, and therefore to a Boolean constant.



The tree of assignments.

The strategy we will employ to compute φ will be a DFS search on this tree from root to leaves using a branch and bound technique. We backtrack from a node *only* if there is no satisfying assignment in its entire subtree. As soon as we find a leaf satisfying φ, we halt returning the assignment.

At a node α we consider x_α. If $x_\alpha \notin G$ (implying that $x_\alpha \notin S$), then φ_α is not satisfiable. This implies that the subtree of α contains no satisfying assignments and we can stop the search on this subtree. If $x_\alpha \in G$, then we continue searching in α's subtree.

At a leaf α we check if the assignment α is satisfiable (note that it is not sufficient to check that $x_\alpha \in G$ since f reduces to S and not to G). This is easy as we merely need to evaluate a Boolean expression in given values.

The key to the polynomial time-bound of the algorithm lies in the sparseness of G. If we visit a number of nodes equal to the number of strings in G of appropriate length, then the algorithm will clearly be polynomial. However, for two different nodes α and β, it may be that $x_\alpha = x_\beta \in G$ and we search both their subtrees resulting in visiting too many nodes. We therefore maintain a set B such that $B \subseteq G - S$ remains invariant throughout. Upon backtracking from a node α (where $x_\alpha \in G$), we place x_α in B. We then check for every node α, that $x_\alpha \notin B$ before searching its subtree, thus preventing a multiple search.

Algorithm: On input $\varphi = \varphi(x_1, ..., x_n)$.

1. $B \leftarrow \emptyset$
2. Tree-Search(λ)
3. In case the above call was not halted, reject φ as non-satisfiable.

In the following procedure, returning from a recursive call on α indicates that the subtree rooted in α contains no satisfying assignment (or, in other words, φ_α is not satisfiable). In case we reach a leaf associated with a satisfying assignment, the procedure halts outputting this assignment.

Procedure Tree-Search(α)

1. determine $\varphi_\alpha(x_{i+1}, ..., x_n) = \varphi(\alpha_1, ..., \alpha_i, x_{i+1}, ..., x_n)$
2. if $|\alpha| = n$: /* at a leaf - φ_α is a constant */
   if $\varphi_\alpha \equiv T$ then output the assignment α and halt
   else return
3. if $|\alpha| < n$:
   (a) compute $x_\alpha = f(\varphi_\alpha)$
   (b) if $x_\alpha \notin G$ /* checkable in poly-time, because $G \in \mathcal{P}$ */
   then return /* $x_\alpha \notin G \Rightarrow x_\alpha \notin S \Rightarrow \varphi_\alpha \notin$ SAT */
   (c) if $x_\alpha \in B$ then return
   (d) Tree-Search($\alpha 0$)
   Tree-Search($\alpha 1$)
   (e) /* We reach here only if both calls in the previous step fail. */
   if $x_\alpha \in G$ then add x_α to B
   (f) return

End Algorithm.

Correctness: During the algorithm B maintains the invariant $B \subseteq G - S$. To see this note that x_α is added to B only if $x_\alpha \in G$ and we are backtracking. Since we are backtracking there are no satisfying assignments in α's subtree, so $x_\alpha \notin S$.

Note that if $x_\alpha \in S$ then $x_\alpha \in G$ ($S \subseteq G$) and $x_\alpha \notin B$ (because B maintains $B \subseteq G - S$). Therefore, if φ is satisfiable then we will find some satisfying assignment since for all nodes α on the path from the root to the appropriate leaf, $x_\alpha \in S$, and its sons are developed.

Complexity: To show that the time complexity is polynomial it is sufficient to show that only a polynomial portion of the tree is "developed". The following claim will yield the desired result.

Claim 8.4.2 *Let α and β be two nodes in the tree such that (1) neither is a prefix/ancestor of the other and (2) $x_\alpha = x_\beta$. Then it is not possible that the sons of both nodes were developed (in Step 3d).*

Proof: Assume we arrived at α first. Since α is not an ancestor of β we arrive at β after backtracking from α. If $x_\alpha \notin G$ then $x_\beta \notin G$ since $x_\beta = x_\alpha$ and we will not develop either. Otherwise, it must be that $x_\alpha \in B$ after backtracking from α. Therefore $x_\beta \in B$ and its sons will not be developed (see Step 3c). $\blacksquare$

Corollary 8.4.3 *Only a polynomial portion of the tree is "developed".*

Proof: There exists a polynomial $q(.)$ that time-bounds the Karp-reduction f. Since every x_α is obtained by an application of f, $x_\alpha \in \cup_{i \le q(n)} \{0,1\}^i$. Yet G is sparse so $|G \cap (\cup_{i \le q(n)} \{0,1\}^i)| \le p(n)$ for some polynomial $p(\cdot)$.

Consider a certain level of the tree. Every two nodes α and β on this level are not ancestors of each other. Moreover on this level of the tree there are at most $p(n)$ different α's such that $x_\alpha \in G$. Therefore by the previous claim the number of x_α's developed forward on this level is bounded by $p(n)$. Therefore the overall number of nodes developed is bounded by $n \cdot p(n)$. $\blacksquare$

SAT $\in \mathcal{P}$ and the proof is complete. $\blacksquare$

Bibliographic Notes

The class P/poly was defined by Karp and Lipton as part of a general formulation of "machines which take advise" [3]. They have noted the equivalence to the traditional formulation of polynomial-size circuits, the effect of uniformity, as well as the relation to Cook-reducibility to sparse sets (i.e., Theorem 8.8).

Theorem 8.4 is atrriburted to Adleman [1], who actually proved $\mathcal{RP} \subseteq \mathcal{P}$/poly using a more involved argument. Theorem 8.9 is due to Fortune [2].

1. L. Adleman, "Two theorems on random polynomial-time", in *19th FOCS*, pages 75–83, 1978.

2. S. Fortune, "A Note on Sparse Complete Sets", *SIAM J. on Computing*, Vol. 8, pages 431–433, 1979.

3. R.M. Karp and R.J. Lipton. "Some connections between nonuniform and uniform complexity classes", in *12th STOC*, pages 302-309, 1980.

Lecture 9

The Polynomial Hierarchy (PH)

Notes taken by Ronen Mizrahi

Summary: In this lecture we define a hierarchy of complexity classes starting from $\mathcal{NP}$ and yet contained in PSPACE. This is done in two ways, the first by generalizing the notion of Cook reductions, and the second by generalizing the definition of $\mathcal{NP}$. We show that the two are equivalent. We then try to make some observations regarding the hierarchy, our main concern will be to learn when does this hierarchy collapse, and how can we relate it to complexity classes that we know already such as BPP and P/Poly.

9.1 The Definition of the class PH

In the literature you may find three common ways to define this class, two of those ways will be presented here. (The third, via "alternating" machines is omitted here.)

9.1.1 First definition for PH: via oracle machines

Intuition

Recall the defintion of a Cook reduction, the reduction is done using a polynomial time machine that has access to some oracle. Requiring that the oracle will belong to a given complexity class C, will raise the question:

> What is the complexity class of all those languages that are Cook reducable to some language from C?

For example:

> Let us set the complexity class of the oracle to be $\mathcal{NP}$, then for Karp reduction we know that every language L, that is Karp reducable to some language in $\mathcal{NP}$ (say SAT), will also be in $\mathcal{NP}$. However it is not clear what complexity class will a Cook reduction (to $\mathcal{NP}$) yield.

Perliminary definitions

Definition 9.1 (the language $L(M^A)$): *The language $L(M^A)$ is the set of inputs accepted by machine M given access to oracle A.*

Notations:

- M^A : The orcale machine M with access to oracle A.
- $M^A(x)$: The output of the orcale machine M^A on input x.

We note the following interesting cases for the above definition:

1. M is a deterministic polynomial time machine. Then M is a Cook reduction of $L(M^A)$ to A.
2. M is a probabilistic polynomial time machine. Then M is a randomized Cook reduction of $L(M^A)$ to A.
3. M is a non-deteministic polynomial time machine (note that the non determinism is related only to M, A is an oracle and as such it always gives the right answer). When we define the polynomial hierarchy we will use this case.

Observe that given one of the above cases, knowing the complexity class of the oracle, will define another complexity class which is the set of languages $L(M^A)$, where A is an oracle from the given comlexity class. The resulting complexity class may be one that is known to us (such as $\mathcal{P}$ or $\mathcal{NP}$), or a new class.

Definition 9.2 (the class M^C): *Let M be an oracle machine. Then M^C is the set of languages obtained from the machine M given access to an oracle from the class of languages C. That is,*

$$M^C \stackrel{\text{def}}{=} \{L(M^A) : A \in C\}$$

For example:

- $M^{\mathcal{NP}} = \{L(M^A) : A \in \mathcal{NP}\}$

  Note: we do not gain much by using $\mathcal{NP}$, rather than any $\mathcal{NP}$-complete language (such as SAT). That is, we know that any language, A, in $\mathcal{NP}$ is Karp reducable to SAT, by using this reduction we can alter M, and obtain a new machine $\tilde{M}$, such that $L(M^A) = L(\tilde{M}^{SAT})$.

In the following definition we abuse notation a little. We write ${C_1}^{C_2}$ but refer to machines natually associated with the class C_1, and to their natural extension to orale machines. We note that not every class has a natural enumeration of machines associated with it, let allow a natural extension of such machines to oracle machines. However, such associations and extensions do hold for the main classes we are interested in such as $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{BPP}$.

Definition 9.3 (the class ${C_1}^{C_2}$ – a fuzzy framework): *Assume that C_1 and C_2 are classes of languages, and also that for each language L in C_1, there exists a machine M_L, such that $L = L(M_L)$. Furthermore, consider the extension of M_L into an oracle machine M so that given access to the empty oracle M behaves as M_L (i.e., $L(M_L) = L(M^{\emptyset})$). Then ${C_1}^{C_2}$ is the set of languages obtained from such machines M_L, where $L \in C_1$, given access to an oracle for a language from the class of languages C_2. That is,*

$${C_1}^{C_2} = \{L(M^A) : L(M^{\emptyset}) \in C_1 \;\&\; A \in C_2\}$$

The above framework can be properly instantiated in some important cases. For example:

- $\mathcal{P}^C = \{L(M^A) : M$ is deterministic polynomial-time oracle machine & $A \in C\}$
- $\mathcal{NP}^C = \{L(M^A)$: same as above but M is non-deterministic$\}$
- $\mathcal{BPP}^C = \{L(M^A)$: same as above but M is probabilistic$\}$

  Here we mean that with probability at least 2/3, machine M on input x and oracle aceess to $A \in C$ correctly decides whether $x \in L(M^A)$.

Back to the motivating question: Observe that saying that L is Cook-reducible to SAT (i.e., $L \propto_C SAT$) is equivalent to writing $L \in \mathcal{P}^{\mathcal{NP}}$. We may now re-address the question regarding the power of Cook reductions. Observe that $\mathcal{NP} \cup \text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$, this is because:

- $\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ holds, because for $L \in \mathcal{NP}$ we can take the oracle A to be an oracle for the language L, and the machine $M \in \mathcal{P}$ to be a trivial machine that takes its input asks the oracle about it, and outputs the oracle's answer.
- $\text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ holds, because we can take the same oracle as above, and a different (yet still trivial) machine $M \in \mathcal{P}$ that asks the oracle about its input, and outputs the boolean complement of the oracle's answer.

We conclude that under the assumption that $\mathcal{NP} \neq \text{co}\mathcal{NP}$, Cook-reductions to $\mathcal{NP}$ give us more power than Karp-reductions to the same class.

> Oded's Note: *We saw such a result already, but it was quite artificial. I refer to that fact that $\mathcal{P}$ is Cook-reducible to the class of trivial langugaes (i.e., the class $\{\emptyset, \{0,1\}^*\}$), whereas non-trivial languages can not be Karp-reduced to trivial ones.*

Actual definition

Definition 9.4 (the class Σ_i): *Σ_i is a sequence of sets and will be defined inductively:*

- $\Sigma_1 \stackrel{\text{def}}{=} \mathcal{NP}$
- $\Sigma_{i+1} \stackrel{\text{def}}{=} \mathcal{NP}^{\Sigma_i}$

Notations:

- $\Pi_i \stackrel{\text{def}}{=} \text{co}\Sigma_i$
- $\Delta_{i+1} \stackrel{\text{def}}{=} \mathcal{P}^{\Sigma_i}$

Definition 9.5 (The hierarchy – PH): $\mathsf{PH} \stackrel{\text{def}}{=} \cup_{i=1}^{\infty} \Sigma_i$

The arbitrary choice to use the Σ_i's (rather than the Π_i's or Δ_i's) is justified by the following observations.

Almost syntaxtic observations

Proposition 9.1.1 $\Sigma_i \cup \Pi_i \subseteq \Delta_{i+1} \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$.

Proof: We prove each of the two containments:

1. $\Sigma_i \cup \Pi_i \subseteq \Delta_{i+1} = \mathcal{P}^{\Sigma_i}$.
   The reason for that is the same as for $\mathcal{NP} \cup \text{co}\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}} = \Delta_2$ (see above)
2. $\mathcal{P}^{\Sigma_i} \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$.
   $\mathcal{P}^{\Sigma_i} \subseteq \mathcal{NP}^{\Sigma_i} = \Sigma_{i+1}$ is obvious. Since $\mathcal{P}^{\Sigma_i}$ is closed under complementation, $L \in \mathcal{P}^{\Sigma_i}$ implies that $\overline{L} \in \mathcal{P}^{\Sigma_i} \subseteq \Sigma_{i+1}$ and so $L \in \Pi_{i+1}$.

■

Proposition 9.1.2 $\mathcal{P}^{\Sigma_i} = \mathcal{P}^{\Pi_i}$ *and* $\mathcal{NP}^{\Sigma_i} = \mathcal{NP}^{\Pi_i}$.

Proof: Given a machine M and an oracle A, it is easy to modify M to $\tilde{M}$ such that: $L(M^{\overline{A}}) = L(\tilde{M}^A)$. The way we build $\tilde{M}$ is by taking M and flipping every answer obtained from the oracle. In particular, if M is deterministic (resp. non-deterministic) polynomail-time then so is $\tilde{M}$. Thus, for such M and any class $\mathcal{C}$ the classes $M^{\text{co}\mathcal{C}}$ and $\tilde{M}^{\mathcal{C}}$ are identical. ■

9.1.2 Second definition for PH: via quantifiers

Intuition

The approach taken here is to recall one of the definitions of $\mathcal{NP}$ and try to generalize it.

Definition 9.6 (polynomially-bounded relation): *a k-ary relation R is called* polynomially bounded *if there exists a polynomial $p(.)$ such that:*

$$\forall (x_1, \ldots, x_k),\ [(x_1, \ldots, x_k) \in R \Longrightarrow (\forall i)\, |x_i| \leq p(|x_1|)]$$

Note: our definition requires that all the elements of the relation are not too long with regard to the first element, but the first element may be very long. We could even require a stronger condition: $\forall i \forall j |x_i| \leq p(|x_j|)$, this will promise that every element of the relation is not too long with regard to every one of the others. We do not make this requirement because the above definition will turn out to be satisfactory for our needs, this is because in our relations the first element is the input word, and we need the rest of the elements in the relation to be bounded in the length of the input. Also the complexity classes, that we will define using the notion of a polynomially bounded k-ary relation, will turn out the same for both the weak and the strong definition of the relation.

We now state again the definition of the complexity class $\mathcal{NP}$:

Definition 9.7 ($\mathcal{NP}$): *$L \in \mathcal{NP}$ if there exists a polynomially bounded and polynomial time recognizable binary relation R_L such that:*

$$x \in L \ \textit{iff}\ \exists y \text{ s.t. } (x, y) \in R_L$$

The way to generalize this definition will be to use a k-ary relation instead of just a binary one.

Actual definition

What we redefine is the sequence of sets Σ_i such that Σ_1 will remain $\mathcal{NP}$. The definition for PH remains the union of all the Σ_i's.

Definition 9.8 (Σ_i): *$L \in \Sigma_i$ if there exists a polynomially bounded and polynomial time recognizable (i+1)-ary relation R_L such that:*

$$x \in L \ \textit{ iff } \ \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i, \ \textit{s.t.} \ (x, y_1, \ldots, y_i) \in R_L$$

- *$Q_i = \forall$ if i is even*
- *$Q_i = \exists$ otherwise*

9.1.3 Equivalence of definitions

We have done something that might seem a mistaken; that is, we have given the same name for an object defined by two different definitions. However, we now intend to prove that the classes produced by the two definitions are equal. A more conventional way to present those two definitions is to state one of them as the definition for PH, and then prove an "if and only if" theorem that characterizes PH according to the other definition.

Theorem 9.9 : *The above two definitions of PH are equivalent. Furthermore, for every i, the class Σ_i as in Definition 9.4 is identical to the one in Definition 9.8.*

Proof: We will show that for every i, the class Σ_i by the two definitions is equal. In order to distinguish between the clases produced by the two definitions we will introduce the following notation:

- Σ_i^1 is the set Σ_i produced by the first definition.
- Σ_i^2 is the set Σ_i produced by the second definition.
- Π_i^1 is the set Π_i produced by the first definition.
- Π_i^2 is the set Π_i produced by the second definition.

Part 1: We prove by induction on i that $\forall i$, $\Sigma_i^2 \subseteq \Sigma_i^1$:

- Base of induction: Σ_1 was defined to be $\mathcal{NP}$ in both cases so there is nothing to prove.
- We assume that the claim holds for i and prove for $i+1$: suppose $L \in \Sigma_{i+1}^2$ then by definition it follows that there exists a relation R_L such that:

  $$x \in L \ \text{ iff } \ \exists y_1 \forall y_2 \exists y_3 \ldots Q_i y_i Q_{i+1} y_{i+1}, \ \text{s.t.} \ (x, y_1, \ldots, y_i, y_{i+1}) \in R_L$$

  In other words this means that:

  $$x \in L \ \text{ iff } \ \exists y_1, \ \text{s.t.} \ (x, y_1) \in L_i$$

  where L_i is defined as follows:

  $$L_i \stackrel{\text{def}}{=} \{(x', y') : \forall y_2 \exists y_3 \ldots Q_i y_i Q_{i+1} y_{i+1}, \ \text{s.t.} \ (x', y', \ldots, y_i, y_{i+1}) \in R_L\}$$

We claim that $L_i \in \Pi_i^2$, this is by complementing the definition of Σ_i^2. If we do this complementation for $L \in \Sigma_i^2$ we get:

$$x \in L \text{ iff } \exists y_1 \forall y_2 \ldots Q_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \in R_L$$

$$x \in \overline{L} \text{ iff } \forall y_1 \exists y_2 \ldots \overline{Q}_i y_i, \text{ s.t. } (x, y_1, \ldots, y_i) \notin R_L$$

This is almost what we had in the definition of L_i except for the "$\notin R_L$" as opposed to "$\in R_L$". Remember that deciding membership in R_L is polynomial time recognizable, and therefore its complement is also so. Now that we have that $L_i \in \Pi_i^2$, we can use the inductive hypothesis $\Pi_i^2 \subseteq \Pi_i^1$. So far we have managed to show that:

$$x \in L \text{ iff } \exists y_1, \text{ s.t. } (x, y_1) \in L_i$$

Where L_i belongs to Π_i^1. We now claim that $L \in \mathcal{NP}^{\Pi_i^1}$, this is true because we can write a non-deterministic, polynomial-time machine, that decides membership in L, by guessing y_1, and using an oracle for L_i. Therefore we can further conclude that:

$$L \in \mathcal{NP}^{\Pi_i^1} \equiv \mathcal{NP}^{\Sigma_i^1} \equiv \Sigma_{i+1}^1.$$

Part 2: We prove by induction on i that $\forall i\,,\ \Sigma_i^1 \subseteq \Sigma_i^2$:

- Base of induction: as before.
- Induction step: suppose $L \in \Sigma_{i+1}^1$ then there exists a non-deterministic polynomial time machine M such that $L \in L(M^{\Sigma_i^1})$ which means that:

$$\exists L' \in \Sigma_i^1, \text{ s.t. }, L = L(M^{L'})$$

From the definition of $M^{L'}$ it follows that:

$$x \in L \text{ iff } \exists y, q_1, a_1, \ldots, q_t, a_t \text{ s.t. } :$$

1. Machine M, with non-determinstic choices y, interacts with its oracle in the following way:
   - 1^{st} query $= q_1$ and 1^{st} answer $= a_1$

     .

     .

     .
   - t^{th} query $= q_t$ and t^{th} answer $= a_t$
2. for every $1 \leq j \leq t$:
   - $(a_j = 1) \Longrightarrow q_j \in L'$
   - $(a_j = 0) \Longrightarrow q_j \notin L'$

where y is a description of the non-determinstic choices of M.

Let us view the above according to the second definition, that is, according to the definition with quantifiers, then the first item is a polynomial time predicate and therefore this potentially puts L in $\mathcal{NP}$. The second item involves L'. Recall that $L' \in \Sigma_i^1$ and that by the inductive hypothesis $\Sigma_i^1 \subseteq \Sigma_i^2$, and therefore we can view membership in L' according to the second definition, and embed this result within what we have above. This will yield that for every $1 \leq j \leq t$:

- $(a_j = 1) \Longrightarrow \exists y_1^{(j,1)} \forall y_2^{(j,1)} \dots Q_i y_i^{(j,1)}$, s.t. $(q_j, y_1^{(j,1)}, \dots, y_i^{(j,1)}) \in R_{L'}$
- $(a_j = 0) \Longrightarrow \forall y_1^{(j,2)} \exists y_2^{(j,2)} \dots Q_i y_i^{(j,2)}$, s.t. $(q_j, y_1^{(j,2)}, \dots, y_i^{(j,2)}) \in \overline{R}_{L'}$

Let us define:

- w_1 is the concatenation of:
  $y, q_1, a_1, \dots, q_t, a_t$, and $y_1^{(j,1)}$ for all j s.t. $(a_j = 1)$.
- w_2 is the concatenation of:
  $y_1^{(j,2)}$ for all j s.t. $(a_j = 0)$, and $y_2^{(j,1)}$ for all j s.t. $(a_j = 1)$.

  .

  .

  .
- w_i is the concatenation of:
  $y_{i-1}^{(j,2)}$ for all j s.t. $(a_j = 0)$, and $y_i^{(j,1)}$ for all j s.t. $(a_j = 1)$.
- w_{i+1} is the concatenation of:
  $y_i^{(j,2)}$ for all j s.t. $(a_j = 0)$.

R_L will be the $(i+1)$-ary relation defined in the following way: $(w_1, \dots, w_{i+1}) \in R_L$ iff for every $1 \le j \le t$:

- $(a_j = 1) \Longrightarrow (q_j, y_1^{(j,1)}, \dots, y_i^{(j,1)}) \in R_{L'}$
- $(a_j = 0) \Longrightarrow (q_j, y_1^{(j,2)}, \dots, y_i^{(j,2)}) \in \overline{R}_{L'}$

where the w_i's are parsed analogously to the above.

Since $R_{L'}$ and $\overline{R}_{L'}$ where polynomially bounded, and polynomial time recognizable, so is R_L.

Altogether we have:

$$x \in L \text{ iff } \exists w_1, \forall w_2, \dots Q_{i+1} w_{i+1}, \text{ s.t. } (w_1, \dots, w_{i+1}) \in R_L$$

It now follows from the definition of Σ_{i+1}^2 that $L \in \Sigma_{i+1}^2$ as needed.

■

9.2 Easy Computational Observations

Proposition 9.2.1 $PH \subseteq PSPACE$

Proof: We will show that $\Sigma_i \subseteq PSPACE$ for all i. Let $L \in \Sigma_i$, then we know by the definition with quantifiers that:

$$x \in L \text{ iff } \exists y_1 \forall y_2 \exists y_3 \dots Q_i y_i, \text{ s.t. } (x, y_1, \dots, y_i) \in R_L$$

Given x we can use i variables to try all the possibilities for $y_1, \dots, y_i$ and make sure that they meet the above requirement. Since the relation R_L is polynomially bounded, we have a polynomial bound on the length of each of the y_i's that we are checking. Thus we have constructed a deterministic machine that decides L.

This machine uses i variables, the length of each of them is polynomially bounded in the length of the input. Since i is a constant, the overall space used by this machine is polynomial. ■

Proposition 9.2.2 *$\mathcal{NP} = \text{co}\mathcal{NP}$ implies $PH \subseteq \mathcal{NP}$ (which implies $PH = \mathcal{NP}$).*

Intuitively the extra power that non-deterministic Cook reductions have over non-deterministic Karp reductions, comes from giving us the ability to complement the oracle's answers for free. What we claim here is that if this power is meaningless then the whole hierarchy collapses.

Proof: We will show by induction on i that $\forall i$, $\Sigma_i = \mathcal{NP}$:

1. $i = 1$: by definition $\Sigma_1 = \mathcal{NP}$.

2. Induction step: by the inductive hypothesis it follows that $\Sigma_i = \mathcal{NP}$ so what remains to be shown is that $\mathcal{NP}^{\mathcal{NP}} = \mathcal{NP}$. Containment in one direction is obvious so we focus on proving that $\mathcal{NP}^{\mathcal{NP}} \subseteq \mathcal{NP}$. Let $L \in \mathcal{NP}^{\mathcal{NP}}$ then there exist a non-deterministic, polynomial-time machine M, and an oracle $A \in \mathcal{NP}$, such that $L = L(M^A)$. Since $\mathcal{NP} = \text{co}\mathcal{NP}$ it follows that $\overline{A} \in \mathcal{NP}$ too. Therefore, there exist relations R_A and $R_{\overline{A}}$ ($\mathcal{NP}$ relations for A and $\overline{A}$ respectively) such that:

   - $q \in A$ iff $\exists w$, s.t. $(q, w) \in R_A$.
   - $q \in \overline{A}$ iff $\exists w$, s.t. $(q, w) \in R_{\overline{A}}$

   Using these relations, and the definition of $\mathcal{NP}^{\mathcal{NP}}$ we get:
   $x \in L$ iff $\exists y, q_1, a_1, \ldots, q_t, a_t$, such that, for all $1 \le j \le t$:

   - $a_j = 1 \Longleftrightarrow q_j \in A \Longleftrightarrow \exists w_j, (q_j, w_j) \in R_A$
   - $a_j = 0 \Longleftrightarrow q_j \in \overline{A} \Longleftrightarrow \exists w_j, (q_j, w_j) \in R_{\overline{A}}$.

   Define:

   - w is the concatenation of: $y, q_1, a_1, \ldots, q_t, a_t, w_1, \ldots, w_t$
   - R_L is a binary relation such that:
     $(x, w) \in R_L$ iff for all $1 \le j \le t$:
     - $a_j = 1 \Longrightarrow (q_j, w_j) \in R_A$
     - $a_j = 0 \Longrightarrow (q_j, w_j) \in R_{\overline{A}}$.

     Since M is a polynomial-time machine, t is polynomial in the length of x. Combining this fact with the fact that both R_A and $R_{\overline{A}}$ are polynomial-time recognizable, and polynomially bounded, we conclude that so is R_L.

   All together we get that there exists an $\mathcal{NP}$ relation R_L such that :

   $$x \in L \text{ iff } \exists w, \text{ s.t. } (x, w) \in R_L$$

   Thus, $L \in \mathcal{NP}$.

■

Generalizing Proposition 9.2.2, we have

Proposition 9.2.3 *For every $k \ge 1$, if $\Pi_k = \Sigma_k$ then $PH = \Sigma_k$.*

A proof is presented in the appendix to this lecture.

9.3 BPP is contained in PH

Not knowing whether $\mathcal{BPP}$ is contained in $\mathcal{NP}$, it is of some confort to know that it is contained in the Polynomial-Hierarchy (which extends $\mathcal{NP}$).

Theorem 9.10 (Sipser and Lautemann): $BPP \subseteq \Sigma_2$.

Proof: Let $L \in BPP$ then there exists a probabilistic polynomial time machine $A(x,r)$ where x is the input and r is the random guess. By the definition of BPP, with some amplification we get, for some polynomial $p(.)$:

$$\forall x \in \{0,1\}^n, \text{ s.t. } \Pr_{r \in_R \{0,1\}^{p(n)}}[A(x,r) \neq \chi(x)] < \frac{1}{3p(n)}$$

where $\chi(x) = 1$ if $x \in L$ and $\chi(x) = 0$ otherwise.

> Oded's Note: *A word about the above is in place. Note that we do* **not** *assert that the error decreases as a fast fixed function of n, where the function is fixed before we determine the randomness complexity of the new algorithm. We saw result of that kind in Lecture 7; but here we claim something different. That is, that the error probability may depend on the randomness complexity of the new algorithm. Still, the dependency required here is easy to achieve. Specifically, suppose that the original algorithm uses $m = \text{poly}(n)$ coins. Then by running it t times and ruling by majority we decrease the error probability to $\exp(-\Omega(t))$. The randomness complexity of the new algorithm is tm. So we need to set t such that $\exp(-\Omega(t)) < 1/3mt$, which can be satisfied with $t = O(\log m) = O(\log n)$.*

The key observation is captured by the following claim

Claim 9.3.1 *Denote $m = p(n)$ then, for every $x \in L \cap \{0,1\}^n$, there exist $s_1, \ldots, s_m \in \{0,1\}^m$ such that*

$$\forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} A(x, r \oplus s_i) = 1 \tag{9.1}$$

Actually, the same sequence of s_i's may be used for all $x \in L \cap \{0,1\}^n$ (provided that $m \geq n$ which holds without loss of generality). However, we don't need this extra property.

Proof: We will show existence of such s_i's by the Probabilistic Method: That is, instead of showing that an object with some property exists we will show that a random object has the property with positive probability. Actually, we will upper bound the probability that a random object does not have the desired property. In our case we look for existence of s_i's satisfying Eq. (9.1), and so we will upper bound the probability, denoted P, that randomly chosen s_i's do not satisfy Eq. (9.1):

$$\begin{aligned}
P &\stackrel{\text{def}}{=} \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\neg \forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} (A(x, r \oplus s_i) = 1)] \\
&= \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\exists r \in \{0,1\}^m, \bigwedge_{i=1}^{m} (A(x, r \oplus s_i) = 0)] \\
&\leq \sum_{r \in \{0,1\}^m} \Pr_{s_1,\ldots,s_m \in_R \{0,1\}^m}[\bigwedge_{i=1}^{m} (A(x, r \oplus s_i) = 0)]
\end{aligned}$$

where the inequality is due to the union bound. Using the fact that the events of choosing s_i's uniformly are independent, we get that the probability of all the events happening at once equals to the multiplication of the probabilities. Therefore:

$$P \le \sum_{r\in\{0,1\}^m} \prod_{i=1}^{m} \mathrm{Pr}_{s_i\in_R\{0,1\}^m}[A(x,r\oplus s_i)=0]$$

Since in the above probability r is fixed, and the s_i's are uniformly distributed then (by a property of the $\oplus$ operator), the $s_i \oplus r$'s are also uniformly distributed. Recall that we consider an arbitrary fixed $x \in L \cap \{0,1\}^n$. Thus,

$$\begin{aligned} P &\le 2^m \cdot \mathrm{Pr}_{s\in_R\{0,1\}^m}[A(x,s)=0]^m \\ &\le 2^m \cdot \left(\frac{1}{3m}\right)^m \ll 1 \end{aligned}$$

The claim holds. ■

Claim 9.3.2 *For any $x \in \{0,1\}^n \setminus L$ and for all $s_1,\ldots,s_m \in \{0,1\}^m$, there exists $r \in \{0,1\}^m$ so that $\bigvee_{i=1}^m A(x,r\oplus s_i)=0$.*

Proof: We will actually show that for all $s_1,\ldots,s_m$ there are many such r's. Let $s_1,\ldots,s_m \in \{0,1\}^m$ be arbitrary.

$$\mathrm{Pr}_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x,r\oplus s_i)=0] = 1-\mathrm{Pr}_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x,r\oplus s_i)=1]$$

However, since $x \notin L$ and $\mathrm{Pr}_{r\in\{0,1\}^m}[A(x,r)=1] < 1/3m$, we get

$$\begin{aligned} \mathrm{Pr}_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x,r\oplus s_i)=1] &\le \sum_{i=1}^{m} \mathrm{Pr}_{r\in\{0,1\}^m}[A(x,r\oplus s_i)=1] \\ &\le m\cdot\frac{1}{3m} = \frac{1}{3} \end{aligned}$$

and so,

$$\mathrm{Pr}_{r\in\{0,1\}^m}[\bigvee_{i=1}^{m} A(x,r\oplus s_i)=0] \ge \frac{2}{3}$$

Therefore there exist (many) such r's and the claim holds. ■

Combining the results of the two claims together we get:

$$x\in L \text{ iff } \exists s_1,\ldots s_m \in\{0,1\}^m, \forall r \bigvee_{i=1}^{m} A(x,r\oplus s_i)=1$$

This assertion corresponds to the definition of Σ_2, and therefore $L \in \Sigma_2$ as needed. ■

Comment: The reason we used the $\oplus$ operator is because it has the property that given an arbitrary fixed r, if s is uniformly distributed then $r \oplus s$ is also uniformly distributed. Same for fixed s and random r. Any other efficient binary operation with this property may be used as well.

9.4 If NP has small circuits then PH collpases

The following result shows that an unlikely event regarding non-uniform complexity (i.e., the class $\mathcal{P}$/poly) implies an unlikely event regarding uniform complexity (i.e., PH).

Theorem 9.11 (Karp & Lipton): *If $\mathcal{NP} \subseteq \mathcal{P}$/poly then $\Pi_2 = \Sigma_2$, and so $PH = \Sigma_2$.*

Proof: We will only prove the first implication in the theorem. The second follows by Proposition 9.2.3. Showing that Σ_2 is closed under complementation, gives us that $\Pi_2 = \Sigma_2$. So what we will actually prove is that $\Pi_2 \subseteq \Sigma_2$.

Let L be an arbitrary language in Π_2, then there exists a trinary polynomially bounded, and polynomial time recognizable relation R_L such that:

$$x \in L \text{ iff } \forall y \exists z \text{ s.t. }, (x, y, z) \in R_L$$

Let us define:

$$L' \stackrel{\text{def}}{=} \{(x', y') : \exists z, \text{ s.t. } (x', y', z) \in R_L\}$$

Then we get that:

- $x \in L$ iff $\forall y, (x, y) \in L'$
- $L' \in \mathcal{NP}$

Consider a Karp reduction of L' to 3SAT, call it f:

$$x \in L \text{ iff } \forall y, f(x, y) \in 3SAT$$

Let us now use the assumption that $\mathcal{NP} \subseteq P/Poly$ for 3SAT, then it follows that 3SAT has small circuits $\{C_m\}_m$, where m is the length of the input. We claim that also 3SAT has small circuits $\{C'_n\}_n$ where n is the number of variables in the formula. This claim holds since the length of a 3SAT formula is of $O(n^3)$ and therefore $\{C'_n\}$ can use the larger sets of circuits $C_1, \ldots, C_{O(n^3)}$. Let us embed the circuits in our statement regarding membership in L, this will yield:

$x \in L$ iff $\exists (C'_1, \ldots, C'_n)$ $(n \stackrel{\text{def}}{=} max_y\{\#var(f(x, y))\})$ s.t.:

- $C'_1, \ldots, C'_n$ correctly computes 3SAT, for formulas with a corresponding number of variables.
- $\forall y,\ C'_{\#var(f(x,y))}(f(x, y)) = 1$

The second item above gives us that $L \in \Sigma_2$, since the quantifiers are as needed. However it is not clear that the first item is also bahaving as needed. We will restate the first item as follows:

$$\forall \phi_1, \ldots, \phi_n, [\bigwedge_{i=2}^{n} C'_i(\phi_i) = (C'_{i-1}(\phi'_i) \vee C'_{i-1}(\phi''_i)) \bigwedge C'_1 \textit{ operates correctly}] \tag{9.2}$$

Where:

$\phi_i(x_1, \ldots, x_i)$ is any formula over i variables.

$\phi'_i(x_1, \ldots, x_{i-1}) \stackrel{\text{def}}{=} \phi_i(x_1, \ldots, x_{i-1}, 0)$

$\phi''_i(x_1, \ldots, x_{i-1}) \stackrel{\text{def}}{=} \phi_i(x_1, \ldots, x_{i-1}, 1)$

A stupid technicality: Note that assigning a value to one of the variables, gives us a formula that is not in CNF as required by 3SAT (as its clauses may contain constants). However this can easily be achieved, by iterating the following process, where in each iteration one of the following rules is applied:

- $x \vee 0$ should be changed to x.
- $x \vee 1$ should be changed to 1.
- $x \wedge 0$ should be changed to 0.
- $x \wedge 1$ can be changed to x.
- $\neg 1$ can be changed to 0.
- $\neg 0$ can be changed to 1.

If we end-up with a formula in which some variables do not appear, we can augment it by adding clauses of the form $x \wedge \neg x$.

> Oded's Note: *An alternative resolution of the above technicality is to extend the definition of CNF so to allow constnats to appear (in clauses).*

Getting back to the main thing: We have given a recursive definition for a correct computation of the circuits (on 3SAT). The base of the recursion is checking that a single variable fromula is handled correctly by C'_1, which is very simple (just check if the single variable formula is satisfiable or not, and compare it to the output of the circuit). In order to validate the $(i+1)^{\text{th}}$ circuit, we wish to use the i^{th} circuit, which has already been validated. Doing so requires us to reduce the number of variables in the formula by one. This is done by assigning to one of the variables both possible values (0 or 1), and obtaining two formulas upon which the i^{th} circuit can be applied. The full formula is satisfiable iff at least one of the reduced formulas is satisfiable. Therefore we combine the results of applying the i^{th} circuit on the reduced formulas, with the $\vee$ operation. It now remains to compare it to the value computed by the $(i+1)^{\text{th}}$ circuit on the full formula. This is done for all formula over $i+1$ variables (by the quantification $\forall \phi_{i+1}$).

So all together we get that:

$$x \in L \text{ iff } \exists (C'_1, \ldots, C'_n), \text{ s.t. } \forall y, (\phi_1, \ldots, \phi_n), \ (x, (C'_1, \ldots, C'_n), (y, \phi_1, \ldots, \phi_n)) \in R_L$$

where R_L is a polynomially-bounded 3-ary relation defined using the Karp reduction f, Eq. (9.2) and the simplifying process above. Specifically, the algorithm recognizing R_L computes the formula $f(x,y)$, determines the formulas ϕ'_i and ϕ''_i (for each i), and evaluates circuits (the description of which is given) on inputs which are also given. Cleraly, this algorithm can be implemented in polynomial-time, and so it follows that $L \in \Sigma_2$ as needed. ■

Bibliographic Notes

The Polynomial-Time Hierarchy was introduced by Stockmeyer [6]. The third equivalent formulation via "alternating machines" can be found in [1].

The fact that $\mathcal{BPP}$ is in the Polynomial-time hierarchy was proven independently by Lautemann [4] and Sipser [5]. We have followed Lautemann's proof. The ideas underlying Sipser's proof found many applications in complexity theory, and will be presented in the next lecture (in the approximation procedure for $\#\mathcal{P}$). Among these applications, we mention Stockmeyer's approximation procedure for $\#\mathcal{P}$ (cf., citel9:S83), the reduction of SAT to uniqueSAT (cf. [8] and next lecture), and the equivalence between public-coin interactive proofs and general interactive proofs (cf. [2] and Lecture 11).

The fact that $\mathcal{NP} \subseteq \mathcal{P}/\text{poly}$ implies a collapse of the Polynomial-time hierarchy was proven by Karp and Lipton [3].

1. A.K. Chandra, D.C. Kozen and L.J. Stockmeyer. Alternation. *JACM*, Vol. 28, pages 114–133, 1981.

2. S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 73–90, 1989. Extended abstract in *18th STOC*, pages 59–68, 1986.

3. R.M. Karp and R.J. Lipton. "Some connections between nonuniform and uniform complexity classes", in *12th STOC*, pages 302-309, 1980.

4. C. Lautemann. BPP and the Polynomial Hierarchy. *IPL*, 17, pages 215–217, 1983.

5. M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th STOC*, pages 330–335, 1983.

6. L.J. Stockmeyer. The Polynomial-Time Hierarchy. *Theoretical Computer Science*, Vol. 3, pages 1–22, 1977.

7. L. Stockmeyer. The Complexity of Approximate Counting. In *15th STOC*, pages 118–126, 1983.

8. L.G. Valiant and V.V. Vazirani. NP Is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47 (1), pages 85–93, 1986.

Appendix: Proof of Proposition 9.2.3

Recall that our aim is to prove the claim:

For every $k \geq 1$, if $\Pi_k = \Sigma_k$ then $PH = \Sigma_k$.

Proof: For an arbitrary fixed k, we will show by induction on i that $\forall i \geq k, \Sigma_i = \Sigma_k$:

1. Base of induction: when $i = k$, there is nothing to show.

2. Induction step: by the inductive hypothesis it follows that $\Sigma_i = \Sigma_k$, so what remains to be shown is that $\mathcal{NP}^{\Sigma_k} = \Sigma_k$. Containment in one direction is obvious so we focus on proving that $\mathcal{NP}^{\Sigma_k} \subseteq \Sigma_k$.

   Let $L \in \mathcal{NP}^{\Sigma_k}$, then there exist a non-deterministic, polynomial-time machine M, and an oracle $A \in \Sigma_k$, such that $L = L(M^A)$. Since $\Pi_k = \Sigma_k$ it follows that $\overline{A} \in \Sigma_k$ too. Therefore, there exist relations R_A and $R_{\overline{A}}$ ($k+1$-ary relations, polynomially bounded, and polynomial time recognizable, for A and $\overline{A}$ respectively) such that :

   - $q \in A$ iff $\exists w_1, \forall w_2, \ldots, Q_k w_k$ s.t. $(q, w_1, \ldots w_k) \in R_A$.

- $q \in \overline{A}$ iff $\exists w_1, \forall w_2, \ldots, Q_k w_k$ s.t. $(q, w_1, \ldots w_k) \in R_{\overline{A}}$.

Using those relations, and the definition of $\mathcal{NP}^{\Sigma_k}$ we get:
$x \in L$ iff $\exists y, q_1, a_1, \ldots, q_t, a_t$ s.t. for all $1 \leq j \leq t$:

- $a_j = 1 \iff q_j \in A \iff \exists w_1^{(j,1)}, \forall w_2^{(j,1)}, \ldots, Q_k w_k^{(j,1)}$ s.t. $(q_j, w_1^{(j,1)}, \ldots w_k^{(j,1)}) \in R_A$.
- $a_j = 0 \iff q_j \in \overline{A} \iff \exists w_1^{(j,0)}, \forall w_2^{(j,0)}, \ldots, Q_k w_k^{(j,0)}$ s.t. $(q_j, w_1^{(j,0)}, \ldots w_k^{(j,0)}) \in R_{\overline{A}}$.

Define:

- w_1 is the concatenation of: $y, q_1, a_1, \ldots, q_t, a_t, w_1^{(1,0)}, \ldots, w_1^{(t,0)}, w_1^{(1,1)}, \ldots, w_1^{(t,1)}$.

  .

  .
- w_k is the concatenation of: $w_k^{(1,0)}, \ldots, w_k^{(t,0)}, w_k^{(1,1)}, \ldots, w_k^{(t,1)}$
- R_L is a $k+1$-ary relation such that:
  $(x, w_1, \ldots, w_k) \in R_L$ iff for all $1 \leq j \leq t$:
  - $a_j = 1 \implies (q_j, w_1^{(j,1)}, \ldots w_k^{(j,1)}) \in R_A$.
  - $a_j = 0 \implies (q_j, w_1^{(j,0)}, \ldots w_k^{(j,0)}) \in R_{\overline{A}}$.

  Since M is a polynomial machine, then t is polynomial in the length of x. R_A and $R_{\overline{A}}$ are polynomial time recognizable, and polynomially bounded relations.
  Therefore R_L is also so.

All together we get that there exists a polynomially bounded, and polynomial time recognizable relation R_L such that :

$$x \in L \text{ iff } \exists w_1, \forall w_2, \ldots, Q_k w_k \text{ s.t. } (x, w_1, \ldots, w_k) \in R_L$$

By the definition of Σ_k, $L \in \Sigma_k$.

■

Lecture 10

The counting class $\#\mathcal{P}$

Notes taken by Oded Lachish, Yoav Rodeh and Yael Tauman

Summary: Up to this point in the course, we've focused on decision problems where the questions are YES/NO questions. Now we are interested in counting problems. In $\mathcal{NP}$ an element was in the language if it had a short checkable witness. In $\#\mathcal{P}$ we wish to count the number of witnesses to a specific element. We first define the complexity class $\#\mathcal{P}$, and classify it with respect to other complexity classes. We then prove the existence of $\#\mathcal{P}$-complete problems, and mention some natural ones. Then we try to study the relation between $\#\mathcal{P}$ and $\mathcal{NP}$ more exactly, by showing we can probabilistically approximate $\#\mathcal{P}$ using an oracle in $\mathcal{NP}$. Finally, we refine this result by restricting the oracle to a weak form of SAT (called $uniqueSAT$).

10.1 Defining $\#\mathcal{P}$

We used the notion of an $\mathcal{NP}$-relation when defining $\mathcal{NP}$. Recall:

Definition 10.1 *($\mathcal{NP}$ relation) : An $\mathcal{NP}$ relation is a relation $R \subseteq \Sigma^\star \times \Sigma^\star$ such that:*

- *R is polynomial time decidable.*
- *There exists a polynomial $p(\cdot)$ such that for every $(x, y) \in R$, it holds that $|y| \leq p(|x|)$.*

Given an $\mathcal{NP}$-relation R we defined:

Definition 10.2 $L_R \stackrel{\text{def}}{=} \{x \in \Sigma^\star \mid \exists y \ s.t. \ (x, y) \in R\}$

We regard the y's that satisfy $(x, y) \in R$ as witnesses to the membership of x in the language L_R. The decision problem associated with R, is the question: Does there exist a witness to a given x? This is our definition of the class $\mathcal{NP}$. Another natural question we can ask is: How many witnesses are there for a given x? This is exactly the question we capture by defining the complexity class $\#\mathcal{P}$. We first define:

Definition 10.3 *For every binary relation $R \subseteq \Sigma^\star \times \Sigma^\star$, the counting function $f_R : \Sigma^\star \to \mathsf{N}$, is defined by:*

$$f_R(x) \stackrel{\text{def}}{=} |\{y \mid (x, y) \in R\}|$$

The function f_R captures our notion of counting witnesses in the most natural way. So, we define $\#\mathcal{P}$ as a class of functions. Specifically, functions that count the number of witnesses in an $\mathcal{NP}$-relation.

Definition 10.4 $\#\mathcal{P} = \{f_R : R \textit{ is an } \mathcal{NP} \textit{ relation}\}$

We encounter some problems when trying to relate $\#\mathcal{P}$ to other complexity classes, since it is a class of functions while all classes we discussed so far are classes of languages. To solve this, we are forced to give a less natural definition of $\#\mathcal{P}$, using languages. For each $\mathcal{NP}$-relation R, we associate a language $\#_R$. How do we define $\#_R$? Our first attempt would be:

Definition 10.5 (Counting language — first attempt) *:* $\#_R = \{(x,k) : |\{y : (x,y) \in R\}| = k\}$

First observe that given an oracle to f_R, it is easy to decide $\#_R$. This is a nice property of $\#_R$, since we would like it to closely represent our other formalism using functions. For the same reason we also want the other direction: Given an oracle to $\#_R$, we would like to be able to calculate f_R efficiently (in polynomial time). This is not as trivial as the other direction, and in fact, is not even necessarily true. So instead of tackling this problem, we alter our definition:

Definition 10.6 (Counting language — actual definition) *:* $\#_R = \{(x,k) : |\{y : (x,y) \in R\}| \geq k\}$. *In other words,* $(x,k) \in \#_R$ *iff* $k \leq f_R(x)$.

We choose the last definition, because now we can prove the following:

Proposition 10.1.1 *For each $\mathcal{NP}$-relation R :*

1. *$\#_R$ is Cook reducible to f_R*
2. *f_R is Cook reducible to $\#_R$*

We denote the fact that problem P Cook reduces to problem Q by $P \; \alpha_c \; Q$.
Proof:

1. *($\#_R$ is Cook reducible to f_R)* : Given (x,k), we want to decide whether $(x,k) \in \#_R$. We use our oracle for f_R, by calling it with parameter x. As an answer we get : $l = |\{y \; : \; (x,y) \in R\}|$. If $l \geq k$ then we accept, otherwise reject.

2. *(f_R is Cook reducible to $\#_R$)* : Given x, we want to find $f_R(x) = |\{y : (x,y) \in R\}|$ using our oracle. We know $f_R(x)$ is in the range $\{0, \ldots, 2^{|p(x)|}\}$, where $p(\cdot)$ is the polynomial bounding the size of the witnesses in the definition of an NP-relation. The oracle given is exactly what we need to implement binary search.

   $BINARY(x, Lower, Upper)$:

   - if $(Lower = Upper)$ output $Lower$.
   - $Middle = \frac{Lower+Upper}{2}$
   - if $(x, Middle) \in \#_R$ output $BINARY(x, Middle, Upper)$
   - else output $BINARY(x, Lower, Middle)$

   Where the branching in the third line is because if $(x, Middle) \in \#_R$, then $f_R(x) \geq Middle$, so we need only search for the result in the range $[Middle, Upper]$. A symmetric argument explains the *else* clause.

   The output is: $f_R(x) = BINARY(x, 0, 2^{p(|x|)})$. Binary search in general, runs in time logarithmic in interval it searches in. In our case : $O(\log(2^{p(|x|)})) = O(p(|x|))$. We conclude, that the algorithm runs in polynomial time in $|x|$.

■

Notice that we could have changed our definition of $\#_R$, to be:

$$\#_R = \{(x,k) : |\{y : (x,y) \in R)\}| \leq k\}$$

The proposition would still hold. We could have also changed it to a strict inequality, and gotten the same result.

¿From now on we will use the more natural definition of $\#\mathcal{P}$: as a class of functions. This doesn't really matter, since we showed that in terms of cook-reducibility, the two definitions are equivalent.

It seems that the counting problem related to a relation R should be harder than the corresponding decision problem. It is unknown whether it is strictly harder, but it is certainly not weaker. That is,

Proposition 10.1.2 *For every $\mathcal{NP}$-relation R, the corresponding language L_R Cook reduces to f_R.*

Proof: Given $x \in \Sigma^\star$, use the oracle to calculate $f_R(x)$. Now, $x \in L_R$ if and only if $f_R(x) \geq 1$. ■

Corollary 10.7 *$\mathcal{NP}$ Cook reduces to $\#\mathcal{P}$*

On the other hand we can bound the complexity of $\#\mathcal{P}$ from above:

Claim 10.1.3 *$\#\mathcal{P}$ Cook reduces to $\mathcal{P}$SPACE*

Proof: Given x, we want to calculate $f_R(x)$ using polynomial space. Let $p(\cdot)$ be the polynomial bounding the length of the witnesses of R. We run over all possible witnesses of length $\leq p(|x|)$. For each one, we check in polynomial time whether it is a witness for x, and sum the number of witnesses. All this can be done in space $O(p(|x|) + q(|x|))$, where $q(\cdot)$ is the polynomial bounding the running time (and therefore space) of the witness checking algorithm. Such a polynomial exists since R is an $\mathcal{NP}$-relation. ■

10.2 Completeness in #P

When one talks about complexity classes, proving the existence, and finding complete problems in the complexity class, is of great importance. It helps reason about the whole class using only one specific problem. Therefore, we are looking for an $\mathcal{NP}$-relation R, s.t. for every other $\mathcal{NP}$-relation Q, there is a Cook reduction from f_Q to f_R. Formally:

Definition 10.8 *($\#\mathcal{P}$-complete) : f is $\#\mathcal{P}$-complete if*

1. *f is in $\#\mathcal{P}$.*
2. *For every g in $\#\mathcal{P}$, g Cook reduces to f.*

With Occam's Razor in mind, we'll try to find a complete problem, such that all other problems are reducible to it using a very simple form of reduction. Note that by restricting the kind of reductions we allow, we may rule out candidates for $\#\mathcal{P}$-complete problems. We take a restricted form of a Levin reduction ϕ from f_Q to f_R:

$$\forall x \in \Sigma^\star : f_Q(x) = f_R(\phi(x))$$

By allowing only this kind of reduction, we can find out several things about our candidates for $\#\mathcal{P}$-complete problems. For example:

$$f_Q(x) \geq 1 \Leftrightarrow f_R(\phi(x)) \geq 1$$

In other words :

$$x \in L_Q \Leftrightarrow \phi(x) \in L_R$$

Which means that ϕ is a Karp reduction from L_Q to L_R. This implies that the decision problem related to R must be $\mathcal{NP}$-complete. Moreover, we require that the reduction preserves the number of witnesses for every input x. We capture this notion in the following definition:

Definition 10.9 *(*Parsimonious*) : A reduction $\phi : \Sigma^\star \rightarrow \Sigma^\star$, is* Parsimonious *w.r.t. $\mathcal{NP}$-relations Q and R if for every x : $|\{y : (x, y) \in Q\}| = |\{y : (\phi(x), y) \in R\}|$.*

Corollary 10.10 *if R is an NP-relation, and for every $\mathcal{NP}$-relation Q there exists $\phi_Q : \Sigma^\star \rightarrow \Sigma^\star$ s.t. ϕ_Q is parsimonious w.r.t. Q and R then f_R is $\#\mathcal{P}$-complete.*

As we've said, a parsimonious reduction from f_Q to f_R must be a Karp reduction from L_Q to L_R. Therefore, we'll try to prove that the Karp reductions we used to prove SAT is $\mathcal{NP}$-complete, are also parsimonious, and thereby $\#SAT$ is $\#\mathcal{P}$-complete.

Definition 10.11

$$R_{SAT} = \left\{ (\psi, \tau) \;\middle|\; \begin{array}{l} \psi \textit{ is a boolean formula on variables } V(\psi) \\ \tau \textit{ is a truth assignment for } V(\psi) : \psi(\tau) = 1 \end{array} \right\}$$

We have proved $SAT \stackrel{\text{def}}{=} L_{R_{SAT}}$ is $\mathcal{NP}$-complete by a series of Karp reductions. All we need to show is that each step is in fact a parsimonious reduction.

Theorem 10.12 *$\#SAT \stackrel{\text{def}}{=} f_{R_{SAT}}$ is $\#\mathcal{P}$-complete.*

Proof: (outline)

1. Obviously $\#SAT$ is in $\#\mathcal{P}$, since R_{SAT} is an NP-relation.

2.
   - The reduction from a generic $\mathcal{NP}$-relation R to Bounded-Halting, is parsimonious because the correspondence between the witnesses is not only one-to-one, it is in fact the identity.
   - The reduction from Bounded-Halting to Circuit-SAT consists of creating for each time unit a set of variables that can describe each possible configuration uniquely. Since a successful run is a specific list of configurations, and corresponds to one witness of Bounded-Halting, we get the same witness translated into one unique representation in binary variables.
   - In the reduction from Circuit-SAT to SAT we add extra variables for each internal gate in the circuit. Each satisfying assignment to the original circuit uniquely determines all the values in the internal gates, and therefore gives us exactly one satisfying assignment to the formula.

■

Notice that we actually proved that the counting problems associated with Bounded-Halting, Circuit-SAT, and SAT are $\#\mathcal{P}$-complete. Not only did we prove $\#SAT$ to be $\#\mathcal{P}$-complete, we also showed that for every f in $\#\mathcal{P}$, there exists a parsimonious reduction from f to $\#SAT$.

The reader might have gotten the impression that every NP-relation R, such that f_R is $\#\mathcal{P}$-complete implies L_R is $\mathcal{NP}$-complete. But the following theorem shows the contrary:

Theorem 10.13 *There exists an $\mathcal{NP}$-relation R s.t. f_R is $\#\mathcal{P}$-complete, and L_R is polynomial time decidable.*

Notice that such a $\#\mathcal{P}$-complete function, does not have the property that we showed $\#SAT$ has: Not all other functions in $\#\mathcal{P}$ have a parsimonious reduction to it. In fact it cannot be that every $\#\mathcal{P}$ problem has a Karp reduction to f_R, since otherwise L_R would be $\mathcal{NP}$-complete.

The idea in the proof is to modify a hard to calculate relation by adding easy to recognize witnesses to every input, so that the question of existence of a witness becomes trivial, yet the counting problem remains just as hard. Clearly, the $\#\mathcal{P}$-hardness will have to be proven by a non-parsimonious reduction (actually even a non-Karp reduction).

Proof: We define :

$$R'_{SAT} = \left\{ (\phi, (\tau, \sigma)) \;\middle|\; \begin{array}{c} (\phi(\tau) = 1) \wedge (\sigma = 1) \\ \vee \\ \sigma = 0 \end{array} \right\}$$

Obviously $L_{R'_{SAT}} = \Sigma^\star$, so it is in $\mathcal{P}$. But $f_{R'_{SAT}}$ is $\#\mathcal{P}$-complete, since for every ϕ : ϕ's witnesses in R'_{SAT} are:

$$\{(\tau, 1) \;:\; \phi(\tau) = 1\} \cup \{(\tau, 0)\}$$

Which means:

$$\#SAT(\phi) + 2^{|\text{Variables}(\phi)|} = f_{R'_{SAT}}(\phi)$$

So given an oracle to $f_{R'_{SAT}}$ we can easily calculate $\#SAT$, meaning that $f_{R'_{SAT}}$ is $\#\mathcal{P}$-complete. ■

We proved the above theorem by constructing a somewhat unnatural NP-relation. We will now find a more natural problem that gives the same result (i.e., which is also $\#\mathcal{P}$-complete).

Definition 10.14 *(*Bipartite Graph*) : $G = (V_1 \cup V_2, E)$ is a* Bipartite Graph *if*

- $V_1 \cap V_2 = \emptyset$
- $E \subseteq V_1 \times V_2$

Definition 10.15 *(*Perfect Matching*) : Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. A* Perfect Matching *is a set of edges $M \subseteq E$, that satisfies:*

1. *every v_1 in V_1 appears in exactly one edge of M.*
2. *every v_2 in V_2 appears in exactly one edge of M.*

Definition 10.16 *(*Perfect Matching — equivalent definition*) : Let $G = (V_1 \cup V_2, E)$ be a bipartite graph. A* Perfect Matching *is a one-to-one and onto function $f : V_1 \to V_2$ s.t. for every v in V_1, $(v, f(v)) \in E$.*

Proof: *(equivalence of definitions) :*

- Assume we have a subset of edges $M \subseteq E$ that satisfies the first definition. Define a function $f : V_1 \to V2$:
$$f(v_1) = v_2 \iff (v_1, v_2) \in M$$
f is well defined, because each v_1 in V_1 appears in exactly one edge of M. It is one-to-one and onto because each v_2 in V_2 appears in exactly one edge of M. Since $M \subseteq E$, f satisfies the condition that for all v_1 in V_1 : $(v_1, f(v_1))$ is in E.

- Assume we have a one-to-one and onto function $f : V_1 \to V_2$ that satisfies the above condition. We construct a set $M \subseteq E$:
$$M = \{(v_1, f(v_1)) : v_1 \in V_1\}$$
$M \subseteq E$ because for every v_1 in V_1 we know that $(v_1, f(v_1))$ is in E. The two conditions are also satisfied:
  1. Since f is a function, every v_1 in V_1 appears in exactly one edge of M.
  2. Since f is one-to-one and onto, every v_2 in V_2 appears in exactly one edge of M.

■

Definition 10.17 $R_{PM} = \{(G, f) : G$ *is a bipartite graph and* f *is a perfect matching of* $G\}$

Fact 10.2.1 $L_{R_{PM}}$ *is polynomial time decidable.*

The idea of the algorithm is to reduce the problem to a network-flow problem which is known to have a polynomial time algorithm. Given a bipartite graph $G = (V_1 \cup V_2, E)$, we construct a directed graph $G' = (V_1 \cup V_2 \cup \{s, t\}, E')$, so that:
$$E' = E \cup \{(s, v_1) : v_1 \in V_1\} \cup \{(v_2, t) : v_2 \in V_2\}$$
where E is viewed as directed edges from V_1 to V_2. What we did, is add a source s and connect it to one side of the graph, and a sink t connected to the other side. We transform it to a flow problem by setting a weight of 1 to every edge in the graph. There is a one to one correspondence between partial matchings and integer flows in the graph: Edges in the matching correspond to edges in E having a flow of 1. Therefore, there exists a perfect matching iff there is a flow of size $|V_1| = |V_2|$.

Theorem 10.18 $f_{R_{PM}}$ *is* $\#\mathcal{P}$*-complete.*

This result is proved by showing that the problem of computing the permanent of a $\{0, 1\}$ matrix is $\#\mathcal{P}$-complete. We will show the reduction from counting the number of perfect matchings to computing the permanent of such matrices. In fact, the two problems are computationally equivalent.

Definition 10.19 *(*Permanent*) : The* permanent *of an* $n \times n$ *matrix* $A = (a_{i,j})_{i,j=1}^n$ *is:*
$$Perm(A) = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)}$$
Where $S_n = \{\pi : \pi$ *is a permutation of* $\{1, \ldots, n\}\}$.

Note that the definition of the permanent of a matrix closely resembles that of the determinant of a matrix. In the definition of determinant, we have the same sum and product, except that each element in the sum is multiplied by the *sign* $\in \{-1, 1\}$ of the permutation π. Yet, computing the determinant is in $\mathcal{P}$, while computing the permanent is $\#\mathcal{P}$-complete, and therefore is believed not to be in $\mathcal{P}$. The main result in this section is the (unproven here) theorem:

Theorem 10.20 *Perm is $\#\mathcal{P}$-complete.*

To show the equivalence of computing $f_{R_{PM}}$ and $Perm$, we use:

Definition 10.21 *(*Bipartite Adjacency Matrix*) : Given a bipartite graph $G = (V_1 \cup V_2, E)$, where $V_1 = \{1, \ldots, n\}$, and $V_2 = \{1, \ldots, m\}$, we define the* Bipartite Adjacency Matrix *of the graph G, as an $n \times m$ matrix $B(G)$, where :*

$$B(G)_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$$

Proposition 10.2.2 *Given a bipartite graph $G = (V_1 \cup V_2, E)$ where $|V_1| = |V_2|$,*

$$f_{R_{PM}}(G) = Perm(B(G))$$

Proof:

$$\begin{aligned} Perm(B(G)) &= |\{\pi \in S_n : \textstyle\prod_{i=1}^{n} b_{i,\pi(i)} = 1\}| \\ &= |\{\pi \in S_n : \forall i \in \{1, \ldots, n\}, b_{i,\pi(i)} = 1\}| \\ &= |\{\pi \in S_n : \forall i \in \{1, \ldots, n\}, (i, \pi(i)) \in E\}| \\ &= |\{\pi \in S_n : \pi \text{ is a perfect matching in } G\}| \\ &= f_{R_{PM}}(G) \end{aligned}$$

■

We just showed that the problem of counting the number of perfect matchings in a bipartite graph Cook reduces to the problem of calculating the permanent of a $\{0, 1\}$ matrix. Notice that the other direction is also true by the same proof: Given a $\{0, 1\}$ matrix, create the bipartite graph that corresponds to it.

Now we will show another graph counting problem, that is equivalent to both of these:

Definition 10.22 *(*Cycle Cover*) : A* Cycle Cover *of a directed graph G, is a set of vertex disjoint simple cycles that cover all the vertices of G. More formally: $C \subseteq E$ is a* cycle cover *of G if for every connected component V_1 of $G' = (V, C)$, there is an ordering $V_1 = \{v_0, \ldots v_{d-1}\}$ s.t. $(v_i, v_j) \in C \Leftrightarrow j = i + 1 (mod\ d)$*

Notice that there is no problem with connected components of size 1, because we allow self loops.

Definition 10.23 *$\#Cycle(G)$ = number of cycle covers of G.*

Definition 10.24 *(*Adjacency Matrix*) : The* Adjacency Matrix *of a directed graph $G = (\{1, \ldots, n\}, E)$ is an $n \times n$ matrix $A(G)$:*

$$A(G)_{i,j} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$$

Proposition 10.2.3 *For every directed graph G, $Perm(A(G)) = \#Cycle(G)$*

In proving this proposition we use the following:

Claim 10.2.4 *C is a cycle cover of G if and only if every $v \in V$ has an out-degree and in-degree of 1 in $G' = (V, C)$.*

Proof: (Claim)

- ($\Longrightarrow$) Every vertex appears in exactly one cycle of C, because the cycles are disjoint. Also, since the cycles are simple, every vertex has an in-degree and out-degree of 1.
- ($\Longleftarrow$) For every connected component V_0 of G', take a vertex $v_0 \in V_0$, and create the directed path : $v_0, v_1, \ldots,$, where for every i, $(v_i, v_{i+1}) \in C$. Since the out-degree of every vertex is 1 in G', this path is uniquely determined, and:

  1. There must exist a vertex v_i that appears twice : $v_i = v_j$. Because V is finite.
  2. We claim that the least such i is 0. Otherwise, the in-degree of v_i is greater than 1.
  3. All the vertices of V_0 appear in our path because it is a connected component of G'.

  Thus each V_0 induces a directed cycle, and so G' is a collection of disjoint directed cycles which cover all V.

■

Proof: (Proposition) We'll define:

$$\Omega \stackrel{\text{def}}{=} \{\pi \in S_n : \forall i \in \{1, \ldots, n\}, (i, \pi(i)) \in E\}$$

It is easy to see that $Perm(A(G)) = |\Omega|$. Every $\pi \in \Omega$ defines

$$C_\pi = \{(i, \pi(i)) : i \in \{1, \ldots, n\}\} \subseteq E$$

and since π is a 1-1 and onto $\{1, \ldots, n\}$, the out-degree and in-degree of each vertex in C_π is 1. So C_π is a cycle cover of G. On the other hand, every cycle cover $C \subseteq G$ defines a mapping: $\pi_C(i) = j$ s.t. $(i, j) \in C$, and by the above claim, this is a permutation. ■

10.3 How close is $\#\mathcal{P}$ to $\mathcal{NP}$?

The main purpose of this lecture, is to study the class $\#\mathcal{P}$, and classify it as best as we can among other complexity classes we've studied. We've seen some examples of $\#\mathcal{P}$ complete problems. We also gave upper and lower complexity bounds on $\#\mathcal{P}$:

$$\mathcal{NP} \;\alpha_c\; \#\mathcal{P} \;\alpha_c\; \mathcal{P}\text{SPACE}$$

We will now try to refine these bounds by showing that $\#\mathcal{P}$ is not as far from $\mathcal{NP}$ as one might suspect. In fact, a counting problem in $\#\mathcal{P}$ can be probabilistically approximated in polynomial time using an NP oracle.

10.3.1 Various Levels of Approximation

We will start by introducing the notion of a range problem. A range problem is a relaxation of the problem of calculating a function. Instead of requiring one value for each input, we allow a full range of answers for each input.

Definition 10.25 *(*Range Problem*) : A* Range Problem Π *is defined by two functions* $\Pi = (\Pi_l, \Pi_u)$. $\Pi_l, \Pi_u : \Sigma^\star \to \mathsf{N}$. *s.t. on input* $x \in \Sigma^\star$, *the problem is to find* $t \in (\Pi_l(x), \Pi_u(x))$, *or in other words, return an integer* t, *s.t.* $\Pi_l(x) < t < \Pi_u(x)$.

Note that there is no restriction on the functions Π_l and Π_u, they can even be non-recursive. Since we are going to use range problems to denote an approximation to a function, we define a specific kind of range problems that are based on a function:

Definition 10.26 *(*Strong Range*) : For* $f : \Sigma^\star \to \mathsf{N}$, *and a polynomial* $p(\cdot)$, *we define the range problem* $StrongRange_p(f) = (l, u)$ *where:*

$$\begin{array}{l} l(x) = f(x) \cdot (1 - \frac{1}{p(|x|)}) \\ u(x) = f(x) \cdot (1 + \frac{1}{p(|x|)}) \end{array}$$

Strong range captures our notion of a good approximation. We will proceed in a series of reductions that will eventually give us the desired result. The first result we prove, is that it is enough to strongly approximate $\#SAT$.

Proposition 10.3.1 *If we can approximate* $\#SAT$ *strongly we can just as strongly approximate any* f *in* $\#\mathcal{P}$. *In other words : For every* f *in* $\#\mathcal{P}$, *and every polynomial* $p(\cdot)$,

$$StrongRange_p(f) \ \alpha_c \ StrongRange_p(\#SAT)$$

Proof: As we've seen, for every f in $\#\mathcal{P}$, there is a parsimonious reduction ϕ_f w.r.t. f and $\#SAT$. Meaning, for all x : $f(x) = \#SAT(\phi_f(x))$. We may assume that $|\phi_f(x)| > |x|$, because we can always pad $\phi_f(x)$ with something that will not change the number of witnesses:

$$\phi_f(x) \wedge z_1 \wedge z_2 \wedge \ldots \wedge z_{|x|}$$

We now use our oracle to $StrongRange_p(\#SAT)$ on $\phi_f(x)$, and get a result t, that satisfies :

$$\begin{array}{rcl} t & \in & (1 \pm \frac{1}{p(|\phi(x)|)}) \cdot \#SAT(\phi_f(x)) \\ & \subseteq & (1 \pm \frac{1}{p(|x|)}) \cdot f(x) \end{array}$$

■

We now wish to define a weaker form of approximation:

Definition 10.27 *(*Constant Range*) : For* $f : \Sigma^\star \to \mathsf{N}$, *and a constant* $c > 0$, *we define the range problem* $ConstantRange_c(f) = (l, u)$ *where:*

$$\begin{array}{l} l(x) = \frac{1}{c} \cdot f(x) \\ u(x) = c \cdot f(x) \end{array}$$

We want to show that approximating $\#SAT$ up to a constant suffices to approximate $\#SAT$ strongly. We'll in fact prove a stronger result: that an even weaker form of approximation is enough to approximate $\#SAT$ strongly.

Definition 10.28 *(*Weak Range*) : For $f : \Sigma^\star \to \mathsf{N}$, and a constant $\epsilon > 0$, we define the range problem $WeakRange_\epsilon(f) = (l, u)$ where:*

$$l(x) = (\tfrac{1}{2})^{|x|^{1-\epsilon}} \cdot f(x)$$
$$u(x) = 2^{|x|^{1-\epsilon}} \cdot f(x)$$

It is quite clear that $ConstantRange$ is a stronger form of approximation than $WeakRange$:

Claim 10.3.2 *For every $0 < \epsilon < 1$ and $c > 0$:*

$$WeakRange_\epsilon(\#SAT) \ \alpha_c \ ConstantRange_c(\#SAT)$$

Proof: Simply because for large enough n:

$$(\frac{1}{2}, 2)^{n^{1-\epsilon}} \subseteq (\frac{1}{c}, c)$$

where we use $(\frac{1}{2}, 2)^{n^{1-\epsilon}}$ to denote the range $((\frac{1}{2})^{n^{1-\epsilon}}, 2^{n^{1-\epsilon}})$. ■

Now we prove the main result:

Proposition 10.3.3 *For every polynomial $p(\cdot)$, and constant $0 < \epsilon < 1$,*

$$StrongRange_p(\#SAT) \ \alpha_c \ WeakRange_\epsilon(\#SAT)$$

Proof: Given ϕ, a boolean formula on variables $\vec{x}$. Define a polynomial $q(n) \geq (2n \cdot p(n))^{\frac{1}{\epsilon}}$, and build ϕ' :

$$\phi' = \bigwedge_{i=1}^{q(|\phi|)} \phi(\vec{x^i})$$

Where each $\vec{x^i}$ is a distinct copy of the variables $\vec{x}$. Obviously $\#SAT(\phi') = (\#SAT(\phi))^{q(|\phi|)}$. Notice $|\phi'| \leq 2|\phi| \cdot q(|\phi|)$. Now, assuming we have an oracle to $WeakRange_\epsilon(\#SAT)$, we call it on ϕ' to get:

$$\begin{array}{rcl} t & \in & (\frac{1}{2}, 2)^{|\phi'|^{1-\epsilon}} \cdot \#SAT(\phi') \\ & \subseteq & (\frac{1}{2}, 2)^{2|\phi| \cdot q(|\phi|)^{1-\epsilon}} \cdot (\#SAT(\phi))^{q(|\phi|)} \end{array}$$

Our result would be $s = t^{\frac{1}{q(|\phi|)}}$. And we have :

$$\begin{array}{rcl} s & \in & (\frac{1}{2}, 2)^{\frac{2|\phi|}{q(|\phi|)^\epsilon}} \cdot \#SAT(\phi) \\ & \subseteq & (\frac{1}{2}, 2)^{\frac{2|\phi|}{2|\phi| p(|\phi|)}} \cdot \#SAT(\phi) \\ & = & (\frac{1}{2}, 2)^{\frac{1}{p(|\phi|)}} \cdot \#SAT(\phi) \\ & \subseteq & (1 \pm \frac{1}{p(|\phi|)}) \cdot \#SAT(\phi) \end{array}$$

where the last containment follows from :

$$\forall x \geq 1 : (1 - \frac{1}{x})^x \leq \frac{1}{2} \ \text{ and } \ 2 \leq (1 + \frac{1}{x})^x$$

■

After a small diversion into proving a stronger result than needed, we conclude that all we have to do is to find a constant $c > 0$, such that we can solve $ConstantRange_c(\#SAT)$.

We still have a hard time solving the problem directly, so we'll do yet another reduction into a relaxed form of decision problems, called promise problems. While machines that solve decision problems are required to give an exact answer for every input, promise problems are only required to do so on a predefined 'promise' set.

Definition 10.29 *(*Promise Problem*) : A* Promise Problem $\Pi = (\Pi_Y, \Pi_N)$*, where* $\Pi_Y, \Pi_N \subseteq \Sigma^\star$*, and* $\Pi_Y \cap \Pi_N = \emptyset$*, is the question: Given* $x \in Promise(\Pi) \stackrel{\text{def}}{=} \Pi_Y \cup \Pi_N$*, decide whether* $x \in \Pi_Y$*.*

Notice that if $x \notin Promise(\Pi)$, there is no requirement. Also, promise problems are a generalization of decision problems, where in decision problems $Promise(\Pi) = \Sigma^\star$, so no promise is made.

Definition 10.30 *($Gap_8\#SAT$) The promise problem $Gap_8\#SAT = (Gap_8\#SAT_Y, Gap_8\#SAT_N)$, where:*

$$\begin{aligned} Gap_8\#SAT_Y &= \{(\phi, K) : \#SAT(\phi) > 8K\} \\ Gap_8\#SAT_N &= \{(\phi, K) : \#SAT(\phi) < \tfrac{1}{8}K\} \end{aligned}$$

We now continue in our reductions. For this we choose $c = 64$, and show we can solve $ConstantRange_{64}(\#SAT)$ using an oracle to $Gap_8\#SAT$.

Proposition 10.3.4 *$ConstantRange_{64}(\#SAT)$ Cook reduces to $Gap_8\#SAT$*

Proof: We run the following algorithm on input ϕ:

- $i = 0$
- While ($Gap_8\#SAT$ answers YES on $(\phi, 8^i)$) do $i = i + 1$
- return $8^{i-\frac{1}{2}}$

Denote $\alpha = \log_8(\#SAT(\phi))$. The result $8^{k-\frac{1}{2}}$, satisfies : $\alpha - 2 < k - \frac{1}{2} < \alpha + 2$, because :

- For all $i < \alpha - 1$, $\#SAT(\phi) > 8 \cdot 8^i$, so $(\phi, 8^i) \in Gap_8\#SAT_Y$. Therefore, we are promised that in such a case the algorithm will increment such an i, and not stop. So, $k \geq \alpha - 1$ follows.
- For all $i > \alpha + 1$, $\#SAT(\phi) < \frac{1}{8} \cdot 8^i$, so $(\phi, 8^i) \in Gap_8\#SAT_N$. Meaning that the algorithm must stop at the first such i or before. The first such i that satisfies $i > \alpha + 1$ also satisfies $i \leq \alpha + 2$. Therefore $k \leq \alpha + 2$.

Now :

$$\begin{array}{ccccc} \alpha - 1 & \leq & k & \leq & \alpha + 2 \\ & & \Downarrow & & \\ \alpha - 2 & < & k - \frac{1}{2} & < & \alpha + 2 \end{array}$$

We conclude:

$$8^{k-\frac{1}{2}} \in (\frac{1}{64}, 64) \cdot \#SAT(\phi)$$

■

So far we've shown the following reductions:

$$StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ StrongRange_{\text{poly}}(\#SAT) \ \alpha_c \ WeakRange_\epsilon(\#SAT)$$
$$\alpha_c \ ConstantRange_{64}(\#SAT) \ \alpha_c \ Gap_8\#SAT$$

Since Cook reductions are transitive, we get :

$$StrongRange_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT$$

We will show how to solve $Gap_8\#SAT$ using an oracle to SAT, but with a small probability of error. So we will show, that in general, if we can solve a problem P using an oracle to a promise problem Q, then if we have an oracle to Q that makes little mistakes, we can solve P with high probability.

Comment : *(Amplification)* : For every promise problem P, and machine M that satisfies:

$$\text{for every } x \in Promise(P) : Prob[M(x) = P(x)] > \frac{2}{3}$$

If on input x that is in $Promise(P)$, we run M on x, $O(n)$ times, then the majority of the results will equal $P(x)$ with probability greater than $1 - 2^{-n}$.

This we proved when we talked about $\mathcal{BPP}$, and the proof stays exactly the same, using Chernoff's bound. Note that we do not care if machine M has an oracle or not, and if so how this oracle operates, as long as different runs of M are independent.

Proposition 10.3.5 *Given a problem P and a promise problem Q, such that P Cook reduces to Q, if we have a probabilistic machine Q' that satisfies:*

$$\textit{for every } x \in Promise(Q) : Prob[Q'(x) = Q(x)] > \frac{2}{3}$$

then for every polynomial $p(\cdot)$, we have a probabilistic polynomial time machine M that uses an oracle to Q', and satisfies:

$$Prob[M^{Q'}(y) \textit{ is a solution of } P \textit{ on input } y] > 1 - 2^{-p(|y|)}$$

Proof: We start by noticing that since the reduction from P to Q is polynomial, there exists a polynomial $q(\cdot)$, such that the oracle Q is called less than $q(|y|)$ times. Since we use Q' and not Q as an oracle, we have a probability of error. If each one of these calls had a probability of error less than : $\frac{1}{q(|y|)} \cdot 2^{-p(|y|)}$, then by using the union bound we would get that the probability that at least one of the oracle calls was incorrect is less than $2^{-p(|y|)}$. The probability of M being correct, is at least the probability that all oracle calls are correct, therefore in this case it is greater than $1 - 2^{-p(|y|)}$.

Using the comment about amplification, we can amplify the probability of success of each oracle call to $1 - \frac{1}{q(|y|)} 2^{-p(|y|)}$, by calling it $O(p(|y|) \cdot \log q(|y|))$ number of times, which is polynomial in the size of the input. ■

In conclusion, all we have to do is show that we can solve $Gap_8\#SAT$ with a probability of error $< \frac{1}{3}$. Then, we showed that we can find a solution to $\#SAT$, that is very close to the real solution ($StrongRange_p(\#SAT)$), with a very high probability of success.

10.3.2 Probabilistic Cook Reduction

In the next sections, we extensively use the notion of probabilistic reduction. Therefore, we'll define it formally, and prove some of it's properties.

Definition 10.31 *(*Probabilistic Cook Reduction*) : Given promise problems P and Q, we say that there is a* Probabilistic Cook Reduction *from P to Q denoted $P \;\alpha_R\; Q$, if there is a probabilistic polynomial time oracle machine M that uses Q as an oracle, and satisfies:*

$$\textit{for every } x \in Promise(P) : Prob[M^Q(x) = P(x)] > \frac{2}{3}$$

where $M^Q(x)$ denotes then operation of machine M on input x when given oracle access to Q. Whenever a query to Q satisfies the promise of Q, the answer is correct, but when the query violates the promise the answer may be arbitrary.

Notice that in the definition, the oracle has no probability of error. We now show that this restriction does not matter, and we can do the same even if the oracle is implemented with bounded probability of error.

Proposition 10.3.6 *If P probabilistically Cook reduces to Q, and we have a probabilistic machine Q' that satisfies*

$$\text{for every } x \in Promise(Q) : Prob[Q'(x) = Q(x)] > \frac{2}{3}$$

then we have a probabilistic polynomial time oracle machine M that uses Q' as an oracle, and satisfies :

$$\text{for every } y \in Promise(P) : Prob[M^{Q'}(y) = P(y)] > \frac{2}{3}$$

Proof: By the definition of a probabilistic Cook reduction, we have a probabilistic polynomial time oracle machine N that satisfies:

$$\text{for every } y \in Promise(P) : Prob[N^{Q}(y) = P(y)] > \frac{3}{4}$$

Where we changed $\frac{2}{3}$ to $\frac{3}{4}$, using the comment about amplification. Machine N runs in polynomial time, therefore it calls the oracle a polynomial $p(|y|)$ number of times. We can assume Q' to be correct with a probability $> \frac{1}{9} \cdot \frac{1}{p(|y|)}$, by calling it each time instead of just once, $O(\log(p(|y|)))$ times, and taking the majority. Using the union bound, the probability that all oracle calls (to this modified Q') are correct is greater than $\frac{8}{9}$.

When all oracle calls are correct, machine N returns the correct result. Therefore with probability greater than $\frac{3}{4} \cdot \frac{8}{9} = \frac{2}{3}$ we get the correct result. ∎

We list some properties of probabilistic cook reductions:

- Deterministic Cook reduction is a special case (i.e., $P \ \alpha_c \ Q \implies P \ \alpha_R \ Q$).
- *Transitivity* : $P \ \alpha_R \ Q \ \alpha_R \ R \implies P \ \alpha_R \ R$

10.3.3 $Gap_8\#SAT$ Reduces to SAT

Our goal, is to show that we can approximate any problem in $\#\mathcal{P}$ using an oracle to SAT. So far we've reduced the problem several times, and got:

$$StrongRange_{\mathrm{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT$$

Now we'll show:

$$Gap_8\#SAT \ \alpha_R \ SAT$$

And using the above properties of probabilistic Cook reductions, this will mean that we can approximate $\#\mathcal{P}$ very closely, with an exponentially small probability of error.

Reminder: $Gap_8\#SAT$ is the promise problem on input pairs (ϕ, k), where ϕ is a boolean formula, and k is a natural number. $Gap_8\#SAT = (Gap_8\#SAT_Y, Gap_8\#SAT_N)$, where:

$$\begin{aligned} Gap_8\#SAT_Y &= \{(\phi,k) : \#SAT(\phi) > 8k\} \\ Gap_8\#SAT_N &= \{(\phi,k) : \#SAT(\phi) < \tfrac{1}{8}k\} \end{aligned}$$

How do we approach the problem? We know, that there is either a very large or a very small number of truth assignment in comparison to the input parameter k. So if we take a random $\frac{1}{k}$

fraction of the assignments, with high probability in the first case at least one of them is satisfying, and in the second, none are. Assume that we have a way of restricting our formula to a random fraction of the assignments S that satisfies : each assignment τ is in the set with probability $\frac{1}{k}$ independently of all other assignments. We set $\phi'(\tau) = \phi(\tau) \wedge (\tau \in S)$. Then we simply check satisfiability of ϕ'. First notice:

$$Prob_S[\phi' \in SAT] \;=\; 1 - Prob_S[\forall\tau \text{ s.t. } \phi(\tau) = 1 \;:\; \tau \notin S] \;=\; 1 - (\tfrac{k-1}{k})^{\#SAT(\phi)}$$

Therefore:

$$\begin{array}{llllllll} \text{If } \#SAT(\phi) > 8k \text{ then} & Prob_S[\phi' \in SAT] & > & 1 - (\frac{k-1}{k})^{8k} & \approx & 1 - \frac{1}{e^8} & > & \frac{2}{3} \\ \text{If } \#SAT(\phi) < \frac{1}{8}k \text{ then} & Prob_S[\phi' \in SAT] & < & 1 - (\frac{k-1}{k})^{\frac{1}{8}k} & \approx & 1 - \frac{1}{e^{\frac{1}{8}}} & < & \frac{1}{3} \end{array}$$

The problem is, we don't have an efficient procedure to choose such a random S. So we weaken our requirements, instead of total independence, we require only pairwise independence. Specifically, we use the following tool:

Definition 10.32 (Universal$_2$ Hashing) *: A family of functions, $H_{n,m}$, mapping $\{0,1\}^n$ to $\{0,1\}^m$ is called* Universal$_2$ *if for a uniformly selected h in $H_{n,m}$, the random variables $\{h(e)\}_{e\in\{0,1\}^n}$ are pairwise independent and uniformly distributed over $\{0,1\}^m$. That is, for every $x \neq y \in \{0,1\}^n$, and $a, b \in \{0,1\}^m$,*

$$Prob_{h\in H_{n,m}}[h(x) = a \;\&\; h(y) = b] \;=\; (2^{-m})^2$$

An efficient construction of such families is required to have algorithms for selecting and evaluating functions in the family. That is,

1. *selecting:* There exists a probabilistic polynomial-time algorithm that on input $(1^n, 1^m)$, outputs a description of a uniformly selected function in $H_{n,m}$.
2. *evaluating:* There exists a polynomial-time algorithm that on input: a description of a function $h \in H_{n,m}$ and a domain element $x \in \{0,1\}^m$ outputs the value $h(x)$.

A popular example is the family of all affine transformations from $\{0,1\}^n$ to $\{0,1\}^m$. That is, all functions of the form $h_{A,b}(x) = Ax + b$, where A is an m-by-n 0-1 matrix, b is an m-dimensional 0-1 vector, and arithmetic is modulo 2. Clearly, this family has an efficient construction. In Appendix A, we will show that this family is Universal$_2$.

Lemma 10.3.7 (Leftover Hash Lemma): *Let $H_{n,m}$ be a family of Universal$_2$ Hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, and let $\epsilon > 0$. Let $S \subseteq \{0,1\}^n$ be arbitrary provided that $|S| \geq \epsilon^{-3} \cdot 2^m$. Then:*

$$Prob_h[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \frac{|S|}{2^m}] > 1 - \epsilon$$

The proof of this lemma appears in Appendix B.

We are now ready to construct a probabilistic Cook reduction from $Gap_8\#SAT$ to SAT, using a Universal$_2$ family of functions. Specifically we will use the family of affine transformations.

Theorem 10.33 $Gap_8\#SAT \;\alpha_R\; SAT$

Proof: We construct a probabilistic polynomial time machine M which is given oracle access to SAT. On input $(\phi, 2^m)$, where ϕ has n variables, M operates as follows:

1. Select uniformly $h \in H_{n,m} = \{\text{Affine transformations from } \{0,1\}^n \text{ to } \{0,1\}^m \}$. The function h is represented by a $\{0,1\}$ matrix $A_{m\times n} = (a_{i,j})_{\substack{i=1,\dots,m \\ j=1,\dots,n}}$ and a $\{0,1\}$ vector $b = (b_i)_{i=1,\dots,m}$.

2. We construct a formula ψ_h, on variables $x_1,...,x_n,y_1,...,y_t$, so that for every $x \in \{0,1\}^n$ $h(x) = 0^m$ iff there exists an assignment to the y_i's so that $\psi_h(x_1,...,x_n,y_1,...,y_t)$ is true. Furthermore, in case $h(x) = 0^m$, there is a unique assignment to the y_i's so that $\psi_h(x_1,...,x_n,y_1,...,y_t)$ is true.

   The construction of ψ_h can be presented in two ways. In the abstract way, we just observe that applying the standard Cook-reduction to the assertion $h(x) = 0^m$, results in the desired formula. (The claimed properties have to be verified indeed.) A more concrete way is to start by the following observations

$$h(x_1,\dots x_n) = 0^m$$
$$\Updownarrow$$
$$\bigwedge_{i=1}^m \left(\sum_{j=1}^n a_{i,j}x_j \equiv b_i \pmod 2\right)$$
$$\Updownarrow$$
$$\bigwedge_{i=1}^m \left((b_i \oplus 1) \oplus \bigoplus_{j=1}^n (a_{i,j} \wedge x_j)\right)$$

   Introducing auxiliary variables, as in the construction of the standard reduction from Circuit–Satisfiability to 3SAT, we obtain the desired formula ψ_h. For example, introducing variables $y_1,...,y_n,y_{1,1},...,y_{m,n}$, the above formula is satisfied for a particular setting of the x_i's iff the following formula is satisfyiable for these x_i's (and furthermore for a unique setting of the y_i's):

$$\bigwedge_{i=1}^m (b_i \oplus 1 \oplus y_i) \ \wedge \ \bigwedge_{i=1}^m \left(y_i = \bigoplus_{j=1}^n y_{i,j}\right) \ \wedge \ \bigwedge_{i=1}^m \bigwedge_{j=1}^n (y_{i,j} = a_{i,j} \wedge x_j)$$

   So all that is left is to write a CNF for $\bigoplus_{j=1}^n y_{i,j}$, by using additional auxiliary variables. To write a CNF for $\bigoplus_{j=1}^n z_j$, we look at a binary tree of depth $\ell \stackrel{\text{def}}{=} \log_2 n$ which computes the XOR in the natrual way. We introduce an auxiliary variable for each internal node, and obtain

$$w_{0,1} \ \wedge \ \bigwedge_{i=0}^{\ell-1} \bigwedge_{j=1}^{2^i} (w_{i,j} = w_{i+1,2j-1} \oplus w_{i+1,2j}) \ \wedge \ \bigwedge_{j=1}^{n} (w_{\ell,j} = z_j)$$

3. Define $\phi' = \phi \wedge \psi_h$. Use our oracle to SAT on ϕ', and return the result.

The validity of the reduction is established via the following two claims.

Claim 1: If $(\phi, 2^m) \in Gap_8\#SAT_Y$ then $\phi' \in SAT$ with probability $> \frac{1}{2}$.

Claim 2: If $(\phi, 2^m) \in Gap_8\#SAT_N$ then $\phi' \in SAT$ with probability $< \frac{1}{8}$.

Before proving these claims, we note that the gap in the probabilities in the two cases (i.e., $(\phi, 2^m) \in Gap_8\#SAT_Y$ and $(\phi, 2^m) \in Gap_8\#SAT_N$) can be "amplified" to obtain the desired probabilities (i.e., $\phi' \in SAT$ with probability at least 2/3 in the first case and at most 1/3 in the second).

Proof Claim 1: We define $S_\phi \stackrel{\text{def}}{=} \{x : \phi(x) = 1\}$. Because $(\phi, 2^m) \in Gap_8\#SAT_Y$, we know that $|S_\phi| > 8 \cdot 2^m$ Now:

$$\begin{aligned}
Prob_h[\phi' \in SAT] &= Prob_h[\{x : \phi(x) = 1 \ \& \ h(x) = 0^m\} \neq \emptyset] \\
&= Prob_h[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset] \\
&\geq Prob_h[|\{x \in S_\phi : h(x) = 0^m\}| \in (1 \pm \tfrac{1}{2})\tfrac{|S_\phi|}{2^m}] \ > \ \tfrac{1}{2}
\end{aligned}$$

The last inequality is an application of the Leftover Hash lemma, setting $\epsilon = \frac{1}{2}$, and the claim follows. □

Proof Claim 2: As $(\phi, 2^m) \in Gap_8\#SAT_N$, we have $|S_\phi| < \frac{1}{8} \cdot 2^m$.

$$\begin{array}{rcl} Prob_h[\phi' \in SAT] & = & Prob_h[\{x \in S_\phi : h(x) = 0^m\} \neq \emptyset] \\ & = & Prob_h[(\bigcup_{x \in S_\phi} \{x : h(x) = 0^m\}) \neq \emptyset] \\ & \leq & \sum_{x \in S_\phi} Prob_h[h(x) = 0^m] \\ & < & \frac{1}{8} \cdot 2^m \cdot 2^{-m} \;\; = \;\; \frac{1}{8} \end{array}$$

The last inequality uses the union bound, and the claim follows. □

Combining the two claims (and using amplification), the theorem follows. ■

In conclusion, we have shown:

$$StrongApprox_{\text{poly}}(\#\mathcal{P}) \ \alpha_c \ Gap_8\#SAT \ \alpha_R \ SAT$$

Which is what we wanted.

10.4 Reducing to $uniqueSAT$

We've introduced the notion of *promise problems* as a means to prove that we can approximate $\#SAT$ using SAT. But promise problems are interesting by their own right, so we will try to investigate them a bit more. We've shown that using an oracle to SAT we can solve $Gap_8\#SAT$. The converse is also true, because we've shown we can approximate (deterministically) $\#SAT$ using $Gap_8\#SAT$, so all we have to do is approximate well enough, to differentiate 0 from positive results, and thus, solve SAT. We will try to refine this result, by showing that a more restricted version of $Gap_8\#SAT$ is enough to solve SAT (and even approximate $\#SAT$).

Definition 10.34 *$Gap_8\#SAT'$ is the promise problem on input pairs (ϕ, k) defined by:*

$$\begin{array}{rcl} Gap_8\#SAT'_Y & = & \{(\phi, k) : 8k < \#SAT(\phi) < 32k\} \\ Gap_8\#SAT'_N & = & \{(\phi, k) : \#SAT(\phi) < \frac{1}{8}k\} \end{array}$$

Claim 10.4.1 *SAT Cook reduces to $Gap_8\#SAT'$*

Proof: Given ϕ, first we will create formula ϕ', s.t. $\#SAT(\phi') = 15 \cdot \#SAT(\phi)$. Take 4 variables $\{x_1, x_2, x_3, x_4\}$ not appearing in ϕ. and define:

$$\begin{array}{rcl} \psi & = & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ \phi' & = & \phi \wedge \psi \end{array}$$

Observe that $\#SAT(\psi) = 15$, and since the variables of ψ do not appear in ϕ, the above equality holds. So we know that :

$$\begin{array}{lcl} \#SAT(\phi') \geq 15 & \iff & \phi \in SAT \\ \#SAT(\phi') = 0 & \iff & \phi \notin SAT \end{array}$$

For every $0 \leq i \leq |Variables(\phi')|$, we call our oracle: $Gap_8\#SAT'(\phi', 2^i)$. We claim : One of the answers is YES iff $\phi \in SAT$.

- Suppose that $\phi \notin SAT$. Then $\#SAT(\phi') = 0 < \frac{1}{8}k$ for all $k > 0$, therefore for all i, $(\phi', 2^i) \in Gap_8\#SAT'_N$, so we are promised to always get a NO answer.
- Suppose $\phi \in SAT$, so as we showed, $\#SAT(\phi') \geq 15$. Therefore, $\log_2(\#SAT(\phi')) \geq \log_2(15) > 3$. There exists an integer $i \geq 0$ s.t.

$$\begin{gathered} i < \log_2(\#SAT(\phi')) - 3 < i + 2 \\ \Downarrow \\ 2^{i+3} < \#SAT(\phi') < 2^{i+5} \\ \Downarrow \\ 8 \cdot 2^i < \#SAT(\phi') < 32 \cdot 2^i \end{gathered}$$

  And for that i, we are guaranteed to get a YES answer.

■

The reader may wonder why we imposed this extra restriction on $Gap_8\#SAT$. We want to show that we can solve SAT using weak oracles. For example $Gap_8\#SAT'$ is a weak oracle. But we wish to continue in our reductions, and our next step is:

Definition 10.35 *$fewSAT$ is the promise problem defined by:*

$$\begin{array}{rcllcl} fewSAT_Y & = & \{\phi : 1 \leq \#SAT(\phi) < 100\} & \subset & SAT \\ fewSAT_N & = & \{\phi : \#SAT(\phi) = 0\} & = & \overline{SAT} \end{array}$$

Proposition 10.4.2 *$Gap_8\#SAT'$ probabilistically Cook reduces to $fewSAT$*

Proof: We will use the same reduction we used when proving $Gap_8\#SAT \ \alpha_R \ SAT$, except we now have $Gap_8\#SAT'$. Recall, we uniformly select $h \in H_{n,m}$, and construct $\phi'(x) = \phi(x) \wedge (h(x) = 0^m)$.

We make analogous claims to the ones stated in the former proof:

- *claim 1* : If $(\phi, 2^m) \in Gap_8\#SAT'_Y$ then $\phi' \in fewSAT$ with probability $> \frac{1}{2}$.
- *claim 2* : If $(\phi, 2^m) \in Gap_8\#SAT'_N$ then $\phi' \in fewSAT$ with probability $< \frac{1}{8}$.

1. Since $(\phi, 2^m) \in Gap_8\#SAT'_Y$, we have:

$$8 \cdot 2^m < |S_\phi \stackrel{\text{def}}{=} \{x : \phi'(x) = 1\}| < 32 \cdot 2^m$$

   So now:

$$\begin{array}{rcl} Prob_h[\phi' \in fewSAT] & = & Prob_h[0 < |\{x : \phi(x) = 1 \ \& \ h(x) = 0^m\}| < 100] \\ & = & Prob_h[0 < |\{x \in S_\phi : h(x) = 0^m\}| < 100] \\ & \geq & Prob_h[(1 - \frac{1}{2}) \cdot 8 < |\{x \in S_\phi : h(x) = 0^m\}| < (1 + \frac{1}{2}) \cdot 32] \\ & \geq & Prob_h[|\{x \in S_\phi : h(x) = 0^m\}| \in (1 \pm \frac{1}{2})\frac{|S_\phi|}{2^m}] > \frac{1}{2} \end{array}$$

2. In the original proof we showed: if $(\phi, 2^m) \in Gap_8\#SAT_N$ then ϕ' is not satisfiable with probability greater than $\frac{7}{8}$. Notice :$Gap_8\#SAT_N = Gap_8\#SAT'_N$, so if $(\phi, 2^m) \in Gap_8\#SAT'_N$ then ϕ' is not satisfiable with probability greater than $\frac{7}{8}$, and in that case, it's in $fewSAT_N$, so we are guaranteed to get a NO answer.

■

As a last step in this endless crusade to understand the complexity of SAT promise problems, we will show that the weakest SAT related promise problem, is in fact as strong as the others.

Definition 10.36 *$uniqueSAT$ is the promise problem on input ϕ defined by:*

$$\begin{array}{rclcl} uniqueSAT_Y & = & \{\phi : \#SAT(\phi) = 1\} & \subset & SAT \\ uniqueSAT_N & = & \{\phi : \#SAT(\phi) = 0\} & = & \overline{SAT} \end{array}$$

Proposition 10.4.3 *$fewSAT$ Cook reduces to $uniqueSAT$*

Proof: Given a formula ϕ, we want to solve $fewSAT$. For each $1 \leq i < 100$ we construct a formula ϕ_i, s.t. :

- $\phi \notin SAT \Rightarrow \phi_i \notin SAT$.
- ϕ_i has a unique satisfying assignment if ϕ has exactly i satisfying assignments.

If we can do this, we can check all these ϕ_i's, with our oracle to $uniqueSAT$. If all of them are NO, then we return NO, otherwise we answer YES. This is correct because if $0 < k \stackrel{\text{def}}{=} \#SAT(\phi) < 100$, then ϕ_k has exactly one satisfying assignment, and therefore $uniqueSAT$ returns YES on ϕ_k. Also, if $\phi \notin SAT$, then all for all i : $\phi_i \in uniqueSAT_N$, so all the answers must be NO.

All that is left is to construct ϕ_i: We create i copies of ϕ, each on a separate set of variables:

$$\psi_i = \wedge_{j=1}^{i} \phi(x_1^j, \ldots, x_n^j)$$

First notice, that if $\phi \notin SAT$, then so is ψ_i. Now assume $\#SAT(\phi) = i$. Every satisfying assignment of ψ_i, corresponds to i satisfying assignments of ϕ. But we want to force them to be different, so we would require that the assignments are different and add this requirement to ψ_i. But then, we will have exactly $i!$ satisfying assignments to the new ψ_i. To solve this, instead of just requiring that they are different, we will impose a lexicographical ordering of the solutions, which will fix one satisfying assignment from the $i!$ possible.

$$\phi_i = \psi_i \wedge \bigwedge_{j=1}^{i-1} (\vec{x^j} <_{lex} \vec{x^{j+1}})$$

■

Just for the heck of it, we'll list all the reductions in order:

$$\begin{array}{c} StrongRange_{\text{poly}}(\#\mathcal{P})\ \alpha_c \\ StrongRange_{\text{poly}}(\#SAT)\ \alpha_c \\ WeakRange_{\epsilon}(\#SAT)\ \alpha_c \\ ConstantRange_{64}(\#SAT)\ \alpha_c \\ Gap_8\#SAT\ \alpha_R \\ SAT\ \alpha_c \\ Gap_8\#SAT'\ \alpha_R \\ fewSAT\ \alpha_c \\ uniqueSAT \end{array}$$

Some collapsing gives us:

$$StrongRange_{\text{poly}}(\#\mathcal{P})\ \alpha_c\ Gap_8\#SAT\ \alpha_R\ uniqueSAT$$

Bibliographic Notes

The counting class $\#\mathcal{P}$ was introduced by Valiant [4], who proved that computing the permanent of 0-1 matrices is $\#\mathcal{P}$-complete. Valiant's proof first establishes the $\#\mathcal{P}$-hardness of computing the permanent of integer matrices (the entries are actually restricted to $\{-1, 0, 1, 2, 3\}$), and next reduces the computation of the permanent of integer matrices to the the permanent of 0-1 matrices. A de-constructed version of Valinat's proof can be found in [1].

The approximation procedure for $\#\mathcal{P}$ is due to Stockmeyer [3], following an idea of Sipser [2]. Our exposition follows further developments in the area. The randomized reduction of SAT to uniqueSAT is due to Valiant and Vazirani [5]. Again, our exposition is a bit different.

1. A. Ben-Dor and S. Halevi. Zeo-One Permanent is #P-Complete, A Simpler Proof. In *2nd Israel Symp. on Theory of Computing and Systems* (*ISTCS93*), IEEE Computer Society Press, pages 108–117, 1993.

2. M. Sipser. A Complexity Theoretic Approach to Randomness. In *15th STOC*, pages 330–335, 1983.

3. L. Stockmeyer. The Complexity of Approximate Counting. In *15th STOC*, pages 118–126, 1983.

4. L.G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, Vol. 8, pp. 189–201, 1979.

5. L.G. Valiant and V.V. Vazirani. NP Is as Easy as Detecting Unique Solutions. *Theoretical Computer Science*, Vol. 47 (1), pages 85–93, 1986.

Appendix A: A Family of Universal$_2$ Hash Functions

In this appendix we show that the family of affine transformations from $\{0,1\}^n$ to $\{0,1\}^m$ is efficiently constructible and is Universal$_2$.

1. *selecting:* Simply selecting uniformly and independently each bit of A and b, will output a uniformly selected affine transformation. This runs in $O(nm + m)$ time, which is polynomial in the length of the input.

2. *evaluating:* Calculating Ax takes $O(mn)$ time, and the addition of b adds $O(m)$ time. All in all, polynomial in the size of the input.

Proposition: *The family of affine transformations from $\{0,1\}^n$ to $\{0,1\}^m$ is Universal$_2$.*

Proof: Given $x_1 \neq x_2 \in \{0,1\}^n$, and $y_1, y_2 \in \{0,1\}^m$. If $x_1 = 0^n$, then

$$\begin{aligned}
Prob_{A,b}[h(x_1) = y_1 \ \&\ h(x_2) = y_2] &= Prob_{A,b}[b = y_1 \ \&\ Ax_2 + b = y_2] \\
&= Prob_{A,b}[b = y_1 \ \&\ Ax_2 = y_2 - y_1] \\
&= Prob_A[Ax_2 = y_2 - y_1] \cdot Prob_b[b = y_1] \\
&= 2^{-m} \cdot 2^{-m} = (2^{-m})^2
\end{aligned}$$

Where $Prob[Ax_2 = y_2 - y_1] = 2^{-m}$, because for a given vector $x_2 \neq 0^n$, a uniformly chosen linear transformation A, maps x_2 uniformly into $\{0,1\}^m$. If $x_2 = 0$ the same argument holds. Assume both are different than 0^m. Since we choose among the linear transformations uniformly, it does not matter in what base we represent them. Since $x_1, x_2 \neq 0$, and they are both in $\{0,1\}^n$, they must be linearly independent. So we may assume they are both base vectors in the representation of A, meaning one column in A : column a_1 in A represents the image of x_1, and a different column a_2 represents the image of x_2.

$$\begin{array}{rcll} Prob_{A,b}[h(x_1) = y_1 \ \& \ h(x_2) = y_2] & = & Prob_{A,b}[Ax_1 + b = y_1 \ \& \ Ax_2 + b = y_2] & \\ & = & Prob_{a_1,a_2,b}[a_1 + b = y_1 \ \& \ a_2 + b = y_2] & \\ & = & Prob_{a_1,a_2}[a_1 = y_1 - b \ \& \ a_2 = y_2 - b] & \text{(for every } b) \\ & = & Prob_{a_1}[a_1 = y_1 - b] \cdot Prob_{a_2}[a_2 = y_2 - b] & \text{(for every } b) \\ & = & 2^{-m} \cdot 2^{-m} = (2^{-m})^2 & \end{array}$$

■

Appendix B: Proof of Leftover Hash Lemma

In this appendix, we prove the Leftover Hash Lemma (Lemma 10.3.7). We first restate the lemma.

The Leftover Hash Lemma: Let $H_{n,m}$ be a family of Universal$_2$ Hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$, and let $\epsilon > 0$. Let $S \subseteq \{0,1\}^n$ be arbitrary provided that $|S| \geq \epsilon^{-3} \cdot 2^m$. Then:

$$Prob_h[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \frac{|S|}{2^m}] > 1 - \epsilon$$

Proof: We define for each $e \in \{0,1\}^n$ a random variable X_e :

$$X_e = \begin{cases} 1 & h(e) = 0^m \\ 0 & \text{otherwise} \end{cases}$$

For each $e_1 \neq e_2 \in \{0,1\}^n$, we claim that X_{e_1}, X_{e_2} are stochastically independent, because they are functions of the independent random variables $h(e_1)$ and $h(e_2)$ respectively. That is, we use the known fact by which if X and Y are independent random variables then, for every function f, $f(X)$ and $f(Y)$ are also independent random variables.

We compute :

$$\begin{array}{rcl} E(x_e) & = & Prob[X_e = 1] \ = \ \frac{1}{2^m} \\ VAR(X_e) & = & Prob[X_e = 1] \cdot (1 - Prob[X_e = 1]) \ = \ \frac{1}{2^m}(1 - \frac{1}{2^m}) \end{array}$$

We define a new random variable $Y = \sum_{e \in S} X_e$. In other words : $Y = |\{e \in S : h(e) = 0^m\}|$. Since the X_e's are pairwise independent we get:

$$\begin{array}{rcl} E(Y) & = & \sum_{e \in S} E(X_e) = \frac{|S|}{2^m} \\ VAR(Y) & = & \sum_{e \in S} VAR(X_e) = \frac{|S|}{2^m}(1 - \frac{1}{2^m}) = (1 - \frac{1}{2^m}) \cdot E(Y) \end{array}$$

We will now use the Chebychev inequality to prove:

$$\begin{array}{rcl} Prob[|\{e \in S : h(e) = 0^m\}| \in (1 \pm \epsilon) \cdot \frac{|S|}{2^m}] & = & Prob[Y \in (1 \pm \epsilon) \cdot E(Y)] \\ & = & Prob[|Y - EY| \leq \epsilon \cdot E(Y)] \\ & \geq & 1 - \frac{VAR(Y)}{(\epsilon \cdot E(Y))^2} = 1 - \frac{(1 - \frac{1}{2^m}) \cdot E(Y)}{\epsilon^2 (E(Y))^2} \\ & = & 1 - \frac{(1 - \frac{1}{2^m}) 2^m}{\epsilon^2 \cdot |S|} \geq 1 - \epsilon \cdot (1 - \frac{1}{2^m}) > 1 - \epsilon \end{array}$$

■

Lecture 11

Interactive Proof Systems

Notes taken by Danny Harnik, Tzvika Hartman and Hillel Kugler

Summary: We introduce the notion of interactive proof systems and the complexity class IP, emphasizing the role of randomness and interaction in this model. The concept is demonstrated by giving an interactive proof system for the graph non-isomorphism language. We discuss the power of the class IP and prove that co$\mathcal{NP} \subseteq \mathcal{IP}$. We discuss issues regarding the number of rounds allowed in a proof system and introduce the class AM capturing languages recognized by Arthur-Merlin games.

11.1 Introduction

A proof is a way of convincing a party of a certain claim. When talking about proofs, we consider two parties: the *prover* and the *verifier*. Given an assertion, the prover's goal is to convince the verifier of it's validity, whereas the verifier's objective is to accept only a correct assertion. In mathematics, for instance, the prover provides a fixed sequence of claims and the verifier checks that they are truthful and that they imply the theorem. In real life, however, the notion of a proof has a much wider interpretation. A proof is a process rather than a fixed object, by which the validity of the assertion is established. For instance, a job interview is a process in which the candidate tries to convince the employer that she should hire him. In order to make the right decision, the employer carries out an interactive process. Unlike a fixed set of questions, in an interview the employer can adapt her questions according to the answers of the candidate, and therefore extract more information, and lead to a better decision. This example exhibits the power of a proof process rather than a fixed proof. In particular it shows the benefits of interaction between the parties.

In many contexts, finding a proof requires creativity and originality, and therefore attracts most of the attention. However, in our discussion of proof systems, we will focus on the task of the verifier – the verification process. Typically the verification procedure is considered to be relatively easy while finding the proof is considered a harder task. The asymmetry between the complexity of verification and finding proofs is captured by the complexity class NP.

We can view NP as a proof system, where the only restriction is on the complexity of the verification procedure (the verification procedure must take at most polynomial-time). For each language $L \in$NP there exists a polynomial-time recognizable relation R_L such that:

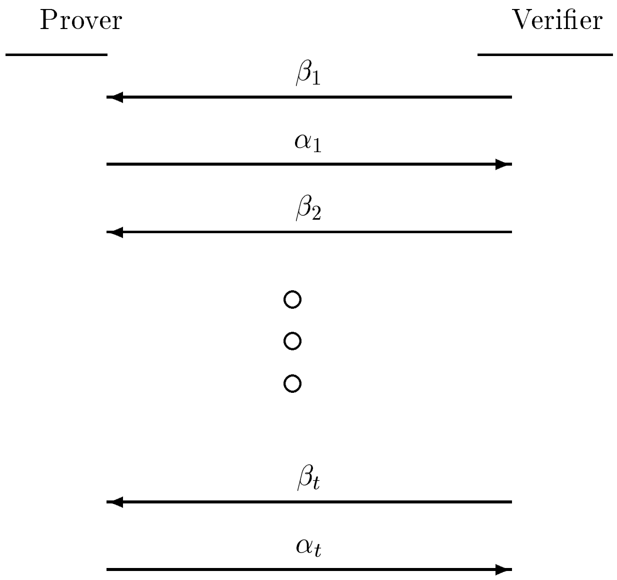
$$L = \{x : \exists y \text{ s.t. } (x, y) \in R_L\}$$

and $(x, y) \in R_L$ only if $|y| \leq poly(|x|)$. In a proof system for an NP language L, a proof for the claim "$x \in L$" consists of the prover sending a witness y, and the verifier checking in polynomial-time whether $(x, y) \in R_L$. Such a witness exists only if the claim is true, hence, only true assertions can be proved by this system. Note that there is no restriction on the time complexity of finding the proof (witness). A good proof system must have the following properties:

1. The verifier strategy is efficient (polynomial-time in the NP case).
2. Correctness requirements:
   - Completeness : For a true assertion, there is a convincing proof strategy (in the case of NP, if $x \in L$ then a witness y exists).
   - Soundness : For a false assertion, no convincing proof strategy exists (in the case of NP, if $x \notin L$ then no witness y exists).

In the following discussion we introduce the notion of *interactive proofs*. To do so, we generalize the requirements from a proof system, adding interaction and randomness.

Roughly speaking, an interactive proof is a sequence of questions and answers between the parties. The verifier asks the prover a question β_i and the prover answers with message α_i. At the end of the interaction, the verifier decides based the knowledge he acquired in the process whether the claim is true or false.



11.2 The Definition of IP

Following the above discussion we define

Definition 11.1 (interactive proof systems): *An* interactive proof system *for a language L is a two-party game between a verifier and a prover that interact on a common input in a way satisfying the following properties:*

1. *The verifier strategy is a probabilistic polynomial-time procedure* (where time is measured in terms of the length of the common input).

2. *Correctness requirements:*

   - Completeness *: There exists a prover strategy P, such that for every $x \in L$, when interacting on the common input x, the prover P convinces the verifier with probability at least $\frac{2}{3}$.*
   - Soundness *: For every $x \notin L$, when interacting on the common input x, any prover strategy P^* convinces the verifier with probability at most $\frac{1}{3}$.*

Note that the prover strategy is computationally unbounded.

Definition 11.2 (The IP Hierarchy): *The complexity class IP consists of all the languages having an interactive proof system.*

We call the number of messages exchanged during the protocol between the two parties, the number of rounds *in the system.*

For every integer function $r(\cdot)$, the complexity class IP($r(\cdot)$) consists of all the languages that have an interactive proof system in which, on common input x, at most $r(|x|)$ rounds are used.

For a set of integer functions R, we denote

$$\mathcal{IP}(R) = \bigcup_{r \in R} \mathcal{IP}(r(\cdot))$$

11.2.1 Comments

- Clearly, $\mathcal{NP} \subseteq \mathcal{IP}$ (actually, $\mathcal{NP} \subseteq \mathcal{IP}(1)$).
  Also, $\mathcal{BPP} = \mathcal{IP}(0)$.

- The number of rounds in IP cannot be more than a polynomial in the length of the common input, since the verifier strategy must run in polynomial-time. Therefore, if we denote by `poly` the set of all integer polynomial functions, then $\mathcal{IP} = \mathcal{IP}(\texttt{poly})$.

- The requirement for completeness, can be modified to require perfect completeness (acceptance probability 1). In other words, if $x \in L$, the prover can always convince the verifier. These two definitions are equivalent. Unlike this, if we require perfect soundness, interactive proof systems collapse to NP-proof systems. These results will be shown in Section 11.5.

- Much like in the definition of the complexity class BPP, the probabilities $\frac{2}{3}$ and $\frac{1}{3}$ in the completeness and soundness requirements can be replaced with probabilities as extreme as $1 - 2^{-p(\cdot)}$ and $2^{-p(\cdot)}$, for any polynomial $p(\cdot)$. In other words the following claim holds:

  Claim 11.2.1 *Any language that has an interactive proof system, has one that achieves error probability of at most $2^{-p(\cdot)}$ for any polynomial $p(\cdot)$.*

  Proof: We repeat the proof system sequentially for k times, and take a majority vote. Denote by z the number of accepting votes. If the assertion holds, then z is the sum of k independent Bernoulli trials with probability of success at least $\frac{2}{3}$. An error in the new protocol happens if $z < \frac{1}{2}k$.

Using Chernoff's Bound :

$$\Pr[z < (1-\delta)E(z)] < e^{-\frac{\delta^2 E(z)}{2}}$$

We choose $k = O(p(\cdot))$ and $\delta = \frac{1}{4}$ and note that $E(z) = \frac{2}{3}k$ (so that $\frac{3}{4} \cdot \frac{2}{3} = \frac{1}{2}$) to get:

$$\Pr\left[z < \frac{1}{2} \cdot k\right] < 2^{-p(\cdot)}$$

The same argument holds for the soundness error (as due to the sequential nature of the interaction we can assert that in each of the k iterations, for any history of prior interactions, the success probability of any cheating strategy is bounded by 1/3). ■

The proof above uses sequential repetition of the protocol to amplify the probabilities. This suffices for showing that the class IP is invariant under the various definitions discussed. However, this method increases the number of rounds used in the proof system. In order to show the invariance of the class IP$(r(\cdot))$, an analysis of the parallel repetition version should be given. (Such an argument is given in Appendix C.1 of [3].)

- Introducing both interaction and randomness in the IP class is essential.
  - By adding interaction only, the interactive proof systems collapse to NP-proof systems. Given an interactive proof system for a prover and a deterministic verifier, we construct an NP- proof system. The prover can predict the verifier's part of the interaction and send the full transcript as an NP witness. The verifier checks that the witness is a valid and accepting transcript of the original proof system. An alternative argument uses the fact that interactive proof systems with perfect soundness are equivalent to NP-proof systems (and the fact that a deterministic verifier necessarily yields perfect soundness).
  - By adding randomness only, we get a proof system in which the prover sends a witness and the verifier can run a BPP algorithm for checking its validity. We obtain a class IP(1) (also denoted MA) which seems to be a randomized (and perhaps stronger) version of NP.

11.2.2 Example – Graph Non-Isomorphism (GNI)

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *isomorphic* (denoted $G_1 \cong G_2$) if there exists a 1-1 and onto mapping $\pi : V_1 \to V_2$ such that $(u,v) \in E_1 \Leftrightarrow (\pi(u), \pi(v)) \in E_2$. The mapping π, if existing, is called an *isomorphism* between the graphs. If no such mapping exists then the graphs are *non-isomorphic* (denoted $G_1 \ncong G_2$).

GNI is the language containing all pairs of non-isomorphic graphs. Formally :

$$GNI = \{(G_1, G_2) : G_1 \ncong G_2\}$$

An interactive proof system for GNI:

- G_1 and G_2 are given as input to the verifier and the prover. Assume without loss of generality that $V_1 = V_2 = \{1, 2, ..., n\}$
- The verifier chooses $i \in_R \{1, 2\}$ and $\pi \in_R S_n$ (S_n is the group of all permutations on $\{1, 2, ..., n\}$).

He applies the mapping π on the graph G_i to obtain a graph H

$$H = (\{1,2,...,n\}, E_H) \text{ where } E_H = \{(\pi(u),\pi(v)) : (u,v) \in E_i\}$$

and sends the graph H to the prover.

- The prover sends $j \in \{1,2\}$ to the verifier.
- The verifier accepts iff $j = i$.

Motivation : if the input graphs are non-isomorphic, as the prover claims, then the prover should be able to distinguish (not necessarily by an efficient algorithm) isomorphic copies of one graph from isomorphic copies of the other graph. However, if the input graphs are isomorphic, then a random isomorphic copy of one graph is distributed identically to a random isomorphic copy of the other graph and therefore, the best choice the prover could make is a random one. This fact enables the verifier to distinguish between the two cases. Formally:

Claim 11.2.2 *The above protocol is an interactive proof system for GNI.*

Comment: We show that the above protocol is an interactive proof system with soundness probability at most $\frac{1}{2}$ rather than $\frac{1}{3}$ as in the formal definition. However, this is equivalent by an amplification argument (see Claim 11.2.1).

Proof: We have to show that the above system satisfies the two properties in the definition of interactive proof systems:

- The verifier's strategy can be easily implemented in probabilistic polynomial time. (The prover's complexity is unbounded and indeed, he has to check isomorphism between two graphs, a problem not known to be solved in probabilistic polynomial time.)
- - Completeness : In case $G_1 \not\cong G_2$, every graph can be isomorphic to at most one of G_1 or G_2 (otherwise, the existence of a graph isomorphic to both G_1 and G_2 implies $G_1 \cong G_2$). It follows that the prover can always send the correct j (i.e. a j such that $j = i$), since $H \cong G_i$ and $H \not\cong G_{3-i}$.
  - Soundness : In case $G_1 \cong G_2$ we show that the prover convinces the verifier with probability at most $\frac{1}{2}$ (the probability ranges over all the possible coin tosses of the verifier, i.e. the choice of i and π). Denote by H the graph sent by the verifier. $G_1 \cong G_2$ implies that H is isomorphic to both G_1 and G_2. For $k = 1,2$ let

    $$S_{G_k} = \{\sigma \in S_n \mid \sigma G_k = H\}$$

    This means that when choosing $i = k$, the verifier can obtain H only by choosing $\pi \in S_{G_k}$.
    Assume $\tau \in S_n$ is an isomorphism between G_2 and G_1, i.e. $G_1 = \tau G_2$. For every $\sigma \in S_{G_1}$ it follows that $\sigma\tau \in S_{G_2}$ (because $\sigma\tau G_2 = \sigma G_1 = H$). Therefore, τ is a 1-1 mapping from S_{G_1} to S_{G_2} (since S_n is a group). Similarly, τ^{-1} is a 1-1 mapping from S_{G_2} to S_{G_1}. Combining the two arguments we get that $|S_{G_1}| = |S_{G_2}|$. Therefore, given that H was sent, the probability that the verifier chose $i = 1$ is equal to the probability of the choice $i = 2$. It follows that for every decision the prover makes he has success probability $\frac{1}{2}$ and therefore, his total probability of success is $\frac{1}{2}$.

■

The above interactive proof system is implemented with only 2 rounds. Therefore,

Corollary 11.3 *GNI $\in$ IP(2).*

11.3 The Power of IP

We have already seen that NP$\subseteq$ IP. The above example suggests that the power of IP is even greater. Since GNI is not known to be in NP we conjecture that NP$\subset$ IP (strict inclusion). Furthermore, the class of languages having interactive proof systems is shown to be equivalent to the powerful complexity class PSPACE. Formally,

Theorem 11.4 *$\mathcal{IP} = \mathcal{PSPACE}$.*

We will only give a partial proof of the theorem. We'll only show that co$\mathcal{NP} \subseteq \mathcal{IP} \subseteq \mathcal{PSPACE}$.

11.3.1 IP is contained in PSPACE

We start by proving the less interesting direction of the theorem (i.e., IP $\subseteq$ PSPACE). This is proven by showing that (for every fixed verifier), an optimal prover strategy exists and can be implemented in polynomial-space.

The Optimal Prover: Given a fixed verifier strategy, there exists an optimal prover strategy; that is, for every common input x, the optimal strategy has the highest possible probability of convincing the verifier. Note that an optimal prover strategy is well-defined, as for every input x and fixed prover strategy, the probability that the prescribed verifier accepts is well-defined (and the number of prover's strategies for input x is finite). A more explicit way of arguing the existence of an optimal prover strategy yields an algorithm for computing it. We first observe that given the verifier strategy and the verifier's coin tosses, we can simulate the whole interaction and it's outcome for any prover strategy. Now, the optimal prover strategy may enumerate all possible outcomes of the verifier's coin tosses, and count how many times each strategy succeeds. The optimal strategy for each input, is one that yields the highest number of successes. Furthermore, this can be done in polynomial-space:

Claim 11.3.1 *The optimal prover strategy can be computed in polynomial-space.*

Proof: We assume without loss of generality that the verifier tosses all his coins before the interaction begins. We also assume that the verifier plays first. Let β_i be the i^{th} message sent by the verifier and α_i be the i^{th} message sent by the prover. Let r be the outcome of all the verifier's coin tosses. Let $R_{\beta_1,\alpha_1,\ldots,\alpha_{i-1},\beta_i}$ be the set of all r's (outcome of coin tosses) that are consistent with the interaction $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$.

Let $F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i)$ be the probability that an interaction (between the optimal prover and the fixed verifier) beginning with $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$ will result in acceptance. The probability is taken uniformly over the verifier's relevant coin tosses (only r such that $r \in R_{\beta_1,\alpha_1,\ldots,\alpha_{i-1},\beta_i}$).

Suppose an interaction between the two parties consists of $\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i$ and it is now the prover's turn to play. Using the function F, the prover can find the optimal move. We show that a polynomial-space prover can recursively compute $F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i)$. Furthermore, in the process, the prover finds an α_i that yields this probability and hence, an α_i that is an optimal move for the prover.

The best choice for α_i is one that gives the highest expected value of $F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})$ over all of the possiblities of verifier's next message (β_{i+1}). Formally :

$$(1) \qquad F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i) = \max_{\alpha_i} E_{\beta_{i+1}}[F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})]$$

Let $V(r, \alpha_1, ..., \alpha_i)$ be the message β_{i+1} that the verifier sends after tossing coins r and receiving messages $\alpha_1, ..., \alpha_i$ from the prover.

The probability for each possible message β_{i+1} to be sent by after $\beta_1, \alpha_1, ..., \alpha_i$ is the portion of possible coins $r \in R_{\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i}$ that yield the message β_{i+1} (i.e. $\beta_{i+1} = V(r, \alpha_1, ..., \alpha_i)$). This yields the following equation for the expected probability :

$$(2) \qquad E_{\beta_{i+1}}[F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})] = \frac{1}{|R_{\beta_1, \alpha_1, ..., \beta_i}|} \sum_{r \in R_{\beta_1, \alpha_1, ..., \beta_i}} F(\beta_1, \alpha_1, ..., \alpha_i, V(r, \alpha_1, ..., \alpha_i))$$

Combining (1) and (2) we get the recursion formula

$$F(\beta_1, \alpha_1, ..., \alpha_{i-1}, \beta_i) = \max_{\alpha_i} \frac{1}{|R_{\beta_1, \alpha_1, ..., \beta_i}|} \sum_{r \in R_{\beta_1, \alpha_1, ..., \beta_i}} F(\beta_1, \alpha_1, ..., \alpha_i, V(r, \alpha_1, ..., \alpha_i))$$

We now show how to compute the function F in polynomial-space:
For each potential α_i, we enumerate all possible values of r. For each r, all of the following can be done in polynomial-space:

- Checking if $r \in R_{\beta_1, \alpha_1, ..., \beta_i}$ by simulating the verifier in the first i interactions (when given r the verifier strategy is polynomial).
- Calculating $\beta_{i+1} = V(r, \alpha_1, ..., \alpha_i)$ again by simulating the verifier.
- Recursively computing $F(\beta_1, \alpha_1, ..., \alpha_i, \beta_{i+1})$.

  In order for the recursion to be polynomial-space computable, we need to show that the recursion stops after polynomially many stages, and that the last stage can be computed in polynomial-space. The recursion stops when reaching a full transcript of the proof system. In such a case the prover can enumerate r and find the probability of acceptance among all consistent r by simulating the verifier. Clearly, this can be done in polynomial-space. Also the depth of the recursion must be at most polynomial, which is obviously the case here, since it is bounded by the number of rounds.

Using polynomial-size counters, we can sum the probabilities for all consistent r, and find the expected probability for each α_i. By repeating this for all possible α_i we can find one that maximizes the expectation. Altogether, the prover's optimal strategy can be calculated in polynomial-space.

Note: All the probabilities are taken over the verifier's coin tosses (no more than a polynomial number of coins). This enables us to use polynomial-size memory for calculating all probabilities with exact resolution (by representing them as rational numbers – storing the numerator and denominator separately). ■

Corollary 11.5 *IP* $\subseteq$ *PSPACE*

Proof: If $L \in$ IP then there exists an interactive proof system for L and hence there exists a polynomial-space optimal prover strategy. Given input x and the verifier‘s coin tosses, we can simulate (in polynomial-space) the interaction between the optimal prover and the verifier and determine this interaction's outcome. We enumerate over all the possible verifier's coin tosses and accept only if more than $\frac{2}{3}$ of the outcomes are accepting. Clearly, we accept if and only if $x \in L$ and this can be implemented in polynomial-space. ■

11.3.2 coNP is contained in IP

As mentioned above, we will not prove that $\mathcal{PSPACE} \subseteq \mathcal{IP}$. Instead, we prove a weaker theorem (i.e., co$\mathcal{NP} \subseteq \mathcal{IP}$), which by itself is already very interesting. The proof of the weaker theorem presents all but one ingrediant of the proof $\mathcal{PSPACE} \subseteq \mathcal{IP}$ (and the missing ingrediant is less interesting).

Theorem 11.6 co$\mathcal{NP} \subseteq \mathcal{IP}$

Proof: We prove the theorem by presenting an interactive proof system for the coNP-complete problem $\overline{3SAT}$ (the same method can work for the problem $\overline{SAT}$ as well). $\overline{3SAT}$ is the set of non-satisfiable $3CNF$ formulae: Given a $3CNF$ formula ϕ, it is in the set if no truth assignment to it's variables satisfies the formula.

The proof uses an arithmetic generalization of the boolean problem, which allows us to apply algebraic methods in the proof system.

The Arithmetization of a Boolean CNF formula: Given the formula ϕ with variables $x_1, ..., x_n$ we perform the following replacements:

Boolean		Arithmetic
T	$\longrightarrow$	positive integers
F	$\longrightarrow$	0
x_i	$\longrightarrow$	x_i
$\overline{x_i}$	$\longrightarrow$	$(1 - x_i)$
$\vee$	$\longrightarrow$	$+$
$\wedge$	$\longrightarrow$	$\cdot$
$\phi(x_1, ..., x_n)$	$\longrightarrow$	$\Phi(x_1, ..., x_n)$

Every boolean $3CNF$ formula ϕ is transformed into a multi-variable polynomial Φ. It is easy to see that for every assignment $x_1, ..., x_n$, we have

$$\phi(x_1, ..., x_n) = F \Longleftrightarrow \Phi(x_1, ..., x_n) = 0$$

Summing over all possible assignments, we obtain an equation for the non-satisfiability of ϕ:

$$\phi \text{ is unsatisfiable } \Longleftrightarrow \sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) = 0$$

Suppose ϕ has m clauses of length three each, thus any 0-1 assignment to $x_1, ..., x_n$ gives $\Phi(x_1, ..., x_n) \leq 3^m$. Since there are 2^n different assignments, the sum above is bounded by $2^n \cdot 3^m$. This fact allows us to move our calculations to a finite field, by choosing a prime q such that $q > 2^n \cdot 3^m$, and working modulo this prime. Thus proving that ϕ is unsatisfiable reduces to proving that

$$\sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) \equiv 0 \pmod q$$

We choose q to be not much larger than $2^n \cdot 3^m$ (this is possible due to the density of the prime numbers). Thus, we obtain that all calculations over the field $GF(q)$ can be done in polynomial-time (in the input length). Working over a finite field will later help us in the task of uniformly selecting an element in the field.

The interactive proof system for $\overline{3SAT}$:

- Both sides receive the common boolean formula ϕ. They perform the arithmetization procedure and obtain Φ.
- The prover picks a prime q such that $q > 2^n \cdot 3^m$, and sends q to the verifier. The verifier checks that q is indeed a prime. If not he rejects.
- The verifier initializes $v_0 = 0$.
- The following is performed n times (i runs from 1 to n):
  - The prover sends a polynomial $P_i(\cdot)$ of degree at most m to the verifier.
  - The verifier checks whether $P_i(0) + P_i(1) \equiv v_{i-1} \pmod q$ and that the polynomial's degree is at most m.

    If not, the verifier rejects.

    Otherwise, he uniformly picks $r_i \in_R GF(q)$, calculates $v_i = P_i(r_i)$ and sends r_i to the prover.
- The verifier accepts if $\Phi(r_1, ...r_n) \equiv v_n \pmod q$ and rejects otherwise.

Motivation: The prover has to find a sequence of polynomials that satisfies a number of restrictions. The restrictions are imposed by the verifier in the following interactive manner: after receiving a polynomial from the prover, the verifier sets a new restriction for the next polynomial in the sequence. These restrictions guarantee that if the claim holds (ϕ is unsatisfiable), such a sequence can always be found (we call it the "Honest prover strategy"). However, if the claim is false, any prover strategy has only a small probability of finding such a sequence (the probability is taken over the verifier's coin tosses). This yields the completeness and soundness of the suggested proof system. The nature of these restrictions is fully clarified in the proof of soundness, but we will first show that the verifier strategy is efficient.

The verifier strategy is efficient: Most steps in the protocol are calculations of polynomials of degree m in n variables, these are easily calculated in polynomial-time. The transformation to an arithmetic field is linear in the formula's length.

Checking primality is known to be in BPP and therefore can be done by the verifier. Furthermore, it can be shown that primality testing is in NP, so the prover can send the verifier an NP-witness to the fact that q is a prime, and the verifier checks this witness in polynomial-time.

Finally, picking an element $r \in_R GF(q)$ can be done in $O(\log q)$ coin tosses, that is polynomial in the formula's length.

The honest prover strategy: For every i define the polynomial:

$$P_i^*(z) = \sum_{x_{i+1}=0,1} \cdots \sum_{x_n=0,1} \Phi(r_1, ..., r_{i-1}, z, x_{i+1}, ..., x_n)$$

Note that $r_1, ..., r_{i-1}$ are constants set by the verifier in the previous stages and known to the prover at the i^{th} stage, and z is the polynomial's variable.

The following facts are evident about P_i^*:

- Calculating P_i^* may take exponential-time, but this is no obstacle for a computationally unbounded prover.
- The degree of P_i^* is at most m. Since there are at most m clauses in ϕ, the highest degree of any one variable is m (if it appears in all clauses).

Completeness of the proof system: When the claim holds, the honest prover always succeeds in convincing the verifier. For $i > 1$:

$$(3.1) \quad P_i^*(0) + P_i^*(1) = \sum_{x_i=0,1} P_i^*(x_i) =_{(1)} \sum_{x_i=0,1} ... \sum_{x_n=0,1} \Phi(r_1, ..., r_{i-1}, x_i, ..., x_n)$$

$$=_{(2)} P_{i-1}^*(r_{i-1}) \equiv_{(3)} v_{i-1} \pmod q$$

Equality (1) is due to the definition of P_i^*. Equality (2) is due to the definition of P_{i-1}^*. Equality (3) is the definition of v_{i-1}.
Also for $i = 1$, since the claim holds we have:

$$P_1^*(0) + P_1^*(1) = \sum_{x_1=0,1} P_1^*(x_1) = \sum_{x_1=0,1} ... \sum_{x_n=0,1} \Phi(x_1, ..., x_n) \equiv v_0 \pmod q$$

And finally: $v_n = P_n^*(r_n) = \Phi(r_1, ..., r_n)$.

We showed that the polynomials of the honest prover pass all of the verifier's tests, obtaining perfect completeness of the proof system.

Soundness of the proof system: If the claim is false, an honest prover will definitely fail after sending P_1^*, thus a prover must be dishonest.

Roughly speaking, we will show that if a prover is dishonest in one round, then with high probability he must be dishonest in the next round as well. In the last round, his dishonesty is revealed. Formally:

Lemma 11.3.2 *If $P_i^*(0) + P_i^*(1) \not\equiv v_{i-1} \pmod q$ then either the verifier rejects in the i^{th} round, or $P_i^*(r_i) \not\equiv v_i \pmod q$ with probability at least $1 - \frac{m}{q}$, where the probability is taken over the verifier's choices of r_i.*

We stress that P_i^* is the polynomial of the honest prover strategy (as defined above), while P_i is the polynomial actually sent by the prover (v_i is set using P_i).
Proof: (of lemma) If the prover sends $P_i = P_i^*$, we get:

$$P_i(0) + P_i(1) \equiv P_i^*(0) + P_i^*(1) \not\equiv v_{i-1} \pmod q$$

and the verifier rejects immeadiately.

Otherwise the prover sends $P_i \neq P_i^*$. We assume P_i passed the verifier's test (if not the verifier rejects and we are done). Since P_i and P_i^* are of degree at most m, there are at most m choices of $r_i \in GF(q)$ such that

$$P_i^*(r_i) \equiv P_i(r_i) \pmod q$$

For all other choices:

$$P_i^*(r_i) \not\equiv P_i(r_i) \equiv v_i \pmod q$$

Since the verifier picks $r_i \in_R GF(q)$, we get $P_i^*(r_i) \equiv v_i \pmod q$ with probability at most $\frac{m}{q}$, ■

Suppose the verifier does not reject in any of the n rounds. Since the claim is false (ϕ is satisfiable), we have $P_1^*(0) + P_1^*(1) \not\equiv v_0 \pmod q$. Applying alternately the lemma and the following equality: for every $i \geq 2 \quad P_{i-1}^*(r_{i-1}) = P_i^*(0) + P_i^*(1)$ (due to equation 3.1) , we get that $P_n^*(r_n) \not\equiv v_n \pmod q$ with probability at least $(1 - \frac{m}{q})^n$. But $P_n^*(r_n) = \Phi(r_1, ..., r_n)$ so the verifier's last test fails and he rejects. Altogether the verifier fails with probability $(1 - \frac{m}{q})^n > 1 - \frac{nm}{q} > \frac{2}{3}$ (by the choice of q). ■

11.4 Public-Coin Systems and the Number of Rounds

An interesting question is how the power of interactive proof systems is affected by the number of rounds allowed. We have already seen that GNI can be proved by an interactive proof with 2 rounds. Despite this example of a coNP language, we conjecture that coNP $\not\subseteq$ IP($O(1)$). Together with the previous theorem we get:

Conjecture 11.7

$$\mathcal{IP}(O(1)) \subset \mathcal{IP}(poly) \quad \textit{(strict containment)}$$

A useful tool in the study of interactive proofs, is the public coin variant, in which the verifier can only ask random questions.

Definition 11.8 (public-coin interactive proofs – $\mathcal{AM}$): *Public coin proof systems (known also as Arthur-Merlin games) are a special case of interactive proof systems, in which, at each round, the verifier can only toss coins, and send their outcome to the prover. In the last round, the verifier decides whether to accept or reject.*

For every integer function $r(\cdot)$, the complexity class $\mathcal{AM}(r(\cdot))$ consists of all the languages that have an Arthur-Merlin proof system in which, on common input x, at most $r(|x|)$ rounds are used.

Denote $\mathcal{AM} = \mathcal{AM}(2)$.

We note that the definition of $\mathcal{AM}$ as Arthur-Merlin games with two rounds is inconsistent with the notation IP= IP($poly$)). (Unfortunately, that's what is found in the literature.)

The difference between the Arthur-Merlin games and the general interactive proof systems can be viewed as the difference between asking tricky questions, versus asking random questions. Surprisingly it was shown that these two versions are essentially equivalent:

Theorem 11.9 (Relating $\mathcal{IP}(\cdot)$ to $\mathcal{AM}(\cdot)$):

$$\forall r(\cdot) \quad \mathcal{IP}(r(\cdot)) \subseteq \mathcal{AM}(r(\cdot)+2)$$

The following theorem shows that power of $\mathcal{AM}(r(\cdot))$ is invariant under a linear change in the number of rounds:

Theorem 11.10 (Linear Speed-up Theorem):

$$\forall r(\cdot) \geq 2 \quad \mathcal{AM}(2r(\cdot)) = \mathcal{AM}(r(\cdot))$$

The above two theorems are quoted without proof. Combining them we get:

Corollary 11.11 $\forall r(\cdot) \geq 2 \quad \mathcal{IP}(2r(\cdot)) = \mathcal{IP}(r(\cdot))$.

Corollary 11.12 (Collapse of constant-round IP to two-round AM):

$$\mathcal{IP}(O(1)) = \mathcal{AM}(2)$$

11.5 Perfect Completeness and Soundness

In the definition of interactive proof systems we require the existence of a prover strategy that for $x \in L$ convinces the verifier with probability at least $\frac{2}{3}$ (analogous to the definition of the complexity class BPP). One can consider a definition requiring *perfect completeness*; i.e., convincing the verifier with probability 1 (analogous to coRP). We will now show that the definitions are equivalent.

Theorem 11.13 *If a language L has an interactive proof system then it has one with perfect completeness.*

We will show that given a public coin proof system we can construct a perfect completeness public coin proof system.

We use the fact that public coin proof systems and interactive proof systems are equivalent (see Theorem 11.9), so if L has an interactive proof system it also has a public coin proof system. We define:

$$AM^0(r(\cdot)) \quad = \quad \{L \mid \text{ L has perfect completeness } r(\cdot) \text{ round public coin proof system}\}$$

So given an interactive proof system we create a public coin proof system and using the following lemma convert it to one with perfect completeness. Thus, the above theorem which refers to arbitrary interactive proofs follows from the following lemma which refers only to public-coin interactive proofs.

Lemma 11.5.1 *If L has a public coin proof system then it has one with perfect completeness*

$$AM(r(\cdot)) \subseteq AM^0(r(\cdot)+1)$$

Proof: Given an Arthur-Merlin proof system, we construct an Arthur-Merlin proof system with perfect completeness and one more round. We use the same idea as in the proof of BPP $\subseteq$ PH.

Assume, without loss of generality, that the Arthur-Merlin proof system consists of $2t$ rounds, and that Arthur sends the same number of coins m in each round (otherwise, ignore the redundant coins). Also assume that the completeness and soundness error probabilities of the proof system are at most $\frac{1}{3tm}$. This is obtained using amplification (see Section 11.2.1).

We denote the messages sent by Arthur (the verifier) $r_1, ..., r_t$ and the messages sent by Merlin (the prover) $\alpha_1, ..., \alpha_t$. Denote by $\langle P, V\rangle_x(r_1, ..., r_t)$ the outcome of the game on common input x between the optimal prover P and the verifier V in which the verifier uses coins $r_1, ..., r_t$: $\langle P, V\rangle_x(r_1, ..., r_t) = 0$ if the verifier rejects and $\langle P, V\rangle_x(r_1, ..., r_t) = 1$ otherwise.

We construct a new proof system with perfect completeness, in which Arthur and Merlin play tm games simultaneously. Each game is like the original game except that the random coins are shifted by a fixed amount. The tm shifts (one for each game) are sent by Merlin in an additional round at the beginning. Arthur accepts if at least one of the games is accepting. Formally, we add an additional round at the beginning in which Merlin sends the shifts $S^1, ..., S^{tm}$ where $S^i = (S^i_1, ..., S^i_t), S^i_j \in \{0,1\}^m$ for every i between 1 and tm. Like in the original proof system Arthur sends messages $r_1, ..., r_t$, where $r_i \in_R \{0,1\}^m$. For game i and round j, Merlin considers the random coins to be $r_j \oplus S^i_j$ and sends as a message α^i_j where α^i_j is computed according to $(r_1 \oplus S^i_1, ..., r_j \oplus S^i_j)$. The entire message in round j is $\alpha^1_j, ..., \alpha^{tm}_j$. At the end of the protocol Arthur accepts if at least one out of the tm games is accepting.

In order to show perfect completeness we will show that for every $x \in L$ there exist $S^1, ..., S^{tm}$ such that for all $r_1, ...r_t$ at least one of the games is accepting. We use a probabilistic argument to show that the complementary event occurs with probability strictly smaller than 1.

$$\begin{aligned}
&\Pr_{S^1,...S^{tm}}\left[\exists r_1, ...r_t \bigwedge_{i=1}^{tm} (\langle P, V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 0)\right] \\
&\leq_{(1)} \sum_{r_1,...r_t \in \{0,1\}^m} \Pr_{S^1,...S^{tm}}\left[\bigwedge_{i=1}^{tm} (\langle P, V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 0)\right] \\
&\leq_{(2)} 2^{tm} \cdot \left(\frac{1}{3tm}\right)^{tm} \quad < \quad 1
\end{aligned}$$

Inequality (1) is obtained using the union bound. Inequality (2) is due to the fact that the $r_j \oplus S_j^i$ are independent random variables so the results of the games are independent, and that the optimal prover fails to convince the verifier on a true assertion with probability at most $\frac{1}{3tm}$.

We still have to show that the proof system suggested satisfies the soundness requirement. We show that for every $x \notin L$ and for any prover strategy P^* and choices of shifts $S^1, ..., S^{tm}$ the probability that one or more of the tm games is accepting is at most $\frac{1}{3}$.

$$\begin{aligned}
&\Pr_{r_1,...r_t}\left[\bigvee_{i=1}^{tm} (\langle P^*, V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 1)\right] \\
&\leq_{(1)} \sum_{i=1}^{tm} \Pr_{r_1,...,r_t}\left[\langle P^*, V\rangle_x(r_1 \oplus S_1^i, ..., r_t \oplus S_t^i) = 1\right] \\
&\leq_{(2)} \sum_{i=1}^{tm} \frac{1}{3tm} \quad = \quad \frac{1}{3}
\end{aligned}$$

Inequality (1) is obtained using the union bound. Inequality (2) is due to the fact that any prover has probability of at most $\frac{1}{3tm}$ of success for a single game (because any strategy that the prover can play in a copy of the parallel game can be played in a single game as well). ■

Unlike the last theorem, requiring *perfect soundness* (i.e. for every $x \notin L$ and every prover strategy P^*, the verifier always rejects after interacting with P^* on common input x) reduces the model to an NP-proof system , as seen in the following proposition:

Proposition 11.5.2 *If a language L has an interactive proof system with perfect soundness then $L \in \mathcal{NP}$.*

Proof: Given an interactive proof system with perfect soundness we construct an NP proof system. In case $x \in L$, by the completeness requirement, there exists an accepting transcript. The prover finds an outcome of the verifier's coin tosses that gives such a transcript and sends the full transcript along with the coin tosses. The verifier checks in polynomial time that the transcript is valid and accepting and if so - accepts. This serves as an NP-witness to the fact that $x \in L$. If $x \notin L$ then due to the perfect soundness requirement, no outcome of verifier's coin tosses yields an accepting transcript and therefore there are no NP-witnesses. ■

Bibliographic Notes

Interactive proof systems were introduced by Goldwasser, Micali and Rackoff [5], with the explicit objective of capturing the most general notion of efficiently verifiable proof systems. The original motivation was the introduction of zero-knowledge proof systems, which in turn were supposed to provide (and indeed do provide) a powerful tool for the design of complex cryptographic schemes.

First evidence that interactive proofs may be more powerful than NP-proofs was given by Goldreich, Micali and Wigderson [4], in the form of the interactive proof for Graph Non-Isomorphism presented above. The full power of interactive proof systems was discoved by Lund, Fortnow, Karloff, Nisan, and Shamir (in [7] and [8]). The basic technique was presented in [7] (where it was shown that co$\mathcal{NP} \subseteq \mathcal{IP}$) and the final result ($\mathcal{PSPACE} = \mathcal{IP}$) in [8]. Our presentation follows [8]. For further discussion of credits, see [3].

Public-coin interactive proofs (also known as Arthur-Merlin proofs) were introduced by Babai [1]. The fact that these restricted interactive proofs are as powerful as general ones was proved by Goldwasser and Sipser [6]. The linear speed-up (in number of rounds) of public-coin interactive proofs was shown by Babai and Moran [2].

1. L. Babai. Trading Group Theory for Randomness. In *17th STOC*, pages 421–429, 1985.

2. L. Babai and S. Moran. Arthur-Merlin Games: A Randomized Proof System and a Hierarchy of Complexity Classes. *JCSS*, Vol. 36, pp. 254–276, 1988.

3. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

4. O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

5. S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SICOMP*, Vol. 18, pages 186–208, 1989. Preliminary version in *17th STOC*, 1985. Earlier versions date to 1982.

6. S. Goldwasser and M. Sipser. Private Coins versus Public Coins in Interactive Proof Systems. *Advances in Computing Research: a research annual*, Vol. 5 (Randomness and Computation, S. Micali, ed.), pages 73–90, 1989. Extended abstract in *18th STOC*, pages 59–68, 1986.

7. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *JACM*, Vol. 39, No. 4, pages 859–868, 1992. Preliminary version in *31st FOCS*, 1990.

8. A. Shamir. IP = PSPACE. *JACM*, Vol. 39, No. 4, pages 869–877, 1992. Preliminary version in *31st FOCS*, 1990.

Lecture 12

Probabilistically Checkable Proof Systems

Notes taken by Alon Rosen and Vered Rosen

Summary: In this lecture we introduce the notion of Probabilistically Checkable Proof (PCP) systems. We discuss some complexity measures involved, and describe the class of languages captured by corresponding PCP systems. We then demonstrate the alternative view of $\mathcal{NP}$ emerging from the PCP theorem, and use it in order to prove two non-approximability results for the problems $max3SAT$ and $maxCLIQUE$.

12.1 Introduction

Loosely speaking, a probabilistically checkable proof system (PCP) for a language consists of a probabilistic polynomial-time verifier having direct access to individual bits of a binary string. This string (called oracle) represents a proof, and typically will be accessed only partially by the verifier. Queries to the oracle are positions on the bit string and will be determined by the verifier's input and coin tosses (potentially, they might be determined by answers to previous queries as well). The verifier is supposed to decide whether a given input belongs to the language.

If the input belongs to the language, the requirement is that the verifier will always accept (i.e. given access to an adequate oracle). On the other hand, if the input does not belong to the language then the verifier will reject with probability at least $\frac{1}{2}$, no matter which oracle is used.

One can view PCP systems in terms of interactive proof systems. That is, one can think of the oracle string as being the prover and of the queries as being the messages sent to him by the verifier. In the PCP setting however, the prover is considered to be memoryless and thus cannot adjust his answers based on previous queries posed to him.

A more appealing interpretation is to view PCP systems as a possible way of generalizing $\mathcal{NP}$. Instead of conducting a polynomial-time computation upon receiving the entire proof (as in the case of $\mathcal{NP}$), the verifier is allowed to toss coins and query the proof only at locations of his choice. This either allows him to inspect very long proofs (looking at no more than polynomially many locations), or alternatively, look at very few bits of a possible proof.

Most surprisingly, PCP systems have been used to fully characterize the languages in $\mathcal{NP}$. This characterization has been found to be useful in connecting the hardness involved in the approximation of some NP-$hard$ problems with the $\mathcal{P} \neq \mathcal{NP}$ question. In other words, very strong non-approximability results for various classical optimization problems have been established using PCP systems for $\mathcal{NP}$ languages.

12.2 The Definition

12.2.1 The basic model

In the definition of PCP systems we make use of the notion of a probabilistic oracle machine. In our setting, this will be a probabilistic Turing machine which, in addition to the usual features, will have direct access (counted as a single step) to individual bits of a binary string (the oracle). ¿From now on, we denote by $M^\pi(x)$ the output of machine M on input x, when given such oracle access to the binary string π.

Definition 12.1 (Probabilistically Checkable Proofs - PCP) *A probabilistic checkable proof system for a language L is a probabilistic polynomial-time oracle machine* (called verifier)*, denoted M, satisfying*

- *Completeness: For every $x \in L$ there exists an oracle π_x such that:*

$$\Pr\left[M^{\pi_x}(x) = 1\right] = 1$$

- *Soundness: For every $x \notin L$ and every oracle π:*

$$\Pr\left[M^{\pi}(x) = 1\right] \leq \frac{1}{2}$$

where the probability is taken over M's internal coin tosses.

12.2.2 Complexity Measures

When considering a randomized oracle machine, some complexity measures other than time may come into concern. A natural thing would be to count the number of queries made by the verifier. This number determines what is the portion of the proof being read by the verifier. Another concern would be to count the number of coins tossed by the randomized oracle machine. This in turn determines what is the total number of possible executions of the verifier (once an oracle is fixed).

It turns out that the class of languages captured by PCP systems varies greatly as the above mentioned resources of the verifier are changed. This motivates a quantitative refinement of the definition of PCP systems which captures the above discussed concept.

Definition 12.2 (Complexity Measures for PCP) *Let $r, q : N \to N$ be integer functions (in particular constant). The complexity class $\mathcal{PCP}(r(\cdot), q(\cdot))$ consists of languages having a probabilistic checkable proof system in which it holds that:*

- Randomness Complexity*: On input $x \in \{0,1\}^*$, the verifier makes at most $r(|x|)$ coin tosses.*
- Query Complexity*: On input $x \in \{0,1\}^*$, the verifier makes at most $q(|x|)$ queries.*

For sets of integer functions R and Q, we let

$$\mathcal{PCP}(R, Q) \stackrel{\text{def}}{=} \bigcup_{r \in R, q \in Q} \mathcal{PCP}(r(\cdot), q(\cdot))$$

In particular, we denote by poly the set of all integer functions bounded by a polynomial and by log the set of all integer functions bounded by a logarithmic function (e.g. $f \in \log$ iff $f(n) = O(\log n)$). From now on, whenever referring to a PCP system, we will also specify its corresponding complexity measures.

12.2.3 Some Observations

- The definition of PCP involves binary queries to the oracle (which is itself a binary string). These queries specify locations on the string whose binary values are the answers to the corresponding queries. From now on, when given a query q to an oracle π the corresponding binary answer will be denoted π_q. Note that an oracle string can possibly be of exponential length (since one can specify an exponentially far location on the string using polynomially many bits).

- A PCP verifier is called *non-adaptive* if its queries are determined solely based on its input and the outcome of its coin tosses. (A general verifier, called adaptive, may determine its queries also based on answers to previously received oracle answers). From now on, whenever referring to a PCP verifier it will be assumed to be adaptive (unless otherwise specified).

- A possible motivation for the introduction of PCP systems would be to provide an alternative view of $\mathcal{NP}$, one that will rid us of the "rigidity" of the conventional view. In this regard randomness seems to be a most important ingredient, it provides us the possibility to be "imprecise" in the acceptance of false instances. This is best seen when taking the probability bound in the soundness condition to be zero. This will cause that no probability is involved in the definition and will make it collapse into $\mathcal{NP}$. To see this, notice that in the above case, the output of the verifier does not vary with the outcome of its coin tosses. This means that in order to determine the verifier's decision on some input, it suffices to examine only one of its possible executions (say, when using the all zero coin sequence). In such an execution only a polynomial portion of the PCP proof is being read by the verifier. It is easy to see, that in this case, the PCP and $\mathcal{NP}$ definitions coincide (just treat the relevant portion of the PCP proof as an $\mathcal{NP}$-*witness*).

  Note that in order to be consistent with the $\mathcal{NP}$ definition we require perfect completeness (i.e. a true instance is always accepted).

- The definition of PCP requires that for every x in L there exists a proof π_x for which it holds that $\Pr[M^{\pi_x}(x) = 1] = 1$. This means that π_x is potentially different for every x. However, we can assume w.l.o.g., that there exists a proof π which is common to all x's in L. This π will simply be the concatenation of all π_x's (according to some ordering of the x's in L). Since the verifier is polynomial we can assume that all π_x's are at most exponentially long (the verifier cannot access more than an exponentially long prefix of his proof). Therefore, the location of π_x within π will not be more than exponential in $|x|$ away, and so can be accessed in poly$(|x|)$ time.

- The oracle in a PCP system is viewed in a somewhat different manner than previously. We demonstrate this by comparing a PCP system to the mechanism of a *Cook-reduction*. Recall that a language L_1 is *Cook-reducible* to L_2 if there exists an oracle machine M such that for **all** $x \in \{0,1\}^*$ it holds that $M^{L_2}(x) = \chi_{L_1}(x)$. Note that the oracle in the *Cook-reduction* mechanism is the language L_2, and is supposed to exist for **all** $x \in \{0,1\}^*$ (regardless of the question whether x is in L or not). In contrast, in the case of PCP systems the oracle

π is supposed **not** to exist whenever x is not in L. That is, every oracle would cause the verifer to reject x with probability at least $\frac{1}{2}$. Therefore, in the PCP case (as opposed to the *Cook-reduction* case) there is a lack of "symmetry" between the positive instances of L and the negative ones.

12.3 The PCP characterization of NP

12.3.1 Importance of Complexity Parameters in PCP Systems

As was already mentioned in subsection 12.2.2, the class of languages captured by PCP systems varies greatly as the appropriate parameters $r(\cdot)$ and $q(\cdot)$ are modified. This fact is demonstrated by the following assertions:

- If $\mathcal{NP} \subseteq \mathcal{PCP}(o(\log), o(\log))$ then $\mathcal{NP} = \mathcal{P}$
- $\mathcal{PCP}(\text{poly}, \text{poly}) = \mathcal{NEXP}$ $(= \mathcal{NTIME}(2^{\text{poly}}))$

By taking either one of the complexity measures to zero the definition of PCP collapses into one of the following degenerate cases:

- $\mathcal{PCP}(\text{poly}, 0) = \text{co}\mathcal{RP}$
- $\mathcal{PCP}(0, \text{poly}) = \mathcal{NP}$

When looking at the above degenerate cases of the PCP definition we do not really gain any novel view on the complexity classes involved (in this case, co$\mathcal{RP}$ and $\mathcal{NP}$). Thus, the whole point of introducing the PCP definition may be missed. What we would like to see are more delicate assertions involving both non-zero randomness and query complexity. In the following subsection we demonstrate how PCP systems can be used in order to characterize the complexity class $\mathcal{NP}$ in such a non-degenerate way. This characterization will lead to a new perspective on $\mathcal{NP}$ and enable us to further investigate the languages in it.

12.3.2 The PCP Theorem

As already stated, the languages in the complexity class $\mathcal{NP}$ are trivially captured by PCP systems using zero randomness and a polynomial number of queries. A natural question arises: can the two complexity measures be traded off, in a way that still captures the class $\mathcal{NP}$? Most surprisingly, not only the answer to the above question is positive, but also a most powerful result emerges. The number of queries made by the verifier can be brought down to a constant while using only a logarithmic number of coin tosses. This result is known as the PCP theorem (it will be cited here without a proof).

Our goal is to characterize $\mathcal{NP}$ in terms of PCP systems. We start by demonstrating how $\mathcal{NP}$ upper bounds a fairly large class in the PCP hierarchy. This is the class of languages having a PCP system whose verifier makes a polynomial number of queries while using a logarithmic number of coin tosses.

Proposition 12.3.1 $\mathcal{PCP}(\log, \text{poly}) \subseteq \mathcal{NP}$

Proof: Let L be a language in $PCP(\log, \text{poly})$. We will show how to use its PCP system in order to construct a non-deterministic machine M which decides L in polynomial-time. This will imply that L is in $\mathcal{NP}$.

Let M' be the probabilistic-polynomial time oracle machine in the above $\mathcal{PCP}(\log, \text{poly})$ system for L. We are guaranteed that on input $x \in \{0,1\}^*$, M' makes $\text{poly}(|x|)$ queries using $O(\log(|x|))$ coin tosses. For the sake of simplicity, we prove the claim for a non-adaptive M' (in order to adjust the proof to the adaptive case, some minor modifications are required).

Denote by $\langle r_1, \ldots, r_m \rangle$ the sequence of all m possible outcomes of the coin tosses made by M' (note that $|r_i| = O(\log(|x|))$ and $m = 2^{O(\log(|x|))} = \text{poly}(|x|)$). Denote by $\langle q_1^i, \ldots, q_{n_i}^i \rangle$ the sequence of n_i queries made by M when using the coin sequence r_i (note that n_i is potentially different for each i, and is polynomial in $|x|$). Since M' is non-adaptive, its queries are determined as a function of the input x and the coin sequence r_i, and do not depend on answers to previous queries.

By the completeness condition we are guaranteed that for every x in L there exists a PCP proof π_x, such that the verifier M' always accepts x when given access to π_x. A natural candidate for an $\mathcal{NP}$*-witness* for x would be π_x. However, as already stated in subsection 12.2.3, π_x might be of exponential size in $|x|$, and therefore unsuitable to be used as an $\mathcal{NP}$*-witness*. We will therefore use a "compressed" version of π_x, this version corresponds to the portion of the proof which is actually being read by the verifier M'.

We now turn to the construction of a witness w, given $x \in L$ and a corresponding oracle π_x (for the sake of simplicity we denote it by π). Consider all possible executions of M' on input x given access to the oracle string π (each execution depends on the coin sequence r_i). Take the substring of π containing all the bits examined by M' during these executions (i.e. $\left\{\langle \pi_{q_1^i}, \ldots, \pi_{q_{n_i}^i} \rangle\right\}_{i=1}^{m}$). Encode each entry in this substring as $\langle index, \pi_{index} \rangle$ (that is, $\langle query, answer \rangle$), denote the resulting encoded string by w_x^π (note that now $|w_x^\pi|$ is polynomial in $|x|$).

We now describe the non-deterministic machine M which decides L in polynomial time. Given input x, and w on the guess tape, M will simulate the execution of M' on input x for all possible r_i's. Every query made by M' will be answered by M according to the corresponding answers appearing in w (by performing binary search on the indices in w). The machine M will accept *if and only if* M' would have accepted x for **all** possible r_i's.

Since M simulates the execution of M' exactly m times (which is polynomial in $|x|$), and since M' is a polynomial time machine, then M is itself a polynomial-time machine, as required. It remains to be seen that $L(M)$ indeed equals L:

- $\forall x \in L$, we show that there exists w such that $M(x, w) = 1$. By the perfect completeness condition of the PCP system for L, there exists an oracle π such that $\Pr[M'^{\pi}(x) = 1] = 1$. Therefore, it holds that for **all** coin sequences r_i, the machine M' accepts x while accessing π. It immediately follows by definition that $M(x, w_x^\pi) = 1$, where w_x^π is as described above.

- $\forall x \notin L$, we show that for **all** w's it holds that $M(x, w) = 0$. By the soundness condition of the PCP system for L, for all oracles π it holds that $\Pr[M'^{\pi}(x) = 1] \le \frac{1}{2}$. Therefore, for at least $\frac{1}{2}$ of the possible coin sequences r_i, M does not accept x while accessing π. Assume, for the sake of contradiction, that there exists a witness w for which it holds that $M(x, w) = 1$. By the definition of M this means that for all possible coin tosses M' accepts x when given answers from w. We can therefore use w in order to construct an oracle, π^w, for which it holds that $\Pr\left[M'^{\pi^w}(x) = 1\right] = 1$, in contradiction to the soundness condition. (the oracle π^w can be constructed as follows: for every index q that appears in w, define π_q^w to be the binary answer corresponding to q. Define the rest of π^w arbitrarily.)

Consider now the case of an adaptive M'. In this case, we can construct w_x^π adaptively. Given an input $x \in L$ and a corresponding oracle π, run M'^{π} on x for every random string r_i, and see what are the queries made by M' (which depend on x, r_i and answers to previous queries). Then take w_x^π to be the substring of π that is defined by all these queries, as before. ■

The essence of the above proof, is that given a PCP proof (of logarithmic randomness) for some x in L we can efficiently "pack" it (compress it into polynomial size) and transform it into an $\mathcal{NP}$-witness for x. This is due to the fact that the **total** portion of the proof used by the verifier (in all possible runs, i.e. over all possible coin sequences) is bounded by a polynomial. In light of the above, any result of the type

$$\mathcal{NP} \subseteq \mathcal{PCP}(\log, q(\cdot))$$

would be interesting, since it implies that for every $x \in L$, we can construct a witness with the additional property, that enables a "lazy" verifier to toss coins, and decide membership in L, based only on a tiny portion of the $\mathcal{NP}$-witness (as will be further discussed in subsection 12.3.3).

It turns out that the polynomial $q(\cdot)$ bounding the number of queries in a result of the above kind can be taken to be a constant. This surprising result is what we refer to as the PCP theorem.

Theorem 12.3 (The PCP Theorem)

$$\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$$

The PCP theorem is a culmination of a sequence of works, each establishing a meaningful and increasingly stronger statement. The proof of the PCP theorem is one of the most complicated proofs in the theory of computation and it is beyond our scope to prove it here. We state as a side remark, that the smallest possible number of queries for which the PCP theorem has been proven is currently 5 (whereas with 3 queries one can get arbitrarily close to soundness error 1/2).

The conclusion is that $\mathcal{NP}$ is *exactly* the set of languages which have a PCP verifier that asks a constant number of queries using a logarithmic number of coin tosses.

Corollary 12.4 (The PCP Characterization of $\mathcal{NP}$)

$$\mathcal{NP} = \mathcal{PCP}(\log, O(1))$$

Proof: Combining Theorem 12.3 with Proposition 12.3.1, we obtain the desired result. ■

12.3.3 The PCP Theorem gives rise to "robust" $\mathcal{NP}$-relations

Recall that every language L in $\mathcal{NP}$ can be associated with an $\mathcal{NP}$-relation R_L (in case the language is natural, so is the relation). This relation consists of all pairs (x, y) where x is a positive instance of L and y is a corresponding $\mathcal{NP}$-*witness*. The PCP theorem gives rise to another (unnatural) relation R'_L with some extra properties. In the following subsection we briefly discuss some of the issues regarding the relation R'_L.

Since every $L \in \mathcal{NP}$ has a $\mathcal{PCP}(\log, O(1))$ system we are guaranteed that for every x in L there exists a PCP proof π_x, such that the corresponding verifier machine M always accepts x when given access to π_x. In order to define our relation we would like to consider pairs of the form (x, π_x). However, in general, π_x might be of exponential size in $|x|$, and therefore unsuitable to be used in an $\mathcal{NP}$-relation. In order to "compress" it into polynomial size we can use the construction introduced in the proof of Proposition 12.3.1 (i.e. of a witness w for the non-deterministic machine M). Denote by π'_x the resulting "compressed" version of π_x. We are now ready to define the relation:

$$R'_L \stackrel{\text{def}}{=} \{(x, \pi'_x) \mid \Pr[M^{\pi_x}(x) = 1] = 1\}$$

By the definition of PCP it is obvious that $x \in L$ *if and only if* there exists π'_x such that $(x, \pi'_x) \in R'_L$. It follows from the details in the proof of proposition 12.3.1 that R'_L is indeed recognizable in polynomial-time.

Although not stated in the theorem, the proof of the PCP theorem actually demonstrates how to efficiently transform an $\mathcal{NP}$-witness y (for an instance x of $L \in \mathcal{NP}$) into an oracle proof $\pi_{x,y}$ for which the PCP verifier always accepts x. Thus, there is a Levin-reduction between the natural $\mathcal{NP}$-relation for L and R'_L.

We conclude that any $\mathcal{NP}$-witness of R_L can be efficiently transformed into an $\mathcal{NP}$-witness of R'_L (i.e. an oracle proof) which offers a trade-off between the portion of the $\mathcal{NP}$-witness being read by the verifier and the amount of certainty it has in its answer. That is, if the verifier is willing to tolerate an error probability of 2^{-k}, it needs to inspect $O(k)$ bits of the proof (the verifier chooses k random strings $r_1, \ldots, r_k$ uniformly among $\{0,1\}^{O(\log)}$. It will be convinced with probability 2^{-k} that the input x is in L, if for every i, M accepts x using randomness r_i and given oracle access to the appropriate $O(1)$ queries).

12.3.4 Simplifying assumptions about $\mathcal{PCP}(\log, O(1))$ verifiers

When considering a $\mathcal{PCP}(\log, O(1))$ system, some simplifying assumptions about the corresponding verifier machine can be made. We now turn to introduce two of them:

1. Any verifier in a $\mathcal{PCP}(\log, O(1))$ system can be assumed to be non-adaptive (i.e. its queries are determined as a function of the input and the random tape only, and do not depend on answers to previous queries). This is due to the fact that any adaptive $\mathcal{PCP}(\log, O(1))$ verifier can be converted into a non-adaptive one by modifying it in such a way that it will consider **all** possible sequences of $\{0,1\}$ answers given to its queries by the oracle. This certainly costs us in an exponential blowup in the query complexity, but, since the number of queries made by the original (adaptive) verifier is constant, so will be the query complexity of the modified (non-adaptive) verifier after the blowup. Note that in general, adaptive verifiers *are* more powerful than non-adaptive ones (in terms of quantitative results). There are constructions in which adaptive verifiers make less queries than non-adaptive ones while achieving the same results.

2. Any verifier in a $\mathcal{PCP}(\log, O(1))$ system can be assumed to always make the same (constant) number of queries (regardless of the outcome of its coin tosses). Take any verifier in a $\mathcal{PCP}(\log, O(1))$ system not satisfying the above property. Let t be the maximal number of queries made in some execution of the above verifier (over all possible outcomes of the coin tosses). For every possible outcome of the coin tosses, modify the verifier in such a way that it will ask a total number of t queries, make him ignore answers to the newly added queries. Clearly, such a verifier will be consistent with the original one, and will still make only a constant number of queries (which is t).

From now on, whenever referring to $\mathcal{PCP}(\log, O(1))$ systems, free use of the above assumptions will be made (without any loss of generality).

12.4 PCP and non-approximability

Many natural optimization problems are known to be $\mathcal{NP}$-*hard*. However, many times an approximation to the exact value of the solution could be sufficient for our needs. In this section we will investigate the existence (or rather, the inexistence) of efficient approximation algorithms for two $\mathcal{NP}$-*complete* problems, namely, $max3SAT$ and $maxCLIQUE$.

An algorithm for a given problem is considered a C-*approximation* algorithm if for every instance it generates an answer that is off the correct answer by a factor of at most C. The question of interest, is given an $\mathcal{NP}$-*complete* problem Π, what is the best C for which there is a C-approximation algorithm for Π.

The PCP characterization of $\mathcal{NP}$ provides us an alternative view of languages in $\mathcal{NP}$. This view is not as rigid as the original one, and thus creates a framework which is apparently more insightful for the study of approximability.

We start by rephrasing the PCP theorem in an alternative way. This in turn will be used in order to derive an immediate non-approximability result for $max3SAT$. While rephrasing the PCP theorem, a new type of polynomial-time reductions, which we call *amplifying*, emerges.

12.4.1 Amplifying Reductions

Consider an unsatisfiable $3CNF$ formula[1]. It may be the case that the formula is very "close" to being satisfiable. For example, there exist unsatisfiable formulae such that by removing only one of their clauses, they suddenly become satisfiable.

In contrast, there exist unsatisfiable $3CNF$ formulae which are much "farther" from being satisfiable than the above mentioned formulae. These formulae may always have a constant fraction of unsatisfied clauses (for all possible truth assignments). As a consequence, they offer us the (most attractive) feature of being able to probabilistically check whether a certain truth assignment satisfies them or not (by randomly sampling their clauses and picking with constant probability a clause which is unsatisfied by this assignment). Not surprisingly, this resembles the features of a PCP system.

Loosely speaking, amplifying reductions of 3SAT [2] to itself are *Karp-reductions*, which, in addition to the conventional properties, have the property that they map unsatisfiable $3CNF$ formulae into unsatisfiable $3CNF$ formulae which are "far" from being satisfiable (in the above sense).

Definition 12.5 (amplifying reduction) *An* amplifying reduction *of* 3SAT *to itself is a polynomial-time computable function f mapping the set of* $3CNF$ *formulae to itself such that for some constant $\epsilon > 0$ it holds that:*

- *f maps satisfiable $3CNF$ formulae to satisfiable $3CNF$ formulae.*
- *f maps non-satisfiable $3CNF$ formulae to (non-satisfiable) $3CNF$ formulae for which every truth assignment satisfies at most an $1 - \epsilon$ fraction of the clauses.*

An amplifying reduction of a language L in $\mathcal{NP}$ to 3SAT, can be defined analogously.

[1] Recall that a t-CNF formula is a boolean formula consisting of a conjunction of clauses, where each clause is a disjunction of up to t literals (a literal is a variable or its negation).

[2] 3SAT is the problem of deciding whether a given $3CNF$ formula has a satisfying truth assignment.

12.4.2 PCP Theorem Rephrased

Amplifying reductions seem like a suitable tool to be used in order to construct a PCP system for every language in $\mathcal{NP}$. Not only they are efficiently computable, but they enable us to map negative instances of any language in $\mathcal{NP}$ into negative instances of 3SAT which we may be able to reject on a probabilistic basis (analogously to the soundness condition in the PCP definition).

It turns out that the converse is also true, given a PCP system for a language in $\mathcal{NP}$ we are also able to construct an amplifying reduction of 3SAT to itself.

Theorem 12.6 (PCP theorem rephrased) *The following are equivalent:*

1. $\mathcal{NP} \subseteq \mathcal{PCP}(\log, O(1))$. *(The PCP Theorem).*
2. *There exists an amplifying reduction of* 3SAT *to itself.*

Proof: We start with the ((1) $\Rightarrow$ (2)) direction. Consider any language $L \in \mathcal{NP}$. By the PCP theorem L has a $\mathcal{PCP}(\log, O(1))$ system, we will now show how to use this system in order to construct an amplifying reduction from L to 3SAT. This will in particular hold for L = 3SAT (which is itself in $\mathcal{NP}$), and the claim will follow.

Let M be the probabilistic polynomial-time oracle machine in the above $\mathcal{PCP}(\log, O(1))$ system for L. We are guaranteed that on input $x \in \{0,1\}^*$, M makes $t = O(1)$ queries using $O(\log(|x|))$ coin tosses.

Denote by $\langle r_1, \ldots, r_m \rangle$ the sequence of all m possible outcomes of the coin tosses made by M (note that $|r_i| = O(\log(|x|))$ and $m = 2^{O(\log(|x|))} = \mathrm{poly}(|x|)$).

Denote by $\langle q_1^i, \ldots, q_t^i \rangle$ the sequence of t queries made by M when using the coin sequence r_i. As mentioned in subsection 12.3.4, we can assume that M is non-adaptive, therefore its queries are determined as a function of the input x and the coin sequence r_i, and do not depend on answers to previous queries (although not evident from the notation q_j^i, the queries do not depend only on r_i, but on x as well).

We now turn to the construction of the amplifying reduction. Given $x \in \{0,1\}^*$, we construct for each r_i a (constant size) $3CNF$ boolean formula, φ_i^x, describing whether M would have accepted the input x (i.e. describing all possible outputs of M on input x, using the coin sequence r_i). We associate to each query q_j^i a boolean variable $z_{q_j^i}$ whose value should be the answer M gets to the corresponding query. Again, since M is assumed to be non-adaptive, when given its input and coin tosses, M's decision is completely determined by the answers it gets to its queries. In other words, M's decision depends only on the values of $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$.

In order to construct φ_i^x, begin by computing the following truth table: to every possible sequence $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$ assign the corresponding boolean decision of M (i.e. the output of M on input x, using the coin sequence r_i, and given answers $z_{q_j^i}$ to queries q_j^i). Clearly, this can be computed in polynomial-time (by simulating M's execution). Therefore, the whole table can be computed in polynomial-time (since the number of possible assignments to $\langle z_{q_1^i}, \ldots, z_{q_t^i} \rangle$ is 2^t, which is a constant). We can now build a $3CNF$ boolean formula, φ_i^x, which is consistent with the above truth table, this is done in the following way:

1. Construct a t-CNF formula $\psi_i^x = \psi_i^x(z_{q_1^i}, \ldots, z_{q_t^i})$ which is consistent with the truth table.
2. Using a constant number of auxiliary variables, transform it to $3CNF$ (denoted φ_i^x).

Since the table size is constant, the above procedure can be executed in constant time. Note that in the transformation of *t-CNF* formulae into *3CNF* formuae, each clause with t literals is substituted by at most t clauses of 3 literals. Since ψ_i^x consists of exactly 2^t clauses we conclude that the number of clauses in φ_i^x is bounded by $t \cdot 2^t$.

Finally, given φ_i^x for $i = 1, \ldots, m$, we let our amplifying reduction f map $x \in \{0,1\}^*$ into the *3CNF* boolean formula:

$$\varphi^x \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \varphi_i^x$$

Since for every $i = 1, \ldots, m$ the (constant size) formula φ_i^x can be computed in polynomial-time (in $|x|$), and since $m = \text{poly}(|x|)$, it follows that the mapping $f : x \mapsto \varphi^x$ is polynomial-time computable, and $|\varphi^x|$ is polynomial in $|x|$ (note also that the number of clauses in φ^x is bounded by $m \cdot t \cdot 2^t$). It remains to be verified that f is indeed an amplifying reduction:

- $\forall x \in L$, we now show that φ^x is in 3SAT, this happens if and only if the corresponding *t-CNF* formula $\psi^x \stackrel{\text{def}}{=} \bigwedge_{i=1}^{m} \psi_i^x(z_{q_1^i}, \ldots, z_{q_t^i})$ is in t-SAT (recall that ψ_i^x was introduced in the construction of φ_i^x). Since $L \in$ PCP, then there exists an oracle π such that $\Pr[M^\pi(x) = 1] = 1$. Therefore, it holds that for every coin sequence r_i, the machine M accepts x while accessing π. Since ψ_i^x is consistent with the above mentioned truth table it follows that for **all** $i = 1, \ldots, m$, it holds that $\psi_i^x(\pi_{q_1^i}, \ldots, \pi_{q_t^i}) = 1$, and thus ψ^x is in t-SAT. We conclude that φ^x is in 3SAT, as required [3].

- $\forall x \notin L$, we now show that every truth assignment satisfies at most an $1 - \epsilon$ fraction of φ^x's clauses. Since $L \in$ PCP, then for all oracles π it holds that $\Pr[M^\pi(x) = 1] \leq \frac{1}{2}$. Therefore, for at least $\frac{1}{2}$ of the possible coin sequences r_i, machine M does not accept x while accessing π. Put in other words, for each truth assignment (which corresponds to some π) at least $\frac{1}{2}$ of the φ_i^x's are unsatisfiable. Since every unsatisfiable boolean formula always has at least one unsatisfied clause, it follows that for every truth assignment φ^x has at least $\frac{m}{2}$ unsatisfied clauses. Since the number of clauses in φ^x is bounded by $m \cdot t \cdot 2^t$, by taking ϵ to be the constant $\frac{1}{2 \cdot t \cdot 2^t}$ we are guaranteed that every truth assignment satisfies at most an $1 - \epsilon$ fraction of φ^x's clauses.

We now turn to the ((2) $\Rightarrow$ (1)) direction. Under the assumption that there exists an amplifying reduction of 3SAT to itself we will show that the PCP theorem holds. Consider any language $L \in \mathcal{NP}$. Since L is *Karp-reducible* to 3SAT, it is sufficient to show that 3SAT $\in \mathcal{PCP}(\log, O(1))$.

Let $f : 3CNF \to 3CNF$ be an amplifying reduction of 3SAT to itself. And let ϵ be the constant guaranteed by Definition 12.5. We now show how to use f in order to construct a $\mathcal{PCP}(\log, O(1))$ system for 3SAT. We start by giving an informal description of the verifier machine M. Given a certain *3CNF* formula φ, M computes $\varphi' = f(\varphi)$. It then tosses coins in order to uniformly choose one of the clauses of φ'. By querying the oracle string (which should be a possible truth assignment for φ') M will assign truth values to the chosen clause's variables. M will accept if and only if the clause is satisfied. The fact that f is an amplifying reduction implies that whenever M gets a negative instance of 3SAT, with constant probability the chosen clause will not be satisfied. In contrast, this will never happen when looking at a positive instance.

We now turn to a more formal definition of the PCP verifier machine M. On input $\varphi \in 3CNF$ and given access to an oracle string π', M is defined in the following way:

[3]Note that all φ_i^x's have disjoint sets of auxiliary variables, hence transforming a satisfying assignment of ψ^x into a satisfying assignment of φ^x causes no inconsistencies.

1. Find the $3CNF$ formula $\varphi' = \varphi'(x_1, \ldots, x_{n'}) \stackrel{\text{def}}{=} f(\varphi)$.
   $\varphi' = \bigwedge_{i=1}^{m'} c_i$ where c_i denotes a clause with 3 literals.

2. Select a clause c_i of φ' uniformly.
   Denote by $\langle x_{i_1}, x_{i_2}, x_{i_3}\rangle$ the three variables whose literals appear in c_i.

3. Query the values of $\langle \pi'_{i_1}, \pi'_{i_2}, \pi'_{i_3}\rangle$ separately, and assign them to $\langle x_{i_1}, x_{i_2}, x_{i_3}\rangle$ accordingly. Verify the truth value of $c_i = c_i(x_{i_1}, x_{i_2}, x_{i_3})$.

4. Repeat stages 2,3 for $\lceil \frac{1}{\epsilon} \rceil$ times independently (note that $\lceil \frac{1}{\epsilon} \rceil$ is constant).

5. Output 1 if and only if in **all** iterations the truth value of c_i was 1.

Clearly, M is a polynomial-time machine. Note that f is computable in polynomial-time (this also implies that $n', m' = \text{poly}(|\varphi|)$). In addition, the number of iterations executed by M is constant, and in each iteration a polynomial amount of work is executed (depending on n', m' which are, as already mentioned, polynomial in $|\varphi|$).

We turn to evaluate the additional complexity measures involved. In terms of randomness, M needs to uniformly choose $\lceil \frac{1}{\epsilon} \rceil$ numbers in the set $\{1, ..., m'\}$. This involves $O(\log(m')) = O(\log(|\varphi|))$ coin tosses, as required. In terms of queries, the number of queries asked by M is exactly $\frac{3}{\epsilon}$ which is constant, again as required. It remains to examine the completeness and soundness of the above PCP system:

- *completeness*: If $\varphi \in$ 3SAT, then $\varphi' \in$ 3SAT (since f is an amplifying reduction). Therefore there exists a truth assignment, π', such that $\varphi'(\pi') = 1$. Now, since every clause of φ' is satisfied by π', it immediately follows that:
$$\Pr\left[M^{\pi'}(\varphi) = 1\right] = 1$$

- *soundness*: If $\varphi \notin$ 3SAT then any truth assignment for φ' satisfies at most an $1 - \epsilon$ fraction of the clauses. Therefore for any possible truth assignment (oracle) π' it holds that
$$\Pr\left[M^{\pi'}(\varphi) = 1\right] = \Pr\left[\bigwedge_{j=1}^{\lceil \frac{1}{\epsilon} \rceil} \left\{ \begin{array}{c} c_{i_j} \textit{ is satisfied by} \\ \textit{the assignment } \pi' \end{array} \right\}\right] \leq (1-\epsilon)^{\lceil \frac{1}{\epsilon} \rceil} \leq \frac{1}{e} < \frac{1}{2}$$

where the probability is taken over M's internal coin tosses (i.e. over the choice of $i_1, \ldots, i_{\lceil \frac{1}{\epsilon} \rceil}$).
■

Corollary 12.7 *There exists an amplifying reduction of* 3SAT *to itself.*

Proof: Combining the PCP Theorem with Theorem 12.6, we obtain the desired result. ■

12.4.3 Connecting PCP and non-approximability

The characterization of $\mathcal{NP}$ using probabilistic checkable proof systems enabled the area of approximability to make a significant progress.

In general, PCP systems for $\mathcal{NP}$ yield strong non-approximability results for various classical optimization problems. The hardness of approximation is typically established using the notion of gap problems, which are a particular case of promise problems. Recall that a promise problem consists of two sets (A, B), where A is the set of YES instances and B is the set of NO instances. A and B need not be complementary, that is, an instance $x \in \{0,1\}^*$ is not necessarily in either A or B. We demonstrate the notion of a gap problem using the promise problem $gapCLIQUE$ as an example.

Denote by $maxCLIQUE(G)$ the size of the maximal clique in a graph G. Let $gapCLIQUE_{\alpha,\beta}$ be the promise problem (A, B) where A is the set of all graphs G with $maxCLIQUE(G) \geq \alpha$, and B is the set of all graphs G with $maxCLIQUE(G) \leq \beta$. The gap is defined as α/β. Typically, a hardness result will specify a value C of the gap for which the problem is $\mathcal{NP}$-*hard*. This means that there is no efficient algorithm that approximates the $maxCLIQUE$ size of a graph G within a factor of C (unless $\mathcal{NP} = \mathcal{P}$).

The gap versions of various other optimization problems are defined in an analogous way. In this subsection we bring two non-approximability results concerning the problems $max3SAT$ and $maxCLIQUE$ that will be defined in the sequel.

An immediate non-approximability result for max3SAT

Definition 12.8 *(max3SAT): Define max3SAT to be the following problem: Given a $3CNF$ boolean formula φ find the maximal number of clauses that can be simultaneously satisfied by any truth assignment to the variables of φ.*

$max3SAT$ is known to be $\mathcal{NP}$-*hard*. Therefore, approximating it would be desirable. This motivates the definition of the corresponding gap problem:

Definition 12.9 $(gap3SAT_{\alpha,\beta})$: *Let $\alpha, \beta \in [0,1]$ such that, $\alpha \geq \beta$.*
Define $gap3SAT_{\alpha,\beta}$ to be the following promise problem:

- *The YES instances are all $3CNF$ formulae φ, such that there exists a truth assignment which satisfies at least an α-fraction of the clauses of φ.*

- *The NO instances are all $3CNF$ formulae φ, such that every truth assignment satisfies less than a β-fraction of the clauses of φ.*

Note that $gap3SAT_{1,1}$ is an alternative formulation of 3SAT (the decision problem).

The following claim states that, for some $\beta < 1$, it is $\mathcal{NP}$-*hard* to distinguish between satisfiable $3CNF$ formulae and $3CNF$ formulae for which no truth assignment satisfies more than a β-fraction of its clauses. This result implies that there is some constant $C > 1$ such that $max3SAT$ could not be approximated within C (unless $\mathcal{NP} = \mathcal{P}$). The claim is an immediate result of Corollary 12.7.

Claim 12.4.1 *There exists a constant $\beta < 1$, such that the promise problem $gap3SAT_{1,\beta}$ is $\mathcal{NP}$-hard.*

Proof: Let $L \in \mathcal{NP}$. We want to manifest that L is *Karp-reducible* to $gap3SAT_{1,\beta}$.

$3SAT$ is $\mathcal{NP}$-complete, therefore there exists a *Karp-reduction* f_1 from L to $3SAT$. By Corollary 12.7 there exists an amplifying reduction f_2 (and a constant $\epsilon > 0$) from $3SAT$ to itself. Now, take any $1 - \epsilon < \beta < 1$:

- For $x \in L$, $\varphi = f_2(f_1(x))$ is satisfiable, and is therefore a YES instances of $gap3SAT_{1,\beta}$.

- For $x \notin L$, $\varphi = f_2(f_1(x))$ is not satisfiable. Furthermore, for every truth assignment, the fraction of satisfied clauses in φ is at most $1-\epsilon$. Therefore, φ is a NO instance of $gap3SAT_{1,\beta}$.

■

Recently, stronger results were proven. These results show that for every $\beta > 7/8$, the problem $gap3SAT_{1,\beta}$ is $\mathcal{NP}$-*hard*. This means that it is infeasible to come with an efficient algorithm that approximates $max3SAT$ within a factor strictly smaller than 8/7. On the other hand, $gap3SAT_{1,7/8}$ is known to be polynomially solvable, and therefore the 8/7-approximation ratio is tight.

MaxCLIQUE is non-approximable within a factor of two

We briefly review the definitions of the problems $maxCLIQUE$ and $gapCLIQUE_{\alpha,\beta}$:

Definition 12.10 *(maxCLIQUE): Define maxCLIQUE to be the following problem: Given a graph G, find the size of the maximal clique of G (a clique is a set of vertices such that every pair of vertices share an edge).*

$maxCLIQUE$ is known to be $\mathcal{NP}$-*hard*. Therefore, approximating it would be desirable. This motivates the definition of the corresponding gap problem:

Definition 12.11 ($gapCLIQUE_{\alpha,\beta}$) : *Let $\alpha, \beta : N \to N$ be two functions, satisfying $\alpha(n) \geq \beta(n)$ for every n. For a graph G, denote $|G|$ to be the number of vertices in G.*

Define $gapCLIQUE_{\alpha,\beta}$ to be the following promise problem:

- *The YES instances are all the graphs G, with max clique greater than or equal to $\alpha(|G|)$.*
- *The NO instances are all the graphs G, with max clique smaller than or equal to $\beta(|G|)$.*

We conclude our discussion on PCP systems by presenting a nice theorem which demonstrates the hardness of approximating $maxCLIQUE$. The theorem implies that it is an infeasible task to approximate $maxCLIQUE$ within a constant smaller than two (unless $\mathcal{NP} = \mathcal{P}$).

Note, however, that this is not the strongest result known. It has been shown recently that given a graph G of size N, the value of $maxCLIQUE$ is non-approximable within a factor of $N^{1-\epsilon}$ (for every $\epsilon > 0$). This result is tight, since an $N^{1-o(1)}$-approximation algorithm is known to exist (the latter is scarcely better than the trivial approximation factor of N).

Theorem 12.12 *There exists a function $\alpha : N \to N$, such that the promise problem $gapCLIQUE_{\alpha,\alpha/2}$ is $\mathcal{NP}$-hard.*

Proof: Let $L \in \mathcal{NP}$ be some language. We want to show that L is *Karp-reducible* to the language $gapCLIQUE_{\alpha,\alpha/2}$ (for some function $\alpha : N \to N$ which is not dependent on L, rather it is common to all L's).

Loosely speaking, given input $x \in \{0,1\}^*$ we construct in an efficient way a graph G_x having the following property: If x is in L then G_x is a YES instance of $gapCLIQUE$, whereas, if x is not in L then G_x is a NO instance of $gapCLIQUE$.

We now turn to a formal definition of the above mentioned reduction. By the PCP-theorem, L has a $PCP(O(\log), O(1))$ system. Therefore, there exists a probabilistic polynomial-time oracle machine M, that on input $x \in \{0,1\}^*$ makes $t = O(1)$ queries using $O(\log(|x|))$ random coin tosses.

Again, we let $\langle r_1, \ldots, r_m \rangle$ be the sequence of all m possible outcomes of the coin tosses made by M (note that $m = \text{poly}(|x|)$).

Let $\langle q_1^i, \ldots, q_t^i \rangle$ denote the t queries made by M when using the coin tosses r_i, and let $\langle a_1^i, \ldots, a_t^i \rangle$ be a possible sequence of answers to the corresponding queries. We now turn to define a graph G'_x that corresponds to machine M and input x:

vertices: For every possible r_i, the tuple

$$(\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$$

is a vertex in G'_x *if and only if* when using r_i, and given answers a_j^i to queries q_j^i, M accepts x. Note that since M is non-adaptive, once r_i is fixed then so are the queries $\langle q_1^i, \ldots, q_t^i \rangle$. This implies that two vertices having the same r_i, also have the same q_j^i's. Therefore, the number of vertices corresponding to a certain r_i is smaller or equal to 2^t, and the total number of vertices in G'_x is smaller or equal to $m \cdot 2^t$.

edges: Two vertices $v = (\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$ and $u = (\mathbf{r_j}, (\mathbf{q_1^j}, a_1^j), \ldots, (\mathbf{q_t^j}, a_t^j))$ will **not** have an edge between them *if and only if* they are not *consistent*, that is, v and u contain the same query and each one of them has a different answer to this query.

Note that if u and v contain the same randomness (i.e. r_i is equal to r_j) they do not share an edge, since they cannot be consistent (as mentioned earlier, vertices having the same randomness have also the same queries, so u and v must differ in the answers to the queries).

Finally, modify G'_x by adding to it $(m \cdot 2^t - |G'_x|)$ isolated vertices. The resulting graph will have exactly $m \cdot 2^t$ vertices and will be denoted G_x. Note that since the above modification does not add any edges to G'_x, it does not change the size of any clique in G'_x (in particular it holds that $maxCLIQUE(G_x) = maxCLIQUE(G'_x)$).

The above reduction is efficient, since the graph G_x can be constructed in polynomial time: There are at most $m \cdot 2^t$ vertices in G'_x(which is polynomial in $|x|$ since t is a constant), and to decide whether $(\mathbf{r_i}, (\mathbf{q_1^i}, a_1^i), \ldots, (\mathbf{q_t^i}, a_t^i))$ is a vertex in G'_x, one has to simulate machine M on input x, randomness r_i, queries $\{q_j^i\}_{j=1}^t$ and answers $\{a_j^i\}_{j=1}^t$ and see whether it accepts or not. This is, of course, polynomial, since M is polynomial. Finally, deciding whether two vertices share an edge can be done in polynomial time.

Let $\alpha(n) \stackrel{\text{def}}{=} n/2^t$. Since $|G_x| = m \cdot 2^t$, it holds that $\alpha(|G_x|) = m$. It is therefore sufficient to show a reduction from the language L to $gapCLIQUE_{m,m/2}$, this will imply that the promise problem $gapCLIQUE_{\alpha,\alpha/2}$ is $\mathcal{NP}$*-hard*.

- For $x \in L$, we show that G_x contains a clique of size m. By the PCP definition there exists a proof π such that for every random string r, machine M accepts x using randomness r and given oracle access to π. Look at the following set of m vertices in the graph G_x: $S = \{(\mathbf{r_i}, (\mathbf{q_1^i}, \pi_{q_1^i}), \ldots, (\mathbf{q_t^i}, \pi_{q_t^i})) \text{ for } 1 \le i \le m\}$. It is easy to see that all the vertices in S are indeed legal vertices, because π is a proof for x. Also, all the vertices in S must be consistent, because all their answers are given according to π, and therefore, every two vertices in S share an edge. This entails that S is an m-clique in G_x, and therefore G_x is a YES instance of $gapCLIQUE_{m,m/2}$.

- For $x \notin L$, we show that G_x does not contain a clique of size greater than $m/2$. Suppose, in contradiction, that S is a clique in G_x of size greater than $m/2$. Define now the following proof π: For every query and answer (q, a) in one of the vertices of S define $\pi_q = a$. For every other query (which is not included in either of the vertices of S) define π_q to be an arbitrary value in {0,1}. Since S is a clique, all its vertices share an edge and are therefore consistent. Note that π is well defined, the consistency requirement implies that same queries have same answers (for all queries and answers appearing in some vertex in S). Therefore, it cannot be the case that we give two incosistent values to the same entry in π during its construction. Now, since all the vertices of S have different r_i's and $|S|$ is greater than $m/2$, it holds that for more than $\frac{1}{2}$ of the possible coin sequences r_i, machine M accepts x while accessing π. In other words, $\Pr[M^{\pi}(x) = 1] > 1/2$, in contradiction to the soundness condition. We conclude that indeed G_x does not have a clique of size greater than $m/2$, and is therefore a NO instance of $gapCLIQUE_{m,m/2}$.

■

Bibliographic Notes

The PCP Characterization Theorem is attributed to Arora, Lund, Motwani, Safra, Sudan and Szegedy (see [2] and [1]). These papers, in turn, built on numerous previous works; for details see the papers themselves or [4]. In general, our presentation of PCP follows follows Section 2.4 of [4], and the interested reader is referred to [4] for a survey of further developments and more refined considerations.

The first connection between PCP and hardness of approximation was made by Feige, Goldwasser, Lovasz, Safra, and Szegedy [3]: They showed the connection to maxClique (presented above). The connection to max3SAT and other "MaxSNP approximation" problems was made later in [1].

We did not present the strongest known non-approximability results for max3SAT and maxClique. These can be found in Hastad's papers, [6] and [5], respectively.

1. S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *JACM*, Vol. 45, pages 501–555, 1998.

2. S. Arora and S. Safra. Probabilistic Checkable Proofs: A New Characterization of NP. *JACM*, Vol. 45, pages 70–122, 1998.

3. U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating Clique is almost NP-complete. *JACM*, Vol. 43, pages 268–292, 1996.

4. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

5. J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. To appear in *ACTA Mathematica.* Preliminary versions in *28th STOC* (1996) and *37th FOCS* (1996).

6. J. Hastad. Getting optimal in-approximability results. In *29th STOC*, pages 1–10, 1997.

Lecture 13

Pseudorandom Generators

Notes taken by Sergey Benditkis, Boris Temkin and Il'ya Safro

Summary: Pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. The latter are indistiguishable from truely random sequences by any efficient observer. We show that, for efficiently sampleable distributions, computational indistiguishability is preserved under multiple samples. We related pseudorandom generators and one-way functions, and show how to increase the stretching of pseudorandom generators.

13.1 Instead of an introduction

Oded's Note: *See introduction and motivation in the Appendix. Actually, it is recommended to read the appendix before reading the following notes, and to refer to the notes only for full details of some statements made in Sections 13.6.2 and 13.6.3 of the appendix.*

Oded's Note: *Loosely speaking, pseudorandom generators are defined as efficient deterministic algorithms which stretch short random seeds into longer pseudorandom sequences. We stress three aspects: (1) the efficiency of the generator; (2) the stretching of seeds to longer strings; (3) the pseudorandomness of output sequences. The third aspect refers to the key notion of computational indistinguishability. We start with a definition and discussion of the latter.*

13.2 Computational Indistinguishability

We have two things which are called "probability ensembles", and denoted by $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$. We are talking about infinite sequences of distributions like we talk about the language and each distribution seats on some final domain. Typicaly the distribution X_n will have as a support strings of length polynomial of n, not more and not much less.

Definition 13.1 (probability ensembles): *A* probability ensemble *X is a family $X = \{X_n\}_{n\geq 1}$ such that X_n is a probability distribution on some finite domain.*

What is to say that these ensembles are computational indistinguishable? We want to look at the particular algorithm A and want to ask what is the probability of the event: when you give to A

an input X_n then it says 1 (or 1 is just arbitrary) and look at the difference of the probabilities for answers of execution this algorithm A for two inputs $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$. And if this difference is negligible, when you look at n as the parameter then we will say that we can not distinguish the first ensamble from the second one.

Definition 13.2 (negligible functions): *The function $f : \mathsf{N} \mapsto [0,1]$ is* negligible *if for all polynomials p, and for all sufficiently large n's, $f(n) < 1/p(n)$.*

Suppose we have two *probability ensambles* $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$, where X_n and Y_n are distributions over some finite domain.

Definition 13.3 (indistinguishability by a specific algorithm): *Consider some probabilistic algorithm A. We will say that $\{X_n\}$ and $\{Y_n\}$ are* indistinguishable by A *if*

$$|\Pr(A(X_n) = 1) - \Pr(A(Y_n) = 1)| < \frac{1}{p(n)}$$

for every polinomial $p()$ and for every sufficiently large n.

13.2.1 Two variants

Our main focus will be on indistinguishability by any probabilistic polynomial-time algorithm. That is,

Definition 13.4 (canonic notion of computational indistinguishability): *Two probability ansambles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are* computationally indistinguishable *if they are indistinguishable by any probabilistic polynomial-time algorithm. That is, for every probabilistic polynomial-time algorithm A, and every polinomial $p()$ there exists N s.t. for all $n > N$*

$$|\Pr(A(X_n) = 1) - \Pr(A(Y_n) = 1)| < \frac{1}{p(n)}$$

Another notion that we talk about is indistinguhability by circuits.

Definition 13.5 *Two probability ensembles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are* indistinguishable by small circuits *if for all families of polynomial-size circuits $\{C_n\}$*

$$|\,[\Pr(C_n(X_n) = 1) - \Pr(C_n(Y_n) = 1)]\,|$$

is neglibible.

13.2.2 Relation to Statistical Closeness

Oded's Note: *This subsection was rewritten by me.*

The notion of Computational Indistinguishability is a relaxation of the notion of *statistical closeness* (or *statistical indistinguishability*).

Definition 13.6 (statistical closeness): *The* statistical difference (or variation distance) *between two distributions, X and Y, is defined by*

$$\Delta(X,Y) \stackrel{\text{def}}{=} \frac{1}{2} \cdot \sum_{\alpha} |\Pr[X=\alpha] - \Pr[Y=\alpha]|$$

Two probability ensembles $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$ are statistical close *if $\Delta(X_n,Y_n)$ is a negligible function of n. That is, for all polynomial $p()$ there exists N s.t. for all $n > N$ $\Delta(X_n,Y_n) < 1/p(n)$.*

An equivalent definition of $\Delta(X_n,Y_n)$, is the maximum over all subsets, S, of $\Pr[X_n \in S] - \Pr[Y_n \in S]$. (A set S which obtains the maximum is the set of all z's satisfying $\Pr[X_n = z] > \Pr[Y_n = z]$, which proves the equivalence.) Yet another equivalent definition of $\Delta(X_n,Y_n)$ is the maximum over all Boolean f's of $\Pr[f(X_n)=1] - \Pr[f(Y_n)=1]$. Thus,

Proposition 13.2.1 *If two probability ensembles are statistical close then they are computationally indistinguishable.*

We note that there are computationally indistinguishable probability ensembles which are not statistical close.

13.2.3 Computational indistinguishability and multiple samples

> Oded's Note: *We show that under certain conditions, computational indistinguishability is preserved under multiple samples.*

Definition 13.7 (constructability of ensembles): *The ensemble $\{Z_n\}_{n\in\mathsf{N}}$ is* probabilistic polynomial-time constructable *if there exists a probabilistic polynomial time algorithm S such that for every n, $S(1_n) \equiv Z_n$.*

Theorem 13.8 *Let $\{X_n\}$ and $\{Y_n\}$ computationally indistinguishable (i.e., indistinguishable by any probabilistic polynomial time algorithm). Suppose they are both probabilistic polynomial time constructable. Let $t()$ be a positive polinomial. Define $\{\overline{X_n}\}_{n\in N}$ and $\{\overline{Y_n}\}_{n\in N}$ in the following way:*

$$\overline{X_n} = X_n^1 \circ X_n^2 \circ \ \ldots \ \circ X_n^{t(n)}, \quad \overline{Y_n} = Y_n^1 \circ Y_n^2 \circ \ \ldots \ \circ Y_n^{t(n)}$$

The X_n^i's (resp. Y_n^i's) are independent *copies of X_n (Y_n). Then $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$ are Probabilistic Polynomial Time indistinguishable.*

Proof: Suppose, there exists a distinguisher D, between $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$.

> Oded's Note: *We use the "hybrid technique": We define hybrid distributions so that the extreme hybrids coincide with $\{\overline{X_n}\}$ and $\{\overline{Y_n}\}$, and link distinguishability of neighboring hybrids to distinguishability of $\{X_n\}$ and $\{Y_n\}$.*

Then define

$$H_n^{(i)} \ = \ \left(X_n^{(1)} \circ X_n^{(2)} \circ \ \ldots \ X_n^{(i)} \circ Y_n^{(i+1)} \circ \ \ldots \ Y_n^{(t(n))}\right)$$

It is easy to see that $H_n^{(0)} = \overline{Y_n}$, $H_n^{(t(n))} = \overline{X_n}$.

> Oded's Note: *Also note that $H_n^{(i)}$ and $H_n^{(i+1)}$ differ only in the distribution of the $i+1^{\text{st}}$ component, which is identical to Y_n in the first hybrid and to X_n in the second. The idea is to distinguish Y_n and X_n by pluging them in the $i+1^{\text{st}}$ component of a distribution. The new distribution will be distributed identically to either $H_n^{(i)}$ or $H_n^{(i+1)}$, respectively.*

Define algorithm D' as follows:

Begin Algorithm Distinguisher
Input α , (taken from X_n or Y_n)
(1) Choose $i \in_R \{1 \,..\, t(n)\}$ (i.e., uniformly in $\{1, ..., t(n)\}$)
(2) Construct $Z = \left(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ \alpha \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))} \right)$
Return $D(Z)$
end.

$$\begin{aligned} \Pr\left[D'(X_n)=1\right] &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ X_n \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}) = 1\right] \\ &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(H_n^{(i)}) = 1\right] \end{aligned}$$

whereas

$$\begin{aligned} \Pr\left[D'(Y_n)=1\right] &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(X_n^{(1)} \circ X_n^{(2)} \circ \ ... \ X_n^{(i-1)} \circ Y_n \circ Y_n^{(i+1)} \circ \ ... \ Y_n^{(t(n))}) = 1\right] \\ &= \frac{1}{t(n)} \sum_{i=1}^{t(n)} \Pr\left[D(H_n^{(i-1)}) = 1\right]. \end{aligned}$$

Thus,

$$\begin{aligned} & \left|\Pr\left[D'(X_n)=1\right] - \Pr\left[D'(Y_n)=1\right]\right| \\ &= \frac{1}{t(n)} \cdot \left| \sum_{i=1}^{t(n)} \Pr[D(H_n^{(i)})=1] - \sum_{i=1}^{t(n)} \Pr[D(H_n^{(i-1)})=1] \right| \\ &= \frac{1}{t(n)} \cdot \left| \Pr[D(H_n^{(t(n))})=1] - \Pr[D(H_n^{(0)})=1] \right| \\ &= \frac{1}{t(n)} \cdot \left| \Pr[D(X_n)=1] - \Pr[D(Y_n)=1] \right| \ \geq \ \frac{1}{t(n) \cdot p(n)} \end{aligned}$$

for some $p()$ and for infinitely many n's ■

Oded's Note: *One can easily show that computational indistinguishability by small circuits is preserved under multiple samples. Here we don't need to assume probabilistic polynomial-time constructability of the ensembles.*

13.3 PRG: Definition and amplification of the stretch function

Intuitively, a pseudo-random generator takes a short, truly-random string and stretches it into a long, pseudorandom one. The pseudorandom string should look "random enough" to use it in place of a truly random string.

Definition 13.9 (PseudoRandom Generator – PRG): *The function $G : \{0,1\}^* \rightarrow \{0,1\}^*$ with stretch function $l(n)$ is a* pseudo-random generator *if:*

- *G is a polynomial time algorithm*
- *for every x, $|G(x)| = l(|x|) > |x|$*
- *$\{G(U_n)\}$ and $\{U_{l(n)}\}$ are computational indistinguishable, where U_m denotes the uniform distribution over $\{0,1\}^m$.*

  Oded's Note: *The above definition is minimalistic regarding its stretch requirement. A generator stretching n bits into $n+1$ bits seems to be of little use. However, as shown next, such minimal stretch generators can be used to construct generators of arbitrary stretch.*

Theorem 13.10 (amplification of stretch function): *Suppose we have a Pseudo-Random Generator G_1 with a stretch function $n + 1$. Then for all polynome $l(n)$ there exists a Pseudo-Random Generator with stretch function $l(n)$.*

Proof:

Construct G as follows: We take the input seed x ($|x| = n$) and feed it through G_1. Then we save the first bit of the output of G_1 (denote it by y_1), and feed the rest of the bits as input to a new invocation of G_1. We repeat this operation $l(n)$ times, in the i-th step we invoke G_1 on input determined in the previous step, save the first output bit as y_i and use the rest n bits as an input to step $i+1$. The output of G is $y = y_1 y_2 \ldots y_{l(n)}$. See Figure 1.

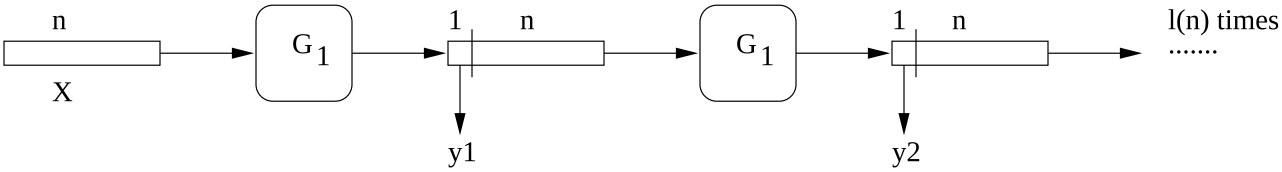


Figure 1.

We claim that G is a Pseudo-Random Generator. The first two requirements for Pseudo-Random Generator are trivial (by construction/definition of G). We will prove the 3rd one. The proof is by contradiction, again using the hybrid method.

Suppose there exists a distinguisher $A : \{0,1\}^* \rightarrow \{0,1\}^{l(n)}$ such that exists polynomial $p()$ and for infinitely namy n's

$$\left| \Pr[\, A(G(U_n)) = 1 \,] \; - \; \Pr\Big[\, A(U_{l(n)}) = 1 \,\Big] \right| \;\geq\; \frac{1}{p(n)}$$

Let us make the following construction . Define sequence of functions $g^{(i)}$:

$g^{(0)}$ is empty

$g^{(i)} = [G_1(x)]_1 \circ g^{(i-1)}([G_1(x)]_{2\ldots(n+1)})$

Where $[y]_i$ is the notation of i-th bit of y and $[y]_{2\ldots(n+1)}$ denotes substring of y from the second bit up to $n+1$-th bit. It is easy to see that $g^{l(n)} = G(x)$.

Construct the class of hybrid distributions $\{H^i\}_{i=1}^{l(n)}$:

$$H^i = U_{l(n)-i} \circ g^i(U_n)$$

One can obeserve that $H^0 = G(U_n)$, and $H^{l(n)} = U_{l(n)}$.
Now we construct the distinguisher D as follows:

Begin Algorithm Distinguisher
Input α , $|\alpha| = n+1$ (taken from $G_1(U_n)$ on U_{n+1})
(1) Choose $i \in_R \{1 .. l(n)\}$
(2) Choose $Z \sim U_{l(n)-i}$
(3) Construct $y = Z \circ \sigma \circ g^{(i-1)}(S)$, where σ is first bit of α and S its n-bit suffix
Return $A(y)$
end.

We denote by $\Pr[A|i]$ the *conitional probability* of event A if particular i was choosen in step (1) of algorithm D. We see that

$$\begin{aligned} & \Pr[\ D(G_1(U_n)) = 1\ ] \ - \Pr[\ D(U_{n+1}) = 1\ ] \\ = & \ \frac{1}{l(n)} \sum_{i=1}^{l(n)} (\ \Pr[\ D(G_1(U_n)) = 1\ |i] \ - \Pr[\ D(U_{n+1}) = 1\ |i]\ ) \,. \qquad (**) \end{aligned}$$

Note that

$$\begin{aligned} \Pr[\ D(G_1(U_n)) = 1|i] &= \Pr\Big[\ A(Z_{1..(l(n)-i)} \circ [G_1(U_n)]_1 \circ g^{(i-1)}([G_1(U_n)]_{(2..n+1)}) = 1\ \Big] \\ &= \Pr\Big[\ A(H^i) = 1\ \Big] \end{aligned}$$

and

$$\begin{aligned} \Pr[\ D(U_{n+1}) = 1|i] &= \Pr\Big[\ A(Z_{1..(l(n)-i)} \circ [U_{n+1}]_1 \circ g^{(i-1)}([U_{n+1}]_{2..n+1})) = 1\ \Big] \\ &= \Pr\Big[\ A(H^{i-1}) = 1\ \Big] \end{aligned}$$

So equation $(**)$ is

$$\begin{aligned} & \frac{1}{l(n)} \sum_{i=1}^{l(n)} \Big(\ \Pr\Big[\ A(H^i) = 1\ \Big] \ - \Pr\Big[\ A(H^{i-1}) = 1\ \Big]\Big) \\ = & \ \frac{1}{l(n)} \Big(\ \Pr\Big[\ A(H^{l(n)}) = 1\ \Big] \ - \Pr\Big[\ A(H^0) = 1\ \Big]\ \Big) \\ = & \ \frac{1}{l(n)} \Big(\ \Pr[\ A(G(U_n)) = 1\ ] \ - \Pr\Big[\ A(U_{l(n)}) = 1\ \Big]\ \Big) \end{aligned}$$

so

$$\Pr[\ D(G_1(U_n)) = 1\ ] \ - \Pr[\ D(U_{n+1}) = 1\ ] \ \geq \ \frac{1}{l(n)p(n)}$$

■

13.4 On Using Pseudo-Random Generators

Suppose we have a probabilistic polynomial time algorithm A, which on input of length n uses $m(n)$, random bits. Algorithm A may solve either search problem for some relation or decision problem for some language L. Our claim will be that for all $\varepsilon > 0$ there exists a probabilistic polynomial time algorithm A' that uses only n^ε random bits and "behaves" in the same way that A does.

The construction of A' bases on assumption that we are given pseudo-random generator $G : \{0,1\}^{n^\varepsilon} \to \{0,1\}^{m(n)}$. Recall that $A(x,R)$ that A is running on input x with coins R.

Algorithm A'
Input $x \in \{0,1\}^n$
Choose $S \in_R \{0,1\}^{n^\varepsilon}$
$R \leftarrow G(s)$ (generate the coin tosses)
Return $A(x,R)$ (run A on input x using coins R)
end.

Proposition 13.4.1 (imformal): *It is infeasible given 1^n to find $x \in \{0,1\}^n$, such that the "behaviour" of $A'(x)$ is substantially diferent from $A(x)$.*

The meaning of this proposition depends on the computational problem solved by A. In case A solves some NP-search problem, the proposition asserts that it is hard (i.e., feasible only with negligable probability) to find large x's such that A can find the solution for x, and $A'(x)$ will fail to do so. In case A computes some function the proposition applies too.

> Oded's Note: *But the proposition may not hold if A solves a search problem in which instances have many possible solutions and are not efficiently verifiable (as in NP-search problems).*

Below we prove the proposition for the case of decision problems (and the proof actually extends to any function computation). We assume that A gives the correct answer with probability bounded away from 1/2.

Proof: Suppose we have a finder F, which works in polynomial time , $F(1^n) = x \in \{0,1\}^n$, such that

$$\Pr\left[A'(x) = X_L(x)\right] \;\le\; \frac{1}{2}$$

where X_L is the characteristic function of a language decideable by A (i.e., $\Pr[A(x) = X_L(x)] \ge 2/3$ for all x's). Construct a distinguisher D as follows:

Begin Algorithm D
Input $\alpha \in \{0,1\}^{m(n)}$
$x \leftarrow F(1^n)$
$v \leftarrow X_L(x)$ with overwhelmingly high probability (i.e., invoke $A(x)$ polynomially many times and take a majority vote).
$w \leftarrow A(x,\alpha)$
If $v = w$ **Then** **Return** 1
Else **Return** 0

end.

D is contradicts the pseudorandomness of G because

$$A(x,\alpha) = \begin{cases} X_L(x), \ w.p. \ \geq \frac{2}{3}, & \alpha \sim U_{m(n)} \\ X_L(x) = A'(x), \ w.p. \ \leq \frac{1}{2}, & \alpha \sim G(U_{n^\varepsilon}) \end{cases} \tag{13.1}$$

Furthermore, with probability at least 0.99, the value v found by $D(x)$ equals $X_L(x)$. Thus,

$$\begin{aligned} \Pr[D(U_{m(n)}) = 1] &> 0.66 - 0.01 = 0.65 \\ \Pr[D(G(U_{n^\epsilon})) = 1] &\leq 0.5 + 0.01 < 0.55 \end{aligned}$$

which provides the required gap. ■

Oded's Note: *Note that for a strong notion of pseudorandom generators, where the output is indistinguishable from random by small circuits we can prove a stronger result; that is, that there are only finitely many x's on which A' behaves differently than A. Thus, in case of decision algorithms, by minor modification to A', we can make A' accept the same language as A.*

13.5 Relation to one-way functions

An important property of a pseudo-random generator $G(S)$ that it turns the seed into the sequence $x = G(S)$ in polynomial time. But the inverse operation of finding the seed S from $G(S)$ would be hard (or else pseudorandomness is violated as shown below). A pseudo-random generator is however, not just a function that hard to invert it also stretches the input into the larger sequence that look random. Still pseudo-random generators can be related to functions which are "only" easy to compute and hard to invert, as defined next.

Definition 13.11 (One-way functions – OWF): *A function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that $\forall x \ |f(x)| = |x|$ is* one-way *if :*

- *there is exists polynomial time algorithm A, such that $\forall x \ A(x) = f(x)$*
- *for all probabilistic polynomial time A' and for all polynome $p()$ and for all sufficiently large n's :*
$$\Pr\left[A'(f(U_n)) = f^{-1} \circ f(U_n)\right] < \frac{1}{p(n)}$$

In other words this function must be easy computed and hard inverted. Note an important feauture: the inversion algotihm must fail almost always. But the probability distribution used here is not uniform over all $f(x)$; rather, it is the distribution $f(x)$ when x is choosen uniformly.

Oded's Note: *The requirement that the function be length preserving (i.e., $|f(x)| = |x|$ for all x's) may be relaxed as long as the length of $f(x)$ is polynomially related to $|x|$. In contrast a function like $f(x) \stackrel{\text{def}}{=} |x|$ would be "one-way" for a trivial and useless reason (on input n in binary one cannot print an n-bit string in polynomial (in $\log n$) time).*

Comment: A popular candidate to be one-way function is based on the conjectured intractability of the integer factorization problem. The length of input and output to the function will not be exactly n, only polynomial in n:

The factoring problem. Let $x, y > 1$ be n-bit integers. Define

$$f(x, y) = x * y$$

When x, y are n-bit primes, it is believed that finding x, y from $x * y$ is computationaly difficult.

> Oded's Note: *So the above should be hard to invert in these cases which occur at density $\approx 1/n^2$. This does not satisfy the definition of one-wayness which requires hardness of inversion almost everywhere, but suitable amplification can get us there. Alternatively, we can redefine the function f so that $f(x, y) = prime(x) * prime(y)$, where $prime()$ is a procedure which uses the input string to generate a large prime so that when the input is a random n-bit string the output is a random $n/O(1)$-bit prime. Such efficient procedures are known to exist. Using less sophisticated methods one can easily construct a procedure which uses n-bits to produce a prime of length $\sqrt[3]{n}/O(1)$.*

Theorem 13.12 *Pseudo-Random Generators exist if and only if One-Way Functions exist.*

So the computational hardness and pseudorandomness are strongly connect each other. If we have the created randomness we can create the hardness, and vice versa. Let us prove one part of the theorem and give hints to special case of other.

PRG $\Longrightarrow$ OWF: Consider pseudo-random generator $G : \{0,1\}^n \rightarrow \{0,1\}^{2n}$. Let us define function $f : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ as follows:

$$f(xy) = G(x) \qquad (|x| = |y| = n).$$

We claim, that f is one-way function, and the proof is by contradiction :

Suppose probabilistic polynomial time algorithm A' inverts f with success probability greater than $\frac{1}{p(n)}$, where $p(n)$ is polynom.

Consider a distinguisher D :
input: α, $\alpha \in \{0,1\}^{2n}$
$xy \leftarrow A'(\alpha)$
if $f(xy) = \alpha$ return 1
otherwise return 0.

$$\begin{aligned}
\Pr[D(G(U_n)) = 1] &= \Pr[D(f(U_n)) = 1] \\
&= \Pr[f(A'(f(U_n))) = f(U_n)] \\
&= \Pr\left[A'(f(U_n)) \in f^{-1}f(U_n)\right] \\
&> \frac{1}{p(n)}
\end{aligned}$$

where the last inequality is due to the contradiction hypothesis. On the other hand, there are at most 2^n strings of length $2n$ which have a preimage under G (and so under f). Thus, a uniformly selected random string of length $2n$ has a preimage under f with probability at most $2^n/2^{2n}$. It follows that

$$\begin{aligned}\Pr\left[D(U_{2n})=1\right] &= \Pr\left[f(A'(U_{2n}))=U_{2n}\right] \\ &\leq \Pr[U_{2n} \text{ is in the image of } f] \\ &\leq \frac{2^n}{2^{2n}} \;=\; 2^{-n}\end{aligned}$$

Thus,

$$\Pr\left[D(G(U_n))=1\right]-\Pr\left[D(U_{2n})=1\right] > \frac{1}{p(n)}-\frac{1}{2^n} > \frac{1}{q(n)}$$

For some polynome $q()$

OWF $\Longrightarrow$ PRG:

> Oded's Note: *The rest of this section is an overview of what is shown in detail in the next lecture (i.e., Lecture 14).*

Let us demonstrate the other direction and build an Pseudo-Random Generator if we have OWF of special form. Suppose the function $f:\{0,1\}^n \rightarrow \{0,1\}^n$ is not only OWF but it is also $1-1$. So it is a permutation of strings of length n. Assume that we can get a random bit b from the input, such that b will be hard to "predict" from the output of f. In this case we can construct a Pseudo-Random Generator as a concatenation of $f(x)$ and b.

Definition 13.13 (Hardcore):
Let f be one-way function, $b:\{0,1\}^* \longrightarrow \{0,1\}$ is a hardcore of f if:

- *$\exists$ polinomial time algorithm A, such that $\forall t A(t)=b(t)$*
- *$\forall$ probabilistic polynomial time algorithm A' $\forall$ polynom $p(.)$ $\forall$ sufficiently large $n's$*

$$\Pr\left[A'(f(U_n))=b(U_n)\right] < \frac{1}{2}+\frac{1}{p(n)}$$

In other words this function must be easy to compute and hard to predict out of $f(x)$.

The following theorem can be proven:

Theorem 13.14 *If f is OW, $f'(x,y) \;=\; f(x)\circ y,\;(|x|=|y|)$ then $b(x,y)=\sum_{i=1}^{n} x_i y_i \;(mod\; 2)$ is a hardcore of f.*

This theorem would be proven in next lecture. Now we can construct a Pseudo-Random Generator G as follows:

$$G(s) \;=\; f'(s)\circ b(s)$$

The two first properties of G (poly-time and stretching) are trivial. The pseudorandomness of G follows from the fact that its first n output bits are uniformly distributed and the last bit is unpredictable. Unpredictability translates to indistinguishability, as will be shown in the next lecture.

Bibliographic Notes

The notion of computational indistinguishability was introduced by Goldwasser and Micali [4] (within the context of defining secure encryptions), and given general formulation by Yao [6]. Our definition of pseudorandom generators follows the one of Yao, which is equivalent to a prior formulation of Blum and Micali [1]. For more details regarding this equivalence, as well as many other issues, see [2]. The latter source presents the notion of pseudorandomness discussed here as a special case (or archetypical case) of a general paradigm.

The discovery that computational hardness (in form of one-wayness) can be turned into a pseudorandomness was made in [1]. Theorem 13.12 (asserting that psedorandom generators can be constructed based on any one-way function) is due to [5]. It uses Theorem 13.14 which is due to [3].

1. M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SICOMP*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd FOCS*, 1982.

2. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

3. O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st STOC*, pages 25–32, 1989.

4. S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

5. J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. Construction of Pseudorandom Generator from any One-Way Function. To appear in *SICOMP*. Preliminary versions by Impagliazzo et. al. in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

6. A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd FOCS*, pages 80–91, 1982.

Oded's Note: *Being in the process of writing an essay on pseudorandomness, it feels a good idea to augment the notes of the current lecture by a draft of this essay. The lecture notes actually expand on the presentation in Sections 13.6.2 and 13.6.3. The other sections in this essay go beyond the lecture notes.*

Appendix: An essay by O.G.

Summary: We postulate that a distribution is pseudorandom if it cannot be told apart from the uniform distribution by an efficient procedure. This yields a robust definition of pseudorandom generators as efficient deterministic programs stretching short random seeds into longer pseudorandom sequences. Thus, pseudorandom generators can be used to reduce the randomness-complexity in any efficient procedure. Pseudorandom generators and computational difficulty are strongly related: loosely speaking, each can be efficiently transformed into the other.

13.6.1 Introduction

The second half of this century has witnessed the development of three theories of randomness, a notion which has been puzzling thinkers for ages. The first theory (cf., [4]), initiated by Shannon [21], is rooted in probability theory and is focused at distributions which are not perfectly random. Shannon's Information Theory characterizes perfect randomness as the extreme case in which the *information content* is maximized (and there is no redundancy at all). Thus, perfect randomness is associated with a unique distribution – the uniform one. In particular, by definition, one cannot generate such perfect random strings from shorter random seeds.

The second theory (cf., [16, 17]), due to Solomonov [22], Kolmogorov [15] and Chaitin [3], is rooted in computability theory and specifically in the notion of a universal language (equiv., universal machine or computing device). It measures the complexity of objects in terms of the shortest program (for a fixed universal machine) which generates the object. Like Shannon's theory, Kolmogorov Complexity is quantitative and perfect random objects appear as an extreme case. Interestingly, in this approach one may say that a single object, rather than a distribution over objects, is perfectly random. Still, Kolmogorov's approach is inherently intractable (i.e., Kolmogorov Complexity is uncomputable), and – by definition – one cannot generate strings of high Kolmogorov Complexity from short random seeds.

The third theory, initiated by Blum, Goldwasser, Micali and Yao [12, 1, 24], is rooted in complexity theory and is the focus of this essay. This approach is explicitly aimed at providing a notion of perfect randomness which nevertheless allows to efficiently generate perfect random strings from shorter random seeds. The heart of this approach is the suggestion to view objects as equal if they cannot be told apart by any efficient procedure. Consequently a distribution which cannot be efficiently distinguished from the uniform distribution will be considered as being random (or rather called pseudorandom). Thus, randomness is not an "inherent" property of objects (or distributions) but rather relative to an observer (and its computational abilities). To demonstrate this approach, let us consider the following mental experiment.

> Alice and Bob play "head or tail" in one of the following four ways. In all of them Alice flips a coin high in the air, and Bob is asked to guess its outcome *before* the coin hits the floor. The alternative ways differ by the knowledge Bob has before making his guess. In the first alternative, Bob has to announce his guess before Alice flips the coin. Clearly, in this case Bob wins with probability 1/2. In the second alternative,

Bob has to announce his guess while the coin is spinning in the air. Although the outcome is *determined in principle* by the motion of the coin, Bob does not have accurate information on the motion and thus we believe that also in this case Bob wins with probability 1/2. The third alternative is similar to the second, except that Bob has at his disposal sophisticated equipment capable of providing accurate *information* on the coin's motion as well as on the environment effecting the outcome. However, Bob cannot process this information in time to improve his guess. In the fourth alternative, Bob's recording equipment is directly connected to a *powerful computer* programmed to solve the motion equations and output a prediction. It is conceivable that in such a case Bob can improve substantially his guess of the outcome of the coin.

We conclude that the randomness of an event is relative to the information and computing resources at our disposal. Thus, a natural concept of pseudorandomness arises – a distribution is *pseudorandom* if no efficient procedure can distinguish it from the uniform distribution, where efficient procedures are associated with (probabilistic) polynomial-time algorithms.

13.6.2 The Definition of Pseudorandom Generators

Loosely speaking, a pseudorandom generator is an *efficient* program (or algorithm) which *stretches* short random seeds into long *pseudorandom* sequences. The above emphasizes three fundamental aspects in the notion of a pseudorandom generator:

1. *Efficiency*: The generator has to be efficient. We associate efficient computations with those conducted within time which is polynomial in the length of the input. Consequently, we postulate that the generator has to be implementable by a deterministic polynomial-time algorithm.

   This algorithm takes as input a *seed*, as mentioned above. The seed captures a bounded amount of randomness used by a device which "generates pseudorandom sequences." The formulation views any such device as consisting of a deterministic procedure applied to a random seed.

2. *Stretching*: The generator is required to stretch its input seed to a longer output sequence. Specifically, it stretches n-bit long seeds into $\ell(n)$-bit long outputs, where $\ell(n) > n$. The function ℓ is called the *stretching measure* (or *stretching function*) of the generator.

3. *Pseudorandomness*: The generator's output has to look random to any efficient observer. That is, any efficient procedure should fail to distinguish the output of a generator (on a random seed) from a truly random sequence of the same length. The formulation of the last sentence refers to a general notion of *computational indistinguishability* which is the heart of the entire approach.

Computational Indistinguishability: Intuitively, two objects are called computationally indistinguishable if no efficient procedure can tell them apart. As usual in complexity theory, an elegant formulation requires asymptotic analysis (or rather a functional treatment of the running time of algorithms in terms of the length of their input).[1] Thus, the objects in question are infinite

[1] We stress that the asymptotic (or functional) treatment is not essential to this approach. One may develop the entire approach in terms of inputs of fixed lengths and an adequate notion of complexity of algorithms. However, such an alternative treatment is more cumbersome.

sequences of distributions, where each distribution has a finite support. Such a sequence will be called a *distribution ensemble*. Typically, we consider distribution ensembles of the form $\{D_n\}_{n\in\mathsf{N}}$, where for some function $\ell : \mathsf{N} \mapsto \mathsf{N}$, the support of each D_n is a subset of $\{0,1\}^{\ell(n)}$. Furthermore, typically ℓ will be a positive polynomial.

> Oded's Note: *In this essay, I've preferred the traditional mathematical notations. Specifically, I have used distributions (over strings) rather than our non-standard "random variables" (which range over strings). For a distribution D, the traditional notation $x{\sim}D$ means x selected according to distribution D.*

Definition 13.6.1 (Computational Indistinguishability [12, 24]): *Two probability ensembles, $\{X_n\}_{n\in\mathsf{N}}$ and $\{Y_n\}_{n\in\mathsf{N}}$, are called* computationally indistinguishable *if for any probabilistic polynomial-time algorithm A, for any positive polynomial p, and for all sufficiently large n's*

$$|\Pr_{x\sim X_n}[A(x)) = 1] - \Pr_{y\sim Y_n}[A(y) = 1]| < \frac{1}{p(n)}$$

The probability is taken over X_n (resp., Y_n) as well as over the coin tosses of algorithm A.

A couple of comments are in place. Firstly, we have allowed algorithm A (called a distinguisher) to be probabilistic. This makes the requirement only stronger, and seems essential to several important aspects of our approach. Secondly, we view events occuring with probability which is upper bounded by the reciprocal of polynomials as *negligible*. This is well-coupled with our notion of efficiency (i.e., polynomial-time computations): An event which occurs with negligible probability (as a function of a parameter n), will occur with negligible probability also if the experiment is repeated for poly(n)-many times.

We note that computational indistinguishability is a strictly more liberal notion than statistical indistinguishability (cf., [24, 10]). An important case is the one of distributions generated by a pseudorandom generator as defined next.

Definition 13.6.2 (Pseudorandom Generators [1, 24]): *A deterministic polynomial-time algorithm G is called a* pseudorandom generator *if there exists a* stretching function, *$\ell : \mathsf{N} \mapsto \mathsf{N}$, so that the following two probability ensembles, denoted $\{G_n\}_{n\in\mathsf{N}}$ and $\{R_n\}_{n\in\mathsf{N}}$, are computationally indistinguishable*

1. *Distribution G_n is defined as the output of G on a uniformly selected seed in $\{0,1\}^n$.*
2. *Distribution R_n is defined as the uniform distribution on $\{0,1\}^{\ell(n)}$.*

That is, letting U_m denote the uniform distribution over $\{0,1\}^m$, we require that for any probabilistic polynomial-time algorithm A, for any positive polynomial p, and for all sufficiently large n's

$$|\Pr_{s\sim U_n}[A(G(s)) = 1] - \Pr_{r\sim U_{\ell(n)}}[A(r) = 1]| < \frac{1}{p(n)}$$

Thus, pseudorandom generators are efficient (i.e., polynomial-time) deterministic programs which expand short randomly selected seeds into longer pseudorandom bit sequences, where the latter are defined as computationally indistinguishable from truly random sequences by efficient (i.e., polynomial-time) algorithms. It follows that any efficient randomized algorithm maintains its performance when its internal coin tosses are substituted by a sequence generated by a pseudorandom generator. That is,

Construction 13.6.3 (typical application of pseudorandom generators): *Let A be a probabilistic algorithm, and $\rho(n)$ denote a* (polynomial) *upper bound on its randomness complexity. Let $A(x,r)$ denote the output of A on input x and coin tosses sequence $r \in \{0,1\}^{\rho(|x|)}$. Let G be a pseudorandom generator with stretching function $\ell:\mathsf{N}\mapsto\mathsf{N}$. Then A_G is a randomized algorithm which on input x, proceeds as follows. It sets $k = k(|x|)$ to be the smallest integer such that $\ell(k) \geq \rho(|x|)$, uniformly selects $s \in \{0,1\}^k$, and outputs $A(x,r)$, where r is the $\rho(|x|)$-bit long prefix of $G(s)$.*

It can be shown that it is infeasible to find long x's on which the *input-output behavior* of A_G is *noticeablly different* from the one of A, although A_G may use much fewer coin tosses than A. That is

Theorem 13.6.4 *Let A and G be as above. Then for every pair of probabilistic polynomial-time algorithms,* a finder F *and* a distinguisher D, *every positive polynomial p and all sufficiently long n's*

$$\sum_{x\in\{0,1\}^n} \Pr[F(1^n)=x]\cdot\Delta_{A,D}(x) \;<\; \frac{1}{p(n)}$$

$$\text{where } \; \Delta_{A,D}(x) \stackrel{\rm def}{=} |\Pr_{r\sim U_{\rho(n)}}[D(x,A(x,r))=1] \;-\; \Pr_{s\sim U_{k(n)}}[D(x,A_G(x,s))=1]|$$

and the probabilities are taken over the U_m's as well as over the coin tosses of F and D.

The theorem is proven by showing that a triplet (A,F,D) violating the claim can be converted into an algorithm D' which distinguishes the output of G from the uniform distribution, in contradiction to the hypothesis. Analogous arguments are applied whenever one wishes to prove that an efficient randomized process (be it an algorithm as above or a multi-party computation) preserves its behavior when one replaces true randomness by pseudorandomness as defined above. Thus, given pseudorandom generators with large stretching function, *one can considerably reduce the randomness complexity in any efficient application.*

Amplifying the stretch function. Pseudorandom generators as defined above are only required to stretch their input a bit; for example, stretching n-bit long inputs to $(n+1)$-bit long outputs will do. Clearly generator of such moderate stretch function are of little use in practice. In contrast, we want to have pseudorandom generators with an arbitrary long stretch function. By the efficiency requirement, the stretch function can be at most polynomial. It turns out that pseudorandom generators with the smallest possible stretch function can be used to construct pseudorandom generators with any desirable polynomial stretch function. (Thus, when talking about the existence of pseudorandom generators, we may ignore the stretch function.)

Theorem 13.6.5 [9]: *Let G be a pseudorandom generator with stretch function $\ell(n) = n+1$, and ℓ' be any polynomially bounded stretch function, which is polynomial-time computable. Let $G_1(x)$ denote the $|x|$-bit long prefix of $G(x)$, and $G_2(x)$ denote the last bit of $G(x)$ (i.e., $G(x) = G_1(x)\,G_2(x)$). Then*

$$G'(s) \;\stackrel{\rm def}{=}\; \sigma_1\sigma_2\cdots\sigma_{\ell'(|s|)}\,,$$
$$\text{where } x_0 = s,\ \sigma_i = G_2(x_{i-1}) \text{ and } x_i = G_1(x_{i-1}),\ \text{for } i=1,...,\ell'(|s|)$$

is a pseudorandom generator with stretch function ℓ'.

Proof Sketch: The theorem is proven using the *hybrid technique* (cf., Sec. 3.2.3 in [5]): One considers distributions H_n^i (for $i = 0, ..., \ell(n)$) defined by $U_i^{(1)} P_{\ell(n)-i}(U_n^{(2)})$, where $U_i^{(1)}$ and $U_n^{(2)}$ are independent uniform distributions (over $\{0,1\}^i$ and $\{0,1\}^n$, respectively), and $P_j(x)$ denotes the j-bit long prefix of $G'(x)$. The extreme hybrids correspond to $G'(U_n)$ and $U_{\ell(n)}$, whereas distinguishability of neighboring hybrids can be worked into distinguishability of $G(U_n)$ and U_{n+1}. Loosely speaking, suppose one could distinguish H_n^i from H_n^{i+1}. Then, using $P_j(s) = G_2(s) P_{j-1}(G_1(s))$ (for $j \geq 1$), this means that one can distinguish $H_n^i \equiv (U_i^{(1)}, G_2(U_n^{(2)}), P_{(\ell(n)-i)-1}(G_1(U_n^{(2)})))$ from $H_n^{i+1} \equiv (U_i^{(1)}, U_1^{(1')}, P_{\ell(n)-(i+1)}(U_n^{(2')}))$. Incorporating the generation of $U_i^{(1)}$ and the evaluation of $P_{\ell(n)-i-1}$ into the distinguisher, one could distinguish $(f(U_n^{(2)}), b(U_n^{(2)})) \equiv G_1(U_n)$ from $(U_n^{(2')}, U_1^{(1')}) \equiv U_{n+1}$, in contradiction to the pseudorandomness of G_1. (For details see Sec. 3.3.3 in [5].) ■

13.6.3 How to Construct Pseudorandom Generators

The known constructions transform computation difficulty, in the form of one-way functions (defined below), into pseudorandomness generators. Loosely speaking, a *polynomial-time computable* function is called one-way if any efficient algorithm can invert it only with negligible success probability. For simplicity, we consider only length-preserving one-way functions.

Definition 13.6.6 (one-way function): *A* one-way function*, f, is a polynomial-time computable function such that for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\Pr_{x \sim U_n}\left[A'(f(x)) \in f^{-1}(f(x))\right] < \frac{1}{p(n)}$$

where U_n is the uniform distribution over $\{0,1\}^n$.

Popular candidates for one-way functions are based on the conjectured intractability of Integer Factorization (cf., [18] for state of the art), the Discrete Logarithm Problem (cf., [19] analogously), and decoding of random linear code [11]. The infeasibility of inverting f yields a weak notion of unpredictability: Let $b_i(x)$ denotes the i^{th} bit of x. Then, for every probabilistic polynomial-time algorithm A (and sufficiently large n), it must be the case that $\Pr_{i,x}[A(i, f(x)) \neq b_i(x)] > 1/2n$, where the probability is taken uniformly over $i \in \{1, ..., n\}$ and $x \in \{0,1\}^n$. A stronger (and in fact strongest possible) notion of unpredictability is that of a hard-core predicate. Loosely speaking, a *polynomial-time computable* predicate b is called a hard-core of a function f if all efficient algorithm, given $f(x)$, can guess $b(x)$ only with success probability which is negligible better than half.

Definition 13.6.7 (hard-core predicate [1]): *A polynomial-time computable predicate $b : \{0,1\}^* \mapsto \{0,1\}$ is called a* hard-core *of a function f if for every probabilistic polynomial-time algorithm A', every positive polynomial $p(\cdot)$, and all sufficiently large n's*

$$\Pr_{x \sim U_n}(A'(f(x)) = b(x)) < \frac{1}{2} + \frac{1}{p(n)}$$

Clearly, if b is a hard-core of a 1-1 polynomial-time computable function f then f must be one-way.[2] It turns out that any one-way function can be slightly modified so that it has a hard-core predicate.

[2] Functions which are not 1-1 may have hard-core predicates of information theoretic nature; but these are of no use to us here. For example, for $\sigma \in \{0,1\}$, $f(\sigma, x) = 0f'(x)$ has an "information theoretic" hard-core predicate $b(\sigma, x) = \sigma$.

Theorem 13.6.8 (A generic hard-core [8]): *Let f be an arbitrary one-way function, and let g be defined by $g(x,r) \stackrel{\rm def}{=} (f(x),r)$, where $|x| = |r|$. Let $b(x,r)$ denote the inner-product mod 2 of the binary vectors x and r. Then the predicate b is a hard-core of the function g.*

See proof in Apdx C.2 in [6]. Finally, we get to the construction of pseudorandom generators:

Theorem 13.6.9 (A simple construction of pseudorandom generators): *Let b be a hard-core predicate of a polynomial-time computable 1-1 function f. Then, $G(s) \stackrel{\rm def}{=} f(s)\,b(s)$ is a pseudorandom generator.*

Proof Sketch: Clearly the $|s|$-bit long prefix of $G(s)$ is uniformly distributed (since f is 1-1 and onto $\{0,1\}^{|s|}$). Hence, the proof boils down to showing that distinguishing $f(s)b(s)$ from $f(s)\sigma$, where σ is a random bit, yields contradiction to the hypothesis that b is a hard-core of f (i.e., that $b(s)$ is *unpredictable* from $f(s)$). Intuitively, such a distinguisher also distinguishes $f(s)b(s)$ from $f(s)\overline{b(s)}$, where $\overline{\sigma} = 1 - \sigma$, and so yields an algorithm for predicting $b(s)$ based on $f(s)$. ■

In a sense, the key point in the proof of the above theorem is showing that the (obvious by definition) unpredictability of the output of G implies its pseudorandomness. The fact that (next bit) unpredictability and pseudorandomness are equivalent in general is proven explicitly in the alternative presentation below.

An alternative presentation. Our presentation of the construction of pseudorandom generators, via Construction 13.6.5 and Proposition 13.6.9, is analogous to the original construction of pseudorandom generators suggested by by Blum and Micali [1]: Given an arbitrary stretch function $\ell:\mathsf{N}\mapsto\mathsf{N}$, a 1-1 one-way function f with a hard-core b, one defines

$$G(s) \stackrel{\rm def}{=} b(x_1)b(x_2)\cdots b(x_{\ell(|s|)})\,,$$

where $x_0 = s$ and $x_i = f(x_{i-1})$ for $i = 1, ..., \ell(|s|)$. The pseudorandomness of G is established in two steps, using the notion of (next bit) unpredictability. An ensemble $\{Z_n\}_{n\in\mathsf{N}}$ is called unpredictable if any probabilistic polynomial-time machine obtaining a prefix of Z_n fails to predict the next bit of Z_n with probability non-negligiblly higher than $1/2$.

Step 1: One first proves that the ensemble $\{G(U_n)\}_{n\in\mathsf{N}}$, where U_n is uniform over $\{0,1\}^n$, is (next-bit) unpredictable (from right to left) [1].

Loosely speaking, if one can predict $b(x_i)$ from $b(x_{i+1})\cdots b(x_{\ell(|s|)})$ then one can predict $b(x_i)$ given $f(x_i)$ (i.e., by computing $x_{i+1}, ..., x_{\ell(|s|)}$ and so obtaining $b(x_{i+1})\cdots b(x_{\ell(|s|)})$), in contradiction to the hard-core hypothesis.

Step 2: Next, one uses Yao's observation by which a (polynomial-time constructible) ensemble is *pseudorandom if and only if it is* (next-bit) *unpredictable* (cf., Sec. 3.3.4 in [5]).

Clearly, if one can predict the next bit in an ensemble then one can distinguish this ensemble from the uniform ensemble (which in unpredictable regardless of computing power). However, here we need the other direction which is less obvious. Still, one can show that (next bit) unpredictability implies indistinguishability from the uniform ensemble. Specifically, consider the following "hybrid" distributions, where the $i^{\rm th}$ hybrid takes the first i bits from the questionable ensemble and the rest from the uniform one. Thus, distinguishing the extreme hybrids implies distinguishing some neighboring hybrids, which in turn implies next-bit predictability (of the questionable ensemble).

A general condition for the existence of pseudorandom generators. Recall that given any one-way 1-1 function, we can easily construct a pseudorandom generator. Actually, the 1-1 requirement may be dropped, but the currently known construction – for the general case – is quite complex. Still we do have.

Theorem 13.6.10 (On the existence of pseudorandom generators [13]): *Pseudorandom generators exist if and only if one-way functions exist.*

To show that the existence of pseudorandom generators imply the existence of one-way functions, consider a pseudorandom generator G with stretch function $\ell(n) = 2n$. For $x, y \in \{0,1\}^n$, define $f(x,y) \stackrel{\text{def}}{=} G(x)$, and so f is polynomial-time computable (and length-preserving). It must be that f is one-way, or else one can distinguish $G(U_n)$ from U_{2n} by trying to invert and checking the result: Inverting f on its range distribution refers to the distribution $G(U_n)$, whereas the probability that U_{2n} has inverse under f is negligible.

The interesting direction is the construction of pseudorandom generators based on any one-way function. In general (when f may not be 1-1) the ensemble $f(U_n)$ may not be pseudorandom, and so Construction 13.6.9 (i.e., $G(s) = f(s)b(s)$, where b is a hard-core of f) cannot be used *directly*. One idea of [13] is to hash $f(U_n)$ to an almost uniform string of length related to its entropy, using Universal Hash Functions [2]. (This is done after guaranteeing, that the logarithm of the probability mass of a value of $f(U_n)$ is typically close to the entropy of $f(U_n)$.)[3] But "hashing $f(U_n)$ down to length comparable to the entropy" means shrinking the length of the output to, say, $n' < n$. This foils the entire point of stretching the n-bit seed. Thus, a second idea of [13] is to compensate for the $n - n'$ loss by extracting these many bits from the seed U_n itself. This is done by hashing U_n, and the point is that the $(n - n' + 1)$-bit long hash value does not make the inverting task any easier. Implementing these ideas turns out to be more difficult than it seems, and indeed an alternative construction would be most appreciated.

13.6.4 Pseudorandom Functions

Pseudorandom generators allow to efficiently generate long pseudorandom sequences from short random seeds. Pseudorandom functions (defined below) are even more powerful: They allow efficient direct access to a huge pseudorandom sequence (which is infeasible to scan bit-by-bit). Put in other words, pseudorandom functions can replace truly random functions in any efficient application (e.g., most notably in cryptography). That is, pseudorandom functions are indistinguishable from random functions by efficient machines which may obtain the function values at arguments of their choice. (Such machines are called oracle machines, and if M is such machine and f is a function, then $M^f(x)$ denotes the computation of M on input x when M's queries are answered by the function f.)

Definition 13.6.11 (pseudorandom functions [7]): *A* pseudorandom function (ensemble)*, with length parameters* $\ell_{\mathrm{D}}, \ell_{\mathrm{R}} : \mathsf{N} \mapsto \mathsf{N}$*, is a collection of functions* $F \stackrel{\text{def}}{=} \{f_s : \{0,1\}^{\ell_{\mathrm{D}}(|s|)} \mapsto \{0,1\}^{\ell_{\mathrm{R}}(|s|)}\}_{s \in \{0,1\}^*}$ *satisfying*

- (efficient evaluation): *There exists an efficient* (deterministic) *algorithm which given a* seed, *s, and an* $\ell_{\mathrm{D}}(|s|)$*-bit* argument, *x, returns the* $\ell_{\mathrm{R}}(|s|)$*-bit long value* $f_s(x)$.

[3] Specifically, given an arbitrary one way function f', one first constructs f by taking a "direct product" of sufficiently many copies of f'. For example, for $x_1, ..., x_{n^2} \in \{0,1\}^n$, we let $f(x_1, ..., x_{n^2}) \stackrel{\text{def}}{=} f'(x_1), ..., f'(x_{n^2})$.

- (pseudorandomness): *For every probabilistic polynomial-time oracle machine, M, for every positive polynomial p and all sufficiently large n's*

$$\left|\Pr_{f\sim F_n}[M^f(1^n)=1]-\Pr_{\rho\sim R_n}[M^\rho(1^n)=1]\right| < \frac{1}{p(n)}$$

where F_n denotes the distribution on F obtained by selecting s uniformly in $\{0,1\}^n$, and R_n denotes the uniform distribution over all functions mapping $\{0,1\}^{\ell_{\mathrm{D}}(n)}$ to $\{0,1\}^{\ell_{\mathrm{R}}(n)}$.

Suppose, for simplicity, that $\ell_{\mathrm{D}}(n)=n$ and $\ell_{\mathrm{R}}(n)=1$. Then a function uniformly selected among 2^n functions (of a pseudorandom ensemble) presents an input-output behavior which is indistinguishable in poly(n)-time from the one of a function selected at random among all the 2^{2^n} Boolean functions. Contrast this with the 2^n pseudorandom sequences, produced by a pseudorandom generator, which are computationally indistinguishable from a sequence selected uniformly among all the $2^{\mathrm{poly}(n)}$ many sequences. Still pseudorandom functions can be constructed from any pseudorandom generator.

Theorem 13.6.12 (How to construct pseudorandom functions [7]): *Let G be a pseudorandom generator with stretching function $\ell(n)=2n$. Let $G_0(s)$ (resp., $G_1(s)$) denote the first (resp., last) $|s|$ bits in $G(s)$, and*

$$G_{\sigma_{|s|}\cdots\sigma_2\sigma_1}(s) \stackrel{\mathrm{def}}{=} G_{\sigma_{|s|}}(\cdots G_{\sigma_2}(G_{\sigma_1}(s))\cdots)$$

Then, the function ensemble $\{f_s:\{0,1\}^{|s|}\mapsto\{0,1\}^{|s|}\}_{s\in\{0,1\}^*}$, where $f_s(x)\stackrel{\mathrm{def}}{=}G_x(s)$, is pseudorandom with length parameters $\ell_{\mathrm{D}}(n)=\ell_{\mathrm{R}}(n)=n$.

The above construction can be easily adapted to any (polynomially-bounded) length parameters $\ell_{\mathrm{D}},\ell_{\mathrm{R}}:\mathsf{N}\mapsto\mathsf{N}$.

Proof Sketch: The proof uses the hybrid technique: The i^{th} hybrid, H_n^i, is a function ensemble consisting of $2^{2^i\cdot n}$ functions $\{0,1\}^n\mapsto\{0,1\}^n$, each defined by 2^i random n-bit strings, denoted $\langle s_\alpha\rangle_{\alpha\in\{0,1\}^i}$. The value of such function at $x=\beta\alpha$, with $|\alpha|=i$, is $G_\beta(s_\alpha)$. The extreme hybrids correspond to our indistinguishability claim (i.e., $H_n^0\equiv f_{U_n}$ and $H_n^n\equiv F_n$), and neighboring hybrids correspond to our indistinguishability hypothesis (specifically, to the indistinguishability of $G(U_n)$ and U_{2n} under multiple samples). ■

We mention that pseudorandom functions have been used to derive negative results in computational learning theory [23] and in complexity theory (cf., Natural Proofs [20]).

13.6.5 The Applicability of Pseudorandom Generators

Randomness is playing an increasingly important role in computation: It is frequently used in the design of sequential, parallel and distributed algorithms, and is of course central to cryptography. Whereas it is convenient to design such algorithms making free use of randomness, it is also desirable to minimize the usage of randomness in real implementations. Thus, pseudorandom generators (as defined above) are a key ingredient in an "algorithmic tool-box" – they provide an automatic compiler of programs written with free usage of randomness into programs which make an economical use of randomness.

Indeed, "pseudo-random number generators" have appeared with the first computers. However, typical implementations use generators which are not pseudorandom according to the above definition. Instead, at best, these generators are shown to pass SOME ad-hoc statistical test (cf., [14]).

However, the fact that a "pseudo-random number generator" passes some statistical tests, does not mean that it will pass a new test and that it is good for a future (untested) application. Furthermore, the approach of subjecting the generator to some ad-hoc tests fails to provide general results of the type stated above (i.e., of the form "for ALL practical purposes using the output of the generator is as good as using truly unbiased coin tosses"). In contrast, the approach encompassed in Definition 13.6.2 aims at such generality, and in fact is tailored to obtain it: The notion of computational indistinguishability, which underlines Definition 13.6.2, covers all possible efficient applications postulating that for all of them pseudorandom sequences are as good as truly random ones.

13.6.6 The Intelectual Contents of Pseudorandom Generators

We shortly discuss some intelectual aspects of pseudorandom generators as defined above.

Behavioristic versus Ontological. Our definition of pseudorandom generators is based on the notion of computational indistinguishability. The behavioristic nature of the latter notion is best demonstrated by confronting it with the Kolmogorov-Chaitin approach to randomness. Loosely speaking, a string is *Kolmogorov-random* if its length equals the length of the shortest program producing it. This shortest program may be considered the "true explanation" to the phenomenon described by the string. A Kolmogorov-random string is thus a string which does not have a substantially simpler (i.e., shorter) explanation than itself. Considering the simplest explanation of a phenomenon may be viewed as an ontological approach. In contrast, considering the effect of phenomena (on an observer), as underlying the definition of pseudorandomness, is a behavioristic approach. Furthermore, there exist probability distributions which are not uniform (and are not even statistically close to a uniform distribution) that nevertheless are indistinguishable from a uniform distribution by any efficient method [24, 10]. Thus, distributions which are ontologically very different, are considered equivalent by the behavioristic point of view taken in the definitions above.

A relativistic view of randomness. Pseudorandomness is defined above in terms of its observer. It is a distribution which cannot be told apart from a uniform distribution by any efficient (i.e. polynomial-time) observer. However, pseudorandom sequences may be distinguished from random ones by infinitely powerful powerful (not at our disposal!). Specifically, an exponential-time machine can easily distinguish the output of a pseudorandom generator from a uniformly selected string of the same length (e.g., just by trying all possible seeds). Thus, pseudorandomness is subjective to the abilities of the observer.

Randomness and Computational Difficulty. Pseudorandomness and computational difficulty play dual roles: The definition of pseudorandomness relies on the fact that putting computational restrictions on the observer gives rise to distributions which are not uniform and still cannot be distinguished from uniform. Furthermore, the construction of pseudorandom generators rely on conjectures regarding computational difficulty (i.e., the existence of one-way functions), and this is inevitable: given a pseudorandom generator, we can construct one-way functions. Thus, (non-trivial) pseudorandomness and computational hardness can be converted back and forth.

13.6.7 A General Paradigm

Pseudorandomness as surveyed above can be viewed as an important special case of a general paradigm. A general treatment is provided in [6].

A generic formulation of pseudorandom generators consists of specifying three fundamental aspects – the *stretching measure* of the generators; the class of distinguishers that the generators are supposed to fool (i.e., the algorithms with respect to which the *computational indistinguishability* requirement should hold); and the resources that the generators are allowed to use (i.e., their own *computational complexity*). In the above presentation we focused on polynomial-time generators (thus having polynomial stretching measure) which fool any probabilistic polynomial-time observers. A variety of other cases are of interest too, and we briefly discuss some of them.

Weaker notions of computational indistinguishability. Whenever the aim is to replace random sequences utilized by an algorithm with pseudorandom ones, one may try to capitalize on knowledge of the target algorithm. Above we have merely used the fact that the target algorithm runs in polynomial-time. However, for example, if we know that the algorithm uses very little work-space then we may able to do better. Similarly, if we know that the analysis of the algorithm only depends on some specific properties of the random sequence it uses (e.g., pairwise independence of its elements). In general, weaker notions of computational indistinguishability such as fooling space-bounded algorithms, constant-depth circuits, and even specific tests (e.g., testing pairwise independence of the sequence), arise naturally: Generators producing sequences which fool such tests are useful in a variety of applications – if the application utilizes randomness in a restricted way then feeding it with sequences of low randomness-quality may do. Needless to say that we advocate a rigorous formulation of the characteristics of such applications and rigorous construction of generators which fool the type of tests which emerge.

Alternative notions of generator efficiency. The above discussion has focused on one aspect of the pseudorandomness question – the resources or type of the observer (or potential distinguisher). Another important question is whether such pseudorandom sequences can be generated from much shorter ones, and at what cost (or complexity). Throughout this essay we've required the generation process to be at least as efficient as the efficiency limitations of the distinguisher.[4] This seems indeed "fair" and natural. Allowing the generator to be more complex (i.e., use more time or space resources) than the distinguisher seems unfair, but still yields interesting consequences in the context of trying to "de-randomize" randomized complexity classe. For example, one may consider generators working in time exponential in the length of the seed. In some cases we loose nothing by being more liberal (i.e., allowing exponential-time generators). To see why, we consider a typical derandomization argument, proceediing in two steps: First one replaces the true randomness of the algorithm by pseudorandom sequences generated from much shorter seeds, and next one goes deterministically over all possible seeds and looks for the most frequent behavior of the modified algorithm. Thus, in such a case the deterministic complexity is anyhow exponential in the seed length. However, constructing exponential time generators may be easier than constructing polynomial-time ones.

References

[4] If fact, we have require the generator to be more efficient than the distinguisher: The former was required to be a fixed polynomial-time algorithm, whereas the latter was allowed to be any algorithm with polynomial running time.

1. M. Blum and S. Micali. How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits. *SIAM Journal on Computing*, Vol. 13, pages 850–864, 1984. Preliminary version in *23rd IEEE Symposium on Foundations of Computer Science*, 1982.

2. L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Science*, Vol. 18, 1979, pages 143–154.

3. G.J. Chaitin. On the Length of Programs for Computing Finite Binary Sequences. *Journal of the ACM*, Vol. 13, pages 547–570, 1966.

4. T.M. Cover and G.A. Thomas. *Elements of Information Theory.* John Wiley & Sons, Inc., New-York, 1991.

5. O. Goldreich. *Foundation of Cryptography – Fragments of a Book.* February 1995. Available from `http://theory.lcs.mit.edu/~oded/frag.html`.

6. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

7. O. Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions. *Journal of the ACM*, Vol. 33, No. 4, pages 792–807, 1986.

8. O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st ACM Symposium on the Theory of Computing*, pages 25–32, 1989.

9. O. Goldreich and S. Micali. Increasing the Expansion of Pseudorandom Generators. Manuscript, 1984. Available from `http://theory.lcs.mit.edu/~oded/papers.html`

10. O. Goldreich, and H. Krawczyk, On Sparse Pseudorandom Ensembles. *Random Structures and Algorithms*, Vol. 3, No. 2, (1992), pages 163–174.

11. O. Goldreich, H. Krawcyzk and M. Luby. On the Existence of Pseudorandom Generators. *SIAM Journal on Computing*, Vol. 22-6, pages 1163–1175, 1993.

12. S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Science*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th ACM Symposium on the Theory of Computing*, 1982.

13. J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. Construction of Pseudorandom Generator from any One-Way Function. To appear in *SIAM Journal on Computing.* Preliminary versions by Impagliazzo et. al. in *21st ACM Symposium on the Theory of Computing* (1989) and Håstad in *22nd ACM Symposium on the Theory of Computing* (1990).

14. D.E. Knuth. *The Art of Computer Programming*, Vol. 2 (*Seminumerical Algorithms*). Addison-Wesley Publishing Company, Inc., 1969 (first edition) and 1981 (second edition).

15. A. Kolmogorov. Three Approaches to the Concept of "The Amount Of Information". *Probl. of Inform. Transm.*, Vol. 1/1, 1965.

16. L.A. Levin. Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. *Inform. and Control*, Vol. 61, pages 15–37, 1984.

17. M. Li and P. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer Verlag, August 1993.

18. A.M. Odlyzko. The future of integer factorization. *CryptoBytes* (The technical newsletter of RSA Laboratories), Vol. 1 (No. 2), pages 5-12, 1995.

19. A.M. Odlyzko. Discrete logarithms and smooth polynomials. In *Finite Fields: Theory, Applications and Algorithms*, G. L. Mullen and P. Shiue, eds., Amer. Math. Soc., Contemporary Math. Vol. 168, pages 269–278, 1994.

20. A.R. Razborov and S. Rudich. Natural proofs. *Journal of Computer and System Science*, Vol. 55 (1), pages 24–35, 1997.

21. C.E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Jour.*, Vol. 27, pages 623–656, 1948.

22. R.J. Solomonoff. A Formal Theory of Inductive Inference. *Inform. and Control*, Vol. 7/1, pages 1–22, 1964.

23. L. Valiant. A theory of the learnable. *Communications of the ACM*, Vol. 27/11, pages 1134–1142, 1984.

24. A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd IEEE Symposium on Foundations of Computer Science*, pages 80–91, 1982.

Lecture 14

Pseudorandomness and Computational Difficulty

Notes taken by Moshe Lewenstein and Yehuda Lindell

Summary: In this lecture we continue our discussion of pseudorandomness and show a connection between pseudorandomness and computational difficulty. More specifically we show how the difficulty of inverting one-way functions may be utilized to obtain a pseudorandom generator. Finally, we state and prove that a hard-to-predict bit (called a hard-core) may be extracted from any one-way function. The hard-core is fundamental in our construction of a generator.

14.1 Introduction

The main theme of this lecture is the utilization of one-way functions in order to construct a pseudorandom generator. Intuitively, a one-way function is a function that is *easy* to compute and *hard* to invert. Generally, "easy" refers to polynomial time and "hard" to the fact that success in the *average case* requires more than polynomial time (for any polynomial). It is critical that the difficulty be in the average case and *not* in the worst case, as with $\mathcal{NP}$-Complete problems. This will become clear later.

How can one-way functions help us construct a pseudorandom generator? The answer lies in the property of unpredictability. This concept will be formalized in the coming lectures but for now we will discuss it informally. Assume that we have a string s and we begin to scan it in some given (computable) order. If at some stage we can predict the next bit with probability significantly greater than one half, then the string is clearly not random (because for a random string, each bit is chosen *independently* with probability exactly one half). On the other hand, it can be shown that if we *cannot* predict any "next" bit during our scan with success significantly more than $\frac{1}{2}$, then the string is pseudorandom.

In this light, the use of computationally difficult problems becomes clear. We rely on the difficulty of inverting a one-way function f. More specifically we show that there exists a function $b : \{0,1\}^n \to \{0,1\}$ such that given x it is easy to compute $b(x)$ yet given only $f(x)$ it is difficult. This function is formally called a hard-core of f. Now, although given $f(x)$, $b(x)$ is fully and deterministically defined, we have no way of finding or predicting its value. Therefore, the computational difficulty of finding $b(x)$ provides us with an unpredictable bit which forms the basis of our generator.

14.2 Definitions

We now recall the concepts necessary for this lecture.

Definition 14.1 Pseudorandom generators: *G is a* pseudorandom generator *if:*

1. *G operates in (deterministic) polynomial time*
2. *For every s, $|G(s)| > |s|$ (w.l.o.g., assume that $\exists l(\cdot)$ such that $\forall s \in \{0,1\}^n$ $|G(s)| = l(n)$).*
3. *$\{G(U_n)\}$ and $\{U_{l(n)}\}$ are probabilistic polynomial time indistinguishable (where U_n is the uniform distribution over $\{0,1\}^n$).*

Definition 14.2 One-way functions: *Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-preserving function (i.e. $\forall x$ $|f(x)| = |x|$). Then f is* one-way *if:*

1. *f is "easy" to compute. Formally, there exists a polynomial time algorithm A such that $\forall x$ $A(x) = f(x)$.*
2. *f is "hard" to invert in the average case. Formally, for every probabilistic polynomial time algorithm A, for every polynomial $p(\cdot)$, and for all sufficiently large n's,*

$$Prob[A(f(U_n)) \in f^{-1}f(U_n)] < \frac{1}{p(n)}$$

The above definition refers to a length-preserving function. This is a simplifying assumption we will use throughout this lecture, but it is generally not necessary as long as $|x| = poly(|f(x)|)$. The requirement that the lengths of x and $f(x)$ be polynomially related is necessary to ensure that the difficulty involved in inverting $f(x)$ is not because x is too long. Since the inverting algorithm must work in time polynomial in $|f(x)|$, if $|f(x)|$ is logarithmic in $|x|$, no algorithm can even write x. In this case there is no computational difficult in *inverting* f and the one-wayness is due to the above technicality. Assuming that f is length-preserving avoids this problem.

As we will see, there is no requirement that f be length-preserving in the hard-core theorem stated and proved in section 4. However, the exact construction of the pseudorandom generator in section 3 relies heavily upon the length-preserving property of the one-way function and the assumption that it is $1-1$. Other constructions exist for the more general case but are more complex.

Although the definition of a one-way function guarantees that it is difficult to find the entire string x given $f(x)$, it may be very easy to obtain some of the bits of x. For example, assuming that f is one-way, consider the function $f'(\sigma \cdot x) = \sigma \cdot f(x)$, where $\sigma \in \{0,1\}$. It is easy to see that f' is also one-way. This is rigorously shown by assuming that f' is *not* one-way and showing how f can be inverted using an algorithm which inverts f'. So we see that despite the fact that f' is one-way, the first bit of the input is completely revealed.

Therefore, in order to obtain unpredictability as desired, we need a specific bit based on x, which is provably hidden given $f(x)$. This bit is called a *hard-core* of f.

Reducibility Arguments: The above-mentioned method of proof (showing that $f'(\sigma \cdot x) = \sigma \cdot f(x)$ is one-way) is called a reduction. It is worthwhile discussing this technique as it will form the basis of many proofs that we will see. The context in which it appears is when we take a certain primitive and construct a new primitive based on it. For example, we will need it to prove our construction of a pseudorandom generator from a one-way function. Although it may be intuitively clear that a new construction is "correct" this sort of intuition may be misleading and so requires a sound proof. We do this by showing that if the newly constructed primitive is not secure, then an algorithm defying it may be used to defy the original primitive as well. More concretely, if our generator is distinguishable, then a distinguishing algorithm for it may be used to invert the one-way function used in the construction. We will see this type of argument many times in the coming lectures.

Definition 14.3 Hard-Core: *The function $b : \{0,1\}^* \rightarrow \{0,1\}$ is a* hard-core *of f if:*

1. *b is easy to compute (in the same sense as above),*

2. *For every probabilistic polynomial time algorithm A, for every polynomial $p(\cdot)$, and for all sufficiently large n's,*
$$Prob[A(f(U_n)) = b(U_n)] < \frac{1}{2} + \frac{1}{p(n)}$$

Note that it is trivial to predict b with probability $\frac{1}{2}$ by simply guessing. The above definition requires that we cannot do significantly better than this guess. A hard-core of a one-way function plays an important role in cryptographic applications (as all information is hidden). However, in our case, we will use the computational difficulty involved as the basis for our construction of a pseudorandom generator. This can be seen in the following way. Given $f(x)$, $b(x)$ is fully-defined, yet completely unpredictable to any polynomially bound algorithm. This extra bit of unpredictable information is what will supply the "stretch" effect necessary for the generator.

The following claim shows that a hard-core can only exist for a one-way function. This is intuitively obvious, since if f is $1-1$ but not one-way then we can invert it and compute b. This is formally shown below.

Claim 14.2.1 *If f is 1-1 and polynomial time computable, and b is a hard-core of f, then f is one-way.*

Proof: By contradiction, assume that b is a hard-core of f, yet f is not one-way. We now show how an algorithm inverting f can be used to predict $b(x)$ from $f(x)$. Note once again the reduction technique we use.

f is not one-way, therefore there exists a probabilistic polynomial time algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's, $Prob[A(f(x)) = x] \geq \frac{1}{p(n)}$ ($|x| = n$).

We now construct an algorithm A' for predicting b from f:

Input: y

- $x' \leftarrow A(y)$ (attempt to invert y, using A).
- If $f(x') = y$ then output $b(x')$
- Otherwise, output 0 or 1 with probability 1/2.

We now calculate the success probability of A'.

$$Prob[A'(f(x)) = b(x)] \ = \ Prob[A(f(x)) = x]\cdot 1 + Prob[A(f(x)) \neq x]\cdot\frac{1}{2}$$

$$\geq \frac{1}{p(n)} \cdot 1 + (1 - \frac{1}{p(n)}) \cdot \frac{1}{2} \quad = \quad \frac{1}{2} + \frac{1}{2p(n)}$$

The success probability of A' is greater than or equal to $\frac{1}{2} + \frac{1}{2p(n)}$ for infinitely many n's, thereby contradicting the fact that b is a hard-core of f. ■

Comment: If f is not 1-1, then the above claim is false. Consider f such that $f(x) = 0^{|x|}$. Clearly $b(x) =$ "the 1st bit of x" is a hard-core. However this is because of information theoretic considerations rather than computational bounds. The function f defined here is *trivially* inverted by taking any arbitrary string of length $|x|$ as the preimage. However, $b(x)$ may clearly not be guessed with any probability better than $\frac{1}{2}$.

14.3 A Pseudorandom Generator based on a 1-1 One-Way Function

In this section we show a construction of a pseudorandom generator given a length-preserving 1-1 one-way function. The construction is based on a hard-core of the one-way function. In the next section, we show how to generically construct a hard-core of any one-way function.

We note that constructions of a pseudorandom generator exist based on *any* one-way function (not necessarily length-preserving and 1-1). However, the constructions and proofs in the more general case are long and complicated and we therefore bring this case only.

Theorem 14.4 *Let f be a length-preserving, 1-1 one-way function and let b be a hard-core of f. Then $G(s) = f(s)b(s)$ is a pseudorandom generator (stretching the input by one bit).*

Proof: We first note that as f is length-preserving and 1-1, $f(U_n)$ is distributed uniformly over $\{0,1\}^n$ and is therefore fully random. It remains to show that for $s \in_R \{0,1\}^n$, the combination of $f(s)$ and $b(s)$ together remains indistinguishable from U_{n+1}. Intuitively, as we cannot predict $b(s)$ from $f(s)$, the string looks random to any computationally bound algorithm.

We will now show how a successful distinguishing algorithm may be used to predict b from f. This then proves that no such distinguishing algorithm exists (because b is a hard-core of f). By contradiction, assume that there exists an algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's

$$|Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(U_{n+1}) = 1]| \geq \frac{1}{p(n)}$$

As $f(U_n)$ is distributed uniformly, this is equivalent to A successfully distinguishing between $\{f(U_n)b(U_n)\}$ and $\{f(U_n)U_1\}$. It follows that A can distinguish between $X_1 = \{f(U_n)b(U_n)\}$ and $X_2 = \{f(U_n)\overline{b(U_n)}\}$.

Let X be the distribution created by uniformly choosing $\sigma \in_R \{1,2\}$ and then sampling from X_σ. Clearly X is identically distributed to $\{f(U_n)U_1\}$. Now:

$$Prob[A(X) = 1] = \frac{1}{2} \cdot Prob[A(X_1) = 1] \; + \; \frac{1}{2} \cdot Prob[A(X_2) = 1]$$

$$\Rightarrow Pr[A(X_2) = 1] = 2 \cdot Prob[A(X) = 1] \; - \; Prob[A(X_1) = 1]$$

Therefore:

$$|\ Prob[A(f(U_n)b(U_n)) = 1] \; - \; Prob[A(f(U_n)\overline{b(U_n)}) = 1]\ |$$

$$= |\ Prob[A(X_1) = 1] - Prob[A(X_2) = 1]\ |$$
$$= |\ Prob[A(X_1) = 1] - 2{\cdot}Prob[A(X) = 1] + Prob[A(X_1) = 1]\ |$$
$$= |\ 2{\cdot}Prob[A(X_1) = 1] - 2{\cdot}Prob[A(X) = 1]\ |$$
$$= 2{\cdot}|\ Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(U_{n+1}) = 1]\ | \geq \frac{2}{p(n)}$$

Assume, without loss of generality, that for infinitely many n's it holds that:

$$Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1] \geq \frac{2}{p(n)}$$

Otherwise we simply reverse the terms here and make the appropriate changes in the algorithm and the remainder of the proof (i.e. we change step 2 of the algorithm below to: If $A(y \cdot \sigma) = 0$ then output σ).

We now construct A' to predict $b(U_n)$ from $f(U_n)$. Intuitively, A' adds a random guess to the end of it's input y (where $y = f(x)$ for some x) and attempts to see if it guessed $b(f^{-1}(y))$ correctly based on A's response to this guess. The algorithm follows:

Input: y

1. Uniformly choose $\sigma \in_R \{0, 1\}$
2. If $A(y \cdot \sigma) = 1$ then output σ
3. Otherwise, output $\overline{\sigma}$

We now calculate the probability that A' successfully computes $b(f^{-1}(y))$. As σ is uniformly chosen from $\{0, 1\}$ we have:

$$Prob[A'(f(U_n)) = b(U_n)] = \frac{1}{2} \cdot Prob[A'(f(U_n)) = b(U_n) \mid \sigma = b(U_n)]$$
$$+\frac{1}{2} \cdot Prob[A'(f(U_n)) = b(U_n) \mid \sigma = \overline{b(U_n)}]$$

Now, by the algorithm (see steps 2 and 3 respectively) we have:

$$Prob[A'(f(U_n)) = b(U_n) \mid \sigma = b(U_n)] = Prob[A(f(U_n)b(U_n)) = 1]$$

and

$$Prob[A'(f(U_n)) = b(U_n) \mid \sigma = \overline{b(U_n)}] = Prob[A(f(U_n)\overline{b(U_n)}) = 0]$$
$$= 1 - Prob[A(f(U_n)\overline{b(U_n)}) = 1]$$

By our contradiction hypothesis:

$$Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1] \geq \frac{2}{p(n)}$$

Therefore:

$Prob[A'(f(U_n)) = b(U_n)]$

$$= \frac{1}{2}\cdot Prob[A(f(U_n)b(U_n)) = 1] + \frac{1}{2}\cdot(1 - Prob[A(f(U_n)\overline{b(U_n)}) = 1])$$
$$= \frac{1}{2} + \frac{1}{2}\cdot(Prob[A(f(U_n)b(U_n)) = 1] - Prob[A(f(U_n)\overline{b(U_n)}) = 1]) \;\geq\; \frac{1}{2} + \frac{1}{p(n)}$$

which is in contradiction to the fact that b is a hard-core of f and cannot be predicted with non-negligible advantage over $\frac{1}{2}$. ■

We remind the reader that in the previous lecture we proved that a generator stretching the seed by even a single bit can be deterministically converted into a generator stretching the seed by any polynomial length. Therefore, it should not bother us that the above construction seems rather weak with respect to its "stretching capability".

At this stage it should be clear why it is crucial that the one-way function be hard to invert in the average case and not just in the worst case. If the function is invertible in the average case, then it is easy to distinguish between $\{f(U_n)b(U_n)\}$ and $\{U_{n+1}\}$ most of the time. This would clearly *not* be satisfactory for a pseudorandom generator.

14.4 A Hard-Core for Any One-Way Function

In this section we present a construction of a hard-core from any one-way function. Here there is no necessity that f be 1-1 or even length-preserving. As we have seen in the previous section, the existence of a hard-core is essential in our construction of a pseudorandom generator.

Theorem 14.5 *Let $f_0 : \{0,1\}^* \to \{0,1\}^*$ be a one-way function. Define $f(x,r) = (f_0(x), r)$ where $|x| = |r|$. Then $b(x,r) = \sum_{i=1}^{n} x_i r_i \bmod 2$ is a hard-core of f.*

Note that since f_0 is one-way, f is clearly one-way (trivially, any algorithm inverting f can be used to invert f_0).

Proof: Assume by contradiction, that there exists a probabilistic polynomial time algorithm A and a polynomial $p(\cdot)$ such that for infinitely many n's

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{p(n)}$$

where the probabilities are taken, uniformly and independently, over $x \in_R \{0,1\}^n$, $r \in_R \{0,1\}^n$ and coin tosses of A (if any).

The following claim shows that there are a significant number of x's for which A succeeds with non-negligible probability. We will then show how to invert f for these x's.

Claim 14.4.1 *Let $S_n \subseteq \{0,1\}^n$ be the set of all x's where $Prob_r[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{2p(n)}$. Then, $Prob_x[x \in S_n] > \frac{1}{2p(n)}$.*

Proof: By a simple averaging argument. Assume by contradiction that $Prob_x[x \in S_n] \leq \frac{1}{2p(n)}$. Then:

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] = Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \in S_n]\cdot Prob[x \in S_n]$$
$$+ Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \notin S_n]\cdot Prob[x \notin S_n]$$

Now, trivially both $Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \in S_n] \leq 1$ and $Prob[x \notin S_n] \leq 1$. Furthermore, by the contradiction hypothesis $Prob[x \in S_n] \leq \frac{1}{2p(n)}$. Finally, based on the definition of S_n, $Prob_{x,r}[A(f(x,r)) = b(x,r) \mid x \notin S_n] < \frac{1}{2} + \frac{1}{2p(n)}$. Putting all these together:

$$Prob_{x,r}[A(f(x,r)) = b(x,r)] < 1 \cdot \frac{1}{2p(n)} + (\frac{1}{2} + \frac{1}{2p(n)}) \cdot 1 = \frac{1}{2} + \frac{1}{p(n)}$$

which contradicts the assumption regarding the success probability of A. ■

It suffices to show that we can retrieve x from $f(x)$ for $x \in S_n$ with probability $\frac{1}{poly(n)}$ (because then we can invert any $f(x)$ with probability $\geq \frac{1}{2p(n)poly(n)} = \frac{1}{poly(n)}$). So, assume that for x: $Prob_r[A(f(x,r)) = b(x,r)] \geq \frac{1}{2} + \frac{1}{2p(n)}$ (as in the claim).

Denote $B_x(r) = A(f(x,r)$. Now x is fixed and $B_x(r)$ is a black-box returning $b(x,r)$ with probability $\geq \frac{1}{2} + \frac{1}{2p(n)}$. We use calls to B_x to retrieve x given $f(x)$. We also denote $\epsilon = \frac{1}{2p(n)}$.

As an exercise, assume that $Prob_r[B_x(r) = b(x,r)] > \frac{3}{4} + \epsilon$. There is no reason to assume that this is true but it will help us with the proof later on. Using this assumption, for every $i = 1, ..., n$ we show how to recover x_i (that is, the i'th bit of x).

Input: $y = f(x)$

1. Uniformly select $r \in_R \{0,1\}^n$
2. Compute $B_x(r) \oplus B_x(r \oplus e^i)$ where $e^i = (0^{i-1}10^{n-i})$ (1 in the i'th coordinate and 0 everywhere else).

Now, $Prob[B_x(r) \neq b(x,r)] < \frac{1}{4} - \epsilon$ by the hypothesis. Therefore, $Prob[B_x(r \oplus e^i) \neq b(x, r \oplus e^i)] < \frac{1}{4} - \epsilon$ (because $r \oplus e^i$ is also uniformly distributed). So, the probability that $B_x(r) = b(x,r)$ and $B_x(r \oplus e^i) = b(x, r \oplus e^i)$ is greater than $\frac{1}{2} + 2\epsilon$ (by summing the errors).
In this case: $B_x(r) \oplus B_x(r \oplus e^i) = b(x,r) \oplus b(x, r \oplus e^i)$. However,

$$b(x,r) \oplus b(x, r \oplus e^i) = \sum_{j=1}^{n} x_j r_j + \sum_{j=1}^{n} x_j (r_j + e^i_j) \; mod \; 2$$

$$= \sum_{j=1}^{n} x_j r_j + x_i + \sum_{j=1}^{n} x_j r_j \; mod \, 2 = x_i$$

So, if we repeat this $O(\frac{n}{\epsilon^2})$ times and take a majority vote, we obtain a correct answer with a very high probability (in the order of $1 - \frac{1}{2^n}$). This is true (but with probability $\geq 1 - \frac{1}{2n}$) even if the different executions are only pairwise independent (this can be derived using Chebyshev's inequality and is important later on). We do the same for all i's and in this way succeed in inverting $f(x)$ with high probability. Note that we can use the same set of random strings r for each i (the only difference is in obtaining $b(x, r \oplus e^i)$ each time).

This inversion algorithm worked based on the unreasonable assumption that $Prob_r[B_x(r) = b(x,r)] > \frac{3}{4} + \epsilon$. This was necessary as we needed to query B_x on two different points and therefore we had to sum the error probabilities. When the probability of success is only ϵ above $\frac{1}{2}$, the resulting error probability is far too high.

In order to solve this problem, remember that we are really interested in calculating $b(x, e^i) = x_i$. However, we cannot query $B_x(e^i)$ because we have no information on B_x's behaviour at any given point (the known probabilities are for $B_x(r)$ where r is uniformly distributed). Therefore we queried B_x at r and $r \oplus e^i$ (2 random points) and inferred x_i from the result.

We now show how to compute $b(x,r) \oplus b(x, r \oplus e^i)$ for $O(\frac{n}{\epsilon^2})$ pairwise independent r's. Based on what we have seen above, this is enough to invert $f(x)$. Our strategy is based on obtaining values of $b(x,r)$ for "free" (that is, without calling B_x). We do this by guessing what the value should be, but in such a way that the probability of being correct is non-negligible.

Let $l = \log_2(m+1)$ where $m = O(\frac{n}{\epsilon^2})$.
Let $s^1, ..., s^l \in_R \{0,1\}^n$ be l uniformly chosen n-bit strings.
Then for every non-empty $I \subseteq \{1, ..., l\}$, define: $r^I = \bigoplus_{i \in I} s^i$. Each r^I is an n-bit string constructed by xoring together the strings s^i indexed by I.

Each r^I is uniformly distributed as it is the xor of random strings. Moreover each pair is independent since for $I \neq J$, $\exists s^i$ s.t. w.l.o.g. $s^i \in I$ and $s^i \notin J$. Therefore, r^I given r^J is uniformly distributed based on the distribution of s^i.

Now, let us uniformly choose $\sigma^1, ..., \sigma^l \in \{0,1\}$. Assume that we were very lucky and that for every i, $\sigma^i = b(x, s^i)$ (in other words, we guessed $b(x, s^i)$ correctly every time). Note that this lucky event happens with the non-neglible probability of $\frac{1}{2^l} = \frac{\epsilon}{n^2} = \frac{1}{poly(n)}$. The following claim shows that in this lucky case we achieve our aim.

Claim 14.4.2 *Let $\tau^I = \bigoplus_{i \in I} \sigma^i$. Then, if for every i, $\sigma^i = b(x, s^i)$ then for every I, $\tau^I = b(x, r^I)$.*

Proof:

$$b(x, r^I) = b(x, \sum_{i \in I} s^i) = \sum_{j=1}^{n} x_j \cdot \sum_{i \in I} s^i_j = \sum_{i \in I} \sum_{j=1}^{n} x_j \cdot s^i_j = \sum_{i \in I} b(x, s^i) = \sum_{i \in I} \sigma^i = \tau^I$$

where all sums are mod 2. ■

The above claim shows that by correctly guessing $\log m$ values of σ^i we are able to derive the value of $b(x, \cdot)$ at m pairwise independent points. This is because under the assumption that we guessed the σ^i's correctly, each τ^I is exactly $b(x, r^I)$ where the r^I's are uniformly distributed and pairwise independent.

Note that since there are $2^l - 1 = m$ different subsets I, we have the necessary number of different points in order to obtain x_i. The algorithm uses these points in order to extract x_i instead of uniformly chosen r's. (An alternative strategy is not to guess the σ^i's but to try every possible combination of them. Since $2^l = m + 1$ which is polynomial, we can do this in the time available.)

The Actual Algorithm:

Input: y

1. Uniformly choose $s^1, ..., s^l \in_R \{0,1\}^n$ and $\sigma^1, ..., \sigma^l \in \{0,1\}$.
2. For every non-empty $I \subseteq \{1, ..., l\}$, define: $r^I = \bigoplus_{i \in I} s^i$ and compute $\tau^I = \bigoplus_{i \in I} \sigma^i$.
3. For every $i \in \{1, ..., n\}$ and non-empty $I \subseteq \{1, ..., l\}$ do
   - $v^I_i \; = \; \tau^i \oplus B_x(r^I \oplus e^i)$
   - Guess $x_i \; = \text{majority}_I \{v^I_i\}$

In order to calculate the probability that the above algorithm succeeds in inverting $f(x)$, assume that for every i, we guessed σ^i such that $\sigma^i = b(x, s^i)$ (in step 1). The probability of this event occurring is $\frac{1}{2^l} = \frac{1}{m+1}$ for $m = O(\frac{n}{\epsilon^2})$ and $\epsilon = \frac{1}{2p(n)}$. In other words, the probability that we are lucky is $\frac{1}{poly(n)}$.

Since we know that for every I, $B_x(r^I \oplus e^i)$ is correctly computed with a probability greater that $\frac{1}{2} + 2\epsilon$, it follows that $b(x, r^I) \oplus B_x(r^I \oplus e^i)$ is also correct with the same probability (as the σ^i's are correct). As previously mentioned, due to the pairwise independence of the events, the probability that we succeed in inverting $f(x)$ in this case is at least $1 - \frac{1}{2n}$ (this is proved using Chebyshev's inequality).

It is easy to see that the probability that we succeed in guessing all σ^i's correctly *and* then proceed to successfully invert f is the product of the above two probabilities, that is $\frac{1}{poly(n)}$.

We therefore conclude that the probability of successfully inverting f is non-negligible. This is in contradiction to the one-wayness of f. Therefore, b as defined is a hard-core of f. ■

Summary of the proof.

We began by assuming the existence of an algorithm A which predicts $b(x, r)$ from $f(x, r)$ with non-negligible success probability (over all choices of x, r and coin tosses of A). We then showed that there is a non-negligible set of x's for which A succeeds and that we proceed to attempt to invert $f(x)$ for these x's only. This enabled us to set x and focus only on the randomness in r.

In the next stage we described how we can obtain x, under the assumption that both $b(x, r)$ and $b(x, r \oplus e^i)$ can be computed correctly with probability $\geq \frac{1}{2} + \frac{1}{poly(n)}$. This was easily shown as $b(x, r) \oplus b(x, r \oplus e^i) = x_i$ and by repeating we can achieve a small enough error so that the probability that we succeed in obtaining x_i for all i is at least $\frac{1}{poly(n)}$.

Finally we showed how to achieve the previous assumption. This involved realizing that pairwise independence for different r's is enough and then showing how $poly(n)$ pairwise independent n-bit strings can be obtained using only $log(poly(n))$ n-bit random strings $s^1, ..., s^l$.

Based on this, we guess the value of $b(x, s^i)$ for each i. The critical point here is that the probability of guessing correctly for all i is $\frac{1}{poly(n)}$ and that the value of $b(x, \cdot)$ for all $poly(n)$ pairwise independent n-bit strings can be immediately derived from these guesses.

In short, we succeeded (with non-negligible probability) in guessing $b(x, \cdot)$ correctly for a polynomial number of pairwise independent strings. These strings are used as r for $b(x, r)$ in the inversion algorithm described in the middle stage. We compute $b(x, r \oplus e^i)$ using A. Assuming that we guessed correctly, we achieve the necessary success probability for computing both $b(x, r)$ and $b(x, r \oplus e^i)$. As we guess once for the entire inversion algorithm and this is independent of all that follows, we are able to extract x from $f(x)$ with overall probability of $\frac{1}{poly(n)}$.

This proves that no such algorithm A exists.

Bibliographic Notes

This lecture is based mainly on [2] (see also Appendix C in [1]).

1. O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness.* Algorithms and Combinatorics series (Vol. 17), *Springer*, 1998.

2. O. Goldreich and L.A. Levin. Hard-core Predicates for any One-Way Function. In *21st STOC*, pages 25–32, 1989.

Lecture 15

Derandomization of BPP

Notes taken by Erez Waisbard and Gera Weiss

Summary: In this lecture we present an efficient deterministic simulation of randomized algorithms. This process, called derandomization, introduce new notions of pseudorandom generators. We extend the definition of pseudorandom generators and show how to construct a generator that can be used for derandomization. The new construction differ from the generator that we constructed in the previous lecture in it's running time (it will run slower, but fast enough for the simulation), but most importantly in it's assumptions. We are not assuming the existence of one-way function but we make another assumption which may be weaker than that.

15.1 Introduction

Randomness plays a key role in many algorithms. However random sources are not always available and we would like to minimize the usage of randomness. A naive way to remove the "randomness element" from algorithm is simply to go over all possible coin tosses it uses and act according to the majority. This however we can't do for BPP in polynomial time with respect to the size of the input. If we could shrink the amount of randomness the algorithm consumes (to logarithmic) then we could imply the naive idea in polynomial time. A way to use small random source to create much more randomness was introduced in the previous lecture - the pseudorandom generator. Pseudorandom generator G stretch out random seed into a polynomial-long pseudorandom sequence that can't be efficiently (in polynomial time) distinguished from a truly random sequence.

$$G : \{0,1\}^k \rightarrow \{0,1\}^{poly(k)}$$

For convenience we reproduce here the formal definition we gave in the previous lecture for pseudorandom generator.

Definition 15.1 *A deterministic polynomial-time algorithm G is called a pseudorandom generator if there exist a stretching function $l : N \rightarrow N$, so that for any probabilistic polynomial-time algorithm D, for any positive polynomial p, and for all sufficiently large k's*

$$|Prob[D(G(U_k)) = 1] - Prob[D(U_{l(k)}) = 1]| < \frac{1}{p(k)}$$

Where U_n denotes the uniform distribution over $\{0,1\}^n$ and the probability is taken over U_k (resp., $U_{l(k)}$) as well as over the coin tosses of D.

Suppose we have such a pseudorandom generator then for every $\epsilon > 0$ we can shrink the amount of randomness used by an algorithm A, deciding a language in BPP, from $poly(n)$ to n^ϵ (where n is the length of the input). The shrinking of randomness will not cause significance change in the behavior of the algorithm, meaning that it is infeasible to find a long enough input on which the algorithm which uses less randomness will decide differently than the original one. The problem is that the above doesn't indicate that there are no inputs for which A will act differently when using the pseudorandom source, but that they are hard to find. Thus in order to derandomize BPP we will need stronger notion of pseudorandom generator. We will need a pseudorandom generator which can fool any small (polynomial size) circuit (and hence any polynomial time algorithm).

Definition 15.2 *A deterministic polynomial-time algorithm G is called a non-uniformly strong pseudorandom generator if there exist a stretching function $l : N \to N$, so that for any family $\{C_k\}_{k \in L}$ of polynomial-size circuits, for any positive polynomial p, and for all sufficiently large k's*

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| < \frac{1}{p(k)}$$

Theorem 15.3 *If such G exist which is robust against polynomial-size circuit then*

$$\forall \epsilon > 0 \quad BPP \subseteq Dtime(2^{n^\epsilon})$$

Proof: $L \in BPP$ implies that the algorithm A for deciding L doesn't only take an input x with size n but also uses randomness R with size l (when we write for short $A(x)$ we really mean $A(x, R)$). The relation between the size of the input and the size of the randomness is $l = poly(n)$. Let us construct a new algorithm A' which will use less randomness than A but will act similar to A' for almost all inputs.

$$A'(x, s) \stackrel{\text{def}}{=} A(x, G(s))$$

Where $s \in \{0,1\}^{n^\epsilon}$.
A' uses little randomness and we claim that A' only differ from A for finitely many x's.

Proposition 15.1.1 *For all but finitely many x's*

$$|Prob[A(x) = 1] - Prob[A'(x) = 1]| < \frac{1}{6}$$

Proof: The idea of the proof is that if there were infinitely many x's for which A and A' differ, then we could distinguish G's output from a random string, in contradiction to the assumption that G is a pseudorandom generator.

In order to contradict Definition 15.2 it suffices to present a family $\{C_k\}$ of small circuits for which

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| \geq \frac{1}{6}$$

Suppose towards contradiction that for infinitely many x's A and A' behave differently, i.e

$$Prob[A(x) = 1] - Prob[A'(x) = 1] \geq \frac{1}{6}$$

Then we incorporate these inputs and A into a family of small circuits as follows

$$x \to C_x(\alpha) \stackrel{\text{def}}{=} A(x, \alpha)$$

This will enable us to distinguish the output of the generator from a uniformly distributed source. The circuit C_k will be one of $\{C_x : A(x) \text{ uses } k \text{ coin tosses}\}$. Note that if there are infinitely many x's on which A and A' differ then there are infinitely many sizes of x's on which they differ. The amount of randomness used by the algorithm is polynomial with respect to the size of the input.

The idea behind this construction is that

$$C_k(G(U_k)) \equiv A'(x) \quad and \quad C_k(U_{l(k)})) \equiv A(x)$$

Hence we have a family of circuits such that

$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_{l(k)}) = 1]| \geq \frac{1}{6}$$

Which is a contradiction to the definition of pseudorandom generator. ■

Saying that algorithm A decides BPP means that if $x \in L$ the probability that A will say $'YES'$ is greater than $\frac{2}{3}$ and if $x \notin L$ the probability that A will say $'YES'$ is smaller than $\frac{1}{3}$. By the above proposition, for all but finitely many x's $|Prob[A(x) = 1] - Prob[A'(x) = 1]| < 1/6$. Thus, for all but finitely many x's

$$x \in L \Rightarrow prob\ [A(x, U_n) = 1] \geq \frac{2}{3} \Rightarrow prob\ [A'(x, s) = 1] > \frac{1}{2}$$

$$x \notin L \Rightarrow prob\ [A(x, U_n) = 1] \leq \frac{1}{3} \Rightarrow prob\ [A'(x, s) = 1] < \frac{1}{2}$$

Now we define the algorithm A'' which incorporate these finitely many inputs, and for all other inputs it loops over all possible $s \in \{0,1\}^{n^\epsilon}$ (seeds of G) and decides by majority.

Algorithm A'': On input x proceed as follows.

```
if x is one of those finitely x's
    return the known answer
else
    for all s ∈ {0,1}^(n^ε)
       run A'(x,s)
    return the majority of A' answers
```

Clearly this A'' deterministicly decides L and run in time $2^{n^\epsilon} \cdot poly(n)$ as required. ■

15.2 New notion of Pseudorandom generator

The time needed for A'' to decide if an input x is in L or not was exponential in the length of it's seed s. For simulation purposes if the random seed is logarithmic in the size of the input, running the pseudorandom generator exponential time in the length of the seed is really running it polynomial time with respect to the length of the input x. Thus the time needed for simulating a randomized algorithm which run in polynomial time remains polynomial even if we allow the pseudorandom generator to run exponential time (with logarithmic size seed). In general, for the purpose of derandomizing BPP we may allow the generator to use exponential time. There seems to be no point insisting that the generator will use only polynomial time in the length of the seed when the running time of the simulating algorithm has an exponential factor in the size of the seed.

Motivated as above, we now introduce a new notion of pseudorandom generator with the following properties.

1. Indistinguishable by a polynomial-size circuit.
2. Can run in exponential time ($2^{O(k)}$ on k-bit seed).

The efficiency of the generator which was one of the key aspects is relaxed in the new notion. This new notion may seem a little unfair at first since we only give polynomial time to the distinguisher and allow the generator to run exponential time. It even suggests that if we give the seed as an extra input, polynomial size circuit wouldn't be able to take advantage of that because it wouldn't be able to evaluate the generator on the given seed. The relaxation allow us to construct pseudorandom generators under seemingly weaker conditions than the ones required for polynomial time pseudorandom generators (the assumption about existence of one-way functions).

15.3 Construction of non-iterative pseudorandom generator

The difference between the definition of pseudorandom generator that we introduced in this lecture and the definition of pseudorandom that we had before (the one usually used for cryptographic purposes) is that we allow the generator to run in time exponential in it's input size.

This difference enables us to construct a random generator under possibly weaker conditions without damaging the derandomization process. In this section we will demonstrate how to construct such a generator using "unpredictable predicate" and a structure called "design" (we will give a precise definition later). In the construction we use two main tools which we assume to exist. We assume the existence of such a predicate and the existence of a design, but later we will show how to construct such a design hence the only real assumption that we make is the existence of a predicate.

In the previous class we proved that pseudorandom generators exist if and only if one-way permutation exist.
We will show (in section 15.3.2 below) that this assumption is not stronger than the previous assumption, i.e the existence of one-way permutation implies the existence of a predicate (but not necessarily the opposite way). So it seems better to use the new assumption which may be true even if there exist no one-way function.

The previous construction uses a one-way permutation $f : \{0,1\}^l \rightarrow \{0,1\}^l$ in the following way:

> From a random string $S_0 = S$ (the seed), compute a sequence $\{S_i\}$ by $S_{i+1} = f(S_i)$. The random bits are then extracted from this sequence using a hard-core predicate. We proved that a small circuit that is not fooled by this bit sequence can be used to

demonstrate that f is not a one-way permutation because it can be used to compute f^{-1}.

The construction that we will give here has a different nature. Instead of creating the bits sequentially we will generate them in parallel. Unpredictable predicates supply an easy way to construct one additional random looking bit from the seed. The problem is to generate more bits. We will do it by invoking the predicate on nearly disjoint subsets of the bits in the seed. This will give us bits which are nearly independent (such that no polynomial-size circuit will notice that they has some dependence).

15.3.1 Parameters

- k - Length of seed.
- m - Length of output (a polynomial in k).
- We want the generator to produce an output that is indistinguishable by any polynomial-size circuit (in k, or equivalently in m).

15.3.2 Tool 1: An unpredictable predicate

The first tool that we will need in order to construct a pseudorandom generator is a predicate that can not be approximated by polynomial sized circuits.

Definition 15.4 *We say that an $exp(l)$-computable predicate $b : \{0,1\}^l \rightarrow \{0,1\}$ is unpredictable by small circuits (or simply unpredictable) if for every polynomial $p(\cdot)$, for all sufficiently large l's and for every circuit C of size $p(l)$:*

$$Prob\,[C(U_l) = b(U_l)] < \frac{1}{2} + \frac{1}{p(l)}$$

This definition require that small circuits attempting to compute b have only negligible fraction of advantage over unbiased coin toss. This is a real assumption, because we don't know of a way to construct such a predicate (unlike the next tool that we will show how to construct later).

To evaluate the strength of our construction we prove, in the next claim, that the existence of unpredictable predicate is guaranteed if one-way permutation exist. The other way is not necessarily true, so it is possible that our construction can be applied (if someone will find a provable unpredictable predicate) while other constructions that depend on the existence of one-way permutations are not applicable (if someone will prove that such functions do not exist).

Claim 15.3.1 *If f_0 is a one-way permutation and b_0 is a hard-core of f_0, then $b(x) \stackrel{\text{def}}{=} b_0\left(f_0^{-1}(x)\right)$ is an unpredictable predicate.*

Proof:
We begin by noting that b is computable in exponential time because it takes exponential time to invert f and together with the computation of b_0, the total time is no more than exponential in the size of x.

The second property that we need to show is that it is impossible to predict b. In order to prove this property we use the variable $y \stackrel{\text{def}}{=} f_0^{-1}(x)$ to get:

$$b\left(f_0(y)\right) = b_0(y)$$

Assume towards contradiction that b is predictable. This means that there exist an algorithm A and a polynomial $p(\cdot)$ such that for infinite number of n's :

$$Prob\left[A(U_n) = b(U_n)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

A is a polynomial-size algorithm which can predict b with a noticeable bias. But f is a permutation so we may write:

$$Prob\left[A\left(f_0(U_n)\right) = b\left(f_0(U_n)\right)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

Hence, from the definition of b we get:

$$Prob\left[A\left(f_0(U_n)\right) = b_0(U_n)\right] \geq \frac{1}{2} + \frac{1}{p(n)}$$

which is a contradiction to the definition of b_0 as a hard core. ■

Recall that we demonstrated a generic hard-core predicate (the inner product *mod* 2) that is applicable to an arbitrary one-way permutation, hence essentially the last claim is only assuming the existence of one-way permutation (because the hard-core predicate can always be found). And we succeeded to show that the existence of unpredictable predicate may only be a weaker assumption than the existence of a one-way permutation. We did not prove that it is really a weaker assumption but we did show that it is not stronger. It may be the case that both assumptions are the same but we don't know of any proof for such a claim.

The assumption that we use to construct a generator is the existence of a "hard" predicate. The word "hard" means that the predicate can not be approximated by small circuits. The hardness of a predicate is measured by two parameters:

- The size of the circuit.
- The closeness of approximation.

In this notes we use polynomial-size circuits and polynomial approximation. Other papers demonstrated that similar process can be carried out with different conditions on this hardness parameters. In particular, the closeness of approximation parameter is greatly relaxed.

15.3.3 Tool 2: A design

The task of generating a single bit from a random bits seems easy if you have the predicate that we assumed to exist in the previous section because the output of the predicate must look random to every polynomial-size circuit. The problem is how can we generate more than one bit. We will do this using a collection of nearly disjoint sets to get random bits that are almost mutually independent (almost means indistinguishable by a polynomial-size circuits). To formalize this idea we introduce the notion of a design:

Definition 15.5 *A collection of m subsets $\{I_1, I_2, ..., I_m\}$ of $\{1, ..., k\}$ is a (k,m,l)-design (or simply design) if the following three properties exist:*

1. *For every $i \in \{1, ..., m\}$,*
$$|I_i| = l$$
2. *For every $i \neq j \in \{1, ..., m\}$,*
$$|I_i \cap I_j| = o(\log\ k)$$
3. *The collection is constructible in $exp(k)$-time.*

In our application the set $\{1, ..., k\}$ is all bit locations in the seed, and the subsets $\{I_1, I_2, ..., I_m\}$ correspond to different subsequences extracted from the seed. For example, one may look at a ten bit seed
$$S = \langle 1010010110 \rangle$$
The subset $I = \{1, 5, 7\} \subset \{1, ..., 10\}$ correspond to the first, the fifth and the seventh bits in the seeds which are, in this example :
$$S[I] = \langle 100 \rangle$$
In general, for $S = \langle \sigma_1 \sigma_2 \cdots \sigma_k \rangle$, and $I = \{i_1, ..., i_l\} \subset \{1, ..., k\}$ we use the notation:
$$S[I] \stackrel{\text{def}}{=} \langle \sigma_{i_1} \sigma_{i_2} \cdots \sigma_{i_k} \rangle$$

15.3.4 The construction itself

We now want to construct a pseudorandom generator with a polynomial stretching function. The size of the seed will be k and the size of the output will be $m = m(k)$ which is a polynomial in k. Suppose we have a design $\{I_1, ..., I_m\}$ which consist of m subsets of the set $\{1, ..., k\}$. If these subset where completely disjoint then it is obvious (from Definition 15.4) that for every unpredictable predicate b, the sequence $\langle b(S[I_1])\ b(S[I_2]) \cdots b(S[I_m]) \rangle$ is unpredictable. Since unpredictability implies pseudorandomness we get pseudorandom generator. We will show that this is also true when the intersection of any two subsets is logarithmic in k and m (i.e, this is a design). Our generator will blow up seeds by applying the unpredictable predicate to every subsequence corresponding to subsets in the design.

Proposition 15.3.2 *Let $b : \{0,1\}^{\sqrt{k}} \to \{0,1\}$ be unpredictable predicate, and $\{I_1, ..., I_m\}$ be a $(k, m, \sqrt{k})$ - design, then the following function is a pseudorandom generator (as defined in section 15.2):*
$$G(S) \stackrel{\text{def}}{=} \langle b(S[I_1])\ b(S[I_2]) \cdots b(S[I_m]) \rangle$$

Proof:
Based on the definition of unpredictable predicate and the definition of a design it follows that it take no more than exponentially many steps to evaluate G.

We will show now that no small circuit can distinguish the output of G from a random sequence. Suppose towards contradiction that there exist a family of polynomial-size circuits $\{C_k\}_{k \in \mathsf{N}}$ and a polynomial $p(\cdot)$ such that for infinitely many k's
$$|Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_m) = 1]| > \frac{1}{p(k)}$$

(This is the negation of G being a pseudorandom generator because the definition of pseudorandom generator demands that for every sufficiently large k's we have $|\cdots| < \frac{1}{p(k)}$ which imply that there are only finitely many k's with $|\cdots| > \frac{1}{p(k)}$)

We will assume that this expression is positive and remove the absolute value, because if we have infinitely many k's such that the above is true, we have that half of them have the same sign. We may take the sign as we want since we can always take other sequence of circuits with the reversed sign. So without loss of generality we assume that for infinitely many k's

$$Prob[C_k(G(U_k)) = 1] - Prob[C_k(U_m) = 1] > \frac{1}{p(k)}$$

For any $0 \leq i \leq m$ we define a "hybrid" distribution H_k^i on $\{0,1\}^m$ as follows: the first i bits are chosen to be the first i bits of $G(U_k)$ and the other $m - i$ bits are chosen uniformly at random :

$$H_k^i \stackrel{\text{def}}{=} G(U_k)_{[1,\ldots,i]} \circ U_{m-i}$$

In this notation we get $H_k^0 = U_m$ and $H_k^m = G(U_k)$. If we look at the function :

$$f_k(i) \stackrel{\text{def}}{=} Prob[C_k(H_k^i) = 1]$$

We get that, since $f_k(m) - f_k(0) > \frac{1}{p(k)}$, there must be some $0 \leq i_k \leq m$ such that:

$$f_k(i_k + 1) - f_k(i_k) > \frac{1}{m} \cdot \frac{1}{p(k)}$$

So we know that there exist a circuit which behaves significantly different if the i_k'th bit is taken randomly or from the generator (the difference is greater than one over the polynomial $p'(k) \stackrel{\text{def}}{=} m \cdot k$). That is:

$$Prob[C_k(H_k^{i_k+1}) = 1] - Prob[C_k(H_k^{i_k}) = 1] > \frac{1}{p'(k)}$$

We will use this circuit to construct another circuit which will be able to guess the next bit with some bias. On i_k bits of input and $m - i_k$ bits of random, the new circuit C'_k will behave as follows:

$$C'_k(\langle y_1, ..., y_{i_k}\rangle, \langle R_{i_k+1}, ..., R_m\rangle) \stackrel{\text{def}}{=} \begin{cases} R_{i_k+1} & if\ C_k(\langle y_1, ..., y_{i_k}, R_{i_k+1}, ..., R_m\rangle) = 1 \\ 1 - R_{i_k+1} & otherwise \end{cases}$$

where $\langle y_1, ..., y_m\rangle \stackrel{\text{def}}{=} G(U_k)$.

The idea behind this construction is that we know that the the probability that C_k will return 1 is significantly different whether R_{i_k+1} equals y_{i_k+1} or not. This is true because when R_{i_k+1} equals y_{i_k+1} we see that C_k is getting $H_k^{i_k+1}$ as input and otherwise it get $H_k^{i_k}$. The consequence of this fact is that we can use the answer of C_k to distinguish these two cases.

To analyze the behavior of C'_k more formally we look at two events (or Boolean variables):

$$A \stackrel{\text{def}}{=} (C_k(\langle y_1, ..., y_{i_k}, R_{i_k+1}, ..., R_m\rangle) = 1)$$
$$B \stackrel{\text{def}}{=} (R_{i_k+1} = y_{i_k+1})$$

Notice that C'_k will return y_{i_k+1} in two distinct scenarios. The first scenario is when $R_{i_k+1} = y_{i_k+1}$ and C_k returns 1, and the second scenario is when $R_{i_k+1} \neq y_{i_k+1}$ and C_k returns 0. Using the above notation we get that:

$$Prob[C'_k = y_{i_k+1}] = Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A^c|B^c]$$

Since

- $Prob[A] = f(i_k) = Prob[C_k(H_k^{i_k}) = 1]$
- $Prob[A|B] = f(i_k + 1) = Prob[C_k(H_k^{i_k+1}) = 1]$
- $Prob[B] = Prob[B^c] = \frac{1}{2}$
- $Prob[A] = Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A|B^c]$

we get that:

$$\begin{aligned}
Prob[C'_k = y_{i_k+1}] &= Prob[B] \cdot Prob[A|B] + Prob[B^c] \cdot Prob[A^c|B^c] \\
&= Prob[B] \cdot Prob[A|B] + Prob[B^c] - Prob[B^c] \cdot Prob[A|B^c] \\
&= Prob[B] \cdot Prob[A|B] + Prob[B^c] - (Prob[A] - Prob[B] \cdot Prob[A|B]) \\
&= \frac{1}{2} + Prob[A|B] - Prob[A] \\
&= \frac{1}{2} + f(i_k + 1) - f(i_k) \\
&> \frac{1}{2} + \frac{1}{p'(k)}
\end{aligned}$$

Hence the conclusion is:

$$[Prob[C'_k(G(U_k)_{[1,\ldots,i_k]}) = G(U_k)_{i_k+1}] > \frac{1}{2} + \frac{1}{p'(k)}$$

(We use subscript notation to take partial bits of a given bit sequence. In this particular case $G(U_k)_{[1,\ldots,i_k]}$ is the first i_k bits of $G(U_k)$ and $G(U_k)_{i_k+1}$ is the $i_k + 1$'s bit of $G(U_k)$.)

That is, C'_k can guess the i_k'th bit of $G(U_k)$ with $\epsilon \stackrel{\text{def}}{=} \frac{1}{p'(k)}$ advantage over unbiased coin toss.

Let us now extend C' to also get complete seed as input (in addition to $S[I_1], S[I_2], \ldots, S[I_{i_k}]$)

$$C''_k(S \circ G(S)_{[1,\ldots,i_k]}) \stackrel{\text{def}}{=} C'_k(G(S)_{[1,\ldots,i_k]})$$

Since $G(U_k)_{i_k+1}$ is defined to be $b(S[I_{i_k+1}])$, we have:

$$Prob_s[C''_k(S \circ G(S)_{[1,\ldots,i_k]}) = b(S[I_{i_k+1}])] > \frac{1}{2} + \epsilon$$

We claim that there exist $\alpha \in \{0,1\}^{k-|I_{i_k+1}|}$ such that:

$$Prob_s[C''_k(S \circ G(S)_{[1,\ldots,i_k]}) = b(S[I_{i_k+1}]) \mid S[\overline{I_{i_k+1}}] = \alpha] > \frac{1}{2} + \epsilon$$

This claim is true because if we look at a random selection of S as two independent selections of $S[I_{i+1}]$ and $S[\overline{I_{i+1}}]$ we see that the average over the second selection is greater than $\frac{1}{2}+\epsilon$ so there must be an element with weight greater than that.

Now we come to the key argument in this proof. By incorporating C''_k with α to a new circuit, we get a circuit that can approximate the value of $b(S[I_{i_k+1}])$ but it need the "help" of $b(S[I_1]), b(S[I_2]), ..., b(S[I_{i_k}])$. We will show now that we can do without this "help" when $\{I_1, ..., I_m\}$ is a design (note that we didn't use the fact that this is a design until now).

To prove that it is possible to build a circuit that doesn't use the "help", we need to show that there exist a polynomial-size circuit that get only $S[I_{i_k+1}]$ and can approximate $b(S[I_{i_k+1}])$. To do this we use the fact that all the bits in $S[I_1], S[I_2], ..., S[I_{i_k}]$ depend only on small fraction of $S[I_{i_k+1}]$ so circuits for computing these bits are relatively small and we can incorporated them in a circuit that include all possible values of these bits. Details follows.

To elaborate the last paragraph, recall that the intersection of any two subsets in a design is at most $O(log\ k)$, hence given $S[\overline{I_{i_k+1}}] = \alpha$ we know that for every $i \leq i_k$ the bits in I_i are fixed except for $O(log\ k)$ bits that may be in $S[I_{i_k+1}]$, and are given as part of the input. Since there are at most $O(log\ k)$ such bits, there exist a polynomial-size circuit that can "precompute" the value of b for every combination of these bits.

The first part of this circuit is a collection of tables, one for every I_i. Each table is indexed by all possible values of the "free bits" of an I_i (i.e., these in I_{i_k+1}). The entry for every such value (of $S[I_{i_k+1} \cap I_i]$) contains the corresponding $b(S[I_i])$ (under $S[\overline{I_{i_k+1}}] = \alpha$).

The second part of this circuit just implements a "selector"; that is, it uses the bits of $S[I_{i_k+1}$ in order to obtain the appropriate values of $b(S[I_1]), b(S[I_2]), ..., b(S[I_{i_k}])$ from the corresponding tables mentioned above.

Since we have polynomially many entries in every table and polynomially many tables we get that this is a polynomial-size circuit.

The conclusion is that we got a circuit that can approximate b with a non negligible advantage over unbiased coin toss. There exist infinitely many such circuits for infinitely many k's, which contradict the assumption of b being unpredictable predicate. ■

15.4 Constructions of a design

In this section we describe how to construct a design that can be used for the generator that we introduced in the preceding section. We need to construct m different subsets of the set $\{1, ..., k\}$ each has size l with small intersections.

15.4.1 First construction: using $GF(l)$ arithmetic

Assume without loss of generality that l is a prime factor and let $k = l^2$ (if l is not a prime factor pick the smallest power of 2 which is greater than l).

For the field $F \stackrel{\text{def}}{=} GF(l)$, we get that the Cartesian product $F \times F$ contains $k = l^2$ elements which we identify, with the k elements of $\{1, ..., k\}$. Every number in $\{1, ..., k\}$ is assigned to a pair in $F \times F$ (in a one-to-one correspondence). In the forgoing discussion we will interchange the pair and it's representative in $\{1, ..., k\}$ freely.

For every polynomial $p(\cdot)$ of degree d over F introduce the subset:

$$I_p \stackrel{\text{def}}{=} \{\langle e, p(e)\rangle : e \in F\} \subseteq F \times F$$

We get that:

1. The size of each set is $|I_p| = |F| = l$.
2. For every two polynomials $p \neq q$ the sets I_p and I_q intersects in at most d points (that is, $|I_p \cap I_q| \leq d$). This is true since:

   $$|\{\langle e, p(e)\rangle : e \in F\} \cap \{\langle e, q(e)\rangle : e \in F\}| = |\{\langle e : p(e) = q(e)\}|$$

   But the polynomial $p(e) - q(e)$ has degree smaller or equal to d so it can only have d zeroes (due to the Fundamental Theorem of Algebra).
3. There are $|F|^{d+1} = l^{d+1}$ polynomials over $GF(l)$ so for every polynomial $P(\cdot)$ we can find d such that the number of sets is greater than $P(l)$.
4. This structure is constructible in exponential (actually polynomial) in k time because all we need to do is simple arithmetic in $GF(l)$.

The conclusion is that we have a design (see Definition 15.5) that can be applied in the construction of pseudorandom generator that we gave above. This design remove the second assumption that we made about the existence of a design, so we get (as promised) that the only assumption needed in order to allow derandomization of BPP is that there exist an unpredictable predicate.

15.4.2 Second construction: greedy algorithm

In this subsection we introduce another construction of a design. We call this algorithm greedy because it just scans all the subsets of the needed size until it find one that doesn't overlap all previously selected subsets too much. This algorithm is "simpler" than the one we gave in the previous subsection but we need to show that it work correctly.

Consider the following parameters:

- $k = l^2$
- $m = poly(k)$
- We want that for all i to have $|I_i| = l$ and for all $i \neq j$, $|I_i \cap I_j| = O(log\ k)$

For these parameters we give a simple algorithm that scans all subsets one by one to find the next set to include in the design:

for $i = 1$ to m

    for all $I \subset [k]$, $|I| = l$ do

        $flag \leftarrow FALSE$

        for $j = 1$ to $i - 1$

if $|I_i \cap I_j| > log\ k$ then $flag \leftarrow TRUE$

if $flag = TRUE$ then $I_i \leftarrow I$

This algorithm run in an exponential time because there are 2^l subsets of size l and we scan them m times. Since $m \cdot 2^l < 2^k$ we get that even if we had to scan all the subsets in every round we could finish the process in time exponential in k.
To prove that the algorithm works it is enough to show that if we have $I_1, I_2, ..., I_{i-1}$ such that

1. $i \leq m$
2. For every $j < i : |I_j| = l$
3. For every $j_1, j_2 < i : |I_{j_1} \cap I_{j_2}| \leq log\ m + 2$

Then there exist another set $I_i \subset [k]$ such that $|I_i| = l$ and for every $j < i : |I_j \cap I_i| < 2 + log\ m$. We prove this claim using the Probabilistic Method. That is, we will show that most choices of I_i will do. In fact we show the following

Claim 15.4.1 *Let $I_1, I_2, ..., I_{i-1} \subset [k]$ each of size l. Then there exists an l-set, I, such that for all j's, it is the case that $|I_j \cap I| \leq \log\ m + 2$.*

Proof: We first consider the probability that a uniformly chosen l-set has a large intersection with a fixed l-set, denoted S. Actually, it will be easier to analyze the intersection of S with a set R selected at random so that for every $i \in [k]$

$$Prob[i \in R] = \frac{2}{l}$$

That is, the size of R is a random variable, with expectation $2k/l$ (which equals $2l$). We will show that with very high probability the intersection of R with S is not too big, and that very high probability $|R| \geq l$ (and so we can find an appropriate l-set). Details follow.
Let s_i be the i'th element in S sorted in any order (e.g., the natural increasing order). Consider the sequence $\{X_i\}_{i=1}^{l}$ of random variables defined as

$$X_i \stackrel{\text{def}}{=} \begin{cases} 1 & if\ s_i \in R \\ 0 & otherwise \end{cases}$$

Since these are independent Boolean random variables with $Prob[X_i = 1] = \frac{2}{l}$ for each i, we can use Chernoff's bound to get:

$$\begin{aligned} Prob\left[|S \cap R| > 2 + log\ m\right] &= Prob\left[\sum_{i=1}^{l} X_i > 2 + log\ m\right] \\ &= Prob\left[\frac{\sum_{i=1}^{l} X_i}{l} > \frac{2}{l} + \frac{log\ m}{l}\right] \\ &< Prob\left[\left|\frac{\sum_{i=1}^{l} X_i}{l} - \frac{2}{l}\right| > \frac{log\ m}{l}\right] \\ &< 2 \cdot e^{-log^2\ m} \ll 1/2m \end{aligned}$$

It follows that for R selected as above, the probability that there exists an I_j so that $|R \cap I_j| > 2 + \log m$ is bounded above by $\frac{i-1}{2m} < \frac{1}{2}$. The only problem is that such an R is not necessarily of size l. We shall show that with high probability the size of R is at least l, and so it contains a subset which will do (as I_i).
Consider the sequence $\{Y_i\}_{i=1}^k$ defined as

$$Y_i \stackrel{\text{def}}{=} \begin{cases} 1 & if\ i \in R \\ 0 & otherwise \end{cases}$$

Then, applying Chernoff's Bound we get:

$$\begin{aligned} Prob[|R| < l] &\leq Prob[|\tfrac{1}{k}\textstyle\sum_{i=1}^{k} Y_i - \tfrac{2}{l}| < \tfrac{1}{l}] \\ &< 2 \cdot e^{-2} \ \ll \ \tfrac{1}{2} \end{aligned}$$

Thus, for R selected as above, the probability that either $|R| < l$ or there exists an I_j so that $|R \cap I_j| > 2 + \log m$ is *strictly smaller than 1*. Therefore , there exists a set R such that $|R| \geq l$ and yet, for every $j < i$, we have $|R \cap I_j| \leq 2 + \log m$. Any l-subset of R qualifies as the set asserted by the claim. ■

We stress that this discussion about a randomly selected set is not part of the algorithm. The algorithm itself is totally deterministic. The randomness is just in our discussion – it serves as a tool to show that the algorithm will always find what it is looking for in every step.

Bibliographic Notes

This lecture is based on [4]. Further derandomization results, building on [4], can be found in [1], [2] and [3]. Specifically, in the latter paper a "full derandomization of BPP" is provided under the assumption that there exists a language $L \in \mathcal{E}$ having almost-everywhere exponential circuit complexity. That is: *Let* $\mathcal{E} \stackrel{\text{def}}{=} \cup_c \text{Dtime}(t_c)$, *with* $t_c(n) = 2^{cn}$. *Suppose that there exists a language* $L \in \mathcal{E}$ *and a constant* $\epsilon > 0$ *such that, for all but finitely many n's, any circuit* C_n *which correctly decides* L *on* $\{0,1\}^n$ *has size at least* $2^{\epsilon n}$. *Then,* $\mathcal{BPP} = \mathcal{P}$.

1. L. Babai, L. Fortnow, N. Nisan and A. Wigderson. BPP has Subexponential Time Simulations unless EXPTIME has Publishable Proofs. *Complexity Theory*, Vol. 3, pages 307–318, 1993.

2. R. Impagliazzo. Hard-core Distributions for Somewhat Hard Problems. In *36th FOCS*, pages 538–545, 1995.

3. R. Impagliazzo and A. Wigderson. P=BPP if E requires exponential circuits: Derandomizing the XOR Lemma. In *29th STOC*, pages 220–229, 1997.

4. N. Nisan and A. Wigderson. Hardness vs Randomness. *JCSS*, Vol. 49, No. 2, pages 149–167, 1994.

Lecture 16

Derandomizing Space-Bounded Computations

Notes taken by Eilon Reshef

Summary: This lecture considers derandomization of space-bounded computations. We show that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, namely, any bounded-probability Logspace algorithm can be deterministically emulated in $O(\log^2 n)$ space. We show that $\mathcal{BPL} \subseteq \mathcal{SC}$, namely, any such algorithm can be deterministically emulated in $O(\log^2 n)$ space and (simultaneously) in polynomial time.

16.1 Introduction

This lecture considers derandomization of space-bounded computations. Whereas current techniques for derandomizing time-bounded computations rely upon unproven complexity assumptions, the derandomization technique illustrated in this lecture stands out in its ability to derive its results exploiting only the pure combinatorial structure of space-bounded Turing machines.

As in previous lectures, the construction yields a pseudorandom generator that "fools" Turing machines of the class at hand, when in our case the pseudorandom generator generates a sequence of bits that looks truly random to any space-bounded machine.

16.2 The Model

We consider probabilistic Turing machines along the lines of the online model discussed in Lecture 5. A probabilistic Turing machine M has four tapes:

1. A read-only bidirectional input tape.
2. A read-only unidirectional random tape.
3. A read-write bidirectional work tape.
4. A write-only unidirectional output tape.

We consider $\mathcal{BPSPACE}(\cdot)$, the family of *bounded probability* space-bounded complexity classes. These classes are the natural two-sided error counterparts of the single-sided error classes $\mathcal{RSPACE}(\cdot)$, defined in Lecture 7 (Definition 7.9).

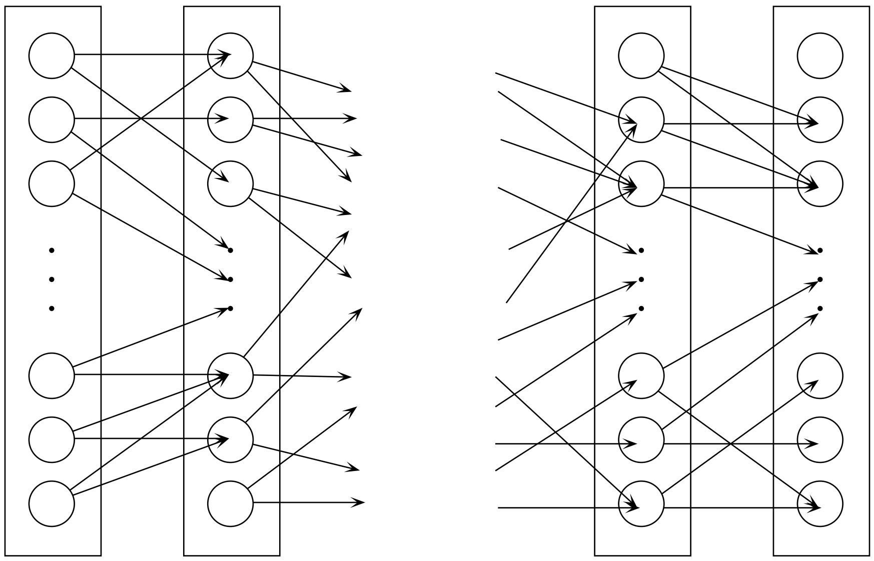


Figure 16.1: An Execution Graph of a Bounded-Space Turing Machine

Formally,

Definition 16.1 *For any function $s(\cdot) : \mathsf{N} \to \mathsf{N}$, the complexity class $\mathcal{BPSPACE}(s(\cdot))$ is the set of all languages L for which there exists a randomized Turing machine M such that on an input x*

1. *M uses at most $s(|x|)$ space.*
2. *The running time of M is bounded by $\exp(s(|x|))$.*
3. *$x \in L \Rightarrow \Pr[M(x) = 1] \geq 2/3$.*
4. *$x \notin L \Rightarrow \Pr[M(x) = 1] < 1/3$.*

Recall that condition (2) is important, as otherwise $\mathcal{NSPACE}(\cdot) = \mathcal{RSPACE}(\cdot) \subseteq \mathcal{BPSPACE}(\cdot)$. As usual, we are interested in the cases where $s(\cdot) \geq \log(\cdot)$. In particular, our techniques derandomize the complexity class $\mathcal{BPL}$, defined as

Definition 16.2 $\mathcal{BPL} \stackrel{\triangle}{=} \mathcal{BPSPACE}(\log)$

Throughout the rest of the discussion we assume that the problems at hand are decision problems, that all functions are space-constructible, and that all logarithms are of base 2.

16.3 Execution Graphs

We represent the set of possible executions of a $\mathcal{BPSPACE}(\cdot)$ Turing machine M on an input x, $|x| = n$, as a layered directed graph $G_{M,x}$ (see Figure 16.1).

A vertex in the i-th layer of $G_{M,x}$ corresponds to a possible configuration of M after it has consumed exactly i random bits, i.e., when the head reading from the random tape is on the i-th cell. Thus, the i-th layer of $G_{M,x}$ corresponds to the set of all such possible configurations. $G_{M,x}$

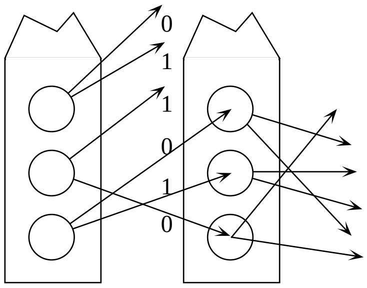


Figure 16.2: Edges in the Execution Graph $G_{M,x}$

contains an edge from a configuration vertex in the i-th layer to a configuration vertex in the $(i+1)$-th layer if there is a possible transition between the two configurations.

Formally, $G_{M,x} = (V_{M,x}, E_{M,x})$ is defined as follows. For each $i = 1, \ldots, D$, with $D \leq \exp(s(n))$, let the i-th layer $V^i_{M,x}$ be the set of all possible configurations of M given that M has consumed exactly i random bits. The vertex set $V_{M,x}$ is a union of all layers $V^i_{M,x}$. For each vertex $v \in V^i_{M,x}$, the edge set $E_{M,x}$ contains two outgoing directed edges to vertices $f_0(v), f_1(v)$ in $V^{i+1}_{M,x}$, where $f_0(v)$ (resp. $f_1(v)$) corresponds to the configuration M reaches following the sequence of transitions carried out after reading a "0" (resp. "1") bit from the random tape, and until the next bit is read (see Figure 16.2). For the convenience of the exposition below, assume that each of the two edges is labeled by its corresponding bit, i.e., "0" or "1".

Note that when the location of the head on the random tape is fixed, a configuration of M is fully determined by the contents of the work tape and by the positions of the heads on the work tape and on the input tape. Thus,

$$|V^i_{M,x}| \leq 2^{s(n)} \cdot s(n) \cdot n \leq \exp(s(n)).$$

The initial configuration of M corresponds to a designated vertex $v_0 \in V^0_{M,x}$. Similarly, the set of final vertices $V^D_{M,x}$ is partitioned into the set of accepting configurations V_{ACC} and the set of rejecting configurations V_{REJ}.

A *random walk* on $G_{M,x}$ is a sequence of steps, emanating from v_0, and proceeding along the directed edges of $G_{M,x}$, where in each step the next edge to be traversed is selected uniformly at random. Under this interpretation, the probability that a machine M accepts x is exactly the probability that a random walk emanating from v_0 reaches a vertex in V_{ACC}.

In contrast, a *guided walk* on $G_{M,x}$ with a *guide* R is a sequence of steps, emanating from v_0, and proceeding along the directed edges of $G_{M,x}$, where in the i-th step the next edge to be traversed is determined by the i-th bit of the guide R, i.e., the edge taken is the one whose label is equal to the value of the i-th bit of R.

Let $ACC(G_{M,x}, R)$ denote the event that a guided walk on $G_{M,x}$ with a guide R reaches a vertex in V_{ACC}. In this view,

$$\Pr[M \text{ accepts } x] \;=\; \Pr_{R \in_R \{0,1\}^D}[ACC(G_{M,x}, R)],$$

when R is selected uniformly at random from the set $\{0,1\}^D$.

Thus, a language L is in $\mathcal{BPL}$ if there exists a Turing machine M with a space bound of $s(n) = O(\log(n))$ such that for $D(n) = \exp(s(n)) = \mathrm{poly}(n)$,

- Whenever $x \in L$, $\Pr_{R \in_R \{0,1\}^{D(n)}}[ACC(G_{M,x}, R)] \geq 2/3$.
- Whenever $x \notin L$, $\Pr_{R \in_R \{0,1\}^{D(n)}}[ACC(G_{M,x}, R)] < 1/3$.

A (D, W)-*graph* G is a graph that corresponds to the execution of some $s(\cdot)$-space-bounded Turing machine on an input x, with a "depth" (number of layers) of D, $D \leq \exp(s(|x|))$ and a "width" (number of vertices in each layer) of W, $W \leq \exp(s(|x|))$. In the sequel, we present a derandomization method that "fools" any (D, W)-graph by replacing the random guide R with a pseudorandom guide R'.

16.4 Universal Hash Functions

The construction below is based upon a universal family of hash functions,

$$\mathcal{H}_\ell \;=\; \{h : \{0,1\}^\ell \to \{0,1\}^\ell\}.$$

Recall the following definition:

Definition 16.3 *A family of functions* $\mathcal{H} = \{h : A \to B\}$ *is called a* universal family of hash functions *if for every* x_1 *and* x_2 *in* A, $x_1 \neq x_2$, $\Pr_{h \in_R \mathcal{H}}[h(x_1) = y_1 \text{ and } h(x_2) = y_2] = \left(\frac{1}{|B|}\right)^2$.

Note that in our case the family $\mathcal{H}_\ell$ is degenerate, since the functions in $\mathcal{H}_\ell$ map ℓ-bit strings to ℓ-bit strings, and thus do not have any "shrinking" behavior whatsoever.

The construction requires that the functions h in $\mathcal{H}_\ell$ have a succinct representation, i.e., $|\langle h\rangle| = 2\ell$. An example of such a family is the set of all linear functions over $GF(2^\ell)$, namely

$$\mathcal{H}_\ell \triangleq \{h_{a,b} \mid a, b \in GF(2^\ell)\},$$

where

$$h_{a,b}(x) \triangleq ax + b,$$

with $GF(2^\ell)$ arithmetic.

Clearly, $|\langle h_{a,b}\rangle| = 2\ell$, and $h_{a,b}$ can be computed in space $O(\ell)$.

For the purpose of our construction, a hash function h is *well-behaved* with respect to two sets A and B, if it "extends well" to the two sets, i.e.,

$$\left|\Pr_{x \in_R \{0,1\}^\ell}[x \in A \text{ and } h(x) \in B] - \rho(A) \cdot \rho(B)\right| \leq 2^{-\ell/5},$$

where for any set $S \subseteq \{0,1\}^\ell$, we denote by $\rho(S)$ the probability that a random element x hits the set S, namely,

$$\rho(S) \triangleq \frac{|S|}{2^\ell} = \Pr_{x \in \{0,1\}^\ell}[x \in S].$$

The following proposition asserts that for any two sets A and B, almost all of the functions in $\mathcal{H}_\ell$ are well-behaved in the above sense with respect to A and B. Formally,

Proposition 16.4.1 *For every universal family* $\mathcal{H}_\ell$ *of hash functions, and for every two sets* $A, B \subseteq \{0,1\}^\ell$, *all but a* $2^{-\ell/5}$ *fraction of the* $h \in \mathcal{H}_\ell$ *satisfy*

$$\left|\Pr_{x \in \{0,1\}^\ell}[x \in A \textit{ and } h(x) \in B] - \rho(A) \cdot \rho(B)\right| \leq 2^{-\ell/5}.$$

16.5 Construction Overview

We now turn to consider an arbitrary (D, W)-graph G representing the possible executions of an $s(\cdot)$-space-bounded Turing machine M on an input x, where $|x| = n$. We construct a pseudorandom generator $H : \{0,1\}^k \to \{0,1\}^D$ that emulates a truly random guide on G.

Formally,

Definition 16.4 *A function $H : \{0,1\}^k \to \{0,1\}^D$ is a (D, W)-*pseudorandom generator *if for every (D, W)-graph G,*

$$\left|\Pr_{R\in\{0,1\}^D}[ACC(G,R)] - \Pr_{R'\in\{0,1\}^k}[ACC(G,H(R'))]\right| \le 1/10.$$

In Section 16.6, we prove the following theorem:

Theorem 16.5 *There exists a (D, W)-pseudorandom generator $H(\cdot)$ with $k(n) = O(\log D \cdot \log W)$. Further, $H(\cdot)$ is computable in space linear in its input.*

In particular,

Corollary 16.6 *There exists a (D, W)-pseudorandom generator $H(\cdot)$ with the following parameters:*

- $s(n) \;=\; \Theta(\log n)$.
- $D(n) \;\le\; \mathrm{poly}(n)$.
- $W(n) \;\le\; \mathrm{poly}(n)$.
- $k(n) \;=\; O(\log^2 n)$.

By trying all possible assignments for the seed of $H(\cdot)$, it follows that

Corollary 16.7 $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$.

In fact, as will be evident from the construction below, the pseudorandom generator H operates in space $O(\log n)$. However, the space complexity of the derandomization algorithm is dominated by the space used for writing down the seed for $H(\cdot)$.

Note that this result is not very striking, since the same result is known to hold for the single-sided error class $\mathcal{RL}$, as $\mathcal{RL} \subseteq \mathcal{NL} \subseteq \mathcal{DSPACE}(\log^2 n)$. However, as shown below, the construction can be extended to yield more striking results.

16.6 The Pseudorandom Generator

In this section, we formally describe a (D, W)-pseudorandom generator as defined in Theorem 16.5. Without loss of generality, we assume $D \le W$. The pseudorandom generator H is based on the universal family of hash functions $\mathcal{H}_\ell$, and extends strings of length $O(\ell^2)$ to strings of length $D \le \exp(\ell)$, for $\ell = \Theta(\log |W|)$. The pseudorandom strings cannot be distinguished from truly random strings by any (D, W)-graph.

The input to H is interpreted as the tuple

$$I \;=\; (r, \langle h_1\rangle, \langle h_2\rangle, \ldots, \langle h_{\ell'}\rangle),$$

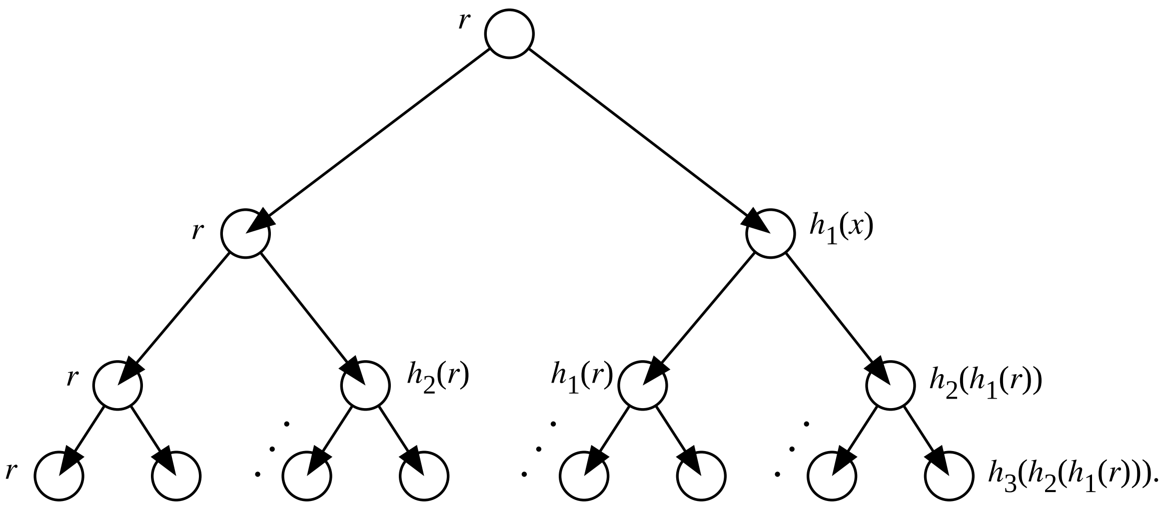


Figure 16.3: The Computation Tree T of $H(\cdot)$

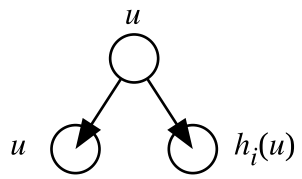


Figure 16.4: A Node in the Computation Tree of H

where $|r| = \ell$, $h_1, \ldots, h_{\ell'}$ are functions in $\mathcal{H}_\ell$, and $\ell' = \log(D/\ell)$. It can be easily observed that the length of the input I is indeed bounded by $O(\ell^2)$.

Now, given an input I, it may be most convenient to follow the computation of the pseudo-random generator H by considering a computation over a complete binary tree T of depth ℓ' (see Figure 16.3). The computation assigns a value to each node of the tree as follows. First, set the value of the root of the tree to be r. Next, given that a node located at depth $i-1$ has a value of u, set the value of its left child to u, and the value of its right child to $h_i(u)$ (see Figure 16.4). Finally, $H(\cdot)$ returns the concatenation of the binary values of the leaves of the tree, left to right.

More formally,

$$H(I) \triangleq \alpha_0 \cdot \alpha_1 \ldots \cdot \alpha_{2^{\ell'}-1},$$

where α_j is defined such that if the binary representation of j is $\tau_1 \cdots \tau_{\ell'}$, then

$$\alpha_j \triangleq h_{\ell'}^{\tau_{\ell'}} \circ \cdots \circ h_1^{\tau_1}(r),$$

where $h_i^1(z) = h_i(z)$ and $h_i^0(z) = z$ for every z.

Yet another way of describing $H(\cdot)$ is recursively as

$$H(r, \langle h_i \rangle, \ldots, \langle h_{\ell'} \rangle) \;=\; H(r, \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle) \cdot H(h_i(r), \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle),$$

where $H(z) \;=\; z$.

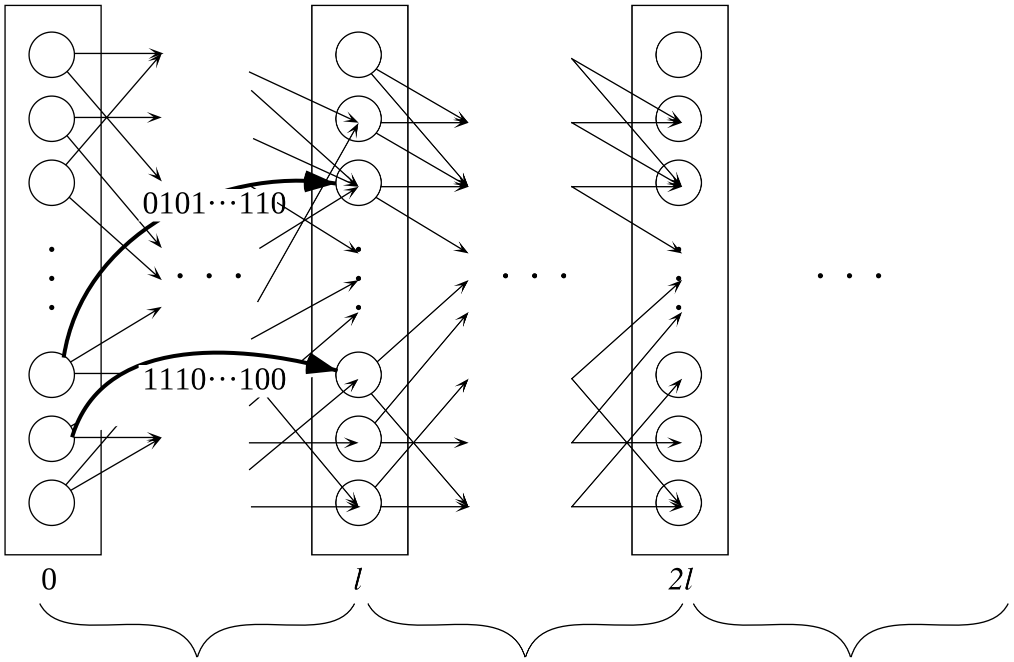


Figure 16.5: Contracting ℓ Layers

Note that the output of $H(\cdot)$ is composed of exactly $2^{\ell'} = D/\ell$ blocks, each of length ℓ, and hence the length of the output $H(I)$ is indeed D.

16.7 Analysis

It remains to see that $H(\cdot)$ is indeed a (D, W)-pseudorandom generator.

Theorem 16.8 (Theorem 16.5 rephrased) *$H(\cdot)$ is a (D, W)-pseudorandom generator.*

Proof: Consider a (D, W) graph $G_{M,x}$ that corresponds to the possible executions of a prospective distinguisher M on an input x. We show that $H(\cdot)$ is a (D, W)-pseudorandom generator by showing that a guided walk on $G_{M,x}$ using the guide $H(I)$ behaves "similarly" to a truly random walk, where $I = (z, \langle h_1 \rangle, \ldots, \langle h_{\ell'} \rangle)$ is drawn uniformly as a seed of H.

In an initial step, prune layers from $G_{M,x}$, so that only each ℓ-th layer remains, contracting edges as necessary (see Figure 16.5). Formally, construct a layered multigraph G_0 whose vertex set is the union of the vertex sets $V^0_{M,x}, V^{\ell}_{M,x}, V^{2\ell}_{M,x}, \ldots, V^D_{M,x}$, and whose edges correspond to directed paths of length ℓ in $G_{M,x}$. Label each edge in the multigraph G_0 by an ℓ-bit string which is the concatenation of the labels of the edges along the corresponding directed path in $G_{M,x}$. Thus, every multiedge e in G_0 corresponds to a subset of $\{0,1\}^{\ell}$. Clearly, a random walk on $G_{M,x}$ is equivalent to a random walk on G_0, when at each step an ℓ-bit string R is drawn uniformly, and the edge traversed is the edge whose label is R.

The analysis associates $H(\cdot)$ with a sequence of coarsenings. At each such coarsening, a new hash function h_i, uniformly drawn from $\mathcal{H}_{\ell}$, is used to decrease the number of truly random bits needed for the random walk by a factor of 2. After ℓ' such coarsenings, the only truly random bits required for the walk are the ℓ random bits of r, with the additional bits used to encode the hash functions $h_1, \ldots, h_{\ell'}$.

We begin by presenting the first coarsening step. In this step, the random guide $R = (R_1, R_2, \ldots, R_{D/\ell})$ is replaced by a "semi-random" guide $R' = (R_1, h_{\ell'}(R_1), R_3, h_{\ell'}(R_3), \ldots, R_{D/\ell-1}, h_{\ell'}(R_{D/\ell-1}))$, where $h_{\ell'}$ and the R_i's are drawn uniformly at random. Below we show that the semi-random guide behaves "almost like" the truly random guide, i.e., that for some ϵ,

$$|\Pr_R[ACC(G_0, R)] - \Pr_{R'}[ACC(G_0, R')]| < \epsilon.$$

We begin with a technical preprocessing step, removing from G_0 edges whose traversal probability is very small. Formally, let E_{light} denote the set of all "light" edges, i.e., edges (u, v) for which $\Pr_{R_i \in_R \{0,1\}^\ell}[u \to v] < 1/W^2$. Create a graph G'_0 whose vertex set is the same as G_0's, but containing only edges in $E \setminus E_{\text{light}}$. We first show that the removal of the light edges have a negligible effect. Formally,

Lemma 16.7.1 *For* $\epsilon_1 = 2/W$,

1. $|\Pr_R[ACC(G'_0, R)] - \Pr_R[ACC(G_0, R)]| < \epsilon_1$.
2. $|\Pr_{R'}[ACC(G'_0, R')] - \Pr_{R'}[ACC(G_0, R')]| < \epsilon_1$.

Proof: For the first part, the probability that a random walk R uses an edge in E_{light} is at most $\epsilon_1 = D \cdot (1/W^2) \leq 1/W$, and hence

$$|\Pr_R[ACC(G_0, R)] - \Pr_R[ACC(G'_0), R)]| < \epsilon_1.$$

For the second part, consider two consecutive steps guided by R' along the edges of G_0. The probability that the first step of R' traverses a light edge is bounded by $1/W^2$. By Proposition 16.4.1 with respect to the sets $\{0,1\}^\ell$ and the set of light edges available for the second step, for all but a fraction of $2^{-\ell/5}$ of the hash functions $h_{\ell'}$, the probability that the second step of R' traverses a light edge is bounded by $1/W^2 + 2^{-\ell/5} \leq 2/W^2$, assuming that the constant for ℓ is large enough. Hence, except for a fraction of $(D/2) \cdot 2^{-\ell/5} < \epsilon_1/2$ of the hash functions, the overall probability that R' traverses a light edge is bounded by $D \cdot (2/W^2) < \epsilon_1/2$. Thus, the overall probability of hitting a light edge is bounded by ϵ_1. ■

It thus remains to show that the semi-random guide R' behaves well with respect to the pruned graph G'_0. Formally, for some ϵ_2 specified below, we show that

Lemma 16.7.2 $|\Pr_R[ACC(G'_0, R)] - \Pr_{R'}[ACC(G'_0, R')]| < \epsilon_2$.

Proof:

Consider three consecutive layers of G'_0, say $V^0_{M,x}$, $V^\ell_{M,x}$, $V^{2\ell}_{M,x}$ (see Figure 16.6), and fix a triplet of vertices $u \in V^0_{M,x}$, $v \in V^\ell_{M,x}$, and $w \in V^{2\ell}_{M,x}$ for which the edges (u, v) and (v, w) are in the edge set of G'_0. Let $E_{u,v}$ denote the set of edges in G'_0 connecting u to v, and let $E_{v,w}$ denote the set of edges in G'_0 connecting v to w.

The probability that a random walk emanating from u visits v and reaches w can be written as

$$P_{u-v-w} \;=\; \Pr_{R_1, R_2 \in \{0,1\}^\ell}\left[R_1 \in E_{u,v} \textbf{ and } R_2 \in E_{v,w}\right].$$

Since the graph G'_0 was constructed such that $\Pr_{R_i \in_R \{0,1\}^\ell}[u \to v] \geq 1/W^2$ and $\Pr_{R_i \in_R \{0,1\}^\ell}[v \to w] \geq 1/W^2$, it follows that $P_{u-v-w} \geq 1/W^4$.

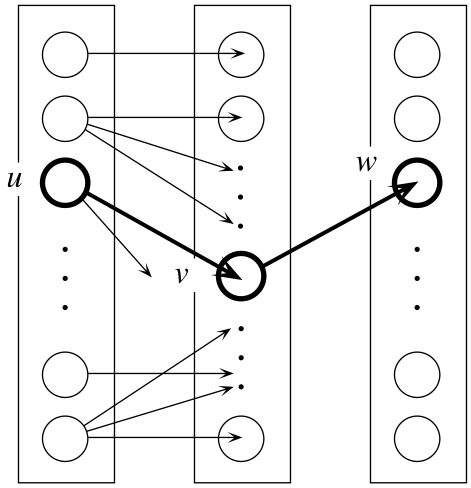


Figure 16.6: Three Consecutive Layers of G'_0

Now, the crux of the construction is that the above random walk can be replaced by a "semi-random" walk, namely, a walk whose first ℓ steps are determined by a random guide R_1, and whose last ℓ steps are determined by $h_{\ell'}(R_1)$, for some function $h_{\ell'}$ in $\mathcal{H}_\ell$. Given $h_{\ell'}$, the probability that a semi-random walk emanating from u reaches w via v is $P^{h_{\ell'}}_{u-v-w}$, where

$$P^h_{u-v-w} \triangleq \Pr_{R_1 \in \{0,1\}^\ell}[R_1 \in E_{u,v} \textbf{ and } h(R_1) \in E_{v,w}]$$

However, Proposition 16.4.1 applied with respect to the sets $E_{u,v}$ and $E_{v,w}$ asserts that except for a fraction of $2^{-\ell/5}$ of the hash functions $h_{\ell'}$,

$$\left|P^{h_{\ell'}}_{u-v-w} - P_{u-v-w}\right| \leq 2^{-\ell/5}. \tag{16.1}$$

Thus, except for a fraction of at most $\epsilon_3 = W^4 \cdot 2^{-\ell/5}$ of the hash functions, Equation (16.1) holds for *every* triplet of vertices u, v and w in every triplet of consecutive layers, i.e.,

$$\forall\ u, v, w \quad \left|P^{h_{\ell'}}_{u-v-w} - P_{u-v-w}\right| \leq 2^{-\ell/5}. \tag{16.2}$$

Next, fix a hash function $h_{\ell'}$ which satisfies Equation (16.2). The overall probability that a truly random walk starting from u reaches w can be written as

$$P_{u-w} \;=\; \sum_v P_{u-v-w},$$

whereas the probability that a semi-random walk starting from u reaches w is

$$P^{h_{\ell'}}_{u-w} \;=\; \sum_v P^{h_{\ell'}}_{u-v-w}.$$

Consequently, with a suitable selection of constants, and since $P_{u-w} \geq 1/W^4$,

$$|P^{h_{\ell'}}_{u-w} - P_{u-w}| \leq W \cdot 2^{-\ell/5} \leq W^5 \cdot 2^{-\ell/5} \cdot P_{u-w} \leq 2^{-\ell/10} \cdot P_{u-w} = \epsilon_4 \cdot P_{u-w},$$

for $\epsilon_4 = 2^{-\ell/10}$.

However, once the hash function $h_{\ell'}$ is fixed, every two-hop walk $u - v - w$ depends only on the corresponding ℓ-bit R_i, and hence for every "accepting path", i.e., a path $P \in \{0,1\}^D$ leading from the initial vertex v_0 to an accepting vertex,

$$\left|\Pr[R' = P] - \Pr[R = P]\right| \leq \epsilon_4 \cdot \Pr[R = P].$$

Since the probability of accepting is a sum over all accepting paths,

$$\left|\Pr_{R'}[ACC(G'_0, R')] - \Pr_R[ACC(G'_0, R)]\right| \leq \epsilon_4 \cdot \Pr_R[ACC(G'_0, R)] \leq \epsilon_4. \tag{16.3}$$

Finally, consider the two events $ACC(G'_0, R)$ and $ACC(G'_0, R')$, where the hash function $h_{\ell'}$ is drawn uniformly at random. The probability that R' hits a "bad" hash function $h_{\ell'}$ is bounded by ϵ_3. Otherwise, Equation (16.3) holds, and thus the lemma holds for $\epsilon_2 = \epsilon_3 + \epsilon_4$. ■

By the above two lemmas, it follows that the semi-random guide R' behaves "well' in the original graph G_0, i.e.,

Corollary 16.9 $|\Pr_R[ACC(G_0, R)] - \Pr_{R'}[ACC(G_0, R')]| < \epsilon$, *where* $\epsilon = 2 \cdot \epsilon_1 + \epsilon_2$. ■

Now, to perform another coarsening step, construct a new multigraph G_1 by contracting each pair of adjacent edge sets as follows:

- The vertex set of G_1 is the union of the vertices in the even layers of G.
- Create an edge for every two-hop path taken by the semi-random walk. Formally, for every two adjacent edges (u, v) and (v, w) labeled σ and $h_{\ell'}(\sigma)$ respectively, create an edge (u, w) in G_1, and label it σ.

Reapply Lemma 16.7.1 and Lemma 16.7.2 on G_1, this time using a new hash function $h_{\ell'-1}$, yielding a new multigraph G_2, and so on. It thus follows that at each step,

$$\left|\Pr[ACC(G_{i+1}, R^{(i+1)})] - \Pr[ACC(G_i, R^{(i)})]\right| < \epsilon.$$

After ℓ' iterations, the resulting graph $G_{\ell'}$ is a bipartite graph, on which a truly random guide r is applied. Since the above analysis corresponds to the behavior of the pseudorandom generator $H(\cdot)$,

$$\left|\Pr_{I \in_R \{0,1\}^{|I|}}[ACC(G, H(I))] - \Pr_{R \in_R \{0,1\}^D}[ACC(G, R)]\right| \quad \leq \quad \ell' \cdot \epsilon \leq 1/10,$$

where $I = (r, \langle h_1 \rangle, \langle h_2 \rangle, \ldots, \langle h_{\ell'} \rangle)$, which concludes the proof. ■

Remark: The analysis requires that the constant for $\ell = \Theta(\log n)$ be large enough. In the underlying computational model, this ensure that the machine M cannot retain even a description of a single function h_i. Otherwise, M could examine the first four blocks of the pseudorandom sequence, i.e., $z, h_{\ell'}(z), z', h_{\ell'}(z')$, and fully determine $h_{\ell'}$ by solving two linear equations with two variables. This would make it possible for M to distinguish between a truly random sequence R and the pseudorandom sequence $H(I)$.

16.8 Extensions and Related Results

16.8.1 $\mathcal{BPL} \subseteq \mathcal{SC}$

Whereas Corollary 16.7 asserts that $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^2 n)$, the running time of the straightforward derandomized algorithm is $\Omega(\exp(\log^2(n)))$, and in particular is not polynomial in n.

In this section we consider the complexity class $\mathcal{TS}(t(\cdot), s(\cdot))$, which denotes the set of all languages that can be recognized by a Turing machine whose running time is bounded by $t(\cdot)$ and whose space is (simultaneously) bounded by $s(\cdot)$. In particular, we consider $\mathcal{SC}$ (a.k.a., "Steve's Class"),

Definition 16.10 $\mathcal{SC} \stackrel{\triangle}{=} \mathcal{TS}(\mathrm{poly}(n), \mathrm{polylog}(n))$.

We state the following theorem:

Theorem 16.11 *$\mathcal{BPL} \subseteq \mathcal{SC}$.*

Proof Sketch: Consider a language $L \in \mathcal{BPL}$ and a corresponding Turing machine M for which $L = L(M)$. Now, instead of trying all $O(\log^2(n))$ possible values of the input $I = (r, \langle h_1 \rangle, \ldots, \langle h_{\ell'} \rangle)$ of H to determine whether an input x is in L, perform the following steps:

- "Magically" find a sequence of "good" functions $h_1, \ldots, h_{\ell'}$.
- Emulate M only over all possibilities of r. Since $|r| = \ell$, the emulation can be carried out in time exponential in ℓ, and hence in polynomial time.

To find a sequence of "good" functions, incrementally fix $h_{\ell'}$, $h_{\ell'-1}$, …, h_1. To fix a single h_j, assume all functions $h_{\ell'}, \ldots, h_{j+1}$ were already fixed and stored in memory. Consider all functions h in $\mathcal{H}_\ell$, and test each such h to determine whether Lemma 16.7.1 and Equation (16.2) hold. Since the existence of a "good" function h_j is asserted by Theorem 16.8, it remains to verify that finding such a function can be carried out in time exponential in ℓ (and hence in polynomial time) and in logarithmic space.

To see that this is the case, recall that the pseudorandom generator $H(\cdot)$ can be written recursively as

$$H(r, \langle h_i \rangle, \ldots, \langle h_{\ell'} \rangle) \;=\; H(r, \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle) \cdot H(h_i(r), \langle h_{i+1} \rangle, \ldots, \langle h_{\ell'} \rangle),$$

where $H(z) \;=\; z$. Consequently, once the functions $h_{\ell'}, \ldots, h_{j+1}$ are fixed, every single probability P_{u-v-w} can be computed directly simply by exhaustively considering all possible random guides R (of length 2ℓ). Similarly, given a candidate hash function $h_{\ell'}$, every single probability $P^{h_{\ell'}}_{u-v-w}$ can also be computed directly by exhaustively considering all semi-random guides R'. Hence, to determine whether Equation (16.2) holds, simply compare P_{u-v-w} and $P^{h_{\ell'}}_{u-v-w}$ for every triplet of vertices u, v and w in adjacent layers. Testing whether Lemma 16.7.1 holds can be carried out in a similar manner. Clearly, each such test can be carried out in time exponential in ℓ, and since the number of candidate functions $h_{\ell'}$ is also exponential in ℓ, the overall running time is exponential in ℓ. Further, testing a single function h can be carried out in space linear in ℓ, and hence the overall space complexity is dominated by the space needed to store the functions $h_{\ell'}, \cdots, h_1$, i.e., by $O(\ell^2)$. ■

Remark: The edge set of the i-th multigraph G_i depends upon the hash functions drawn in previous steps. Thus, although "almost all" functions h in $\mathcal{H}_\ell$ would satisfy Equation (16.2), one cannot consider fixing a function h_j before committing on the functions $h_{\ell'}, \ldots, h_{j+1}$.

16.8.2 Further Results

Below we state, without a proof, two related results:

Theorem 16.12 $\mathcal{BPL} \subseteq \mathcal{DSPACE}(\log^{1.5} n)$.

Theorem 16.13 (Informal) *Every random computation that can be carried out in polynomial time and in linear space can also carried out in polynomial time and linear space, but using only a linear amount of randomness.*

Bibliographic Notes

The main result presented in this lecture is due to Noam Nisan: The generator itself was presented and analyzed in [1], and the $\mathcal{SC}$ derandomization was later given in [2].

Theorems 16.12 and 16.13 are due to [4] and [3], respectively.

1. N. Nisan. Pseudorandom Generators for Space Bounded Computation. *Combinatorica*, Vol. 12 (4), pages 449–461, 1992.

2. N. Nisan. $\mathcal{RL} \subseteq \mathcal{SC}$. *Journal of Computational Complexity*, Vol. 4, pages 1-11, 1994.

3. N. Nisan and D. Zuckerman. Randomness is Linear in Space. To appear in *JCSS*. Preliminary version in *25th STOC*, pages 235–244, 1993.

4. M. Saks and S. Zhou. $RSPACE(S) \subseteq DSPACE(S^{3/2})$. In 36th *FOCS*, pages 344–353, 1995.

Lecture 17

Zero-Knowledge Proof Systems

Notes taken by Michael Elkin and Ekaterina Sedletsky

Summary: In this lecture we introduce the notion of zero-knowledge interactive proof system, and consider an example of such a system (Graph Isomorphism). We define perfect, statistical and computational zero-knowledge and present a method for constructing zero-knowledge proofs for NP languages, which makes essential use of bit commitment schemes. Presenting a zero-knowledge proof system for an NP-complete language, we obtain zero-knowledge proof systems for every language in NP. We consider a zero-knowledge proof system for one NP-complete language, specifically Graph 3-Colorability. We mention that zero-knowledge is preserved under sequential composition, but is not preserved under the parallel repetition.

Oded's Note: *For alterantive presentations of this topic we refer the reader to either Section 2.3 in [2] (approx. 4 pages), or the 30-page paper [3], or the first 4–5 sections in Chapter 4 of [1] (over 50 pages).*

17.1 Definitions and Discussions

Zero-knowledge (ZK) is quite central to cryptography, but it is also interesting in context of this course of the complexity theory. Loosely speaking, zero-knowledge proof systems have the remarkable property of being convincing and yielding nothing beyond the validity of the assertion.

We say that the proof is zero-knowledge if the verifier does not get from it anything that he can not compute by himself, when it assumes that the assertion is true.

Traditional proof carries with it something which is beyond the original purpose. The purpose of the proof is to convince somebody, but typically the details of proof give the verifier *more* than merely conviction in the validity of the assertion and it is not clear whether it is essential or not.

But there is an extreme case, in which the prover gives the verifier nothing beyond being convinced that the assertion is true. If the verifier assumed a-priori that the assertion is true, then actually the prover supplied no new information.

The basic paradigm of zero-knowledge interactive proof system is that whatever can be efficiently obtained by interacting with a prover, could also be computed without interaction, just by assuming that the assertion is true and conducting some efficient computation.

Recall that in the definition of interactive proof system we have considered properties of the verifier, whereas no requirements on the prover were imposed. In zero-knowledge definition we talk about some *feature* of the prescribed prover, which captures prover's robustness against attempts to gain knowledge by interacting with it. Verifier's properties are required to ensure that we have a proof system. A straightforward way of capturing the informal discussion follows.

Definition 17.1 *Let A and B be a pair of interactive Turing machines, and suppose that all possible interactions of A and B on each common input terminate in a finite number of steps. Then $\langle A, B\rangle (x)$ is the random variable representing the (local) output of B when interacting with machine A on common input x, when the random-input to each machine is uniformly and independently chosen.*

Definition 17.2 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* zero-knowledge *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ holds*

$$\{\langle P, V^*\rangle (x)\}_{x\in L} = \{M^*(x)\}_{x\in L}\,,$$

where the equality "=" between the ensembles of distributions can be interpreted in one of three ways that we will discuss later.

Machine M^* is called a simulator *for the interaction of V^* with P.*

We stress that we require that *for every* V^* interacting with P, not merely for V, there exists a simulator M^*. This simulator, although not having access to the interactive machine P, is able to simulate the interaction of V^* with P. This fact is taken as evidence to the claim that V^* did not gain any knowledge from P (since the same output could have been generated without any access to P).

V^* is an interactive machine, potentially something more sophisticated than V, which is the prescribed verifier. What V^* is interested in, is to extract from the prover more information than the prover is willing to tell. The prover wants to convince the verifier in the fact that $x \in L$, but the verifier is interested to get *more* information. Any efficient way, which the verifier may try to do it, is captured by such interacting process or strategy V^*.

We are talking here about probability ensembles, but unlike our definitions of the pseudo-randomness, now probability ensembles are defined using index which is not a natural number, but rather less trivial. We have to modify the formalism slightly and to enable indexing by any countable set. It is important to understand that for every x we have two distributions $\langle P, V^*\rangle (x)$ and $M^*(x)$.

¿From now on the distribution ensembles are indexed by strings $x \in L$, so that each distribution contains only strings of length polynomial in the length of the index.

The question is when these two probability ensembles are equal or close.

There are three natural notions:

Definition 17.3 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* perfect zero-knowledge(PZK) *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ the distributions $\{\langle P, V^*\rangle (x)\}_{x\in L}$ and $\{M^*(x)\}_{x\in L}$ are identical, i.e.*

$$\{\langle P, V^*\rangle (x)\}_{x\in L} \equiv \{M^*(x)\}_{x\in L}\,.$$

We emphasize that a distribution may have several different ways of being generated. For example, consider the uniform distribution over n bits. The normal way of generating such a distribution would be to toss n coins and to write down their outcome. Another way of doing it would be to toss $2*n$ coins, to ignore the coins in the even positions, and only to output the values of the coins in odd positions. These are two different ways of producing the same distribution.

Back to our definition, the two distributions $\{\langle P, V^* \rangle (x)\}_{x \in L}$ and $\{M^*(x)\}_{x \in L}$ have totally different ways of being produced. The first one is produced by interaction of the two machines and the second is produced by a traditional probabilistic polynomial time machine.

Consider a "honest" verifier. The two distributions are supposed to be exactly the same when x is in L. As we have seen in the example of uniform distribution, the two generated in different ways distributions may be identical, and our definition, indeed, requires them to be identical. The verifier has several other parameters, except of random coins and partial message history, and we do not fix them. The probability that the verifier accepts x, when $x \in L$, depends on these parameters. However, by completeness requirement of interactive proof system, this probability should be close to 1 for any possible values of the parameters.

When x is in L, M^* accepts with very high probability, but when x is not in L, it is not required that distribution $\{M^*(x)\}$ would be similar to $\{\langle P, V^* \rangle (x)\}$, and it may not happen. So machine M^*, which is called a simulator, simulates the interaction, assuming that x is in L. When x is not in L, it can do total rubbish.

To emphasize this point consider the following example.

Example 1.1: Consider a verifier V that satisfies the definition of interactive proof system, and so when $x \in L$, V will accept with very high (close to 1) probability. When $x \notin L$, V will accept with very low (close to 0) probability. Such $\langle P, V \rangle$ can be simulated by a trivial machine M which always accepts. When $x \in L$ the distributions $\langle P, V \rangle (x)$ and $M(x)$ are very close (the first is very close to 1 and the second is identically 1), whereas when $x \notin L$ the two distributions are totally different (the first one is close to 1 and the second one is identically 0). This behavior should not surprise, because if we would be able to simulate $\langle P, V \rangle$ by some non-interactive probabilistic polynomial time M it would follow that $IP \subseteq BPP$, whereas the definition we have introduced is much more general.

Example 1.2: Before we introduce an example of zero-knowledge interactive proof system, let us first describe an interactive proof system that "hides" some information from the verifier. Although, the system is not zero-knowledge, the verifier can not determine any particular bit of the "real" proof with probability bigger than 1/2.

Consider an NP relation R' that satisfies $(x, w') \in R$ if and only if $(x, \overline{w'}) \in R$. Such a relation can be created from any NP relation R by the following modification

$$R' \underline{\underline{\triangle}} \{(x, 0w) : (x, w) \in R\} \cup \{(x, 1w) : (x, \overline{w}) \in R\}.$$

Now we are suggesting the following interactive proof system by which the prover just sends to the verifier a witness w'.

$$\underline{Prover} \quad x \quad \underline{Verifier}$$

$$\xrightarrow[w']{\qquad\qquad\qquad}$$

Given the input x, the prover selects uniformly at random either w' or $\overline{w'}$, and sends one of them to the verifier. Both w' and $\overline{w'}$ are witnesses for x and so it is a valid interactive proof system. But if the verifier is interested to get some individual bit of the witness w', he has no way to do it using the data that he obtained from the prover. Indeed, each individual bit that the verifier receives from the prover is distributed uniformly and the verifier could produce such distribution by itself without interaction with the prover. However, we observe that the verifier gets some info about the witness, since it knows that he received either the witness or its complement.

Except of perfect zero-knowledge we define several more liberal notions of the same class. One of them requires that the distributions will be statistically close. By statistically close we mean that the variation distance between them is negligible as the function of the length of input x.

Definition 17.4 *The distribution ensembles $\{A_x\}_{x\in L}$ and $\{B_x\}_{x\in L}$ are* statistically close *or have* negligible variation distance *if for every polynomial $p(\cdot)$ there exists integer N such that for every $x \in L$ with $|x| \geq N$ holds*

$$\sum_{\alpha} |Prob\,[A_x = \alpha] - Prob\,[B_x = \alpha]| \leq \frac{1}{p(|x|)}.$$

Definition 17.5 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* statistical zero-knowledge *proof system (SZK) or* almost perfect *interactive proof system if for every probabilistic polynomial time verifier V^* there exists non-interactive probabilistic polynomial time machine M^* such that the ensembles $\{\langle P, V^*\rangle(x)\}_{x\in L}$ and $\{M^*(x)\}_{x\in L}$ are statistically close.*

Even more liberal notion of zero-knowledge is computational zero-knowledge interactive proof system.

Definition 17.6 *Two ensembles $\{A_x\}_{x\in L}$ and $\{B_x\}_{x\in L}$ are computationally indistinguishable if for every probabilistic polynomial time distinguisher D and for every polynomial $p(\cdot)$ there exists integer N such that for every $x \in L$ with $|x| \geq N$ holds*

$$|Prob\,[D(x, A_x) = 1] - Prob\,[D(x, B_x) = 1]| \leq \frac{1}{p(|x|)}.$$

The probabilistic polynomial time distinguisher D is given an index of the distribution in the ensemble. This is a general notion of indistinguishability of ensembles indexed by strings.

Definition 17.7 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is computational* zero-knowledge *proof system (CZK) if for every probabilistic polynomial time verifier V^* there exists non-interactive probabilistic polynomial time machine M^* such that the ensembles $\{\langle P, V^*\rangle(x)\}_{x\in L}$ and $\{M^*(x)\}_{x\in L}$ are computationally indistinguishable.*

We should be careful about the order of quantifying in this definition. First we have to determine the verifier V^* and the simulating machine M^*, and then we should check whether the distributions are indistinguishable by "trying all possible" probabilistic polynomial time distinguishers D.

Typically, when we say zero-knowledge, then we mean computational zero-knowledge. This is the most liberal notion, but from the point of view of cryptography applications it is good enough, because, basically, it says that the non-interactive machine M^* can simulate the interactive system $\langle P, V^* \rangle$ in such a way, that "no one" can distinguish between them. Essentially, the generated distributions are close in the same sense as distribution generated by pseudo-random generator is close to the uniform distribution. The idea is that if the machine M^* is able to generate by itself, without interaction, the distribution, that is very close in computational sense to the distribution generated by V^*, that interacts with P, then V^* gains nothing from interaction with P.

Observe that zero-knowledge interactive proof system definition imposes three requirements. Completeness and soundness requirements follows from interactive proof system definition, and zero-knowledge definition imposes the additional condition.

The completeness condition fixes both the prover and the verifier, and states that when the both parties follow the prescribed protocol (both parties are "honest") and $x \in L$, then the verifier accepts with high probability. Observe that if either prover or verifier are "dishonest" the condition may not hold. Indeed, we can not quantify this condition over all verifiers, since some verifier may always reject and then of course the probability of accepting will be zero. On the other hand, if the prover is "dishonest", or in other words, there is another prover instead of P, then it may send some rubbish instead of witness in the NP case or instead of following the prescribed protocol in the general case, and the verifier will accept only with low probability, hence the condition will not hold.

The soundness condition fixes only the verifier and protects his interests. It says, that the verifier doesn't need to trust the prover to be honest. Soundness condition quantifies over all possible provers and says, that no matter how the prover behaves he has very small probability of convincing the verifier to accept a wrong statement. Of course, it is correct only for the fixed verifier V and not for a general one, since we may think about a verifier that always accepts. For such a verifier the probability to accept a wrong statement is 1, hence the soundness condition does not hold for him.

And the zero-knowledge condition protects the prover. It tells the prover: "You are entering the protocol which enables you to convince the other party, even if the other party does not trust you and does not believe you. You can be convinced, that you do not really give the other party more than you intended to (i.e. that the statement is true, nothing beyond that)."

The zero-knowledge condition fixes the prover and quantifies over all verifiers. It says that for any verifier, sophisticated and dishonest as much as he may be, he can not gain from the prover (which obeys the protocol, again bad prover may send all the information to the verifier and then, of course, zero-knowledge condition will not hold) more that the verifier could gain without interacting with the prover.

We finish the section by proving that BPP is contained in PZK. Indeed, any probabilistic polynomial time algorithm may be viewed as an interactive proof system without prover. In such a system no verifier can gain knowledge from the prover, since there is no prover. Thus the system is zero-knowledge. We formalize these considerations in the following claim.

Proposition 17.1.1 *$BPP \subseteq PZK$.*

Proof: Consider an interactive proof system in which the verifier V is just the probabilistic polynomial time algorithm, that decides the language L. Such V exists, since $L \in BPP$. The prover P is just a deterministic machine, that never sends data to the verifier. $< P, V >$ is an interactive proof system, since V will accept with probability $\geq \frac{2}{3}$, when $x \in L$, and will accept with probability $\leq \frac{1}{3}$, when $x \notin L$, since $L \in BPP$, and V decides L. Hence, the completeness and soundness conditions hold. Clearly, it is perfect zero-knowledge since for every V^*, the distribution that $< P, V^* >$ generates is identical to the distribution generated by V^* itself. ∎

17.2 Graph Isomorphism is in Zero-Knowledge

Let $ISO \underline{\underline{\triangle}} \{(\langle G_1 \rangle, \langle G_2 \rangle) \mid G_1 \cong G_2\}$.

We assume that $ISO \notin BPP$, since otherwise $ISO \in ZK$, by Claim 1.1. So we are interested in showing zero-knowledge interactive proof systems for languages that are not in BPP, or at least are conjectured not to be in BPP.

Next, we introduce an interactive protocol proving that two graphs are isomorphic. The trivial interactive proof would be that the prover will send to the verifier the isomorphism, but this gives more information that the mere fact that the two graphs are isomorphic.

Instead of sending the isomorphism, which is not a good idea, we will use the following construction.

Construction 2.1 (Perfect Zero-Knowledge Proof for Graph Isomorphism)

- Common Input: A pair of two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Let ϕ be an isomorphism between the input graphs, namely ϕ is a 1-1 and onto mapping of the vertex set V_1 to the vertex set V_2 so that $(u, v) \in E_1$ if only if $(\phi(u), \phi(v)) \in E_2$. Suppose that $|V_1| = |V_2| = n$ and the vertices of the both graphs are the numbers from 1 to n.

- Prover's first Step (P1): The prover selects a random isomorphic copy of G_2, and sends it to the verifier. Namely, the prover selects at random, with uniform probability distribution, a permutation π from the set of permutations over the vertex set V_1, and constructs a graph with vertex set V_1 and edge set

  $$F \stackrel{def}{=} \{(\pi(u), \pi(v)) : (u, v) \in E_1\}$$

  The prover sends $H = (V_1, F)$ to the verifier.

- Motivating Remark: If the input graphs are isomorphic, as the prover claims, then the graph sent in step P1 is isomorphic to both input graphs. However, if the input graphs are not isomorphic then no graph can be isomorphic to both of them.

- Verifier's first Step (V1): Upon receiving a graph, $G' = (V', E')$, from the prover, the verifier asks the prover to show an isomorphism between G' and one of the input graph, chosen at random by the verifier. Namely, the verifier uniformly selects $\sigma \in \{1, 2\}$, and sends it to the prover (who is supposed to answer with an isomorphism between G_σ and G').

- Prover's second Step (P2): If the message, σ, received from the verifier equals 1 then the prover sends π to the verifier. Otherwise (i.e., $\sigma \neq 1$), the prover sends $\pi \circ \phi$ (i.e., the composition of π on ϕ) to the verifier. (Remark: the prover treats any $\sigma \neq 1$ as $\sigma = 2$.)

- Verifier's second Step (V2): If the message, denoted ψ, received from the prover is an isomorphism between G_σ and G' then the verifier outputs 1, otherwise it outputs 0.

For the schematic representation of the protocol, see the Diagram 2.1.

The verifier program presented above is easily implemented in probabilistic polynomial time. In case the prover is given an isomorphism between the input graphs as auxiliary input, also the prover's program can be implemented in probabilistic polynomial time. We now show that the above pair of interactive machines constitutes an almost perfect zero-knowledge interactive proof system for the language ISO.

Diagram 2.1

$$\begin{array}{ccc}
\underline{Prover} & & \underline{Verifier} \\
\pi \in_R Sym([n]) & & \\
H \longleftarrow \pi G_1 & & \\
 & \xrightarrow[H]{\qquad} & \\
 & & \sigma \in_R \{1,2\} \\
 & \xleftarrow[\sigma]{\qquad} & \\
\text{if } \sigma = 1 \text{ then send } \psi = \pi, & & \\
\text{otherwise } \psi = \pi \circ \phi^{-1} & & \\
 & \xrightarrow[\psi]{\qquad} & \\
 & & \text{Accept if and only if} \\
 & & H = \psi(G_\sigma)
\end{array}$$

Theorem 17.8 *The construction above is an almost perfect zero-knowledge interactive proof system.*

Proof:

1. Completeness.

If the two graphs are isomorphic, we claim that the verifier will always accept. Indeed, if $\sigma = 1$, then the verifier comes to the end of the protocol with $H = \pi(G_1)$ and permutation $\psi = \pi$. The verifier checks whether

$$H = \psi(G_\sigma). \tag{17.1}$$

We observe that $\pi(G_1) = H$, and $\psi(G_\sigma) = \psi(G_1) = \pi(G_1)$, implying (17.1), hence the verifier accepts.

If $\sigma = 2$, then the verifier comes to the end of the protocol with $H = \pi(G_1)$ and permutation $\psi = \pi \circ \phi$. Again he verifies whether (l24:eq) holds. We observe that $\pi(G_1) = H$ and $\psi(G_\sigma) = \pi\phi(G_2) = \pi(G_1)$, again implying (l24:eq), hence the verifier accepts in both cases with probability 1.

The intuition is very simple. If the two graphs are isomorphic and the prover created the isomorphic copy to one of them, he should have no problem in showing isomorphism to each of the graphs.

2. Soundness.

Let $G_1 \ncong G_2$. Consider any prover P^*. If it sends to V a graph H, which is not isomorphic neither to G_1 nor to G_2, then this prover will have no way later to present an isomorphism from G_σ (no matter whether $\sigma = 1$ or 2) to H, *since there is no such isomorphism.* So, in this case, the probability of P^* to convince the verifier that $(\langle G_1 \rangle, \langle G_2 \rangle) \in ISO$, is zero. Suppose P^* sends an H that is isomorphic either to G_1 or to G_2. Without loss of generality, assume $H \cong G_1$. Then if the verifier randomly selected $\sigma = 1$, then P^* will be able to show the isomorphism between H and $G_\sigma = G_1$. Otherwise, if the verifier randomly selected $\sigma = 2$, then there is no isomorphism between $H \cong G_1$ and $G_\sigma = G_2$ (as otherwise G_1 and G_2 would be isomorphic), hence P^* will not be able to find one, despite his unlimited computational power. Hence, in this case, P^* will have probability of exactly $\frac{1}{2}$ to convince the verifier that $(\langle G_1 \rangle, \langle G_2 \rangle) \in ISO$ and we have shown that it is his optimum strategy. So

$$Prob[< P^*, V > (\langle G_1 \rangle, \langle G_2 \rangle) = accept \mid G_1 \ncong G_2] \leq \frac{1}{2}$$

for every prover P^*. By executing this protocol twice sequentially, we obtain

$$Prob[< P^*, V > (\langle G_1 \rangle, \langle G_2 \rangle) = accept \mid G_1 \ncong G_2] \leq \frac{1}{4},$$

hence, satisfying the soundness condition.

3. Zero-knowledge.

There is no other way to prove that the protocol is zero-knowledge, except of building a simulator and proving that it really generates the same distribution.

Simulator M^*. By definition of zero-knowledge we have to show that the distributions are the same when we are given an input from the language. On input $x \stackrel{def}{=} (\langle G_1 \rangle, \langle G_2 \rangle)$, simulator M^* proceeds as follows:

1. Setting the random-tape of V^*: *Let $q(\cdot)$ denote a polynomial bounding the running time of V^*. The simulator M^* starts by uniformly selecting a string $r \in \{0,1\}^{q(|x|)}$, to be used as the contents of the random-tape of V^*.*

2. Simulating the prover's first step (P1): *The simulator M^* selects at random,with uniform probability distribution, a "bit" $\tau = \{1,2\}$ and a permutation ψ from the set of permutations over the vertex set V_τ. It then constructs an isomorphic copy G'' of the graph G_τ, i.e. $G'' = \psi(G_\tau)$.*

3. Simulating the verifier's first step (V1): *The simulator M^* initiates the execution of V^* by placing x on V^*'s common-input-tape, placing r (selected in step (1) above) on V^*'s random-tape, and placing G'' (constructed in step (2) above) on V^*'s incoming message-tape. After executing a polynomial number of steps of V^*, the simulator can read the outgoing message of V^*, denoted σ. Let us assume, without loss of generality, that the message sent by V^* is either 1 or 2. Indeed, if V^* sends $\sigma \notin \{1,2\}$, then the prover has nothing to do with it and we may augment the prover P with a rule to ignore $\sigma \notin \{1,2\}$ and just to wait for a "valid" σ. This would be very easy to simulate.*

4. Simulating the prover's second step (P2): *If $\sigma = \tau$ then the simulator halts with output (x, r, G'', ψ).*

5. Failure of the simulation: *Otherwise (i.e. $\sigma \neq \tau$), the simulator halts with output $\perp$.*

As could be seen from (4) we output the full view of the verifier. We stress that the simulator "has no way to know" whether V^* will ask to see an isomorphism to G_1 or to G_2.

This description of the simulator machine may confuse. Indeed, the definition of zero-knowledge considers the distributions of two random variables $\langle P, V^* \rangle (x)$ and $M^*(x)$, the outputs of $\langle P, V^* \rangle$ and M^* respectively. On the other hand, here M^* returns its whole view, that consists of all the data it possesses, specifically, x, r, G'' and ψ. This inconsistency can be treated by considering $\langle P, V^* \rangle (x)$ and $M^*(x)$ in the zero-knowledge definition as the views of V^* and M^* respectively, and by showing that this approach is equivalent to our definition.

Definition 17.9 *Let (P, V) be an interactive proof system for some language L. We say that (P, V), actually P, is* perfect zero-knowledge(PZK) by view *if for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^* so that for every $x \in L$ holds*

$$\left\{ view_{(P,V^*)}(x) \right\}_{x \in L} \equiv \left\{ M^*(x) \right\}_{x \in L},$$

where $view_{(P,V^*)}(x)$ is the final view of V^* after running $\langle P, V^* \rangle$ on input x and $M^*(x)$ is, as usual, the output of M^* after running on input x.

Claim 17.2.1 *An interactive proof system is perfect zero-knowledge if and only if it is perfect zero-knowledge by view.*

Proof: One direction is straightforward. Suppose there is a probabilistic polynomial time machine M^*, which for every input $x \in L$ outputs $M^*(x)$, that is distributed identically to the view of V^* at the end of execution of $\langle P, V^* \rangle$ on x. We observe that the last step of V^*, i.e. printing the output, is done without interaction with the prover. Note also that M^* may, instead of printing the output, write it down on its work-tape. Then M^* has on its work-tape the final view of V^*. Hence, it is capable to perform the last step of V^* and output the result and so the modified $M^*(x)$ is identical to $\langle P, V^* \rangle$, completing the proof of this direction.

In the opposite direction, we suppose that for every V^* there is a non-interactive probabilistic polynomial time machine M^*, which prints the same output, when it runs on x (for every $x \in L$), as V^* when $\langle P, V^* \rangle$ machine runs on x.

Consider some particular V^*. We need to show that there is a machine that for every $x \in L$ prints at the end of its execution on x the output identical to the view V^* at the end of execution of $\langle P, V^* \rangle$ on x. To see it, consider a verifier V^{**}, that behaves exactly like V^*, but outputs its whole view (i.e., it emulates V^* except that at the end it outputs the view of V^*). There is a machine M^{**}, such that its output $M^{**}(x)$ is distributed identically to the output of V^{**}, i.e. to the view of V^*. Thus M^{**} is the required machine. It completes the proof of the second direction, establishing the equivalency of the definitions. ■

Recall that the Definitions 17.3 and 17.9 both require that for every probabilistic polynomial time interactive machine V^* there exists an (*ordinary*) probabilistic polynomial time machine M^*

with certain properties. Observe that in the proof of the non-trivial (i.e., second) direction of the above claim we use the fact that for every V^{**} (constructed out of V^*) there is a corresponding simulator. We stress that this was not used in the first direction in which we did not modify V^* (but rather M^*).

Statistical zero-knowledge by view and computational zero-knowledge by view are defined analogously. Similar claims about equivalency between statistical zero-knowledge and statistical zero-knowledge by view, and between computational zero-knowledge and computational zero-knowledge by view can be proved using the same argument as in the proof of Claim 17.2.1.

We claim that, when two graphs are isomorphic, then H gives no information on τ, because we can draw a correspondence between the possible mappings that can generate H from G_1 and the possible mappings that can generate H from G_2 by the isomorphism between the two graphs. It follows that

Claim 17.2.2 *Let $x = (\langle G_1\rangle, \langle G_2\rangle) \in ISO$. Then for every string r, graph H, and permutation ψ, it holds that*

$$Prob\left[view_{(P,V^*)}(x) = (x,r,H,\psi)\right] = Prob\left[M^*(x) = (x,r,H,\psi) \mid (M^*(x) \neq \perp)\right].$$

Proof: Let $m^*(x)$ describe $M^*(x)$ conditioned on its not being $\perp$. We first observe that both $m^*(x)$ and $\mathrm{view}_{(P,V^*)}(x)$ are distributed over quadruples of the form $(x,r,\cdot,\cdot)$, with uniformly distributed $r \in \{0,1\}^{q(|x|)}$, for some polynomial $q(\cdot)$. Let $v(x,r)$ be a random variable describing the last two elements of $\mathrm{view}_{(P,V^*)}(x)$ conditioned on the second element equals r. Similarly, let $\mu(x,r)$ describe the last two elements of $m^*(x)$ (conditioned on the second element equals r). Clearly, it suffices to show that $v(x,r)$ and $\mu(x,r)$ are identically distributed, for every x and r. Observe that once r is fixed the message σ sent by V^* on common input x, random-tape r, and incoming message H, is uniquely defined. Let us denote this message by $v^*(x,r,H)$. We need to show that both $v(x,r)$ and $\mu(x,r)$ are uniformly distributed over the set

$$C_{x,r} \stackrel{def}{=} \{(H,\psi) : H = \psi(G_{v^*(x,r,H)})\}.$$

The proof is slightly non-trivial because it relates (at least implicitly) to the automorphism group of the graph G_2 (i.e., the set of permutations π for which $\pi(G_2)$ is identical, not just isomorphic, to G_2). For simplicity, consider the special case in which the automorphism group of G_2 consists of merely the identity permutation (i.e., $G_2{=}\pi(G_2)$ if and only if π is the identity permutation). In this case, $(H,\psi) \in C_{x,r}$ if and only if H is isomorphic to both G_1 and G_2 and ψ is the isomorphism between H and $G_{v^*(x,r,H)}$. Hence, $C_{x,r}$ contains exactly $|V|!$ pairs, each containing a different graph H as the first element, proving the claim in the special case.

For the proof of the general case we refer the reader to [3] (or to [1]). ■

Recall that to prove perfect zero-knowledge we need to show that $\mathrm{view}_{(P,V^*)}(x)$ and $M^*(x)$ are identically distributed. Here, it is not the case. Although, $\mathrm{view}_{(P,V^*)}(x)$ and $M^*(x)|(M^*(x) \neq\perp)$ are identically distributed, when $M^*(x) =\perp$ the distributions are totally different. A common way to overcome this difficulty is to change the definition of perfect zero-knowledge, at least a bit. Suppose we allow the simulator to output a special symbol which we call "failure", but we require that this special symbol is outputted with probability at most 1/2. In this case the construction would satisfy the definition and we could conclude that it is a perfect zero-knowledge proof (under the changed definition).

But recall that Theorem 17.8 states that the construction is almost perfect zero-knowledge. This is indeed true, without any change of the definitions, if the simulator reruns steps (2)-(4) of the construction $|x|$ times. If, at least once, at step (4) σ is equal to τ, then output (x, r, G'', ψ). If at all $|x|$ trials $\sigma \neq \tau$, then output rubbish. In such a case the simulation will not be perfect, but will be statistically close, because the statistical difference will be $2^{-|x|}$.

It remains to show that the running time of the simulator is polynomial in $|x|$. In the case when we run it $|x|$ times, it is obvious, concluding our proof that the interactive proof system is almost perfect zero-knowledge. In the case when we change the definition of perfect zero-knowledge in the described above way, we are done by one iteration, hence the running time is against polynomial

Another possibility is to allow the simulator to run expected polynomial time, rather than strict polynomial time, in such a case the interpretation would be to rerun steps (2)-(4), until the output is not $\perp$. Every time we try, we have a success probability of exactly 1/2. Hence, the expected number of trials is 2.

This concludes the proof of the Theorem 17.8. ■

These definitions, one allowing the failure probability and another allowing an expected polynomial time, are not known to be equivalent. Certainly, if we have a simulator which with probability at most 1/2 outputs the failure, then it can be always converted to one which runs in expected polynomial time. But the opposite direction is not known and is not clear.

17.3 Zero-Knowledge Proofs for NP

17.3.1 Zero-Knowledge NP-proof systems

We want to show why it was essential to introduce the interactive proofs in order to discuss zero-knowledge (in a non-trivial way). One can also define zero-knowledge for NP-proofs, but by the following claim such proofs exist only for BPP (and are thus "useless").

Proposition 17.3.1 *Let L be a language that admits zero-knowledge NP-proof system. Then $L \in BPP$.*

Proof: For the purpose of the proof we use only the fact that an "honest" verifier V, which outputs its whole view, can be simulated by a probabilistic polynomial time machine M.

(One may think that in view of Claim 17.2.1 there is no point to specify that V outputs its whole view. It is not correct. In the proof of the claim we used the fact that if an interactive proof system is zero-knowledge, then *for any* verifier V^* there is a simulator M^* with certain properties. Once we fix a verifier, such kind of argument no longer works. It is the reason that we specify the output of V.)

Let L be the language that the NP proof system decides and R_L its NP relation. Let $x \in L$. The view of the verifier when interacting with a prover on input x will be the input itself and the message, call it w, that he received. Since we consider an honest prover, the following holds

$$view_{(V,P)}(x) = (x, w) \in R_L.$$

Simulator M simulates this view on input x. We show that $(x, M(x)) \in R_L$ with high probability, specifically

$$Prob[(x, M(x)) \in R_L] \geq \frac{2}{3}. \tag{17.2}$$

Also, we will show that for $x \notin L$

$$Prob[(x, M(x)) \in R_L] = 0 < \frac{1}{3}, \tag{17.3}$$

hence, M is the probabilistic polynomial time algorithm for L, implying $L \in BPP$.

First, discuss the case of $x \in L$ and suppose for contradiction that

$$Prob[(x, M(x)) \in R_L] < \frac{2}{3}. \tag{17.4}$$

Then we claim that there is a deterministic polynomial time distinguisher D, that distinguishes between the two distributions $\langle P, V \rangle (x)$ and $M(x)$ with non-negligible probability. Consider $D(\alpha)$ which is defined as 1 if $\alpha \in R_L$, and 0 otherwise, where α is a pair of the form (x, w). Obviously, for $x \in L$,

$$Prob[D(x, \langle P, V \rangle (x)) = 1] = 1$$

since P is a "honest" prover, i.e. a prover that supplies witness for $x \in L$.

Also, from (4) follows that

$$Prob[D(x, M(x)) = 1] < \frac{2}{3}.$$

So there is a non-negligible gap between the two cases. It contradicts the assumption, that the distribution of $M(x)$ is polynomially indistinguishable from the distribution of $\langle P, V \rangle (x)$.

On the other hand, when $x \notin L$, then by the definition of NP, there is no witness y, such that $(x, y) \in R_L$. Particularly, $(x, M(x)) \notin R_L$. In other words,

$$Prob[(x, M(x)) \in R_L] = 0.$$

Concluding the proof of (2) and (3), hence $L \in BPP$. ■

17.3.2 NP $\subseteq$ ZK (overview)

Now, we are going to show that NP has a zero-knowledge interactive proof ($NP \subseteq ZK$).

We will do it assuming, that we have some magic boxes, called commitment schemes, and later we will describe their implementation.

Commitment schemes are used to enable a party to commit itself to a value while keeping it secret. In a latter stage the commitment is "opened" and it is guaranteed that the "opening" can yield only a single value determined in the committing phase. Commitment schemes are the digital analogue of non-transparent sealed envelopes. Nobody can look inside the envelopes and know the value. By putting a note in such an envelope a party commits itself to the contents of the note while keeping it secret.

We present a zero-knowledge proof system for one NP-complete language, specifically *Graph 3-Coloring.*

The language *Graph 3-Coloring*, denoted $G3C$, consists of all simple graphs (i.e., no parallel edges or self-loops) that can be *vertex-colored* using 3 colors so that no two adjacent vertices are given the same color. Formally, a graph $G = (V, E)$, is $3 - colorable$, if there exists a mapping $\phi : V \longmapsto \{1, 2, 3\}$, so that $\phi(u) \neq \phi(v)$ for every $(u, v) \in E$.

In general, if we want to build a zero-knowledge interactive proof system for some other NP language, we may just use standard reduction and run our protocol on the reduced instance of the graph colorability. Thus, if we can show a zero-knowledge proof for one $NP-$complete language, then we can show for all. Basically it is correct, although inaccurate. For more details see [3].

One non-zero-knowledge proof is to send the coloring to the verifier, which would check it, and this would be a valid interactive proof system, but, of course, not a zero-knowledge one.

The instructions that we give to the prover can actually be implemented in polynomial time, if we give the prover some auxiliary input, specifically the 3-coloring of the graph, which is the NP-witness that the graph is in $G3C$. Let ψ be a 3-coloring of G. The prover selects at random some permutation. But this time it is not a permutation of the vertices, but rather permutation of the colors

$$\pi \in_R Sym([3]).$$

Then it sets $\phi(v) \underline{\underline{\triangle}} \pi(\psi(v))$, for each $v \in V$ and puts each of these permuted colors in a separate locked box (of the type that was discussed before), and the boxes have marks, so that both the prover and verifier know the number of each box.

$$\boxed{\phi(1)}^1, \boxed{\phi(2)}^2, ..., \boxed{\phi(i)}^i, ..., \boxed{\phi(n)}^n$$

The ϕ-color of vertex i is sent in the box number i. The verifier can not see the contents of the boxes, but the prover claims, that he has sent a legal coloring of the graph, which is a permutation of the original one.

The verifier selects an edge $e = (u, v)$ at random, and sends it to the prover.

$$\underline{P} \quad \overleftarrow{\quad e = (u, v) \qquad e \in_R Edge(G) \quad} \quad \underline{V}$$

Basically, it asks to inspect the colors of vertices u and v. It expects to see two different colors, and if they are not, then the verifier knows that something is wrong.

The prover sends back the key to box number u and the key to box number v.

$$\underline{P} \quad \overrightarrow{\qquad key_u \quad and \quad key_v \qquad} \quad \underline{V}$$

The verifier uses the keys to open these two boxes (other boxes remain locked), and looks inside. If he finds two different colors from the set $\{1, 2, 3\}$, then it accepts. Otherwise, he rejects. This completes the description of interactive proof system for $G3C$, that we call $G3C$ protocol.

Even more drastically then before, this will be a very weak interactive proof. In order to make any sense we have to repeat it a lot of times. Every time we repeat it, a prover selects a new permutation π, so crucially the color that the verifier sees in one iteration have nothing to do with the colors, that he sees in other iterations.

Of course, if the prover would always use the same colors, then the verifier would know the original coloring by just asking sufficient number of times, a different edge each time. So it is crucial, that in every iteration the coloring is totally random. We observe that the randomness of the coloring may follow also from a random choice of the original coloring ϕ and not only from the randomness of the permutation π, which has very small sample space (only 3!=6). A computationally unbounded prover will have no problem with randomly selecting a 3-coloring for a graph. On the other hand, a polynomial time bounded prover has to be feeded with all the 3-colorings as auxiliary inputs, in order to be able to select randomly at uniform an original coloring ϕ.

Observe that zero-knowledge property (when augmenting the definition a bit) is preserved under sequential composition. The error probabilities also decrease if we apply the protocol in parallel. But in general, it is not known that zero-knowledge is preserved under parallel composition. In particular, the protocol which is derived by running $G3C$ protocol in parallel many times is in some sense not zero-knowledge, or at least probably is not zero-knowledge.

Proposition 17.3.2 *G3C protocol is zero-knowledge interactive proof system.*

Proof:

1. Completeness.

If the graph is 3-Colorable, and both the prover and the verifier follow the protocol, then the verifier will always accept. Indeed, since ϕ is a legal coloring, for every edge $e = (u, v)$ holds $\phi(u) \neq \phi(v)$. Hence, for $G \in G3C$

$$Prob[\langle P, V\rangle\,(\langle G\rangle) = accept] = 1,$$

and we are done.

2. Soundness.

Let ϕ be a color-assignment of graph G that the prover uses in his trial to convince the verifier that the graph is 3-colorable. If $G \notin G3C$, then by definition of 3-Colorability there exists an edge

$$e_0 = (u, v) \in Edges(G),$$

such that either $\phi(u) = \phi(v)$ or $\phi(u) \notin \{1, 2, 3\}$. Without loss of generality, suppose $\phi(x) \in \{1, 2, 3\}$ for all x's. If the verifier asked to open e_0 then by commitment property (specifically, by unambiguity requirement of a commitment scheme, to be discuss it in the next subsection) he will reveal that $\phi(u) = \phi(v)$ and thus he will reject. I.e.

$$\text{Prob}\left[\begin{array}{c}\text{a randomly selected edge}\\ e = (w, z) \text{ satisfies } \phi(w) \neq \phi(z)\end{array}\right] \leq \frac{|Edges(G)\backslash\{e_0\}|}{|Edges(G)|} = 1 - \frac{1}{|Edges(G)|}.$$

Hence, for any prover P^*, for $G \notin G3C$

$$Prob[\langle P^*, V\rangle\,(G) = accept] \leq 1 - \frac{1}{|Edges(G)|}.$$

By repeating the protocol sequentially, sufficiently many times, specifically $\lceil \log_{(\frac{|Edges(G)-1|}{|Edges(G)|})} \frac{1}{3} \rceil$, we reduce this probability to

$$Prob[\langle P^*, V\rangle\,(G) = accept|\; G \notin G3C] < \frac{1}{3}$$

for any prover P^*.

3. Zero-Knowledge.

In order to show that $G3C$ protocol is a zero-knowledge proof system, we devise for every verifier V^* the following simulating non-interactive machine M^*, that for every $G \in G3C$ generates a distribution $m^*(\langle G \rangle) = M^*(\langle G \rangle) | (M^*(G) \neq \perp)$ identical to the distribution $\langle P, V^* \rangle (\langle G \rangle)$.

Description of M^*

(1) Fix the random coins r of V^*.

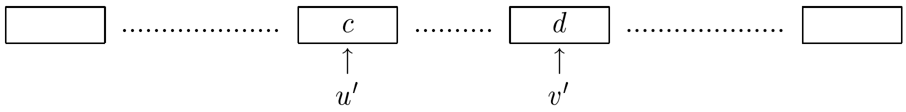
(2) Select at random an edge $e' = (u', v')$,

$$e' = (u', v') \in_R Edges(G).$$

(3) V^* sends to the M^* (in fact, it is inter-routine communication, since V^* is incorporated in M^*) boxes that are filled in the following way. All the boxes are filled with garbage, except the two boxes of u' and v', which are filled with two different, randomly selected numbers between 1 and 3, namely

$$c \neq d \in_R \{1, 2, 3\}.$$

Put c in u' box, d in v' box.



(4) If V^* chooses e', M^* sends V^* the keys and the simulation is completed. If the verifier sends a different edge, we rerun everything from step (1).

Totally, rerun everything at most $|Edges(G)|$ times. If at least once V^* selected e' in that iteration, print the output of V^*. Otherwise, output $\perp$.

V^* has no way to know what boxes are filled with garbage and what boxes are filled with something meaningful, since the boxes are "non-transparent" (see below about secrecy requirement of a commitment scheme). Thus the probability, that in each iteration V^* will select e' is $\frac{1}{|Edges(G)|}$. By doing $|Edges(G)|$ iterations we reach a constant probability (of about e^{-1}) of generating distribution exactly identical to $\langle P, V^* \rangle (\langle G \rangle)$.

Consider the distribution of

$$m^*(\langle G \rangle) = M^*(\langle G \rangle) \mid (M^*(\langle G \rangle) \neq \perp).$$

For every $G \in G3C$, if M^* did not output $\perp$, then one of the iterations was successful. In this case V^* has selected e', which is a randomly selected edge from the graph G. M^* prints the output of V^* on e'. Thus, the distribution of $m^*(\langle G \rangle)$ is identical to the distribution of the output of V^* when it is given a randomly selected edge from the graph G. Now consider the distribution of $\langle P, V^* \rangle (\langle G \rangle)$. The prescribed prover P is fixed and by the construction it supplies to V^* a randomly selected edge from the graph G. Thus the distribution of $\langle P, V^* \rangle (\langle G \rangle)$ is also identical to the distribution of the output of V^* when it is given a randomly selected edge from the graph. Hence,

$$m^*(\langle G \rangle) = \langle P, V^* \rangle (\langle G \rangle).$$

So if the boxes are perfectly sealed, then $G3C \in PZK$. ■

17.3.3 Digital implementation

In reality we use *commitment scheme* instead of these "sealed envelopes", that we discussed previously. In general, a *commitment scheme* is an efficient *two-phase* two-party protocol through which one party, called the *sender* (S), can commit itself to a *value* so the following two conflicting requirements are satisfied.

1. *Secrecy*: At the end of the first phase, the other party, called the *receiver* (R), does not gain any knowledge of the sender's value. This requirement has to be satisfied for any polynomial-time receiver.

2. *Unambiguity*: Given the transcript of the interaction in the first phase, there exists at most one value which the receiver may later (i.e., in the second phase) accept as legal "opening" of the commitment. This requirement has to be satisfied even if the sender tries to cheat (no matter what strategy it employs).

In addition, we require that the protocol is *viable* in the sense that if both parties follow it then, at the end of the second phase, the receiver gets the value committed to by the sender.

Denote by $S(s,\sigma)$ the message that the sender sends to the receiver when he wants to commit himself to a bit σ and his random coins are s. The secrecy requirement states that, for random s, the distributions of the random variables $S(s,0)$ and $S(s,1)$ are indistinguishable by polynomial-size circuits.

Let view of S (R), denoted as $View(S)$ ($View(R)$), be the collection of all the information known to the sender (receiver). Denote by r the random coins of R and let $\overline{m}$ be the sequence of messages that R received from S. Then $View(R) = (r, \overline{m})$. In case of single-round commit, $\overline{m} = S(s,\sigma)$. When the sender S wants to commit itself to a bit σ and his random coins sequence is s, then $View(S) = (s,\sigma)$. The unambiguity requirement states that for all but a negligible fraction of r's, there is no such $\overline{m}$ for which there exist two sequences of random coin tosses of S, s and s' such that

$$View(S) = (s,0) \ \ and \ \ View(R) = (r,\overline{m})$$

and

$$View(S) = (s',1) \ \ and \ \ View(R) = (r,\overline{m}).$$

The intuition of this formalism is that if such s and s' would exist, then the receiver's view would not be monosemantic. Instead, it would enable to sophisticated sender to claim that he committed either to zero or to one, and the receiver would not be able to prove that the sender is cheating.

If we think about the analogy of sealed envelopes, we send them and we believe that their contents is already determined. However, if we do not open them, they look the same whatever the contents is.

In the rest of this section we will need commitment schemes with a seemingly stronger secrecy requirement that defined above. Specifically, instead of requiring secrecy with respect to all polynomial-time machines, we will require secrecy with respect to all (not necessarily uniform) families of polynomial-size circuits. Assuming the existence of non-uniformly one-way functions commitment schemes with nonuniform secrecy can be constructed, following the same constructions used in the uniform case.

Proposition 17.3.3 *The interactive proof system for G3C that uses bit commitment schemes instead of the "magic boxes" for sending colors is still zero-knowledge.*

Proof:

1. Completeness.

If the graph is 3-Colorable, the prover (the sender) will have no problem to convince the verifier (the receiver) by sending the "right keys" to the commitment schemes, that contain the colors of the endpoints of the edge that the verifier asked to inspect. More formally, by sending the "right keys" we mean performing the reveal phase of the commitment scheme. It takes the following form:

1. The prover sends to the verifier the bit σ (the contents of the sealed envelope) and its random coins, s, that he has used when he committed to the bit (we call it the *commit phase*).

2. The verifier checks that σ and s and his own random coins r indeed yield messages that the verifier has received in the commit phase. This verification is done by running the segment of the prover's program in which it committed to the bit σ and the verifier's program. Both programs are now run by the verifier with fixed coins and take polynomial time. Observe that the verifier could not run the whole prover's program, since it need not to be polynomial. As previously, the probability of accepting a 3-Colorable graph is 1.

2. Soundness.

The unambiguity requirement of the commitment scheme definition ensures that the soundness is satisfied too. We will have a very small increase in the probability to accept a non-3-Colorable graph, relatively to the case when we used magic boxes. As in the proof of Claim 3.2 let G be a non-3-Colorable graph and ϕ the assignment of colors that the prover uses. If the interactive proof system is based on magic boxes, then the probability to accept G is exactly equal to the probability of selecting a properly colored by ϕ edge from G while selecting one edge uniformly at random. As we have seen in the proof of Claim 3.2, this probability (further denoted p_0) is bounded by $\frac{|Edges(G)|-1}{|Edges(G)|}$. Intuitively, p_0 is the probability that the verifier asked to inspect an edge that is properly colored by ϕ, although ϕ is not a proper coloring.

If the proof system is based on commitment schemes rather than on magic boxes, then except of p_0 there is a probability that the verifier asked to inspect a non-properly colored edge, but the prover has succeeded to cheat him. It may happen only if the verifier's random coins r belong to the fraction of all possible random coins of the verifier, for which there are random coins of the prover, which enable him to pretend that he committed both to 0 and to 1. Unambiguity requirement of the commitment scheme ensures that this fraction is negligible, and hence the probability (further denoted p_1) that r belongs to this fraction is negligible too. Thus we can bound p_1 by $\frac{1}{2*|Edges(G)|}$. So, the total probability to accept a non-3-Colorable graph is bounded by

$$p_0 + p_1 \leq \frac{|Edges(G)|-1}{|Edges(G)|} + \frac{1}{2*|Edges(G)|} = \frac{|Edges(G)|-1/2}{|Edges(G)|}.$$

By repeating the protocol sufficiently many times we can make this probability smaller than $\frac{1}{3}$, thus satisfying the soundness property.

3. Zero-Knowledge.

The zero-knowledge, that will be guaranteed now, is a computational zero-knowledge.

To show this we prove that M^* outputs $\perp$ with probability at most $\frac{1}{2}$, and that, conditioned on not outputting $\perp$, the simulator's output is computationally indistinguishable from the verifier's view in a "real interaction with the prover".

Claim 17.3.4 *For every sufficiently large graph, $G = (V, E)$, the probability that $M^*(\langle G\rangle) = \perp$ is bounded above by $\frac{1}{2}$.*

Proof: As above, n will denote the cardinality of the vertex set of G. Let $e_1, e_2, ..., e_n$ be the contents of the n "sealed envelopes" that the prover/the simulator sends to the verifier. Let $s_1, s_2, ..., s_n$ be the random coins that the prover used in the commit phase while committing to $e_1, e_2, ..., e_n$. Let us denote by $p_{u,v}(G, r, (e_1, e_2, ..., e_n))$ the probability, taken over all the choices of the $s_1, s_2, ..., s_n \in \{0,1\}^n$, that V^*, on input G, random coins r, and prover message $(C_{s_1}(e_1), ..., C_{s_n}(e_n))$, replies with the message (u, v). We assume, for simplicity, that V^* always answers with an edge of G (since otherwise its message is anyhow treated as if it were an edge of G). We claim that for every sufficiently large graph, $G = (V, E)$, every $r \in \{0,1\}^{q(n)}$, every edge $(u, v) \in E$, and every two sequences $\alpha, \beta \in \{1,2,3\}^n$, it holds that

$$|p_{u,v}(G, r, \alpha) - p_{u,v}(G, r, \beta)| \leq \frac{1}{2|E|}$$

This is proven using the non-uniform secrecy of the commitment scheme.
For further details we refer the reader to [3] (or to [1]). ■

Claim 17.3.5 *The ensemble consisting of the output of M^* on input $G = (V, E) \in G3C$, conditioned on it not being $\perp$, is computationally indistinguishable from the ensemble $view_{(P,V^*)}(\langle G\rangle)_{G\in G3C}$. Namely, for every probabilistic polynomial-time algorithm, A, every polynomial $p(\cdot)$, and every sufficiently large graph $G = (V, E)$,*

$$\left|Pr(A(m^*(\langle G\rangle)) = 1) - Pr(A(view_{(P,V^*)}(\langle G\rangle)) = 1)\right| < \frac{1}{p(|V|)}$$

We stress that these ensembles are very different (i.e., the statistical distance between them is very close to the maximum possible), and yet they are computationally indistinguishable. Actually, we can prove that these ensembles are indistinguishable also by (non-uniform) families of polynomial-size circuits. In first glance it seems that Claim 17.3.5 follows easily from the secrecy property of the commitment scheme. Indeed, Claim 17.3.5 is proven using the secrecy property of the commitment scheme, yet the proof is more complex than one anticipates at first glance. The difficulty lies in the fact that the above ensembles consist not only of commitments to values, but also of an opening of some of the values. Furthermore, the choice of which commitments are to be opened depends on the entire sequence of commitments.
Proof: The proof can be found in [3] (or to [1]). ■

This completes the proof of computational zero-knowledge and of Proposition 17.3.3. ■

Such a bit commitment scheme is not difficult to implement. We observe that the size of the intersection between the space of commitments to 0 and the space of commitments to 1 is a negligible fraction of the size of the union of the two spaces, since a larger intersection would violate the unambiguity requirement. Hence, the space of commitments to 0 and the space of commitments to 1 almost do not intersect. On the other hand, commitments to 0 should be indistinguishable from commitments to 1. Using mechanisms of one-way functions and hard-core bits, we can satisfy the seemingly conflicting requirements.

Consider the following construction.

Let $f : \{0,1\}^n \longrightarrow \{0,1\}^n$ be one-way permutation, and $b : \{0,1\}^n \longrightarrow \{0,1\}$ be its hard-core bit, where n is a security parameter. To commit itself to a bit $v \in \{0,1\}$, the sender S selects uniformly at random $s \in_R \{0,1\}^n$ and sends $(f(s), b(s) \oplus v)$ to the receiver R. R stores this pair as $\alpha = f(s)$ and $\sigma = b(s) \oplus v$.

In the second phase (i.e. when sending the "keys"), S reveals its random coins s. R calculates $v = \sigma \oplus b(s)$, and accepts v if $\alpha = f(s)$. Otherwise, R rejects, since if $\alpha \neq f(s)$, then the sender tries to cheat.

Proposition 17.3.6 *The protocol is a bit commitment scheme.*

Proof:

Secrecy: For every receiver R^* consider the distribution ensembles $\langle S(0), R^*\rangle (1^n)$ and $\langle S(1), R^*\rangle (1^n)$. Observe that

$$\langle S(0), R^*\rangle (1^n) = (f(s), b(s))$$

and

$$\langle S(1), R^*\rangle (1^n) = (f(s), \overline{b(s)}).$$

By definition of hard-core bit $b(\cdot)$ of a one-way function $f(\cdot)$, for every probabilistic polynomial time algorithm A, every polynomial $p(\cdot)$ and for every sufficiently large random string s

$$Pr[A(f(s)) = b(s)] < \frac{1}{2} + \frac{1}{p(|s|)}.$$

In other words, the bit $b(s)$ is *unpredictable* by probabilistic polynomial time algorithm given $f(s)$. Thus the distributions $(f(s), b(s))$ and $(f(s), \overline{b(s)})$ are probabilistic polynomial time indistinguishable. (For the proof of equivalency between indistinguishability and unpredictability see the previous lecture.)

Hence for any probabilistic polynomial time distinguisher D

$$\left| Prob[D(f(s), b(s)) = 1] - Prob[D(f(s), \overline{b(s)} = 1]) \right| < \frac{1}{p(|s|)}$$

proving the secrecy.

Unambiguity: We claim that there is no r for which $(r, \overline{m})$ is ambiguous, where $\overline{m}$ is the sequence of messages that S sent to R. Suppose,that $(r, \overline{m})$ is a possible 0-commitment, i.e. there exist a string s such that, $\overline{m}$ describes the messages received by R, where R uses local coins r and interacts with S, which uses local coins s and has input $v = 0$ and security parameter n. Also, suppose for contradiction that $(r, \overline{m})$ is a possible 1-commitment.

Then there exists s_1 such that $View(S) = (s_1, 0, 1^n)$, $View(R) = (r, \overline{m})$.

And there exists s_2 such that $View(S) = (s_2, 1, 1^n)$, $View(R) = (r, \overline{m})$.

But then $\overline{m}$=$(f(s_1), b(s_1))$=$(f(s_2), \overline{b(s_2)})$. I.e. $f(s_1) = f(s_2)$, implying $s_1 = s_2$ since $f(\cdot)$ is a permutation. But then $b(s_1) = b(s_2)$, contradicting $b(s_1) = \overline{b(s_2)}$. Hence, the assumption that there exists ambiguous $(r, \overline{m})$ leads to the contradiction. Therefore, the unambiguity requirement is satisfied, implying that the protocol is a one-bit commitment scheme. ∎

17.4 Various comments

17.4.1 Remark about parallel repetition

Recall that the definition of zero-knowledge requires that for every V^* there exist a probabilistic polynomial time simulator M^* such that for every $x \in L$ the distributions of $\langle P, V^* \rangle (x)$, $M^*(x)$ are identical/statistically close/computationally indistinguishable.

A particular case of this concept is a blackbox zero-knowledge.

Definition 17.10 (Blackbox Zero-Knowledge): *Let (P, V) be an interactive proof system for some language L. We say that (P, V) is* perfect/statistical/computational blackbox zero-knowledge *if there exists an oracle machine M, such that for every probabilistic polynomial time verifier V^* and for every $x \in L$, the distributions of $\langle P, V^* \rangle (x)$, $M^{V^*}(x)$ are identical/statistically close/computationally indistinguishable.*

Recall that M^{V^*} is an oracle Turing machine M with access to oracle V^*. The following theorem is given without proof.

Theorem 17.11 *If there is an interactive proof system (P, V)* (with negligible error probability) *for language L that satisfies*

- *The prescribed verifier V, sends the outcome of each coin it tosses (i.e. the interactive proof system is of public-coin type).*
- *The interactive proof system consists of constant number of rounds.*
- *The protocol is blackbox zero-knowledge.*

Then $L \in BPP$.

Observe that $G3C$ protocol is blackbox zero-knowledge, but its failure probability is $1 - \frac{1}{|E|}$, hence it is not negligible. Blackbox zero-knowledge is preserved under sequential composition. Thus by repeating $G3C$ polynomially many times we obtain a blackbox zero-knowledge protocol with negligible error probability, but the number of rounds of this protocol is no more constant, thus violating the second condition of the theorem.

Blackbox zero-knowledge is not preserved under parallel composition. Indeed, there exist zero-knowledge proofs that when repeated twice in parallel do yield knowledge. Repeating G3C protocol polynomially many times in parallel clearly satisfies all conditions of the Theorem 17.11 except blackbox zero-knowledge. Thus, unless $NP \subseteq BPP$, this parallel protocol is not black-box zero-knowledge.

More generally, our unability to construct an interactive proof system that satisfies all the conditions of Theorem 17.11 $G3C$ does not surprise, since if we could construct such an interactive proof system, then by Theorem 17.11, $G3C$ would belong to BPP and since $G3C$ is NP-complete it would imply $NP \subseteq BPP$.

> Oded's Note: *All known zero-knowledge proof systems are proven to be so via a black-box argument, and it seems hard to conceive an alternative. Thus practially speaking, Theorem 17.11 may be understood as saying that zero-knowledge proofs (with negligible error) for languages outside BPP should either use non-constant number of rounds or use "private coins" (i.e., not be of public-coin type).*

17.4.2 Remark about randomness in zero-knowledge proofs

In interactive proofs it was important that the verifier took random steps. But as we mentioned, the prover could be deterministic. The only advantage of probabilistic prover in an interactive proof system is that it may be more efficient.

On the other hand, the prover in a zero-knowledge proof system has to be randomized, not because it has not enough power to do things deterministically, but rather because a deterministic prover will not be able to satisfy the zero-knowledge requirement. In $G3C$ example, suppose that the prover selected the permutation π in a very complicated and secret way, but deterministically. Then a simple verifier will just exhaust all the edges, when repeating the protocol several times. Hence, such a protocol would not be a zero-knowledge. In general, we may prove that if a language has a zero-knowledge proof in which either prover or verifier is deterministic, then the language is in BPP.

Bibliographic Notes

For the comprehensive discussion of zero-knowledge and commitment scheme see Chapter 4 in [1]. Theorem 17.11 appear in [4]

1. Oded Goldreich, *Foundations of Cryptography – fragments of a book*. February 1995. Revised version, January 1998. Both versions are available from `http://theory.lcs.mit.edu/~oded/frag.html`.

2. Oded Goldreich, *Modern Cryptography, Probabilistic proofs and Pseudorandomness*, Springer Verlag, Berlin, 1998.

3. O. Goldreich, S. Micali, A. Wigderson, *Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems*. *JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

4. O. Goldreich and H. Krawczyk, *On the Composition of Zero-Knowledge Proof Systems*, *SIAM Journal on Computing*, Vol.25, No. 1, February 1996, pages 169-192.

Lecture 18

NP in PCP[poly,O(1)]

Notes taken by Tal Hassner and Yoad Lustig

Summary: The major result in this lecture is $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. In the course of the proof we introduce an $\mathcal{NPC}$ language "Quadratic Equations", and show it to be in $\mathcal{PCP}(poly, O(1))$, in two stages : first assuming properties of the proof (oracle) and then testing these properties. An intermediate result that might invoke independent interest is an efficient probabilistic algorithm that distinguishes between linear and far-from-linear functions.

18.1 Introduction

In this lecture we revisit $\mathcal{PCP}$ in hope of giving more flavor of the area by presenting the result $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$. Recall that last semester we were shown without proof "the $\mathcal{PCP}$ theorem" $\mathcal{NP} = \mathcal{PCP}(log(n), O(1))$ (Corollary 12.3 in Lecture 12). (Clearly $\mathcal{PCP}(log(n), O(1)) \subseteq \mathcal{NP}$ as given only logarithmic randomness the verifier can only use its randomness to choose from a polynomial selection of queries to the oracle, therefor the answers to all the possible queries can be encoded in a polynomial size witness (for a more detailed proof see proposition 3.1 in lecture 12).

Two of the intermediate results on the way to proving the $\mathcal{PCP}$ theorem are $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$ and $\mathcal{NP} \subseteq \mathcal{PCP}(log, polylog)$. In this lecture we will only see a proof of $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$.

We recall the definition of a PCP proof system for a language L. A Probabilistically Checkable Proof system for a language L is a probabilistic polynomial-time oracle machine M (called verifier) satisfying:

1. Completeness : For every $x \in L$ there exists an oracle π_x such that : $\Pr[M^{\pi_x}(x) \text{ Accepts}] = 1$.
2. Soundness : For every $x \notin L$ and for every oracle π : $\Pr[M^{\pi}(x) \text{ Accepts}] \leq \frac{1}{2}$.

In addition to limiting the verifier to polynomial time we are interested in two additional complexity measures namely, the *randomness* M uses, and the *number of queries* M may perform. We denote by $\mathcal{PCP}(f(\cdot), g(\cdot))$ the class of languages for which there exists a PCP proof system utilizing at most $f(|x|)$ coin tosses and $g(|x|)$ queries for every input x.

We should note that an easy result is that $\mathcal{PCP}(0, poly) \subseteq \mathcal{NP}$. In $\mathcal{PCP}$ we allow the verifier two advantages over an $\mathcal{NP}$ verifier. The first is probabilistic soundness as opposed to perfect one, (which is not relevant in this case as the verifier is deterministic). The second major advantage is that the $\mathcal{PCP}$ "proof" (oracle) is not limited to polynomial length but is given as an oracle. One may look at the oracle as a proof of exponential length for which the verifier machine has random access (reading a bit from the memory address *addr* equivalent to performing a query about *addr*). However the verifier is limited to polynomial time and therefor actually reads only polynomial number of bits. In case the verifier is completely deterministic the bits it will read are known in advance and therefor there is no need to write down the rest of the bits i.e. a polynomial witness (as in $\mathcal{NP}$) will suffice.

18.2 Quadratic Equations

Back to our goal of proving $\mathcal{NP} \subseteq \mathcal{PCP}(poly, O(1))$, it is easy to see that for any $L \in \mathcal{NPC}$ proving $L \in \mathcal{PCP}(poly, O(1))$ will suffice, since given any language L_1 in $\mathcal{NP}$ we can decide it by reducing every instance x to it's corresponding instance x_L and using our $\mathcal{PCP}(poly, O(1))$ proof system to decide whether x_L is in L (iff $x \in L_1$).
In this section we introduce an $\mathcal{NPC}$ language that we will find convenient to work with.

Definition 18.1 (Quadratic Equations): *The language* Quadratic Equations *denoted QE consists of all satisfiable sets of quadratic equations over* $GF(2)$.

Since in GF(2) $x = x^2$ for all x we assume with out loss of generality all the summands are either of degree 2 or constants.
QE formulated as a problem :

Problem: QE
Input: A sequence of quadratic equations $\{\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)}\}_{k=1}^{m}$.
Task: Is there an assignment to $x_1 \dots x_n$ satisfying all equations ?

Clearly QE is in $\mathcal{NP}$ as a satisfying assignment is an easily verifiable witness. To see that it is $\mathcal{NP} - hard$ we will see a reduction from $3 - SAT$.

Given an instance of $3 - SAT$, $\bigwedge_{i=1}^{m} (l_{i1} \vee l_{i2} \vee l_{i3})$ where each l_{ij} is a literal i.e. either an atomic proposition p or it's negation $\neg p$. We will associate with each clause $C_i = (l_{i1} \vee l_{i2} \vee l_{i3})$ (for example $(p_4 \vee \neg p_7 \vee p_9)$) a cubic equation in the following way:

With each atomic proposition p we associate a variable x_p. Now, looking at C_i if l_{ij} is an atomic proposition p then we set the corresponding factor y_{ij} to be $(1 - x_p)$ otherwise l_{ij} is the negation of an atomic proposition $\neg p$ in which case we set the corresponding factor y_{ij} to be x_p . Clearly, the factor y_{ij} equals 0 iff the literal l_{ij} is true, and the expression $y_{i1}y_{i2}y_{i3}$ equals zero iff the disjunction $l_{i1} \vee l_{i2} \vee l_{i3}$ is true (In our example the clause $(p_4 \vee \neg p_7 \vee p_9)$ gets transformed to the equation "$(1 - x_4)x_7(1 - x_9) = 0$"). Therefor the set of equations $\{y_{i1}y_{i2}y_{i3} = 0\}_{i=1}^{m}$ is satisfiable iff our $3 - CNF$ is satisfiable.

We have reduced $3 - SAT$ to a satisfiability of a set of cubic equations but still have to reduce it to quadratic ones (formally we have to open parenthesis to get our equations to the normal form this can be easily done).

What we need is a way to transform a set of cubic equations into a set of quadratic ones (such that one set is satisfiable iff the other is). The latter can be done as follows:
For each pair of variables x_i, x_j in the cubic system introduce a new variable z_{ij} and a new equation $z_{ij} = x_i x_j$, now go through all the equations and whenever we find a summand of degree 3 $(x_i x_j x_k)$ replace it by a summand of degree 2 $(z_{ij} x_k)$. Our new equation system is composed of two parts, the first part introduces the new variables $(\{z_{ij} = x_i x_j\}_{i,j=1}^{n}$ quadratic in the size of the original

input), and the second part is the transformed set of equations all of degree 2 (linear in the size of the input)
In our example:
$(1-x_4)x_7(1-x_9)=0 \iff x_7 - x_4x_7 - x_7x_9 + x_4x_7x_9 = 0$ may be replaced by:
$x_4x_7 = z_{4\,7}$ and $x_7 - x_4x_7 - x_7x_9 + z_{4\,7}x_9 = 0$.
The last technical step is to go over the equation set and replace every summand of degree 1 (i.e. x_i) by its square (i.e. x_i^2). Since in GF(2) for all a it holds that $a = a^2$ this is purely a technical transformation that does not change in any way the satisfiability of the equations.
Clearly the new set of equations is satisfiable iff the original set is.

Since the entire procedure of reducing a $3-CNF$ to a set of quadratic equations can be done in polynomial time we have reduced $3-SAT$ to QE.

Note that the trick used to reduce the degree of summands in the polynoms can be iterated to reduce the degree of higher degree summands, however the new equations of the kind $z_{ij} = x_ix_j$ are of degree 2, and therefor such a trick cannot be used to reduce the degree of equations to degree 1. In fact degree 1 equations are linear equations for which we know an efficient procedure of deciding satisfiability (Gaussian Elimination). Thus there can be no way of reducing a set of quadratic equations to a set of linear ones in polynomial time, unless $\mathcal{P} = \mathcal{NP}$.

18.3 The main strategy and a tactical maneuver

The task standing before us is finding a $\mathcal{PCP}(poly, O(1))$ proof system for QE. Intuitively this means finding a way in which someone can present a proof for satisfiability of an equation set (the oracle), which might be very long but one may verify its correctness (at least with high probability) taking only a constant number of "glimpses" at the proof regardless of the size of the equation system.

The existence of such a proof system seems counter-intuitive at first since in the proof systems we are familiar with, any mistake however small in any place of the proof cause the entire proof to be invalid. (However existence of such proof systems is exactly what the "PCP Theorem" asserts.)

In this section we try to develop an intuition of how such a proof system can exist by outlining the main ideas of the proof that $QE \in \mathcal{PCP}(poly, O(1))$. We will deal with the "toy example" of proving that one linear expression can get the value 0 over GF(2) (of course since such a question is decidable in polynomial time there is a trivial proof system in which the oracle gives no information at all (always answers 1), but we will not make use of that triviality).

To develop an intuition we adopt a convention that the proof of validity for an equation system (the oracle in the $\mathcal{PCP}$ setting) is written by an adversary who tries to cheat us into accepting non-satisfiable equations. The goal of our system is to overcome such tries to deceive us (while still enabling the existence of proofs for satisfiable proof systems).

Suppose first that we can restrict our adversary to writing only proofs of certain kinds, i.e. having special properties. For example we may restrict the proofs to encode assignments in the following way :
Fix some encoding of linear expressions in the variables $x_1, \ldots, x_n$ into natural numbers (denote by C(ex) the natural number encoding the expression 'ex'). A reasonable encoding for the linear expression $\sum_{i=1}^{n} \lambda_i x_i$ would be simply the number whose binary expansion is the sequence of bits $\lambda_1\lambda_2\ldots\lambda_n$).

Suppose the adversary is restricted to writing proofs in which he encodes assignments. To encode an assignment $x_1 = a_0, \ldots, x_n = a_n$, the adversary evaluates all the linear expressions over

$x_1, \ldots, , x_n$ with that specific assignment, and writes the value $ex(a_0, \ldots, a_n)$ at the place $C(ex)$ in the proof. In the $\mathcal{PCP}$ setting this means the oracle π answers $ex(a_0, \ldots, a_n)$ on the query $C(ex)$. For example $\pi(C(x_1 + x_3))$ would have the value $a_1 + a_3$. If we can trust the adversary to comply to such a restriction, all we have to do given an expression ex, is to calculate $C(ex)$ and query the oracle π.

The problem of course is that we cannot restrict the adversary. What we can do is check whether the proof given to us is of the kind we assume it is. To do that we will use properties of such proofs. For example in our case we may try to use the linearity of adding linear expressions: for every two expressions e_1, e_2 it holds that $(e_1 + e_2)(a_0, \ldots, a_n) = e_1(a_0, \ldots, a_n) + e_2(a_0, \ldots, a_n)$. Therefor if the proof is of the kind we assume it to be, then the corresponding values in the proof will also respect addition, i.e. the oracle π must satisfy $\pi(C(e_1 + e_2)) = \pi(C(e_1)) + \pi(C(e_2))$.

In general we look for a characteristic of a special 'kind' of proofs. We want that assuming a proof is of that 'kind', one would be able to check its validity using only $O(1)$ queries, and that checking whether a proof is of that special 'kind' will also be feasible using $O(1)$ queries.

The main strategy is to divide the verification of a proof to two parts :

1. Check whether the proof is of a 'good kind' (the oracle answers have a special property), otherwise reject.
2. Assuming that the proof is of the 'good kind', determine its validity, (for example is the proof encoding a satisfying assignment to the equation system)

Up to this point we have developed the general strategy which we intend to use in our proof. Our characteristic of a 'good' proof would be that it encodes an assignment to the variables in a special way. In Section 4 we will define exactly when is a proof of the 'good kind', and see how 'good' proofs' validity can be checked using $O(1)$ queries. In Section 5 we will see how to check whether a proof is of the 'good kind' in $O(1)$ queries.

The bad news is that our strategy as stated above is infeasible if taken literally. We search for a characteristic of proofs that will enable us to distinguish between 'good' proofs and 'bad' proofs in $O(1)$ queries, but suppose we take a 'good' proof and switch just one bit. The probability that such a 'flawed' proof can be distinguished from the original proof in $O(1)$ queries is negligible. (The probability of even querying about that specific bit is the number of queries over the size of the proof, but the size of the proof is all the queries one may pose to an oracle which is exponential in our case).

On the other hand this problem should not be critical, if the adversary changes only one bit, it seems he doesn't have a very good chance of deceiving us, since we probably won't read that bit anyway and therefor it would look to us as if we were given a proof of the 'good' kind.

It seems that even if distinguishing between 'good' proofs and 'bad' ones is infeasible, it may suffice to distinguish between 'good' proofs and proofs which are 'very different' from 'good' proofs. We should try to find a "kind" of proofs for which we can verify that a given proof "looks similar" to. A proof will be "similar" to "good" if with high probability sampling from it will yield the same results as sampling from some really "good proof".

Since we only sample the proof given to us we can only check that the proof behaves "on average" as a proof with the property we want. This poses a problem in the other part of the proof checking process. When we think of it, the adversary does not really have to cheat in many places to deceive us, just in the few critical ones (e.g. the location $C(ex)$ where ex is the expression we try to evaluate). (Remember our adversary is a fictitious all powerful entity - it knows all about our algorithm. Our only edge is that the adversary must decide on a proof before we choose our random moves).

For example in our "toy system" we decide whether the equation is satisfiable based on the single bit $C(ex)$ (which one can calculate from the expression ex) so the adversary only has to change that bit to deceive us. In general during the verification process we do not ask random queries but ones that have an intent, usually this means our queries have a structure i.e. they belong to a small subset of the possible queries. So the adversary can gain a lot by cheating only on that small subset while "on the average" the proof will still look as of the good kind.

Here comes our tactical maneuver: What we would like is to enjoy both worlds, ask random queries and yet get answers to the queries that interest us (which are from a small subset). It turns out that sometimes this can be done, what we need is another property of the proofs. Suppose that we want to evaluate a linear function L on a point x. We can choose a random shift r and evaluate $L(x+r)$ and $L(r)$ we can now evaluate $L(x) = L(x+r) - L(r)$. If r is uniformly distributed so is $x+r$ (also they are dependent), so we have reduced evaluating L at a specific point (i.e. x) into evaluating L at two random points (i.e. r and $x+r$).

In general we want to calculate the value of a function f in a point x by applying f only to random points, we need some method of calculating $f(x)$ from our point x the shift r and the value of the function on the random points $f(x+r)$ and $f(r)$. If on a random point the chance of a mistake is small enough (i.e. p) then the chance of a mistake in one of the two points $x+r, r$ is at most $2p$ which means we get the value on x with probability greater or equal then $1-2p$. If one can apply such a trick to "good" proofs , we can ask the oracle only random queries, that way neutralizing an adversary attempt to cheat us at "important places". This is called *self-correction* since we can "correct" the value of a function (at an important point) using the same function.

18.4 Testing satisfiability assuming a nice oracle

This section corresponds to the second part of our strategy i.e. we are going to check satisfiability while assuming the oracle answers are nice enough.

The property we are going to assume is as follows:
The oracle π fixes some assignment $x_1 = a_1, \ldots, x_n = a_n$ and when given a sequence of coefficients $\{b_{ij}\}_{i,j=1}^{n}$ answers $\sum_{i,j=1}^{n} b_{ij} a_i a_j$. We assume that the testing phase (to be described in Section 6) ensures us that the oracle π has this property "on the average" i.e. on at most a fraction of 0.01 of the queries we might get an arbitrary answer. The idea is that the oracle encodes a satisfying assignment (if the equations are satisfiable) and that our task is verifying that the assignment encoded by the oracle is indeed satisfying.

Note that we may get the assignment a_i to any specific variable x_i by setting $b_{ii} = 1$ and all the other coefficients $b_{kl} = 0$; But we cannot just find the entire assignment (and evaluate the equations) since that would take a linear number of queries.

We have to check whether the set $\{\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)}\}_{k=1}^{m}$ is satisfiable using only a constant number of queries, the tools at our disposal allow us to evaluate any quadratic equation with the assignment the oracle encodes (also we might have to use our self correction trick to ensure the oracle does not cheat on the questions we are likely to ask. The self correction will work since the assignment is fixed and therefor $\sum_{i,j=1}^{n} b_{ij} a_i a_j$ is just a linear function of the b_{ij}s).

The naive approach of checking every equation in our set will not work for the obvious reason that the number of queries using this approach equals the number of equations which might be linear in the size of the input. We must find a way to "check many equations at once". Our trick will be random summations i.e. we will toss a coin for each equation and sum the equations for which the result of the toss is 1. If all the equations are satisfied clearly the sum of the equations will be satisfied (we sum both sides of the equations).

<u>Random summation test</u>

1 Choose a random subset S of the equations $\{\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j = c^{(k)}\}_{k=1}^{m}$

2 Sum the linear expressions at the left hand side of the equations in S and denote $ex_{rsum} = \sum_{k \in S} (\sum_{i,j=1}^{n} c_{i,j}^{(k)} x_i x_j)$.
After rearranging the summands present ex_{rsum} in normal form.

3 Sum the constants at the right hand side of the equations in S and denote $c_{rsum} = \sum_{k \in S} c^{(k)}$.
(Note that if an assignment $a_1, \ldots, a_n$ is satisfying then $ex_{rsum}(a_1, \ldots, a_n) = c_{rsum}$).

4 Query about ex_{rsum} using self correction technique and compare the answer to c_{rsum}. Accept if they are equal and reject otherwise.

What is the probability of discovering that not all the equations were satisfied ? If only one equation is not satisfied we would discover that with probability $\frac{1}{2}$ since if we include that equation in the sum (set S), the left hand side of the sum is not equal to the right hand side no matter which other equations we sum. But what happens if we have more then one unsatisfied equation?
In this case two wrongs make a right since if we sum two unsatisfied equations the left hand side will be equal to the right hand side (this is the wonderful world of GF(2)).

To see that in any case the probability of discovering the existence of an unsatisfied equation is $\frac{1}{2}$, consider the last toss of coin for an unsatisfied equation (i.e. for all the other unsatisfied equations it was already decided whether to include them in the sum or not). If the sum up to now is satisfied then with probability $\frac{1}{2}$ we will include that equation in the sum and therefor have an unsatisfied sum at the end, and if the sum up to now is not satisfied then with probability $\frac{1}{2}$ we will not include that last equation in the sum and remain with an unsatisfied sum at the end.

We have seen that if an assignment does not satisfy all the equations, then with probability $\frac{1}{2}$ it fails the random sum test. All that is left to completely analyze the test (assuming the oracle was tested and found nice) is a little book keeping.

First we must note that the fact the oracle passed the "oracle test" and found nice, does not assure us that all its answers are reliable, there is 0.01 probability of getting an arbitrary answer. In fact since our queries have a structure (they are all partial sums of the original set of equations), we must use self correction which means asking two queries for every query we really want to ask (of the random shift and of the shifted point of interest). During a random summation test we need to ask only one query (of the random sum), using self correction that becomes two queries (of the shifted point and of the random shift) and therefor we have a probability of 0.02 that our result will be arbitrary. Assuming the answers we got were not arbitrary, the probability of detecting an unsatisfying assignment is 0.5 . Therefor the overall probability of detecting a bad assignment is $0.98 * 0.5 = 0.49$ making the probability of failure 0.51. Applying the test twice and rejecting if one of the trials fail we get failure probability of $0.51^2 < 0.3$.

The result of our "book keeping" is that if we can test that the oracle has our property with failure probability smaller then 0.2, then the probability of the entire procedure to fail is smaller than 0.5, and we satisfy the $\mathcal{PCP}$ satisfiability requirement. (In our case the completeness requirement is easy, just note that an oracle encoding a satisfying assignment always pass).

18.5 Distinguishing a nice oracle from a very ugly one

In this section our objective is to present a method of testing whether the oracle behaves "typically well". We shall say that an oracle π behaves "well" if it encodes an assignment in the way defined above, and that π behaves "typicly well" if it agrees on more then 99% of the queries with a function that encodes an assignment that way. Our test should detect an oracle that does not behave "typicly well" with probability greater then 0.8 .

As described in the strategy our approach is to search some characterizing properties of the "good oracles". What does a "good oracle" look like ? Assume π is a good oracle corresponding to an assignment $x_1 = a_1, \ldots, x_n = a_n$. We may look at the oracle as a function $\pi : \{0,1\}^{n^2} \rightarrow \{0,1\}$ denoting its argument $\underline{b} = (b_{11}, \ldots, b_{1n}, b_{21}, \ldots, b_{nn})^\top$. Then, since $a_1 \ldots, a_n$ are fixed, π is a linear function of $\underline{b}$ i.e. $\pi(\underline{b}) = \sum_{i,j=1}^{n} \lambda_{ij} b_{ij}$. Furthermore π's coefficients λ_{ij} have a special structure - there exists a sequence of constants $\{a_i\}_{i=1}^n$ s.t. $\lambda_{ij} = a_i a_j$. It turns out that these properties characterize the "good oracles", since if π^* is a linear function for which there exists a sequence of constants $\{a_i^*\}_{i=1}^n$ s.t. π^*'s coefficients are $\lambda_{ij}^* = a_i^* a_j^*$, then π^* is encoding the assignment $x_1 = a_1^*, \ldots, x_n = a_n^*$.

Our "oracle test" is composed of two stages: testing the linearity of π and testing that the linearity coefficients λ_{ij}'s have the correct structure.

18.5.1 Tests of linearity

In order to devise a test and prove its correctness we begin with some formal definitions.

Definition 18.2 (linearity of a function)
A function $f : \{0,1\}^m \rightarrow \{0,1\}$ is called linear *if there exist constants $a_1^f, \ldots, a_m^f$ s.t. for all $\underline{\sigma} = (\sigma_1, \ldots, \sigma_m)^\top \in \{0,1\}^m$ it holds that $f(\underline{\sigma}) = \sum_{i=1}^{m} a_i^f \sigma_i$.*

Claim 18.5.1 (alternative definition of linearity)
A function $f : \{0,1\}^m \rightarrow \{0,1\}$ is linear *iff for every two vectors $\underline{\sigma^1}, \underline{\sigma^2} \in \{0,1\}^m$ and every $\lambda_1, \lambda_2 \in \{0,1\}$ it holds that: $\lambda_1 f(\underline{\sigma^1}) + \lambda_2 f(\underline{\sigma^2}) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$.*

Proof: Suppose first that f is linear by the definition i.e. there exists constants $a_1^f, \ldots, a_m^f$ s.t. for all $\underline{\sigma} = (\sigma_1, \ldots, \sigma_m)^\top$, $f(\underline{\sigma}) = \sum_{i=1}^{m} a_i^f \sigma_i$. Then for every $\underline{\sigma^1}, \underline{\sigma^2}$ it holds that $\lambda_1 f(\underline{\sigma^1}) + \lambda_2 f(\underline{\sigma^2}) = \sum_{i=1}^{m} a_i^f \lambda_1 \sigma_i^1 + \sum_{i=1}^{m} a_i^f \lambda_2 \sigma_i^2 = \sum_{i=1}^{m} a_i^f (\lambda_1 \sigma_i^1 + \lambda_2 \sigma_i^2) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$.

Suppose now that the claim holds i.e. for every two vectors $\sigma^1, \sigma^2 \in \{0,1\}^m$ $\lambda_1 f(\sigma^1) + \lambda_2 f(\sigma^2) = f(\lambda_1 \underline{\sigma^1} + \lambda_2 \underline{\sigma^2})$. Denote by a_i^f the value $f(\underline{e^i})$ where $\underline{e^i}$ is the i^{th} element in the standard basis i.e. all $\underline{e^i}$'s coordinates but the i^{th} are 0 and the i^{th} coordinate is 1.
Every $\underline{\sigma} \in \{0,1\}^m$ can be expressed as $\underline{\sigma} = \sum_{i=1}^{m} \sigma_i \underline{e^i}$.

Then, $f(\underline{\sigma}) = f(\sum_{i=1}^{m} \sigma_i \underline{e}^i)$ and by the claim we get :

$f(\sum_{i=1}^{m} \sigma_i \underline{e}^i) \;=\; \sum_{i=1}^{m} \sigma_i f(\underline{e}^i) \;=\; \sum_{i=1}^{m} \sigma_i a_i^f$ ■

Definition 18.3 (distance from linearity)

Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance at least δ *(or* δ far*) if :* $\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] \geq \delta$

Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance at most δ *(or* δ close*) if :* $\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] \leq \delta$

Two functions $f, g : \{0,1\}^m \to \{0,1\}$ *are said to be* at distance δ *if :* $\Pr_{\sigma \in_R \{0,1\}^m}[f(\sigma) \neq g(\sigma)] = \delta$

A function $f : \{0,1\}^m \to \{0,1\}$ *is said to be* at distance at most δ from linear *if there exists some linear function* $L : \{0,1\}^m \to \{0,1\}$ *s.t.* f *is at distance at most* δ *from* L.

In a similar fashion we define distance at least ϵ from linear *and* distance ϵ from linear.

Notice that since there are only finitely many linear functions $L : \{0,1\}^m \to \{0,1\}$ for every function f there is a closest linear function (not necessarily unique) and the distance from linearity is well defined.

We define now a verification algorithm $A^{(\cdot)}(\cdot)$ that accepts as input a distance parameter ϵ and oracle access to a function f (therefor we actually run $A^f(\epsilon)$), and tries to distinguish between f's which are at distance at least ϵ from linear and linear functions. A iterates a "basic procedure" $T^{(\cdot)}$ that detects functions which are "bad" (ϵ-far from linear) with small constant probability. Using enough iterations ensures the detection of "bad" f's with the desired probability.

<u>Basic procedure T^f:</u>

1 Select at random $\underline{a}, \underline{b} \in_R \{0,1\}^n$.

2 Check whether $f(\underline{a}) + f(\underline{b}) = f(\underline{a} + \underline{b})$, if not reject.

<u>Linearity Tester $A^f(\epsilon)$:</u>

1 Repeat for $\lceil \frac{36}{\epsilon} \rceil$ times the basic procedure $T^{(f)}$

2 Reject if $T^{(f)}$ rejects even once, accept otherwise.

Theorem 18.4 (Linearity testing): *If* f *is linear* T^f *always accepts, and if* f *is at distance at least* ϵ *from linear then* $\Pr[T^f \;\; rejects] \geq p_{rej} \stackrel{\rm def}{=} \frac{\epsilon}{4}$.

Proof: Clearly if f is linear T will accept. If f is not linear we deal separately with functions close to linear and with functions relatively far from linear.

Denote by δ the distance of f from linearity and let L be a linear function at distance δ from f.

Denote by $G = \{\underline{a} | f(\underline{a}) = L(\underline{a})\}$ the set of "good" inputs on which the functions agree. Clearly $|G| = (1-\delta)2^m$. We shall try to bound from below the probability that an iteration of A rejects.

Since the value of a linear function of every two from the three points $\underline{a}, \underline{b}, \underline{a}+\underline{b}$ fixes the value of third, it is easy to see that the algorithm will reject if two of these points are in G and the third is not in G. Therefor the probability that one iteration rejects is greater or equal to $\Pr[\underline{a} \in G, \underline{b} \in G, (\underline{a}+\underline{b}) \notin G] + \Pr[\underline{a} \in G, \underline{b} \notin G, (\underline{a}+\underline{b}) \in G] + \Pr[\underline{a} \notin G, \underline{b} \in G, \underline{a}+\underline{b} \in G]$.

The three events are clearly disjoint. What might be less obvious is that they are symmetric. It is easy to see $\Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G] = \Pr[\underline{a} \in G, \underline{b} \notin G, (\underline{a}+\underline{b}) \in G]$. Notice also that instead of choosing $\underline{b}$ at random we may choose $(\underline{a}+\underline{b})$ at random and then $\underline{a}+(\underline{a}+\underline{b}) = \underline{b}$, so the third event is also symmetric and therefor $\Pr[\underline{a} \in G, \underline{b} \in G, (\underline{a}+\underline{b}) \notin G] = \Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G]$.

Thus the probability of rejection is greater or equal to $3\Pr[\underline{a} \notin G, \underline{b} \in G, (\underline{a}+\underline{b}) \in G]$.
The latter can be presented as $3\Pr[\underline{a} \notin G] \cdot \Pr[\underline{b} \in G, (\underline{a}+\underline{b}) \in G | \underline{a} \notin G]$. By definition of δ it holds that $\Pr[\underline{a} \notin G] = \delta$. It is also clear that
$\Pr[\underline{b} \in G, (\underline{a}+\underline{b}) \in G | \underline{a} \notin G] = 1 - \Pr[\underline{b} \notin G \text{ or } (\underline{a}+\underline{b}) \notin G | \underline{a} \notin G]$.
Therefor the probability of rejection is greater or equal to $3\delta(1 - \Pr[\underline{b} \notin G \text{ or } (\underline{a}+\underline{b}) \notin G | \underline{a} \notin G])$. Using the symmetry of $\underline{b}$ and $\underline{a}+\underline{b}$ explained above and union bound the probability of rejection is greater or equal to $3\delta(1 - 2\Pr[\underline{b} \notin G | \underline{a} \notin G])$. Since $\underline{a}$ and $\underline{b}$ were chosen independently the probability of rejection is greater or equal to $3\delta(1 - 2\Pr[\underline{b} \notin G]) = 3\delta(1-2\delta)$.

Notice that the analysis above is good for small δ; i.e., for functions which are quite close to linear functions. However the function $3\delta(1-\delta)$ drops to 0 at $\frac{1}{2}$, therefor the analysis is not good enough for functions which are quite far from linear functions. The obvious reason for the failure of the analysis is that if the function is far enough from a linear function then the probability that two of the three points will fall inside the "good" set of points (whose f value is identical to their value under the closest linear function) is small. Thus, we need an alternative analysis for the case of big δ. Specifically, we show that if f is at distance greater or equal $\frac{3}{8}$ then the probability of rejection is at least $\frac{1}{8}$. As the proof is rather long and technical it is given in Appendix B.

Combining both cases, of functions relatively close to linear and functions relatively far from linear we get the desired result. That is, let δ be the distance of f from linear. Note that $\delta \leq \frac{1}{2}$, (since for every function f the expected distance of f from a random linear function is $\frac{1}{2}$). In case $\delta > \frac{3}{8}$ we have a rejection probability of at least $\frac{1}{8} \geq \frac{\delta}{4}$. Otherwise, $\delta \leq \frac{3}{8}$, and we have a rejection probability of at least $3\delta(1-2\delta) > 3\delta(1 - 2\cdot\frac{3}{8}) = \frac{3}{4}\delta$. ■

Corollary 18.5 *If f is linear then the verification algorithm $A^{(f)}(\epsilon)$ always accepts it. If f is ϵ far from linear then $A^{(f)}(\epsilon)$ rejects it with probability larger then 0.99*

Proof: If f is linear it always passes the basic procedure T. Suppose f is ϵ far from linear then by Theorem 4 the probability that one iteration of the basic procedure T^f rejects is bigger then $\frac{\epsilon}{4}$. Therefor the probability that f passes all the iterations A invokes is smaller then $(1-\frac{3}{4}\delta)^{36\cdot\frac{1}{\epsilon}}$. By the inequality $1 - x \leq e^{-x}$ the probability that A accepts is smaller then $e^{-\frac{3}{4}\epsilon\cdot\frac{36}{\epsilon}} = e^{-9} < 0.01$.

(To prove the $1 - x \leq e^{-x}$ inequality notice equality holds for 0, and differentiate both sides to see that the right hand side drops slower then the left hand side). ■

18.5.2 Assuming linear π testing π's coefficients structure

If π passes the linearity tester we assume that there exists a linear function $L_\pi : \{0,1\}^{n^2} \to \{0,1\}$ at distance smaller or equal 0.01 from π. For the rest of the discussion L_π stands for some such fixed linear function.

For π to be nice it is not enough that it is close to a linear L_π, but L_π must also be of the special structure that encodes an assignment. We saw that this means that there exists a sequence $\{a_i\}_{i=1}^n$ s.t. $L_\pi(\underline{b}) = \sum_{i,j=1}^{n} a_i a_j b_{ij}$ in other words L_π's linearity coefficients $\{\lambda_{ij}\}_{i,j=1}^n$ have the special structure which is $\lambda_{ij} = a_i a_j$.

L_π is a linear function from $\{0,1\}^{n^2}$ to $\{0,1\}$ and therefor its natural representation is as a row vector of length n^2 ($1 \times n^2$ matrix). However if we rearrange L_π's n^2 coefficients to an $n \times n$ matrix form the constraint on L_π can be phrased in a very elegant form, namely there exists a vector $\underline{a} = (a_1, \ldots, a_n)^\top$ s.t. $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$. (Notice this is not a scalar product but an outer product and the result is an $n \times n$ matrix). Notice that $(\underline{a}\,\underline{a}^\top)_{ij} = a_i a_j$ which is exactly what we want of λ_{ij}.

So what has to be checked is that (λ_{ij}) really has this structure i.e. there exists a vector $\underline{a}$ s.t. $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$. How can this be done ?

Notice that if (λ_{ij}) has this special structure then the vector $\underline{a}$ is exactly the diagonal of the matrix (λ_{ij}). This is the case since $\lambda_{ii} = a_i a_i = a_i^2$ however in GF(2) $x^2 = x$ for every x, leading us to $\lambda_{ii} = a_i$.

The last observation means that we can find out every coefficient a_i in $\underline{a}$ simply by querying with $b_{ii} = 1$ and the rest of the $b_{kl} = 0$ (we will have to use self correction as always). This seems to lead us to a reasonable test. First obtain $\underline{a}$ by querying its coefficients a_i, then construct $\underline{a}\,\underline{a}^\top$. What has to be done now is to check whether (λ_{ij}) is indeed equal to the matrix we have constructed $\underline{a}\,\underline{a}^\top$. A natural approach for checking this equality will be sampling: we have access to (λ_{ij}) as a function since we can query π, we can try to simulate $\underline{a}\,\underline{a}^\top$ as a function and compare their results on random inputs. The problem with this natural approach is that querying about every coefficient a_i would cost a linear number of queries and we can only afford a constant number of queries.

It seems we will have to get by without explicitly constructing the entire $\underline{a}$. As the "next best thing" we may try to "capture" $\underline{a}$ by a random summation of its coefficients. A random sum of $\underline{a}$ coefficients defined by a random string $\underline{r} \in_R \{0,1\}^n$ is the sum of the coefficients a_i for which $r_i = 1$ i.e. $\sum_{i=1}^{n} r_i a_i$. This is the scalar product of $\underline{a}$ with $\underline{r}$ denoted $< \underline{a}, \underline{r} >$. The random sum is of course a function of the random string $\underline{r}$ so we introduce the notation $A(\cdot) =< \underline{a}, \cdot >$ where $A(\cdot)$ is the random sum as a function of $\underline{r}$.

Just as we can find any coefficient a_i of $\underline{a}$ by querying π with the appropriate bit on the diagonal b_{ii} turned on, we can find the result of a sum of coefficients. To get the result of a sum of the coefficients $i_1, \ldots, i_k$ all we have to do is query with $b_{i_1 i_1} = 1, \ldots, b_{i_k i_k} = 1$ and all the other bits 0. The result will be $\sum_{l=1}^{k} x_{i_l}^2 = \sum_{l=1}^{k} x_{i_l}$ which is what we want. (As always we have to use self correction).

How is all this going to help us to compare the matrices $\underline{a}\,\underline{a}^\top$ to (λ_{ij}) ?
Most of us immediately identify an $n \times n$ matrix M with its corresponding linear function from $\{0,1\}^n$ to $\{0,1\}^n$. However such a matrix can also stand for a bilinear function
$f_M : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, (where $f_M(\underline{x}, \underline{y}) = \underline{x}^\top M \underline{y}$).

For matrices of the form $\underline{a}\,\underline{a}^\top$ the operation becomes very simple :
$f_{(\underline{a}\,\underline{a}^\top)}(\underline{x}, \underline{y}) = \underline{x}^\top (\underline{a}\,\underline{a}^\top) \underline{y} = (\underline{x}^\top \underline{a})(\underline{a}^\top \underline{y}) =< \underline{x}, \underline{a} >< \underline{a}, \underline{y} >= A(\underline{x}) A(\underline{y})$
(The same result can also be developed in the more technical way of opening the summations).

Our access to $A(\cdot)$ enables us to evaluate the bilinear function represented by $\underline{a}\,\underline{a}^\top$. We can also evaluate the bilinear function represented by (λ_{ij}) since :

$f_{(\lambda_{ij})}(\underline{x}, \underline{y}) = x^\top(\lambda_{ij})y = \sum_{i,j=1}^{n} \lambda_{ij}x_i y_j$.

So in order to evaluate $f_{(\lambda_{ij})}(\underline{x}, \underline{y})$ we only have to feed $b_{ij} = x_i y_j$ into π. Once again we will use our self correcting technique, this time the structure of the queries does not stand out as in the the case of $A(\cdot)$. Nonetheless, the distribution of queries is not uniform (for example there is a skew towards 0s as it is enough that one of the coordinates x_i or y_j is 0 so that b_{ij} will be 0).

It seems reasonable to test whether $(\lambda_{ij}) = \underline{a}\,\underline{a}^\top$ by testing whether the bilinear functions represented by the two matrices are the same, and do that by sampling. The idea is to sample random vectors $\underline{x}, \underline{y} \in \{0,1\}^n$ and check if the functions agree on their value.

Structure test for (λ_{ij}):

1 Select at random $\underline{x}, \underline{y} \in_R \{0,1\}^n$.
2 Evaluate $A(\underline{x})$ and $A(\underline{y})$ by querying with coefficients corresponding to a matrix U where $\underline{x}$ (resp. $\underline{y}$) is the diagonal i.e. $U_{ii} = x_i$ and the rest of the coefficients are 0s. (The queries should be presented using self correction).
3 Compute $f_{(\underline{a}\,\underline{a}^\top)}(\underline{x}, \underline{y}) = A(\underline{x}) \cdot A(\underline{y})$
4 Query $f_{(\lambda_{ij})}(\underline{x}, \underline{y})$ by querying π with $\{x_i y_j\}_{i,j=1}^{n}$ as the query bits. (Again self correction must be used).
5 Accept if $f_{(\underline{a}\,\underline{a}^\top)}(\underline{x}, \underline{y}) = f_{(\lambda_{ij})}(\underline{x}, \underline{y})$, Reject otherwise.

We are left to prove that if the matrices (λ_{ij}) and $\underline{a}\,\underline{a}^\top$ differ then if we sample we will get different results of the bilinear functions with reasonably high probability.

Given two different matrices M, N we want to bound from below the probability that for two random vectors $\underline{x}, \underline{y}$ the bilinear functions $f_M(\underline{x}, \underline{y})$ and $f_N(\underline{x}, \underline{y})$ agree.

The question is when $\underline{x}^\top M\underline{y} = \underline{x}^\top N\underline{y}$?

Clearly this is equivalent to $\underline{x}^\top(M-N)\underline{y} = 0$.

Suppose we choose $\underline{y}$ first, if $(M-N)\underline{y} = \underline{0}$ then whatever $\underline{x}$ we choose the result $\underline{x}^\top\underline{0}$ will always be 0 and does not depend on $\underline{x}$. If on the other hand $(M-N)\underline{y} \neq \underline{0}$ then it might be the case that $\underline{x}^\top((M-N)\underline{y}) = 0$ depending on the choice of $\underline{x}$. We will analyze the probabilities for these two events separately.

To analyze the probability of choosing a $\underline{y}$ s.t. $(M-N)\underline{y} = \underline{0}$, denote the column dimension of $(M-N)$ by d. There exist d linearly independent columns $i_1, \ldots, i_d$ of $(M-N)$ assume without loss of generality these are the last d columns $n-d+1, \ldots, n$. Also assume the choice of $\underline{y}$ is made by tossing coins for its coordinates' values one by one and that the coins for the last coordinates $n-d+1, \ldots, n$ are tossed last. Lets look at the situation just before the last d coins are tossed (the rest of the coins have already been chosen).

The value of $(M-N)\underline{y}$ will be the sum of columns corresponding to coordinates of value 1, however at our stage not all the coordinates values have been chosen. At this stage we may look of at the last d columns' "contribution" to the final sum $((M-N)\underline{y})$ as a random variable. Denote by $(M-N)_{\downarrow j}$ the j^{th} column in $(M-N)$ and by $\underline{v_{rand}}$ the random variable $\sum_{k=n-d+1}^{n} y_k \cdot (M-N)_{\downarrow k}$. (The sum of columns corresponding to coordinates of value 1 out of the last d coordinates). The rest of the coordinates "contribution" has already been set. Denote by $\underline{v_{set}}$ the vector $\sum_{k=1}^{n-d} y_k \cdot (M-N)_{\downarrow k}$ (The contribution of the rest of the coordinates).

Clearly $(M-N)\underline{y} = \underline{v_{set}} + \underline{v_{rand}}$ i.e. $(M-N)\underline{y} = \underline{0}$ iff $\underline{v_{set}} = -\underline{v_{rand}}$. The question is can this happen and at what probability ? First note that this can happen - otherwise $\underline{v_{set}}$ is independent of columns $i_1, \ldots, i_d$ which means the columns dimension is bigger then d. Second notice that since columns $i_1, \ldots, i_d$ are independent there is only one linear combination of them that equals $\underline{v_{set}}$. This means that there is only one result of the coin tosses for coordinates $i_1, \ldots, i_d$ that will make $\underline{v_{rand}}$ equal $\underline{v_{set}}$, i.e. the probability is $2^{-d} = 2^{-dim(M-N)}$.

Assuming $(M-N)\underline{y} \neq \underline{0}$ what is the probability that $\underline{x}^\top((M-N)\underline{y}) = 0$? The question is given a fixed vector that is not $\underline{0}$ what is the probability that its inner product with a randomly chosen vector is 0 ? By exactly the same reasoning used above (and in the random sum argument in Section 4), the probability is $\frac{1}{2}$. (Toss coins for the random vector coordinates one by one and look at the coin tossed for a coordinate which is not 0 in $(M-N)\underline{y}$, there is always a result of the coin flip that would bring the inner product to 0).

Overall the two vectors have the same image if either $(M-N)\underline{y} = 0$ (with probability bounded by $\frac{1}{2}$) or if $\underline{x}^\top((M-N)\underline{y}) = 0$ (with probability $\frac{1}{2}$). Therefor the probability that the two vectors differ is at least $\frac{1}{4}$.

To finish the probability "book keeping" we must take into account the probability of getting wrong answers from the oracle. The test itself has a failure probability of 0.75 . The test uses three queries ($A(\underline{x}), A(\underline{y})$ and $f_{(\lambda_{ij})}(\underline{x}, \underline{y})$) but using self correction each of those becomes two queries. We get a probability of 0.06 of getting an arbitrary answer leading us to probability of 0.81 for the test to fail. Repeating the test 10 independent times we get a failure probability of $0.81^{10} \leq \frac{1}{8}$ and the probability of failure in either the linearity testing or structure test is bounded by $0.125 + 0.01 \leq 0.15$.

The goal of detecting "bad oracles" with probability greater than 0.8 is achieved.

18.5.3 Gluing it all together

Basicly our verification of a proof consists of three stages:

1. linearity testing : a phase in which if the oracle π is 0.01 far from linear we have a chance of 0.99 to catch it (and reject).
   To implement this stage we use the linearity tester $A^\pi(0.01)$. (Distance parameter set to 0.01).

2. structure test : assuming the oracle π is close to linear we test that it encodes an assignment using the "structure test". To boost up probability we repeat the test 10 independent times. At the end of this stage we detect bad oracles with probability greater than 0.8

3. satisfiability test : assuming "good oracle" we use the "random summation" test to verify that the assignment encoded by the oracle is indeed satisfying. (This test is repeated twice).

Bibliographic Notes

This lecture is mostly based on [1], where $\mathcal{NP} \subseteq \mathcal{PCP}[\text{poly}, O(1)]$ is proven (as a step towards proving $\mathcal{NP} \subseteq \mathcal{PCP}[\log, O(1)]$). The linearity tester is due to [4], but the main analysis presented here follows [3]. For further studies of this linearity tester see [2].

1. S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof Verification and Intractability of Approximation Problems. *JACM*, Vol. 45, pages 501–555, 1998. Preliminary version in *33rd FOCS*, 1992.

2. M. Bellare, D. Coppersmith, J. Hastad, M. Kiwi and M. Sudan. Linearity testing in characteristic two. *IEEE Transactions on Information Theory*, Vol. 42, No. 6, November 1996, pages 1781–1795.

3. M. Bellare, S. Goldwasser, C. Lund and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *25th STOC*, pages 294–304, 1993.

4. M. Blum, M. Luby and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. *JCSS*, Vol. 47, No. 3, pages 549–595, 1993. Preliminary version in *22nd STOC*, 1990.

Appendix A: Linear functions are far apart

In this section we intend to prove that linear functions are far apart. In addition to the natural interest such a result may invoke, we hope the result may shed some light as to why linear functions are good candidates for $\mathcal{PCP}$ proof systems.

When looking for a $\mathcal{PCP}$ proof system it is clear that the proofs must be robust in the sense that given a good proof π a small change effecting a constant number of bits must yield another good proof π_1, this is so since the probability of the verifier's to detect such a small change is negligible. This means that π_1 must carry "the same information" as π (at least with respect to the information the verifier can "read" from it). This formulation has a very strong scent of error correcting codes, we do know that error correcting codes have this property. In the case of a $\mathcal{PCP}$ proof system for a language in $\mathcal{NP}$ we even have a natural candidate for the information to be coded namely the witness.

One should note that our discussion is just a plausibility argument as there are major differences between a $\mathcal{PCP}$ verifier and an error correcting decoder. On one hand the $\mathcal{PCP}$ verifier does not have to decode at all. The verifier's task is to be convinced that there exists an information word with some desired properties (the witness). On the other hand the verifier is significantly weaker (computationally) then an efficient error correcting decoder, since it may only look at part of the code word (proof).

If the reader is convinced that the error correcting approach is a good one to begin with, note that in addition to being an error correcting code linear functions have another desired property : we can use self correction in a natural way since $L(x) = L(x+r) - L(r)$.

Proposition 18.5.2 *If $f, g : \{0,1\}^m \to \{0,1\}$ are both linear and $f \neq g$ then the distance of f from g is exactly $\frac{1}{2}$*

Proof: Note that $(f-g)$ is also linear, clearly $f(\underline{x}) \neq g(\underline{x})$ iff $(f-g)(\underline{x}) \neq 0$. All we have left to prove is that for every linear function $h \neq 0$ it holds that $\Pr_{\underline{x}}[h(\underline{x}) \neq 0] = \frac{1}{2}$.
Denote $h(\underline{x}) = \sum_{i=1}^{m} a_i^h x_i$. Since $h \neq 0$ there exists an a_i^h that does not equal 0. Assume the bits of $\underline{x}$ are chosen one by one (x_1 is chosen before x_2 and so on), denote by l the last i for which a_i^h is not 0 (i.e. $a_l = 1$ and for all $j > l$ $a_j = 0$). Clearly $h(\underline{x}) = (\sum_{i=1}^{l-1} a_i^h x_i) + x_l$. If $\sum_{i=1}^{l-1} a_i^h x_i = 0$ then with probability $\frac{1}{2}$ we choose $x_l = 1$ and we get $h(\underline{x}) = 1$. If on the other hand $\sum_{i=1}^{l-1} a_i^h x_i = 1$ then with probability $\frac{1}{2}$ we choose $x_l = 0$ and again we get $h(\underline{x}) = 1$. ■

Corollary 18.6 *If a function $f : \{0,1\}^m \rightarrow \{0,1\}$ is at distance less than $\frac{1}{4}$ from linear then there exists a* unique *linear function L_f s.t. f is at distance less then $\frac{1}{4}$ from L_f.*

Proof: Otherwise by triangle inequality we would get two linear functions at distance smaller than $\frac{1}{2}$ ■

Appendix B: The linearity test for functions far from linear

Recall that our objective is to bound the probability of failure of one iteration of the basic procedure T in case of functions far from linear. The result we need is that given a function $f : \{0,1\}^m \rightarrow \{0,1\}$ which is at distance at least $\frac{3}{8}$ from linear the probability that $f(x+y) \neq f(x) + f(y)$ for randomly chosen x, y is bigger than some constant c. Denote by η the probability of failing one iteration of T (i.e. $\eta \stackrel{\text{def}}{=} \Pr_{x,y}[f(x+y) \neq f(x) + f(y)]$).

Intuitively if the probability of choosing x and y s.t. $f(x) + f(y) = f(x+y)$, is very big than there must be some linear function L which agree with f on "a lot" of points so f cannot be "too far" from linear. The problem is that it is not clear how to find this L or even those points on which L and f agree.

Since a linear function behavior on the entire space $\{0,1\}^n$ is fixed by its behavior on any n independent vectors, we should somehow see how "most of the points would like f to behave". Formalizing this intuition is not as hard as it may seem. Given a point a we would like to find how "most of the points want f to behave on a", the natural way to define that, is that the point x "would like" $f(a)$ to take a value that would make $f(a) = f(x+a) - f(x)$ true.

Definition 18.7 ($L_f(\cdot)$):
For every $a \in \{0,1\}^n$ define $L_f(a)$ to be either 0 or 1 s.t. $\Pr_{x \in_R \{0,1\}^n}[L_f(a) = f(x+a) - f(x)] \geq \frac{1}{2}$
(In cases were $L_f(a)$ might be either 0 or 1 (probability $\frac{1}{2}$) define $L_f(a)$ arbitrarily.)

What we would like to see now is that L_f is indeed linear, and that L_f is reasonably close to f (depending on η the probability to fail the basic procedure T). We would indeed prove those claims but before embarking on that task we have to prove a technical result.

By definition $\Pr_{x,a}[f(x) + f(x+a) = L_f(a)] \geq 1 - \eta$ therefor by an averaging argument $L_f(a)$ behaves nice "on the average a" (i.e. for most a's $\Pr_x[f(x) + f(x+a) = L_f(a)] \geq 1 - 2\eta$). However there might have been a few bad points b on which $\Pr_x[f(x) + f(x+b) = L_f(b)] = \frac{1}{2}$. We would like to show L_f behaves nicely on all the points.

Claim 18.5.3 *For all points $a \in \{0,1\}^n$ it holds that:* $\Pr_{x \in_R \{0,1\}^n}[f(x) + f(x+a) = L_f(a)] \geq 1 - 2\eta$.

Proof: Look at two points x, y chosen at random, what is the probability that "they vote the same for $L_f(a)$" i.e. $f(x) + f(x+a) = f(y) + f(y+a)$?
Clearly if we denote $p = \Pr_x[f(x) + f(x+a) = L_f(a)]$ then
$\Pr_{x,y}[f(x) + f(x+a) = f(y) + f(y+a)] = p^2 + (1-p)^2$ (x and y might both go with $L_f(a)$ or against it). From Another perspective:

$$
\begin{aligned}
&\Pr_{x,y}[f(x) + f(x+a) = f(y) + f(y+a)] = \\
&= \Pr_{x,y}[f(x) + f(x+a) + f(y) + f(y+a) = 0] = \\
&= \Pr_{x,y}[f(x) + f(y+a) + f(x+y+a) + f(y) + f(x+a) + f(x+y+a) = 0] \leq \\
&\leq \Pr_{x,y}[(f(x) + f(y+a) + f(x+y+a) = 0) \wedge (f(y) + f(x+a) + f(x+y+a) = 0)] =
\end{aligned}
$$

$= \ 1 - \Pr_{x,y}[(f(x) + f(y+a) + f(x+y+a) \neq 0) \vee (f(y) + f(x+a) + f(x+y+a) \neq 0)] \leq$
$\leq \ 1 - 2\eta$

The last inequality is true since all $x, y, a+x, a+y$ are uniformly distributed (also dependent), from the definition of η and using the union bound. We got that $p^2 + (1-p)^2 \geq 1 - 2\eta$. Simple manipulation brings us to :
$1 - 2p + 2p^2 \geq 1 - 2\eta \iff p(p-1) \geq -\eta \iff p(1-p) \leq \eta$.
Note that since $L_f(a)$ value (0 or 1) was defined to maximize $p = \Pr[f(x) + f(x+a) = L_f(a)]$, p is always bigger or equal to $\frac{1}{2}$. So $\frac{1}{2}(1-p) \leq p(1-p) \leq \eta$ and from that $p \geq 1 - 2\eta$ follows. ■

Claim 18.5.4 *If the failure probability η is smaller then $\frac{1}{6}$ then the function $L_f(\cdot)$ is linear.*

Proof: Given any $a, b \in \{0,1\}^n$ we have to prove $L_f(a+b) = L_f(a) + L_f(b)$. The strategy here is simple, we will prove (by the probabilistic method) the existence of "good intermediate" points in the sense that L_f behaves nicely on these points and this forces L_f to be linear.
Suppose there exists x, y s.t. the following events happen simultaneously :

E1 : $L_f(a+b) = f(a+b+x+y) + f(x+y)$.
E2 : $L_f(b) = f(a+b+x+y) + f(a+x+y)$.
E3 : $L_f(a) = f(a+x+y) + f(x+y)$.

Then :

$$L_f(a+b) = f(a+b+x+y) + f(x+y) = L_f(b) + f(a+x+y) + f(x+y) = L_f(b) + L_f(a)$$

.

To prove the existence of these points choose x, y at random. By Claim 7.1 the probability that each of the events E1, E2, E3 will not happen is smaller or equal 2η, so the probability that one of these will not happen is smaller or equal to 6η. Since $\eta < \frac{1}{6}$ the probability that some of those events do not happen is smaller then 1, i.e. there exists x and y for which the events E1, E2, E3 happen and therefor L_f is linear with regard to a, b. ■

Claim 18.5.5 *The function $L_f(\cdot)$ is at most at distance 3η from f.*

Proof: Basicly we show that since L_f is defined as "how f should behave if it wants to be linear" then if $f(a) \neq L_f(a)$ then "f is not linear on a" since f is close to linear it must be close to L_f.

Denote by p the probability that f agrees with L_f i.e. $\Pr_x[f(x) = L_f(x)] = p$. (Notice that the distance between f and L_f equals $1-p$ by definition).

Choose two vectors x, y and denote by E the event "$f(x+y) = L_f(x+y)$" and by F the event "$f(x) + f(y) = f(x+y)$".
$\Pr_{x,y}[F \wedge E^c] = \Pr_{x,y}[(f(x) + f(y) = f(x+y)) \wedge (f(x+y) \neq L_f(x+y))] =$
$= \ \Pr_{x,y}[(f(x+y+y) + f(y) = f(x+y)) \wedge (f(x+y) \neq L_f(x+y))]$

Clearly whenever $f(x+y+y) + f(y) = f(x+y)$ and $f(x+y) \neq L_f(x+y)$ it holds that $f(x+y+y) + f(y) \neq L_f(x+y)$ therefor :
$\Pr_{x,y}[F \wedge E^c] \ \leq \ \Pr_{x,y}[f((x+y)+y) + f(y) \neq L_f(x+y)] \leq 2\eta$
Where the last inequality is by claim 7.1

Now notice $\Pr[F] = (1-\eta)$ and $\Pr[E] = p$ by definition.
Looking at $\Pr[F]$ from another perspective:

$$\Pr[F] = \Pr[F \wedge E] + \Pr[F \wedge E^c] \leq \Pr[E] + \Pr[F \wedge E^c] \leq p + 2\eta$$

Comparing the two evaluations of $\Pr[F]$ we get $1-\eta \leq p+2\eta$ i.e. $p \geq 1-3\eta$. Since the distance between f and L_f is $1-p$ our result is that f's distance from L_f is smaller or equal to 3η ■

To conclude if η, the probability to fail T, is smaller then $\frac{1}{8}$ then f is at distance at most $\frac{3}{8}$ from linear and this means :

Corollary 18.8 *If f is at distance bigger than $\frac{3}{8}$ from linear then the probability to fail one iteration of the basic procedure T is bigger or equal to $\frac{1}{8}$.*

> Oded's Note: *The above analysis is not the best possible. One may show that if f is at distance bigger than $1/4$ from linear then T rejects with probability at least $2/9$.*

Lecture 19

Dtime vs Dspace

Notes taken by Tamar Seeman and Reuben Sumner

Summary: In this lecture, we prove that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. That is, we show how to simulate any given deterministic multi-tape Turing Machine (TM) of time complexity t, using a deterministic TM of space complexity $t/\log t$. A main ingrediant in the simulation is the analysis of a pebble game on directed bounded-degree graphs.

19.1 Introduction

We begin by defining Dtime(t) and Dspace(t).

Definition 19.1 (Dtime): *$Dtime(t(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM M which decides whether or not $x \in L$ in less than $t(|x|)$ Turing machine steps.*

Definition 19.2 (Dspace): *$Dspace(s(\cdot))$ is the set of languages L for which there exists a multi-tape deterministic TM which decides whether or not $x \in L$ while never going to the right of cell $s(|x|)$ on any of its tapes.*

Since any Turing machine step can move the head(s) by at most one position, it is immediately clear that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot))$. Furthermore NP is easily in $Dspace(p(\cdot))$ for some polynomial p, but is not believed to be in $Dtime(q(\cdot))$ for any polynomial q. Thus it seems that space is much more powerful than time. In this lecture we will further refine this intuition by proving that $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$. It follows that $Dtime(t(\cdot))$ is a strict subset of $Dspace(t(\cdot))$, since it has already been shown that $Dspace(o(t))$ is a strict subset of $Dspace(t)$.

Note: The multi-tape TM consists of one read-only, bi-directional input tape, one optional write-only output tape (in the case of a machine to decide a language we will not include such a tape and determine acceptance based on the exact state that the Turing machine terminates in) together with a fixed number of bi-directional read/write work tapes. The number of work tapes is irrelevant for $Dspace()$, though, since a TM M with k work tapes can be simulated by a TM M' with just a single work tape and the same amount of space. This is done by transforming k work tapes into one work tape with k tracks. This transformation then simulates the work of the original Turing machine by using polynomially more time, but the same amount of space. However, in the case of $Dtime()$ the number of tapes *does* matter.

19.2 Main Result

Theorem 19.3 *If $t(\cdot)$ is constructible in space $t(\cdot)/\log t(\cdot)$ and $t(\cdot)$ is at least linear, then $Dtime(t(\cdot)) \subseteq Dspace(t(\cdot)/\log t(\cdot))$.*

In order to make the proof of this theorem as readable as possible we state some results (without proof) in place they are needed so that their motivation is clear, but we prove them only later so as not to disturb the flow of the proof.

Proof: Let L be a language accepted by a TM M in $Dtime(t(\cdot))$. Let x be an input and $t = t(|x|)$. Divide each tape into $t^{1/3}$ blocks of $t^{2/3}$ cells. (This ensures that $(\#\text{ blocks})^2 \ll t$, a necessary feature.) Similarly, partition time into periods of $t^{2/3}$ steps. During a particular period, M visits at most two blocks of space on each tape, since the number of steps M can take does not exceed the size of a single block.

Lemma 19.2.1 (Canonical Computation Lemma): *Without loss of generality such a machine moves between block boundaries only on the very last move of a time period. This holds with at most a linear increase in the amount of time used, and a constant number of additional tapes.*

The proof is postponed to the next section. Therefore without loss of generality we assume that in each time period our machine stays within the same block on each work tape.

Our goal now is to compute the final state of the machine, which will indicate whether or not our input is in the language. We therefore have to somehow construct the blocks that we are in at the final time period while never storing the complete state of the work tapes of the machine since these would potentially exceed our space bound. To do so, we establish a dependency graph between different states and find a sufficiently efficient method of evaluating the graph node corresponding to the final machine state.

We introduce some notation to describe the computation of the machine.

$h_i(j)$ is the block location of the i^{th} tape head during the j^{th} period. $h_i(j) \in \left\{1, \ldots, t^{1/3}\right\}$.

$h(j)$ is the vector $h_1(j), h_2(j), \ldots, h_k(j)$ for a k-tape machine.

$c_i(j)$ is the content of block $h_i(j)$ on the i^{th} tape, together with the Turing machine state at the end of the period and head position on tape i. $c_i(j) \in \{0,1\}^{t^{2/3}} \times O(1) \times \{0,1\}^{O(\log t)}$. [1]

$c(j)$ is the vector $c_1(j), c_2(j), \ldots, c_k(j)$.

$l_i(j)$ is the last period where we visited block $h_i(j)$ on tape i. This is $\max\{j' < j \text{ such that } h_i(j') = h_i(j)\}$.

In order to compute $c(j)$ we will need to know:

- $c(j-1)$ to determine what state to start our computation in.
- $c_1(l_1(j)), \ldots, c_k(l_k(j))$ so that we know the starting content of the blocks we are working with.

[1] We assume that $\Sigma = \{0,1\}$, where Σ is the tape alphabet

Figure 19.1: An interesting pebbling problem

It is immediately clear then that we need at most $k+1$ blocks from past stages of computation to compute $c(j)$.

Define a directed graph $G = (V, E)$ where $V = \{1, \ldots, t^{1/3}\}$ and $E = \{(j-1, j) | j > 1\} \cup \{(l_i(j), j)\}$. So vertex i represents knowledge of $c(i)$. There is an edge $(j-1) \to j$ since $c(j)$ depends on $c(j-1)$. Similarly there is an edge $l_i(j) \to j$ since $c(j)$ depends on $c_i(l_i(j))$ for all i. Hence this graph represents exactly the dependency relationship that we have just described.

Consider the example in Figure 19.1. The most obvious way to reach node 6 would be

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and keep state 2
- calculate state 4 from state 3 and keep state 3 as well as still keeping state 2
- calculate state 5 from states 3 and 4 and erase states 3 and 4
- calculate state 6 from states 2 and 5 and erase states 2 and 5

Notice that when calculating state 5 we had in memory states 2,3 and 4, a total of three prior states. We can do better.

- calculate state 1 from the starting state
- calculate state 2 from state 1 and erase state 1
- calculate state 3 from state 2 and *erase* state 2
- calculate state 4 from state 3 and keep state 3
- calculate state 5 from states 3 and 4 and then erase both states 3 and 4
- calculate state 1 from the starting state (for the second time!)
- calculate state 2 from state 1 and erase state 1
- calculate state 6 from states 2 and 5 and then erase both

This second calculation required two more steps to compute but now we never needed to remember more than two previous states, rather than three. It is exactly this tradeoff of time versus space which enables us to achieve our goal.

In general on a directed acyclic graph of bounded degree we can play a pebble game. The rules of the game are simple:

1. A pebble may be placed on any node all of whose parents have pebbles.
2. A pebble may be removed from any node.
3. A pebble may be placed on any node having no parents. This is a special case of the first rule.

The goal of the pebble game is to pebble a given target node while minimizing the number of pebbles used at any one time.

This game corresponds to calculations of our Turing machine. A pebble on vertex j in the game corresponds to a saved value for $c(j)$. When the game tells us to pebble node j then we can recover the contents of $c(j)$ as follows:

1. Load contents of $c_1(l_1(j)), \ldots, c_k(l_k(j))$ from storage. Since there was an edge from $l_i(j) \to j$ in the graph, Rule 1 guarantees that node $l_i(j)$ has a pebble on it. Since $l_i(j)$ has a pebble in the pebble game, then in our computation we have $c(l_i(j)$ and therefore $c_i(l_i(j))$ in storage.
2. Load the starting state and head position for period j from $c(j-1)$, again guaranteed to be available by the pebble game.
3. Based on all the above, it is easy to reconstruct $c(j)$.

Note that in order to determine the answer of the original Turing machine, it suffices to reconstruct $c(t^{1/3})$. Our aim is to do so using $O(t/\log t)$ space, which can be reduced to pebbling the target node $t^{1/3}$ using $t^{1/3}/\log t$ pebbles. We first state the following general result regarding pebbling:

Theorem 19.4 (Pebble Game Theorem): *For any fixed d, any directed acyclic graph $G = (V, E)$ with in-degree bound d, and any $v \in V$, we can pebble v while never using more than $O(|V|/\log|V|)$ pebbles simultaneously.*

The proof is postponed to the next section. Notice, however, that the theorem does not state anything about how efficient it is to compute this pebbling.

Using the above result, we simulate the machine M on input x as follows.

1. Compute $t = t(|x|)$
2. Loop over all possible guesses of $h(1), \ldots, h(t^{1/3})$. For each one do:
   (a) Compute and store the dependency graph.
   (b) Execute a pebbling strategy to reach node $t^{1/3}$ on the graph as follows:
      i. Determine next pebble move
      ii. if the move is "Remove i" then erase $c(i)$
      iii. if the move is "Pebble i" then set the machine to the initial state for period i by loading the contents of the work tape from storage. Calculate $c(i)$. Verify that in step 2 we did guess $h(i+1)$ correctly. If not, then abort the whole calculation so far and return to the next guess in step 2. Otherwise save $c(i)$ for future use.
   (c) Having just executed "Pebble $t^{1/3}$", terminate and accept or reject according to the original Turing machine.

We need to show that all of the above computation can be performed within our space bound.

- Step 1 can be performed within our space bound by our hypothesis regarding the (space constructability of the) function $t(\cdot)$.
- In step 2a we store a graph $G = (V, E)$ where $|V| = t^{1/3}$ and $|E| \le |V| \cdot (k+1)$. A simple list representation of the graph will therefore take $(k+1) \cdot t^{1/3} \cdot \log_2(t^{1/3}) \ll t/\log t$ space.

...	$Rev(b_i(j-2))$	$Rev(b_i(j-1))$	$Rev(b_i(j))$	...
	$b_i(j-1)$	$b_i(j)$	$b_i(j+1)$	
	$Rev(b_i(j))$	$Rev(b_i(j+1))$	$Rev(b_i(j+2))$	

Figure 19.2: The new work tape

- Step 2(b)ii actually frees space rather than using space.
- Step 2(b)iii needs space only for k blocks copied from storage, and space to store the result. Since our pebble game guarantees that we will never need more than $t^{1/3}/\log t^{1/3}$ pebbles, we will never need more than $(t^{1/3}/\log t^{1/3}) \cdot (k+1) \cdot t^{2/3} = O(t/\log t)$ space, our space bound.

So aside from step 2(b)i all of step 2b can be calculated within our space bound.

We need to describe now how to perform step 2(b)i. Consider a non-deterministic algorithm. We set a counter to be the maximum number of possible pebble moves to reach our target. Since it never makes sense to return to an earlier pebbling configuration of the graph, we can bound the number of steps by $2^{t^{1/3}}$. Such a counter will only require $t^{1/3}$ space, which is within our bound. We now non-deterministically make our choice of valid pebbling move and decrement the counter. We accept if we pebble the target vertex, reject if the counter reaches its limit first. The dominant space required by the routine is therefore the space required to store a working copy of the graph, $O(t^{1/3} \cdot \log t)$.

In Lecture 5 we proved that $Nspace(s) \subseteq Dspace(s^2)$. Therefore, the above non-deterministic subprogram which runs in $Nspace(t^{1/3} \cdot \log t)$ can be converted to a deterministic subprogram running in $Dspace(t^{2/3} \cdot \log^2 t \ll t/\log t)$. ■

19.3 Additional Proofs

We now prove two results stated without proof in the previous section.

19.3.1 Proof of Lemma 19.2.1 (Canonical Computation Lemma)

Our new machine M' which simulates the operation of M works as follows. Firstly each work tape is replaced by a three-track work tape. Three additional work tapes are added:

- one uses a unary alphabet and is used as a counter
- the second is a k-track "copy" tape
- the last is a k-track binary "position" tape

After calculating the block bound $t^{2/3}$, store it in unary on the counter tape. Now mark the work tapes into blocks of length $t^{2/3}$ by putting a special end-of-block marker where needed (each block will thus be one cell larger to accommodate the end of cell marker). Let $b_i(j)$ and $b'_i(j)$ denote the contents of the j^{th} block of tape i in the original machine and new machine respectively. Then $b'_i(j)$ is the tuple $(Reverse(b_i(j-1)), b_i(j), Reverse(b_i(j+1)))$. Here $Reverse$ means reversing the order of the tape cells. When simulating the computation we start by working on the middle track. If we see the end-of-block marker when going left, then we start using the first track instead and reverse the directions of each head move (on this track). Similarly, if we go off the end of the block while going right we switch to using the last track and again reverse the direction of head moves. Throughout we keep moving the head on the counter tape until it indicates that we have reached the end of the period.

At the end of the period we have to do some housekeeping. First we save the state, either within the state itself or on a special tape somewhere; either way this is only a constant amount of information. We store the head position within each block on the "position" tape as follows. Scan left along all work tapes not moving past the beginning of the block. For all work tapes not at the beginning of the block write a 1 in the corresponding track of the "position" tape, and for all others write a 0. Continue in this way until every head position is at the start of block for all the work tapes; the number of 1's on track i of the "position" tape will thus equal the head position, within the block, on work tape i. This takes one additional time period.

Consider Figure 19.2. At this point only the center block is up to date; the values for $b_i(j-1)$ and $b_i(j)$ in the left block are 'stale' since we may have modified them in our last time period. Similarly the values for $b_i(j)$ and $b_i(j+1)$ in the right block are also 'stale'. We update these stale values by scanning back and forth over all three cells and using the "copy" tape as a temporary area to store the contents of one block (all three tracks) at a time. Altogether this process requires about 6 time periods.

Finally we return the heads to their correct positions, but this time if they were working on the first track of the working block we place them in the corresponding cell in the previous block, and if they were in the last track then we place them in the next block. This may require an additional time period or two. Altogether this simulation of a single time period has cost us a little bit of extra space (more work tapes) and a constant factor increase of about 14 in the number of periods. ∎

19.3.2 Proof of Theorem 19.4 (Pebble Game Theorem)

Denote by $R_d(p)$ the minimum number of edges in a di-graph of in-degree bound d which requires p pebbles (that is, at least one node in the graph needs p pebbles). We will first show that $R_d(p) = \Omega(p \log p)$ and then that this fact implies our theorem.

Consider a graph $G = (V, E)$ with the minimal possible number of edges requiring p pebbles. Let V_1 be the set of vertices which can be pebbled using at most $p/2$ pebbles. Let G_1 be the subgraph induced by V_1 having edge set $E_1 = E \cap (V_1 \times V_1)$. Let $V_2 = V - V_1$ and let G_2 be the subgraph induced by V_2 and having edge set $E_2 = E \cap (V_2 \times V_2)$. Let $A = E - E_1 - E_2$ be all edges between the two subgraphs. Notice that there cannot be any edge from V_2 to V_1 since then the vertex at the head of the edge (in V_1) would require at least as many pebbles as the vertex at its tail, which is more than $p/2$.

There exists $v \in V_2$ which requires $p/2 - d$ pebbles in G_2

Assume not. Then we show that we can pebble any node in G with fewer than p pebbles, contradicting the hypothesis regarding G. For any node $v \in V_2$ we have assumed that in G_2 we can pebble it with $< p/2 - d$ pebbles. Invoke the strategy to pebble v in the graph G_2 on the original graph G. Whenever we want to pebble a vertex $u \in G_2$ that has a parent in G_1, bring a pebble to the parent in G_1, using $\leq p/2$ pebbles. Since some u might have as many as d parents in G_1, we actually use as many as $d - 1 + p/2$ pebbles when pebbling the d^{th} parent in G_1 (using $d-1$ pebbles to cover the first $d-1$ parents and $p/2$ to do the actual pebbling). Once all the parents in G_1 are pebbled, pebble u and then lift the pebbles on all the vertices in G_1. Thus we can pebble our target v using at most $(p/2-d)+(p/2+d-1)=p-1$ pebbles. Since we can also pebble any $v \in V_1$ with at most $p/2 < p$ we can pebble any $v \in V$ with fewer than p pebbles, a contradiction to our original choice of G.

If v with in-degree k requires m pebbles then it has a parent needing $m-k$

We show that in general, for any $G=(V,E)$ where G is an acyclic di-graph, any node $v \in V$ requiring m pebbles, with an in-degree of k, has a parent u requiring at least $m-k+1$ pebbles. Suppose to the contrary that each parent needs $m-k$ or fewer pebbles. Then simply bring a pebble to each of them in arbitrary order, each time removing all pebbles not on parents of v. When bringing a pebble to the i^{th} parent we have $i-1$ pebbles covering the other parents and use at most $m-k$ pebbles for the pebbling itself. Thus at any time we use at most $m-k+i-1$ pebbles. Over all parents then, the maximum number of pebbles that we use is $m-k+k-1=m-1$ which may occur only when pebbling the k^{th} parent. Now, however, we have k pebbles on the parents of v and we can pebble v itself having used at most $m-1$ pebbles, a contradiction.

There exists $u \in G_1$ which requires $p/2-d$ pebbles (in both G and G_1)

Consider any $v \in G_2$. Repeatedly replace v by a parent in G_2 until you can go no further. Since G is acyclic this is guaranteed to stop. When it stops, since the new v requires, by virtue of being in G_2, more than $p/2 > 0$ pebbles, v has at least one parent in G_1 (and no parents in G_2). Let $m > p/2$ be the number of pebbles needed to pebble v and let k be the in-degree of v in the original graph G. By the above claim v has a parent which requires $m-k+1 > p/2-k+1$ pebbles. Futhermore, since $k \le d$ we see that v has a parent requiring at least $p/2-d+1$ pebbles. Therefore (as v has no parents in G_2) there is a vertex $u \in G_1$ requiring at least $p/2-d+1 > p/2-d$ pebbles.

$|E_2|+|A| \ge R_d(p/2-d)+\frac{p}{4d}$

Since G_2 requires at least $p/2-d$ pebbles, $|E_2| \ge R_d(p/2-d)$. If $|A| \ge \frac{p}{4} \ge \frac{p}{4d}$ then we are done.

Otherwise $|A| < \frac{p}{4}$. We will ignore $|A|$ and show that $|E_2| \ge R_d(p/2-d)+\frac{p}{4d}$. Pebble each of the $< p/4$ vertices in V_1 with children in V_2 in succession. Independently pebbling each would require at most $p/2$ pebbles. By pebbling one at a time we can pebble them all using at most $p/4-1+p/2=3p/4-1$ pebbles. When done we are left with less than $p/4$ pebbles on the graph, leaving more than $3p/4$ pebbles free. Since we know that there must exist a $v \in G_2$ requiring p pebbles in G then it must require at least $3p/4$ pebbles in G_2. Consider a vertex v with out-degree of 0 which requires $3p/4$ pebbles in G_2, and remove it. As proven in section 19.3.2 the resulting graph must still require at least $3p/4-d$ pebbles. Repeating this process i times, we have a graph requiring $3p/4-i \cdot d$ pebbles with at least i fewer edges. So for $i=\frac{p}{4d}$ times, we have a graph requiring at least $3p/4-p/4=p/2$ pebbles with $\frac{p}{4d}$ fewer edges. This subgraph will have at least $R_d(p/2) \ge R_d(p/2-d)$ edges. Together with the $\frac{p}{4d}$ that we removed we see that $|E_2|+|A| \ge |E_2| \ge R_d(p/2-d)+\frac{p}{4d}$ as required.

Putting it together

So $|E|=|E_1|+|E_2|+|A|$. By Section 19.3.2 we have $|E_1| \ge R_d(p/2-d)$. By Section 19.3.2 we have $|E_2|+|A| \ge R_d(p/2-d)+\frac{p}{4d}$. Therefore $R_d(p)=|E| \ge 2R_d(p/2-d)+\frac{p}{4d}$, where the equality is due to the hypothesis that G has minimum number of edges among grapohs requiring p pebbles.

To solve the recurrence notice that

$$R_d(p) \geq 2R_d(\frac{p}{2} - d) + \frac{p}{4d} \geq 2R_d(\frac{p}{2} - 2d) + \frac{p}{4d}$$

For any i we get

$$R_d(\frac{p}{2^i} - 2d) \geq 2R_d(\frac{\frac{p}{2^i} - 2d}{2} - d) + \frac{\frac{p}{2^i} - 2d}{4d} = 2R_d(\frac{p}{2^{i+1}} - 2d) + \frac{\frac{p}{2^i} - 2d}{4d}$$

So

$$R_d(p - 2d) \geq 2^i R_d(\frac{p}{2^i} - 2d) + \sum_{j=0}^{i-1} 2^j \frac{\left(\frac{p}{2^j} - 2d\right)}{4d} = 2^i R_d(\frac{p}{2^i} - 2d) + \sum_{j=0}^{i-1} \frac{p - 2^{j+1}d}{4d}$$

Setting $i = \log_2(p/2d)$ we get

$$\begin{aligned}
R_d(p) \geq R_d(p - 2d) &\geq 2^i R_d(0) + \sum_{j=0}^{\log_2(p/2d)-1} \frac{p - 2^{j+1}d}{4d} \\
&\geq \sum_{j=0}^{\log_2(p/2d)-1} \frac{p - 2^{\log_2(p/2d)}d}{4d} \\
&= \log_2(p/2d)\frac{p - \frac{p}{2d}d}{4d} \\
&= \log_2(p/2d)\frac{p/2}{4d} \\
&= \Omega(p \log p)
\end{aligned}$$

So, for some constant $c > 0$, $R_d(p) \geq c \cdot p \log p$.

Now consider our original question of how many pebbles one needs to pebble a graph of n vertices. With $p = kn/\log n$ pebbles, we can certainly pebble all graphs with less than $c \cdot p \log p$ edges. Now,

$$\begin{aligned}
cp \log p &= c\frac{kn}{\log n} \log \frac{kn}{\log n} \\
&= c\frac{kn}{\log n}(\log k + \log n - \log\log n) \\
&> ckn\left(1 - \frac{\log\log n}{\log n}\right) \\
&> ckn/2
\end{aligned}$$

for all sufficiently large n such that $\frac{\log\log n}{\log n} < 1/2$. Letting $k = \frac{2d}{c}$, we can pebble all graphs with less than $\frac{cn}{2} \cdot \frac{2d}{c} = dn$ edges, using $p = kn/logn$ pebbles. Since any graph on n vertices, with in-degree bound of d, has less than dn edges, it may be pebbled by $O(dn/\log n)$ pebbles. Since d is a constant, the theorem follows. ■

Bibliographic Notes

This lecture is based on [1].

1. J.E. Hopcroft, W. Paul, and L. Valiant. On time versus space. *J. of ACM*, Vol. 24, No. 2, pages 332–337, 1977.

Lecture 20

Circuit Depth and Space Complexity

Notes taken by Vered Rosen and Alon Rosen

Summary: In this lecture we study some of the relations between Boolean circuits and Turing machines. We define the complexity classes NC and AC, compare their computational power, and point out the possible connection between $uniform$-NC and "efficient" parallel computation. We conclude the discussion by establishing a strong connection between space complexity and depth of circuits with bounded fan-in.

20.1 Boolean Circuits

Loosely speaking, a *Boolean Circuit* is a directed acyclic graph with three types of labeled vertices: *inputs*, *outputs*, and *gates*. The inputs are sources in the graph (i.e. vertices with in-degree 0), and are labeled with either Boolean variables or constant Boolean values. The outputs are sinks in the graph (i.e. vertices with out-degree 0). The gates are vertices with in-degree $k > 0$, which are labeled with Boolean functions on k inputs. We refer to the in-degree of a vertex as its *fan-in* and its out-degree as its *fan-out*. A general definition of Boolean circuits would allow the labeling of gates with arbitrary Boolean functions. We restrict our attention to circuits whose gates are labeled with the boolean functions AND, OR, and NOT (denoted $\bigwedge, \bigvee, \neg$ respectively).

20.1.1 The Definition

Definition 20.1 (Boolean Circuit): *A* **Boolean Circuit** *is a directed acyclic graph with labeled vertices:*

- *The* input *vertices, labeled with a variable x_i or with a constant (0 or 1), and have fan-in 0.*
- *The* gate *vertices, have fan-in $k > 0$ and are labeled with one of the boolean functions $\bigwedge, \bigvee, \neg$ on k inputs (in the case that the label is $\neg$, the fan-in k is restricted to be 1).*
- *The* output *vertices, labeled 'output', and have fan-out 0.*

Given an assignment $\sigma \in \{0,1\}^m$ to the variables $x_1, ..., x_m$, $C(\sigma)$ will denote the value of the circuit's output. The value is defined in the natural manner, by setting the value of each vertex according to the boolean operation it is labeled with. For example, if a vertex is labeled $\bigwedge$ and the vertices with a directed edge to it have values a and b, then the vertex has value $a \bigwedge b$.

We denote by $size(C)$ the number of gates in a circuit C, and by $depth(C)$ the maximum distance from an input to an output (i.e. the longest directed path in the graph).

A circuit is said to have *bounded* fan-in if there exists an a-priori upper bound on the fan-in of its AND and OR gates (NOT gates, must have fan-in 1 anyway). If there is no a-priori bound on the fan-in (other than the size of the circuit), the circuit is said to have *unbounded* fan-in.

20.1.2 Some Observations

- We have already seen how to construct a circuit which simulates the run of a Turing machine M on some input $x \in \{0,1\}^n$ (see the proof of Cook's Theorem , Lecture 2). Using this construction the resulting circuit will be of size quadratic in $T_M(n)$ (the running time of M on input length n), and of depth bounded by $T_M(n)$.

- Circuits may be organized into disjoint layers of gates, where each layer consists of gates having equal distance from the input vertices. Once presented this way, a circuit may be thought of as capturing a certain notion of parallel computation. We could associate each path starting from an input vertex with a computation performed by a single processor. Note that all such computations can be performed concurrently. Viewed in this way, the circuit's depth corresponds to parallel time, whereas the size corresponds to the total amount of parallel work.

- Any bounded fan-in circuit can be transformed into a circuit whose gates have fan-in 2 while paying only a constant factor in its depth and size. A gate with constant fan-in c, can be converted into a binary tree of gates of the same type which computes the same function as the original gate. Since c is a constant so will be the tree's size and depth. We can therefore assume without loss of generality that all gates in bounded fan-in circuits have fan-in 2. Note that the same transformation in the case of unbounded fan-in will increase the depth by a factor logarithmic in the size of the circuit.

- Any circuit can be modified in such a way that all negations (i.e. $\neg$ gates) in it appear only in the input layer. Using De-Morgan's laws (that is, $\neg \bigvee x_i = \bigwedge \neg x_i$), each $\bigvee$ gate followed by a negation can be transformed into an $\bigwedge$ gate whose inputs are negations of the original inputs (the same argument applies symmetrically for $\bigwedge$ gates). This way, we can propagate all negations in the circuit towards the input layer without changing the value of the circuit's output, and without changing its depth. Thus, without loss of generality, all "internal" gates in a Boolean circuit are labeled with $\bigwedge$ or $\bigvee$. When measuring the depth of the circuit, negations will be counted as a single layer.

- In an unbounded fan-in circuit, two consecutive $\bigwedge$ (resp. $\bigvee$) gates having identical labels can be merged into a single gate with the same label without changing the value of the circuit's output. We can therefore assume that unbounded fan-in circuits are of the special form where all $\bigwedge$ and $\bigvee$ gates are organized into alternating layers with edges only between adjacent layers.

  Note, however, that the above argument does not necessarily work for bounded fan-in circuits since the merging operation might cause a blow-up in the fan-in of the resulting gates which will exceed the specified bound.

20.1.3 Families of Circuits

Eventhough a circuit may have arbitrarily many output vertices we will focus on circuits which have only one output vertex (unless otherwise specified). Such circuits can be used in a natural way to define a language (subset of $\{0,1\}^*$). Since we are interested in deciding instances of arbitrary size, we consider *families* of Boolean circuits with one different circuit for each input size.

Definition 20.2 (Family of Circuits): *A language $L \subseteq \{0,1\}^*$ is said to be decided by a* family of circuits, *$\{C_n\}$, when C_n takes n variables as inputs, if and only if for every n, $C_n(x) = \chi_L(x)$ for all $x \in \{0,1\}^n$.*

Given a family of circuits, we might be interested in measuring the growth-rate of the size and depth of its circuits (as a function of n). This may be useful when trying to connect circuit complexity with some other abstract model of computation.

Definition 20.3 (Depth and Size of Family) *Let D and S be sets of integer functions ($N \to N$), we say that a family of circuits $\{C_n\}$, has depth D and size S if for all n, $depth(C_n) \leq d(n)$ and $size(C_n) \leq s(n)$ for some $d(\cdot) \in D$ and $s(\cdot) \in S$.*

If we wish to correlate the size and depth of a family $\{C_n\}$ that decides a language L with its Turing machine complexity, it is necessary to introduce some notion of *uniformity*. Otherwise, we could construct a family of circuits which decides a non-recursive language (see Lecture 8). The notion of uniformity which we will make use of is *logspace uniformity*. Informally, we require that a description of a circuit can be obtained by invoking a Turing machine using space which is logarithmic in the length of its output (i.e. the circuit's description). A description of a circuit (denoted $desc(C_n)$), is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of $desc(C_n)$ is at most quadratic in $size(C_n)$.

Definition 20.4 (logspace uniformity): *A family of circuits, $\{C_n\}$, is called* logspace uniform *if there exists a deterministic Turing machine, M, such that for every n, $M(1^n) = desc(C_n)$ while using space which is logarithmic in the length of $desc(C_n)$.*

The reason we require M to work in space which is logarithmic in its *output* length (rather than its input length) lies in the fact that $size(C_n)$ (and therefore the length of $desc(C_n)$) might be super-polynomial in n. The problem is that the description of a circuit with super-polynomial size, *cannot* be produced by Turing machines working with space which is logarithmic in n (the input size), and therefore such a circuit (one with super-polynomial size) would have been overlooked by a definition using the input length as parameter. Note that if we restrict ourselves to circuits with polynomial size, then the above remark does not apply, and it is sufficient to require that M is a deterministic logspace Turing machine.

Based on the above definitions, the class $\mathcal{P}/poly$ is the class of all languages for which there exists a family of circuits having polynomial depth and polynomial size (see Lecture 8). We have already seen how to use the transformation from Turing machines into circuits in order to prove that $uniform$-$\mathcal{P}/poly$ contains (and in fact equals) $\mathcal{P}$. We note that the above transformation can be performed in logarithmic space as well. This means that $logspace$-$uniform$-$\mathcal{P}/poly$ also equals $\mathcal{P}$. In the sequel, we choose logspace uniformity to be our preferred notion of uniformity.

20.2 Small-depth circuits

In this section we consider polynomial size circuits whose depth is considerably smaller than n, the number of inputs. By depth considerably smaller than n we refer to *poly-logarithmic* depth, that is, bounded by $O(\log^k n)$ for some $k \geq 0$. We are interested in separating the cases of unbounded and bounded fan-in, specifically, we will investigate the relation between two main complexity classes. As we will see, these classes will eventually turn out to be subsets of $\mathcal{P}$ which capture the notion of what is "efficiently" computable by parallel algorithms.

20.2.1 The Classes NC and AC

The complexity class NC is defined as the class of languages that can be decided by families of bounded fan-in circuits with polynomial size, and poly-logarithmic depth (in the number of inputs to the circuits). The actual definition of NC introduces an hierarchy of classes decided by circuit families with increasing depth.

Definition 20.5 (NC): *For $k \geq 0$, define NC^k to be the class of languages that can be decided by families of circuits with* bounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define NC to be the union of all NC^k's (i.e. $NC \stackrel{\text{def}}{=} \bigcup_k NC^k$).*

A natural question would be to ask what happens to the computational power of the above circuits if we remove the bound on the fan-in. For instance, it is easy to see that the decision problem "is the input string $\sigma \in \{0,1\}^n$ made up of all 1's?" (the AND function) can be solved by a depth-1 unbounded fan-in circuit, whereas in a bounded fan-in circuit this problem would require depth at least $\Omega(\log n)$, just so all the input bits could *effect* the output gate. The classes based on the unbounded fan-in model (namely AC) are defined analogously to the classes in the NC hierarchy.

Definition 20.6 (AC): *For $k \geq 0$, define AC^k to be the class of languages that can be decided by families of circuits with* unbounded *fan-in, polynomial size, and depth $O(\log^k n)$. Define AC to be the union of all AC^k's (i.e. $AC \stackrel{\text{def}}{=} \bigcup_k AC^k$).*

As we will see in the sequel, AC equals NC. Note however, that such a result does not necessarily rule out a separation between the computational power of circuits with bounded and respectively unbounded fan-in. We have already seen that the class NC^0 of languages decided by constant depth circuits with bounded fan-in, is strictly contained in AC^0, its unbounded fan-in version. We are therefore motivated to look deeper into the NC^k and AC^k hierarchies and try to gain a better understanding of the relation between their corresponding levels.

Theorem 20.7 *For all $k \geq 0$,*

$$NC^k \subseteq AC^k \subseteq NC^{k+1}$$

Proof: The first inclusion is trivial, we turn directly to prove the second inclusion. Since any gate in an unbounded fan-in circuit of size poly(n) can have fan-in at most poly(n), each such gate can be converted into a tree of gates of the same type with fan-in 2, such that the output gate of the tree computes the same function as the original gate (since this transformation can be performed in logarithmic space, the claim will hold both for the uniform and the nonuniform settings). The resulting depth of the tree will be $\log(\text{poly}(n)) = O(\log n)$. By applying this transformation to each gate in an unbounded fan-in circuit of depth $O(\log^k n)$ we obtain a bounded fan-in circuit deciding the same language as the original one. The above transformation will cost us only a logarithmic factor in the depth and a polynomial factor in the size (i.e. the depth of the resulting circuit will be $O(\log^{k+1} n)$, and the size will remain polynomial in n). Thus, any language in AC^k is also in NC^{k+1}. ■

Corollary 20.8 $AC = NC$

In light of Theorem 20.7, it might be interesting to ask how far does the NC (resp. AC) hierarchy extend. Is it infinite, or does it collapse to some level? Even if a collapse seems unlikely, at present no argument is known to rule out this option. One thing which **is** known at present, is that AC^0 is strictly contained in NC^1.

Theorem 20.9 *$AC^0 \subset NC^1$ (strictly contained).*

Note that $uniform$-NC is contained in $\mathcal{P}$. Theorem 20.9 implies that $uniform$-AC^0 is *strictly* contained in $\mathcal{P}$. An interesting open question is to establish what is the exact relationship between $uniform$-NC and $\mathcal{P}$, it is not currently known whether both classes are equal or not (analogously to the $\mathcal{P}$ vs. $\mathcal{NP}$ problem). As a matter of fact, it is not even known how to separate $uniform$-NC^1 from $\mathcal{NP}$.

20.2.2 Sketch of the proof of $AC^0 \subset NC^1$

We prove the Theorem by showing that the decision problem "does the input string have an even number of 1's?" can be solved in NC^1 but **cannot** be solved in AC^0. In this context it will be more convenient to view circuits as computing functions rather than deciding languages.

Definition 20.10 (Parity): *Let $x \in \{0,1\}^n$, the function Parity is defined as:*

$$Parity(x_1, \ldots, x_n) \stackrel{\text{def}}{=} \sum_{i=1}^{n} x_i \ (mod\ 2)$$

Claim 20.2.1 *$Parity \in NC_1$.*

Proof: We construct a circuit computing $Parity$, using a binary tree of logarithmic depth where each gate is a xor gate. Each xor gate can then be replaced by the combination of 3 legal gates: $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$. This transformation increases the depth of the circuit by a factor of 2, and the size of the circuit by a factor of 3. Consequently, $Parity$ is computed by circuits of logarithmic depth and polynomial size, and is thus in NC^1. ■

Claim 20.2.2 *$Parity \notin AC_0$.*

In order to prove claim 20.2.2 we show that every constant depth circuit computing $Parity$ must have sub-exponential size (and therefore $Parity$ cannot belong to AC^0), more precisely:

Theorem 20.11 *For every constant d, a circuit computing $Parity$ on n inputs with depth d, must have size $exp(\Omega(n^{\frac{1}{d-1}}))$.*

Proof: **(sketch)**: The Theorem is proven by induction on d and proceeds as follows:

1. Prove that parity circuits of depth 2 must be of large size.
2. Prove that depth d parity circuits of small size can be converted to depth $d-1$ parity circuits of small size (thus contradicting the induction hypothesis).

The first step is relatively easy, whereas the second step (which is the heart of the Theorem) is by far more complicated. We will therefore give only the outline of it.

Base case ($d = 2$): Without loss of generality, we assume that the circuit given to us is an OR of ANDs (in case it is an AND of ORs the following arguments apply symmetrically). Then, any AND gate evaluating to "1" determines the value of the circuit. Consider now an AND gate (any gate in the intermediate layer). Note that all the variables $x_1, \ldots, x_n$ must be connected to that gate. Otherwise, assume there exists an i such that there is no edge going from x_i (and from $\neg x_i$) into the gate: Then, take an assignment to the variables going into the gate, causing it to evaluate to "1" (we assume the gate is not degenerate). Under this assignment, the circuit will output "1", regardless of the value of x_i, which is impossible.
Due to that, we can associate each AND gate with a certain assignment to the n variables (determined by the literals connected to that gate). We can say that this gate will evaluate to "1" iff the variables will have this assignment.
This argument shows, that there must be at least 2^{n-1} AND gates. Otherwise, there exists an assignment σ to $x_1, \ldots, x_n$, such that $Parity(\sigma) = 1$, of which there is no AND gate identified with σ. This means that the circuit evaluated on the assignment σ will output "0", in contradiction.

The induction step: The basic idea of this step lies in a lemma proven by Hastad. The lemma states that given a depth 2 circuit, say an AND of ORs, then if one gives random values to a randomly selected subset of the variables, then it is possible to write the resulting induced circuit as an OR of relatively few ANDs with very high probability. We now outline how use this lemma in order to carry on the induction step:

Given a circuit of depth d computing parity, we assign random values to some of its inputs (only a large randomly chosen subset of the inputs are preset, the rest stay as variables). Consequently, we obtain a simplified circuit that works on fewer variables. This circuit will still compute parity (or the negation of parity) of the remaining variables.

By virtue of the lemma, it is possible to interchange two adjacent levels (the ones closest to the input layer) of ANDs and ORs. Then, by merging the two now adjacent levels with the same connective, we decrease the depth of the circuit to $d-1$. This is done without increasing significantly the size of the circuit.

Let us now make formal what we mean by randomly fixing some variables.

Definition 20.12 (random restriction): *A* random restriction *with a parameter ϵ to the variables $x_1, \ldots, x_n$, treats every variable x_i (independently) in the following way:*

$$x_i = \begin{cases} w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 0 \\ w.p.\ \frac{1-\epsilon}{2} & set\ x_i \leftarrow 1 \\ w.p.\ \epsilon & leave\ it\ \text{"alive"} \end{cases}$$

Observe that the expected number of variables remaining is $m = n\epsilon$. Obviously, the smaller ϵ is the more we can simplify our circuit, but on the other hand we have fewer remaining variables.

In order to contradict the induction hypothesis, we would like the size of the transformed circuit (after doing a random restriction and decreasing its depth by 1) to be smaller than $exp(o(m^{\frac{1}{d-2}}))$. It will suffice to require that $n^{\frac{1}{d-1}} < m^{\frac{1}{d-2}}$, or alternatively, $m > n \cdot n^{\frac{-1}{d-1}}$. Thus, a wise choice of the parameter ϵ, would be $\epsilon = n^{\frac{-1}{d-1}}$. ∎

20.2.3 NC and Parallel Computation

We now turn to briefly examine the connection between the complexity class $uniform\text{-}NC$ and parallel computation. In particular, we consider the connection to the parallel random-access machine (PRAM), which can be viewed as a parallel version of the RAM. A RAM is a computing device (processor), consisting of a program, a finite set of registers, and an infinite memory (whose cells are of the same size as the registers). The program is a finite sequence of instructions which are executed sequentially, where at the execution of each instruction the processor reads and writes values to some of its registers or memory cells, as required by the instruction.

The PRAM consists of several independent sequential processors (i.e. RAMs), each having its own set of private registers. In addition there is an infinite shared memory accessible by all processors. In one unit time (of a clock common to all processors), each processor executes a single instruction, during which it can read and write to its private set of registers, or to the shared memory cells. PRAMs can be classified according to restrictions on global memory access. An Exclusive-Read Exclusive-Write (EREW) PRAM forbids simultaneous access to the same memory cell by different processors. A Concurrent-Read Exclusive-Write (CREW) PRAM allows simultaneous reads but no simultaneous writes, and a Concurrent-Read Concurrent-Write (CRCW) PRAM allows simultaneous reads and writes (in this case one has to define how concurrent writes are treated). Despite this variety of different PRAM models, it turns out that they do not differ very widely in their computational power.

In designing algorithms for parallel machines we obviously want to minimize the time required to perform the concurrent computation. In particular, we would like our parallel algorithms to be *dramatically* faster than our sequential ones. Improvements in the running time would be considered dramatic if we could achieve an exponential drop in the time required to solve a problem, say from polynomial to logarithmic (or at least poly-logarithmic). Denote by PRAM($t(\cdot), p(\cdot)$) the class of languages decidable by a PRAM machine working in parallel time $t(\cdot)$ and using $p(\cdot)$ processors, then we have the following:

Theorem 20.13 $uniform\text{-}NC$ = PRAM(polylog,poly)

Hence, the complexity class $uniform\text{-}NC$ captures the notion of what is "efficiently" computable by PRAM machines (just as the class $\mathcal{P}$ captures the notion of what is "efficiently" computable by RAM machines, which are equivalent to Turing machines). Note however, that the PRAM cannot be considered a physically realizable model, since, as the number of processors and the size of the global memory scales up, it quickly becomes impossible to provide a constant-length data path from any processor to any memory cell. Nevertheless, the PRAM has proved to be an extremely useful vehicle for studying the logical structure of parallel computation. Algorithms developed for other, more realistic models, are often based on algorithms originally designed for the PRAM. Moreover, a transformation from the PRAM model into some other more realistic model will cost us only a logarithmic factor in the parallel complexity.

Finally, we would like to note that the analogy of NC to parallel computation has some aspects missing. First of all, it ignores the issue of the communication between processors. As a matter of fact, it seems that in practice this is the overshadowing problem to make large scale parallel computation efficient (note that the PRAM model implicitly overcomes the communication issue since two processors can communicate in $O(1)$ just by writing a message on the memory). Another aspect missing is the fact that the division of NC into subclasses based on running times seems to obscure the real bottleneck for parallel computing, which is the number of processors required. An algorithm that requires n processors and $log^2 n$ running time is likely to be far more useful than one that requires n^2 processors and takes $\log n$, but the latter is in NC^1, the more restrictive (and hence presumably better) class.

20.3 On Circuit Depth and Space Complexity

In this section we point out a strong connection between depth of circuits with **bounded** fan-in and space complexity. It turns out that circuit depth and space complexity are *polynomially related.* In particular, we are able to prove that L (and in fact NL) falls within NC. For the sake of generality, we introduce two families of complexity classes, which can be thought of as a generalized version of NC. From now on, we assume that all circuits are uniform.

Definition 20.14 ($\mathcal{DEPTH}/\mathcal{SIZE}$): *Let d, s be integer functions. Define $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), s(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$ and size $s(\cdot)$.*

In particular, if we denote by poly the set of all integer functions bounded by a polynomial and by polylog the set of all integer functions bounded by a poly-logarithmic function (e.g. $f \in$ polylog iff $f(n) = O(\log^k n)$ for some $k \geq 0$), then using the above notation, the class NC can be viewed as $\mathcal{DEPTH}/\mathcal{SIZE}(\text{polylog}, \text{poly})$.

Definition 20.15 ($\mathcal{DEPTH}$): *Let d be an integer function. Define $\mathcal{DEPTH}(d(\cdot))$ to be the class of all languages that can be decided by a uniform family of* bounded *fan-in circuits with depth $d(\cdot)$.*

Clearly, $NC \subseteq \mathcal{DEPTH}(\text{polylog})$. Note that the size of circuits deciding the languages in $\mathcal{DEPTH}(d(\cdot))$ is not limited, except for the natural upper bound of $2^{d(\cdot)}$ (due to the bounded fan-in)[1]. However, circuits deciding languages in NC are of polynomial size. Therefore, the class $\mathcal{DEPTH}(\text{polylog})$ contains languages which potentially do not belong to NC. We are now ready to connect circuit depth and space complexity.

Theorem 20.16 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}/\mathcal{SIZE}(O(s^2), 2^{O(s)})$$

Proof: Given a non-deterministic $s(n)$-space Turing machine M, we construct a uniform family of circuits, $\{C_n\}$, of depth $O(s^2)$ and size $2^{O(s)}$ such that for every $x \in \{0,1\}^*$, $C_{|x|}(x) = M(x)$.

Consider the configuration graph, $G_{M,x}$, of M on input x (see Lecture 6). Recall that the vertices of the graph are all the possible configurations of M on input x, and a directed edge leads from one configuration to another if and only if they are possible consecutive configurations of a computation on x. In order to decide whether M accepts x it is enough to check whether there exists a directed path in $G_{M,x}$ leading from the initial configuration vertex to the accepting configuration vertex. The problem of deciding, given a graph and two of its vertices, whether there exists a directed path between them, is called the directed connectivity problem (denoted $CONN$, see also Lecture 6). It turns out that $CONN$ can be decided by circuits with poly-logarithmic depth. More precisely:

Claim 20.3.1 $CONN \in NC^2$

[1]Thus, an alternative way to define $\mathcal{DEPTH}(d(\cdot))$ would be $\mathcal{DEPTH}/\mathcal{SIZE}(d(\cdot), 2^{d(\cdot)})$.

Proof: Let G be a directed graph on n vertices and let A be the adjacency matrix corresponding to it; that is, A is a boolean matrix of size $n \times n$, and $A_{i,j} = 1$ iff there is a directed edge from vertex v_i to vertex v_j in G. Now, let B be $A + I$, i.e. add to A all the self loops. Consider now the boolean product of B with itself, defined as

$$B^2_{i,j} = \bigvee_{k=1}^{n} (B_{i,k} \wedge B_{k,j}) \tag{20.1}$$

The resulting matrix will satisfy that $B^2_{i,j} = 1$ if and only if there is a directed path of length 2 or less from v_i to v_j in G. Similarly, B^4's entries will denote the existence of paths in G of length up to 4, and so on. Using $\log n$ boolean multiplications we can compute the matrix B^n, which is the adjacency matrix of the transitive closure of A - containing the answers to all the possible connectivity questions (for every pair of vertices in G). Squaring a matrix of size $n \times n$ can be done in AC^0 (see Equation 20.1) and therefore in NC^1. Hence, computing B^n can be done via repeated squaring in NC^2. ■

The circuit we build is a composition of two circuits. The first circuit takes as input some $x \in \{0,1\}^n$, a description of M (which is of a constant length) and outputs the adjacency matrix of $G_{M,x}$. The second circuit takes as input the adjacency matrix of $G_{M,x}$ and decides whether there exists a directed path from the initial configuration vertex to the accepting configuration vertex (i.e. decides $CONN$ on $G_{M,x}$). We start by constructing the first circuit. Given x and the description of M we generate all the possible configurations of M on x (there are $2^{O(s)}$ such configurations, each is represented by $O(s)$ bits). Then, for each pair of configurations we can decide if there should be a directed edge between them (i.e. whether these are possible consecutive configurations). This is done basically by comparing the contents of the work tape in the two configurations, and requires depth $O(\log s)$ (note that the size of the resulting circuit will be $2^{O(s)}$). As for the second circuit, since $G_{M,x}$ is of size $2^{O(s)}$, we have (by Claim 20.3.1) that $CONN$ can be decided on $G_{M,x}$ in depth $O(s^2)$ and size $2^{O(s)}$. Overall, we obtain a circuit C_n of depth $O(s^2)$ and of size $2^{O(s)}$, such that $C_n(x) = M(x)$. ■

Corollary 20.17 $NL \subseteq NC^2$

Proof: Take $s(n) = \log n$, and get that NL can be decided by circuits of polynomial size and of depth $O(\log^2 n)$. ■

Using the fact that $\mathcal{DEPTH}/\mathcal{SIZE}(O(s^2), 2^{O(s)})$ is contained in $\mathcal{DEPTH}(O(s^2))$, we can conclude:

Corollary 20.18 *For every integer function $s(\cdot)$ which is at least logarithmic,*

$$\mathcal{NSPACE}(s) \subseteq \mathcal{DEPTH}(O(s^2))$$

We are now ready to establish a reverse connection between circuit depth and space complexity. This is done by proving a result which is close to being a converse to Corollary 20.18 (given that for every function $s(\cdot)$, $\mathcal{DSPACE}(s) \subseteq \mathcal{NSPACE}(s)$).

Theorem 20.19 *For every function $d(\cdot)$ which is at least logarithmic,*

$$\mathcal{DEPTH}(d) \subseteq \mathcal{DSPACE}(d)$$

Proof: Given a uniform family of circuits $\{C_n\}$ of depth $d(n)$, we construct a deterministic $d(n)$-space Turing machine M such that for every $x \in \{0,1\}^*$, $M(x) = C_{|x|}(x)$. Our algorithm will be the composition of two algorithms, each using $d(n)$ space. The following lemma states that the above composition will give us a $d(n)$-space algorithm, as required (this is a special case of the Composition lemma - the search version, see Lecture 6).

Lemma 20.3.2 *Let M_1 and M_2 be two $s(n)$-space Turing machines. Then, there exists an $s(n)$-space Turing machine M, that on input x outputs $M_2(M_1(x))$.*

Our algorithm is (given input $x \in \{0,1\}^n$):

1. Obtain a description of C_n.

2. Evaluate $C_n(x)$.

A description of a circuit is a list of its gates, where for each gate we specify its type and its list of predecessors. Note that the length of the description of C_n might be exponential in $d(n)$ (since the number of gates in C_n might be exponential in $d(n)$), therefore we must use Lemma 20.3.2. The following claims establish the Theorem:

Claim 20.3.3 *A description of C_n can be generated using $O(d(n))$ space.*

Claim 20.3.4 *Circuit evaluation for bounded fan-in circuits can be solved in space=O(circuit depth).*

Proof: (of Claim 20.3.3) By the uniformity of $\{C_n\}$, there exists a deterministic machine M such that $M(1^n) = desc(C_n)$, while using $\log(|desc(C_n)|)$ space. Since $|desc(C_n)| \leq 2^{O(d(n))}$, we get that M uses $O(d(n))$ space, as required. ■

Proof: (of Claim 20.3.4) Given a circuit C of depth d and an input x, we want to compute $C(x)$. Our implementation will be recursive. A natural approach would be to use the following algorithm:

Denote by $VALUE(C_x, v)$ the value of vertex v in the circuit C when assigning x to its inputs, where v is encoded in binary (i.e. using $\log(size(C)) = O(d)$ bits). Note that $VALUE(C_x,\text{'}output\text{'})$ is equal to the desired value, $C(x)$. The following procedure obtains $VALUE(C_x, v)$:

1. If v is a leaf then return the value assigned to it.
   Otherwise, let u and w be v's predecessors and **op** be v's label.

2. Compute recursively $\sigma \leftarrow VALUE(C_x, u)$ and $\tau \leftarrow VALUE(C_x, w)$.

3. Return σ **op** τ.

Notice that Step 2 in the algorithm requires two recursion calls. Since we are going only one level down each recursion call, one may hastily conclude that the space consumed by the algorithm is $2^{O(d)}$. Remember however, that we are dealing with space, this means that the space consumed in the first recursion call can be reused by the second one, and therefore the actual space consumption is $O(d^2)$ (since there are $O(d)$ levels in the recursion, and on each level we need to memorize the vertex name which is of length $O(d)$).

This is still not good enough, remember that our goal is to design an algorithm working in space $O(d)$. This will be done by representing the vertices of C in a different manner: Each vertex will be specified by a path reaching it from the output vertex. The output vertex will be represented by the empty string ϵ. Its left predecessor will be represented by "0", and its right predecessor by "1", the left predecessor of "1" will be represented by "10", and so on. Since there might be several paths reaching a vertex, it might have multiple names assigned to it, but this will not bother us.

Consequently, each vertex is represented by a bit string of length $O(d)$. Moreover, obtaining from a vertex name its predecessor's or successor's name is done simply by concatenation of a bit or deletion of the last bit. The following procedure computes $VALUE(C_x, \overline{path})$ in $O(d)$ space:

1. Check whether $\overline{path}$ defines a leaf. If it does, then return the value assigned to it. Otherwise, let **op** be the label of the corresponding vertex.
2. Compute recursively $\sigma \leftarrow VALUE(C_x, \overline{path} \circ 0)$ and $\tau \leftarrow VALUE(C_x, \overline{path} \circ 1)$.
3. Return σ **op** τ.

When computing this procedure, C_x will be written on the input tape and $\overline{path}$ will be written on the work tape. At each level of the recursion, $\overline{path}$ determines precisely all the previous recursion levels, so the space consumption of this algorithm is $O(d)$, as required. ■

■

Corollary 20.20 $NC^1 \subseteq L \subseteq NL \subseteq NC^2$

Proof: The first inclusion follows from Theorem 20.19, the second inclusion is trivial and the third inclusion is just Corollary 20.17. ■

Bibliographic Notes

This lecture is mostly based on [1]. For wider perspective see Cook's old survey [2].

1. A. Borodin. On relating time and space to size and depth. *SIAM J. on Computing*, Vol. 6, No. 4, pages 733–744, 1977.
2. S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, Vol. 64, pages 2–22, 1985.

Lecture 21

Communication Complexity

Lecture given by Ran Raz
Notes taken by Amiel Ferman and Noam Sadot

Summary: This lecture deals with Communication Complexity, which is the analysis of the amount of information that needs to be communicated by two parties which are interested in reaching a common computational goal. We start with some basic definitions and simple examples. We continue to consider both deterministic and probabilistic models for the problem, and then we develop a combinatorial tool to help us with the proofs of lower bounds for communication problems. We conclude by proving a probabilistic linear communication complexity lower bound for the problem of computing the inner product of two vectors where initially each party holds one vector.

21.1 Introduction

The communication problem arises when two or more parties (i.e. processes, systems etcetera) need to carry out a task which could not be carried out alone by each of them because of lack of information. Thus, in order to achieve some common goal, defined as a function of their inputs, the parties need to communicate. Often, the formulation of a problem as a communication problem serves merely as a convenient abstraction; for example, a task that needs to share information between diefferent parts of the same CPU could be formulated as such. Communication complexity is concerned with analaysing the amount of information that must be communicated between the different parties in order to correctly perform the intended task.

21.2 Basic model and some examples

In order to investigate the general problem of communication we state a few simplified assumptions on our model:

1. There are only two parties (called player 1 and player 2)
2. Each party has unlimited computing power and we are only concerned with the communication complexity
3. The task is a computation of a predefined function of the input

As we shall see, this model is rich enough to study some non-trivial and interesting aspects of communication complexity.

The input domains of player 1 and player 2 are the (finite) sets X and Y respectively. The two players start with inputs $x \in X$ and $y \in Y$, and their task is to compute some predefined function $f(x, y)$. At each step, the communication protocol specifies which bit is sent by one of the players (alternately), and this is based on information communicated so far as well as on the initial inputs of the players.

Let us see a few examples:

1. Equality Function (denoted EQ):

   The function $f(x, y)$ is defined as:

   $f(x, y) = 1$ if $x = y$

   $f(x, y) = 0$ if $x \neq y$

   That is, the two players are interested to know wheather their initial inputs are equal.

2. Disjointness:

   The inputs are subsets: $x, y \subseteq \{1, \ldots, n\}$

   $f(x, y) = 1$ iff $x \cap y \neq \emptyset$

3. Inner Product (denoted IP):

   The inputs are: $x, y \in \{0, 1\}^n$

   $f(x, y) = \sum_i^n x_i \cdot y_i \bmod 2$

21.3 Deterministic versus Probabilistic Complexity

We begin with some definitions:

Definition 21.1 *A deterministic protocol P with domain $X \times Y$ and with range Z (where X and Y are the input domains of player 1 and player 2 respectively and Z is the domain of the function f) is defined as a deterministic algorithm in which at each step it specifies a bit to be sent from one of the players to the other. The output of the algorithm, denoted $P(x, y)$ (on inputs x and y), is the output of each of the players at the end of the protocol and it is required that:*

$$\forall x \in X, y \in Y \qquad : \qquad P(x, y) = f(x, y)$$

Definition 21.2 *The communication complexity of a deterministic protocol is the worst case number of bits sent by the protocol for some inputs.*

Definition 21.3 *The communication complexity of a function f is the minimum complexity of all deterministic protocols which compute f. This is denoted by $CC(f)$.*

A natural relaxation of the above defined deterministic protocol would be to allow each player to toss coins during his computation. This means that each player has an access to a random string and the protocol that is carried out depends on this string. The way to formulate this is to determine a distribution Π from which the random strings each player uses are sampled uniformly, once the strings are chosen, the protocol that is carried out by each of the players is completely deterministic. We consider the Monte-Carlo model, that is, the protocol should be correct for a large fraction of the strings in Π.

Note that the above description of a randomized protocol implicitly allows for two kinds of possibilities: one in which the string that is initially sampled from Π is common to both players, and the other is that each player initially samples his own private string so that the string sampled by one player is not visible to the other player. These two possibilites are called respectively the public and the private model. How are these two models related ? First of all, it is clear that any private protocol can be simulated as a public protocol: the strings sampled privately by each user are concatenated and serve as the public string. It turns out that a weaker reduction exists in the other direction: any public protocol can be simulated as a private protocol with a small increase in the error and an additive of $O(\log n)$ bits of communication; the idea of the proof is to show that any public protocol can be transformed to a protocol which uses the same amount of communication bits but only $O(\log n + \log \delta^{-1})$ random bits with an increase of δ in the error. Next, each player can sample a string of that length and send it to the other player thus causing an increase of $O(\log n + \log \delta^{-1})$ in the communication complexity and of δ in the error. In view of these results we shall confine ourselves to the public model.

Definition 21.4 *A randomized protocol P is defined as an algorithm which initially samples uniformly a string from some distribution Π and then carries on exactly as in the deterministic case. The sampled string is common to both player, i.e. - this is the public model. It is required that an ϵ - error protocol will satisfy:*

$$\forall x \in X, y \in Y \qquad Pr_{r \in_R \Pi}[P(x,y) = f(x,y)] \geq 1 - \epsilon$$

Note that in a randomized protocol, the number of bits communicated may vary for the same input (due to different random strings). Hence the communication complexity is defined with respect to the strings sampled from Π. One can define the communication complexity of a protocol, viewed as a random variable (with respect to the distribution Π, and the worst possible input), in the average case. However we prefer the somewhat stronger worst-case behaviour:

Definition 21.5 *The communication complexity of a randomized protocol P on input (x, y) is the maximum number of bits communicated in the protocol for any choice of initial random strings of each player. The communication complexity of a randomized protocol P is the maximum communication complexity of P over all possible inputs (x, y)*

Definition 21.6 *The communication complexity of a function f computed with error probability ϵ, denoted $CC_\epsilon(f)$ is the minimum communication complexity of a protocol P which computes f with error probability ϵ.*

In Lecture 3 we have actually considered a private case of the above definition, one in which there is no error; i.e., $CC_0(f)$.

21.4 Equality revisited and the Input Matrix

Recall the Equality Function defined in Section 2, in which both players wish to know wheather their inputs (which are n-bit strings) are equal (i.e. whether $x = y$). Let us first present a randomized protocol which computes EQ with a constant error probability and a constant communication complexity:

Protocol for player i (i = 1,2) ($input_1 = x$ and $input_2 = y$):

1. sample uniformly an n-bit string r (this r is common to both players - public model)
2. compute $< input_i, r >_2$ (the inner product of $input_i$ and r mod 2)
3. send the product computed and receive the product computed by the other player (single bit)
4. if the two bits are equal then output 1 else output 0

If the inputs are equal, i.e. $x = y$, then clearly $< x, r >_2 = < y, r >_2$ for all r-s and thus each player will receive and send the same bit and will decide 1. However, if $x \neq y$, then for a string r sampled uniformly we have that $< x, r >_2 = < y, r >_2$ with probability exactly one half. Thus, the error probability of a single iteration of the above protocol is exactly one half. Since at each iteration we sample a random string r independantly from other iterations we get that after carrying out the protocol for exactly C times the error probability is exactly 2^{-C}. Furthermore, since the number of bits communicated in each iteration is constant (exactly two bits), we get that after C iterations of the above protocol, the communication complexity is $O(C)$. Hence, if C is a constant, we get both a constant error probability and a constant communication complexity (2^{-c} and $O(1)$ respectively for a constant c). However, if we choose C to be equal to $log(n)$ then the error probability and the communication complexity will be, respectively, $\frac{1}{n}$ and $O(log(n))$.

We now present an alternative protocol for solving EQ that also achieves an error probabilty of $\frac{1}{n}$ and communication complexity of $O(log(n))$. Interestingly, this protocol is (already) in the private model:

We present both (n-bit strings) inputs as the coefficients of polynomials over $GF(p)$ where p is an arbitrary fixed prime between n^2 and $2n^2$ (results in number theory guarantee the existence of such a prime). So, both inputs may be viewed as:

input of player 1:

$$A(x) = \sum_{i=0}^{n-1} a_i \cdot x^i \bmod p$$

input of player 2:

$$B(x) = \sum_{i=0}^{n-1} b_i \cdot x^i \bmod p$$

Protocol for player 1 (For player 2: just reverse A and B)

1. choose uniformly a number t in $GF(p)$
2. compute $A(t)$
3. send both t and $A(t)$ to the other player
4. receive s and $B(s)$ from other player
5. if $A(s) = S$ then decide 1 else decide 0

Clearly, if the inputs are equal then so are the polynomials and thus necessarily $A(t) = B(t)$ for every $t \in GF(p)$. If however, $A \neq B$, then these polynomials have at most $n - 1$ points on which they agree (i.e. t-s for which $A(t) = B(t)$) since their difference is a polynomial of degree

$n-1$ which can have at most $n-1$ roots. So the probability of error in this case is $\frac{n-1}{p} \leq \frac{n}{n^2} = \frac{1}{n}$. Notice that since t and $B(t)$ are $O(lgn)$ bits long, we may conclude that $CC_{\frac{1}{n}}(EQ) = O(lgn)$.

Proofs of lower bounds which relate to certain families of algorithms usually necessitate a formalization that could express in a non-trivial way the underlying structure. In our case, a combinatorial view proves to be effective: (Recall that the input domains of both parties are denoted X and Y) we may view the protocol as a process which in each of its steps partitions the input space $X \times Y$ into disjoint sets such that at step t each set includes exactly all input pairs which according to their first t bits cause the protocol to "act" the same (i.e., communicate exactly the same messages during the algorithm). Intuitively, each set at the end of this partioning process is comprised of exactly all pairs of inputs that "cause" the protocol to reach the same conclusion (i.e., compute the same output).

A nice way to visualize this is to use a matrix: each row corresponds to a $y \in Y$ and each column corresponds to a $x \in X$. The value of the matrix in position (i,j) is simply $f(i,j)$, where f is the function both parties need to compute. This matrix is called the Input Matrix. Since the Input Matrix is just another way to describe the function f,we may choose to talk about the communication complexity of an Input Matrix A - dentoed $CC(A)$ instead of the communication complexity of the corresponding function f. For example, the matrix corresponding to the Equality Function is the indentity matrix (since the output of the Equality Function must be 1 iff the inputs are of the form (i,i) for each input pair). The above mentioned partioning process can now be viewed as a partioning of the matrix into sets of matrix elements. It turns out that these sets have a special structure, namely rectangles. Formally, we define

Definition 21.7 *A* rectangle *in $X \times Y$ is a subset $R \subseteq X \times Y$ such that $R = A \times B$ for some $A \subseteq X$ and $B \subseteq Y$.* (Note that elements of the rectangle, as defined above, need not be adjacent in the input matrix.)

However, in order to relate our discussion to this definition we need an alternative characterization of rectangles given in the next proposition:

Proposition 21.4.1 *$R \subseteq X \times Y$ is a rectangle iff $(x_1,y_1) \in R$ and $(x_2,y_2) \in R \;\Rightarrow\; (x_1,y_2) \in R$*

Proof: $\Rightarrow$ If $R = A \times B$ is a rectangle then from $(x_1,y_1) \in R$ we get that $x_1 \in A$ and from $(x_2,y_2) \in B$ we get that $y_2 \in B$ and so we get that $(x_1,y_2) \in A \times B = R$.

$\Leftarrow$ We define the sets $A = \{x | \exists y \ s.t. \ (x,y) \in R\}$ and $B = \{y | \exists x \ s.t. \ (x,y) \in R\}$. On the one hand it is clear that $R \subseteq A \times B$ (directly from A and B's definition). On the other hand, suppose $(x,y) \in A \times B$. Then since $x \in A$ there is a y' such that $(x,y') \in R$ and similiarly there is an x' such that $(x',y) \in R$ from this, according to the assumption we have that $(x,y) \in R$. ■

We shall now show that the sets of matrix elements partitioned by the protocol in the sense described above actually form a partition of the matrix into rectangles: Suppose both pairs of inputs (x_1,y_1) and (x_2,y_2) cause the protocol to exchange the same sequence of messages. Since the first player (with input x_1) cannot distinguish at each step between (x_1,y_1) and (x_1,y_2) (he computes a function of x_1 and the messages so far in any case) then he will communicate the same message to player 2 in both cases. Similarly, player 2 cannot distinguish at each step between (x_2,y_2) and (x_1,y_2) and will act the same in both cases. We showed that if the protocol acts the same on inputs (x_1,y_1) and (x_2,y_2) then it will act the same on input (x_1,y_2), which, using proposition 21.4.1 establishes the fact that the set of inputs on which the proctocol behaves the same is a rectangle.

Since the communication is the same during the protocol for the pair of inputs (x_1, y_1) and (x_2, y_2) (and for the pairs of inputs in the rectangle defined by them, as was explained in the last paragraph) then the protcol's output must be the same for these pairs, and this implies that the value of the f function must be the same too. Thus, a deterministic protocol partitions the Input Matrix into rectangles whose elements are identical, that is, the protocol computes the same output for each pair of inputs in the rectangle. We say that a deterministic protocol partitions the Input Matrix into rectangles of monochromatic rectangles (where color is identified with the input matrix value). Since at each step the protocol partitions the Input Matrix into two (usually not equal in size) parts we have the following:

Fact 21.4.2 *A deterministic protocol P of communication complexity k partitions the Input Matrix into at most 2^k monochromatic rectangles*

Recalling the fact that the Input Matrix of a protocol to the equality problem is the identity matrix, then since the smallest monochromatic rectangle that contains each entry of 1 in the matrix is the singleton matrix which contains exactly one element and since the matrix is of size $2^n \times 2^n$ (for inputs of size n), we get that every protocol for the equality problem must have partitioned the Input Matrix into at least $2^n + 1$ monochromatic rectangles (2^n for the 1's and at least 1 for the zeros). Thus, from Fact 21.4.2 and from the trivial protocol for solving EQ in which player 1 sends its input to the player 2 and player 2 sends to player 1 the bit 1 iff the inputs are equal ($n+1$ bits of communication), we get the following corollary:

Corollary 21.8 $CC(EQ) = n + 1$

21.5 Rank Lower Bound

Using the notion of an Input Matrix developed in the previous section, we now state and prove a useful theorem regarding the lower bound of communication complexity:

Theorem 21.9 *Let A be an Input Matrix for a certain function f, then $CC(A) \geq log_2(r_A)$ where r_A is the rank of A over any fixed field F*

Proof: The proof is by induction on $CC(A)$.

Induction Base: If $CC(A) = 0$ then this means that both sides were able to compute the function f without any communication. This means that for every pair of inputs, both sides compute a constant function which could be either $f(x, y) = 1$ or $f(x, y) = 0$ for all x and y. This implies that A must be the all 0-s or the all 1-s matrix, and so, by definition $r_A \in 0, 1$. Thus, indeed, $CC(A) \geq log_2(r_A)$ as required.

Induction Step: Suppose the claim is true for $CC(A) \leq n - 1$ and we shall prove the claim for $CC(A) = n$. Consider the first bit sent: This bit actually partitions A into two matrices A_0 and A_1 such that the rest of the protocol can be seen as a protcol that relates to only one of these matrices. Since the maximal communication complexity needed for both matrices cannot surpass $n - 1$ (otherwise $CC(A)$ could not have been equal to n), we get the following equation:

$$CC(A) \geq 1 + Max\{CC(A_0), CC(A_1)\} \geq 1 + Max\{log_2(r_{A_0}), log_2(r_{A_1})\} \tag{21.1}$$

where the second inequality is by the induction hypothesis. Now, since $r_A \leq r_{A_0} + r_{A_1}$, we have that $r_A \leq 2 \cdot Max\{r_{A_0}, r_{A_1}\}$. Put differently, we have $Max\{log_2(r_{A_0}), log_2(r_{A_1})\} \geq log_2(r_A) - 1$. Combining this with Eq. (21.1) we get that $CC(A) \geq 1 + log_2(r_A) - 1 = log_2(r_A)$. ■

Applying this theorem to the Input Matrix of the equality problem (the identity matrix), we easily get the lower bound $CC(EQ) \geq n$. A linear lower bound for the deterministic communication complexity of the Inner Product problem can also be achieved by applying this theorem. In the next section we'll see a linear lower bound on the randomized communication complexity of the Inner Product function.

21.6 Inner-Product lower bound

Recalling Inner-Product problem from section 21.2, we prove the following result:

Theorem 21.10 $CC_\epsilon(IP) = \Omega(n)$

To simplify the proof, and the mathematical techniques needed for the proof, we will assume that $0 < \epsilon < (\frac{1}{4} - \tau)$, for arbitarary small $\tau > 0$.

To prove the above theorem, we assume that there is a probabilistic communication protocol P in the public coin model using random string R that uses less than $n\epsilon$ communication bits, and we will show a contradiction. By definition, we know that

$$\mathrm{Pr}_R[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

for every string x and y. Since it is true for each pair of strings, the following property is true:

$$\mathrm{Pr}_{(x,y),R}[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

in which the probability measure over (x, y) is taken as the uniform probability over all such pairs. Changing the order of the probability measures, we obtain:

$$\mathrm{Pr}_{R,(x,y)}[P_R(x,y) = f(x,y)] \geq 1 - \epsilon$$

And since $\epsilon < 1$, we conclude that there is a fixed random string r that for the deterministic protocol induced by P_r, the following property exists:

$$\mathrm{Pr}_{(x,y)}[P_r(x,y) = f(x,y)] \geq 1 - \epsilon$$

By this method, we produce a deterministic protocol on which we can work now and prove lower bound. In contrast to the previous section, the protocol P_r should work well only for most of the inputs, but not necessarily for all of them. Proving lower bound on this deterministic protocol P_r will immediately give the lower bound of the original randomized protocol P.

The method for proving the lower bound is to show that there are not "big" enough rectangles that are not balanced in the input matrix after $n\epsilon$ time, and hence to conclude that we need more than that time. The following definition will be helpful:

Definition 21.11 *A rectangle $U \times V \subset \{0,1\}^n \times \{0,1\}^n$ is* big *if its size satisfies $|U \times V| \geq 2^{2n(1-\epsilon)}$. Otherwise, the rectangle is* small.

Note that there must be at least one big rectangle in the above matrix. Otherwise, using Fact 21.4.2 and the fact that we have at most $2^{n\epsilon}$ rectangles (for at most $n\epsilon$ communiaction), we infer that the size of the entire matrix is not more than $2^{n\epsilon} \cdot 2^{2n(1-\epsilon)} = 2^{2n(1-\frac{\epsilon}{2})} < 2^{2n}$ which leads to a contradiction.

Claim 21.6.1 *If P_r works in $n\epsilon$ time then there exists a big rectangle $U \times V \subset \{0,1\}^n \times \{0,1\}^n$ such that $f(x,y)$ is the same for at least $1-2\epsilon$ fraction of the elements in the rectangle $U \times V$.*

Proof: To prove the claim, we recall that $\Pr_{(x,y)}[P_r(x,y) = f(x,y)] \geq 1-\epsilon$. In other words, at most ϵ fraction of the elements in the matrix do not satisfy $P_r(x,y) = f(x,y)$. In addition, after at most $n\epsilon$ time, we will have a partition of the matrix into at most $2^{n\epsilon}$ rectangles. By Definition 21.11, each small rectangle has size less than $2^{2n(1-\epsilon)}$, and so the number of the elements that belong to small rectangles is less than $2^{n\epsilon} \cdot 2^{2n(1-\epsilon)} = 2^{2n(1-\frac{\epsilon}{2})} < 2^{2n-1}$, and so big rectangles contain more than half of the matrix elements. Thus, if all big rectangles have more than 2ϵ error, the total error due only to these rectangles would be more than ϵ, which leads to a contradiction. The claim follows. ■

By using this claim, we fix a big rectangle R that satisfies the conditions of the previous claim. Without loss of generality, we can assume that the majority of this rectangle is 0. If it is not 0, we just switch every element in the matrix.

Let us denote by B_n the $2^n \times 2^n$ input matrix for Inner Product problem, which looks like that:

$$B_n = \left\{ x \cdot y \bmod 2 \right\}_{(x,y)\in\{0,1\}^n\times\{0,1\}^n}$$

The inner elements are scalar products on the field $GF(2)$. The matrix B_n contains two types of elements: zeroes and ones. By switching each 0 into 1 and each 1 into -1, we get a new matrix H_n (which is a version of Hadamard matrix) that looks like that:

$$H_n = \left\{ (-1)^{x\cdot y} \right\}_{(x,y)\in\{0,1\}^n\times\{0,1\}^n}$$

This matrix has the following property:

Claim 21.6.2 *H_n is an orthogonal matrix over the reals.*

Proof: We will prove that each two rows in the matrix H_n are orthogonal. Let r_x be a row corresponds to a string x and r_z be a row corresponds to a string $z \neq x$. The scalar product between these two rows is

$$\sum_{y\in\{0,1\}^n} (-1)^{x\cdot y} \cdot (-1)^{z\cdot y} =$$

$$\sum_{y\in\{0,1\}^n} (-1)^{\Delta\cdot y}$$

where $\Delta = x \oplus z$. Since $x \neq z$ there is an index $j \in \{1,\ldots,n\}$ such that $\Delta_j = 1$, then the previous expression is equal

$$\sum_{y\in\{0,1\}^n} (-1)^{\sum_{i\neq j} y_i\Delta_i + y_j} \tag{21.2}$$

Let us denote by $y' = y_1 \ldots y_{j-1}y_{j+1}\ldots y_n$, then we can write (21.2) as:

$$\sum_{y'}\sum_{y_j}(-1)^{\sum_{i\neq j} y_i\Delta_i + y_j} =$$

$$\sum_{y'}(-1)^{\sum_{i\neq j} y_i\Delta_i}\sum_{y_j}(-1)^{y_j}$$

Clearly,

$$\sum_{y_j \in \{0,1\}} (-1)^{y_j} = -1 + 1 = 0$$

which proves the claim. ■

We have 2^n rows and columns in the matrix H_n. Let us enumerate the rows by r_i, for $i = 0, 1, \ldots, 2^n - 1$. Then by the previous claim, we have the following properties, where here $\cdot$ denotes inner product over the reals:

1. $r_i \cdot r_j = 0$ for $i \neq j$
2. $r_i \cdot r_i = \|r_i\|^2 = 2^n$ for $i = 0, 1, \ldots, 2^n - 1$. This follows easily from the fact that the absolute value of each element in H_n is 1.

Thus, the rows in the matrix define an orthogonal base over the reals.

The following definition will be helpful in the construction of the proof of Theorem 21.10:

Definition 21.12 (discrepency): *The* discrepency *of a rectangle $U \times V$ is defined as*

$$D(U \times V) = \left| \sum_{(x,y) \in U \times V} (-1)^{f(x,y)} \right|$$

Let R_0 be a big rectangle (of small error) as guaranteed by Claim 21.6.1. Suppose without loss of generality that R_0 has a majority of zeros (i.e., at least $1 - 2\epsilon$ fraction of 0's). Recall that the size of R_0 is at least $2^{2n(1-\epsilon)}$. Thus, R_0 has a big discrepency; that is,

$$D(R_0) \geq (1 - 2\epsilon - 2\epsilon) \cdot 2^{2n(1-\epsilon)} = (1 - 4\epsilon) \cdot 2^{2n(1-\epsilon)} \tag{21.3}$$

On the other hand, we have an upper bound on the discrepency of any rectangle (an in particular of R_0):

Lemma 21.6.3 *The discrepency of any rectangle R is bounded from above by $2^{-\frac{n}{2}} \cdot 2^{2n}$*

Proof: Let us denote $R = U \times V$. The matrix H_n has the property that each bit b in B_n changes into $(-1)^b$ in H_n. Let us consider the following charactaristic vector $I_U : \{0,1\}^n \to \{0,1\}^n$ that is defined in the following way:

$$I_U(x) = \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise} \end{cases}$$

Observe that $I_U \cdot r_j$ is exactly the number of 1's minus the number (-1)'s in r_j, so

$$\begin{aligned} D(U \times V) &= \left| \sum_{j \in V} I_U \cdot r_j \right| \leq \sum_{j \in V} |I_U \cdot r_j| \\ &\leq \sum_{j \in \{0,1\}^n} |I_U \cdot r_j| \end{aligned}$$

where both inequalities are trivial. Using Cauchy-Schwartz inequality (for the second line), we obtain

$$\begin{aligned} D(R) &\leq \sum_{j \in \{0,1\}^n} 1 \cdot |I_U \cdot r_j| \\ &\leq \sqrt{2^n \cdot \sum_{j \in \{0,1\}^n} |I_U \cdot r_j|^2} \end{aligned} \tag{21.4}$$

Recalling that H_n is an orthogonal matrix, and the norm of each row is $\sqrt{2^n}$, we denote $\hat{r_j} = \frac{1}{\sqrt{2^n}} r_j$ which define an orthonormal base. With this notation, Eq. (21.4) can be written as:

$$\begin{aligned}\sqrt{2^n \cdot \sum_{j\in\{0,1\}^n} |I_U \cdot \sqrt{2^n}\hat{r_j}|^2} &= \sqrt{2^n \cdot 2^n \cdot \sum_{j\in\{0,1\}^n} |I_U \cdot \hat{r_j}|^2} \\ &= 2^n \cdot \sqrt{\sum_{j\in\{0,1\}^n} |I_U \cdot \hat{r_j}|^2}\end{aligned}$$

Since $\{\hat{r_j}\}_{j=0,1,\ldots,2^n-1}$ is an orthonormal base, the square root above is merely the norm of I_U (as the norm is invariant over all orthonormal bases). However, looking at the "standard" (point-wise) base, we have that the norm of I_U is $\sqrt{|U|} \leq \sqrt{2^n}$ (since each element in the vector I_U is 0 or 1). To conclude, we got that:

$$D(R) \leq 2^n \cdot \sqrt{2^n} = 2^{\frac{3n}{2}} \tag{21.5}$$

which proves the lemma. ■

We now derive a contradiction by contrasting the upper and lower bounds provided for R_0. By Eq. (21.3), we got that:

$$D(R_0) \geq (1-4\epsilon) \cdot 2^{2n(1-\epsilon)}$$

which is greater than $2^{3n/2}$ for any $0 < \epsilon < \frac{1}{4}$ and all sufficiently large n's (since for such ϵ the exponent is strictly bigger than $3n/2$ which for sufficiently big n's compensates for the small positive factor $1-4\epsilon$). In contrast, Lemma 21.6.3 applies also to R_0 and implies that $D(R_0) \leq 2^{3n/2}$, in contradiction to the above bound (of $D(R_0) > 2^{3n/2}$).

To conclude, we showed a contradiction to our initial hypothesis that the comminication complexity is lower than ϵn. Theorem 21.10 thus follows.

Bibliographic Notes

For further discussion of Communication Complexity see the textbook [2]. Specifically, this lecture corresponds to Chapters 1 and 3.

Communication Complexity was first defined and studied by Yao [4], who also introduced the "rectangle-based" proof technique. The rank lower bound was suggested in [3]. The lower bound on the communication complexity of the Inner Product function is due to [1].

1. B. Chor and O. Goldreich. Unbiased Bits From Sources of Weak Randomness and Probabilistic Communication Complexity. *SIAM J. Comp.*, Vol. 17, No. 2, April 1988, pp. 230–261.

2. E. Kushilevitz and N. Nisan. *Communication Complexity*, Cambridge University Press, 1996.

3. K. Mehlhorn and E. Schmidt. Las-Vegas is better than Determinism in VLSI and Distributed Computing. In *Proc. of 14th STOC,* pp. 330-337, 1982.

4. A.C. Yao. Some Complexity Questions Related to Distributive Computing. In *Proc. of 11th STOC*, pp. 209-213, 1979.

Lecture 22

Monotone Circuit Depth and Communication Complexity

Lecture given by Ran Raz
Notes taken by Yael Tauman and Yoav Rodeh

Summary: One of the main goals of studying circuit complexity is to prove lower bounds on the size and depth of circuits computing specific functions. Since studying the general model gave few results, we will concentrate on monotone circuits. The main result is a tight nontrivial bound on the monotone circuit depth of *st-Connectivity*. This is proved via a series of reductions, the first of which is of significant importance: A connection between circuit depth and communication complexity. We then get a communication game and proceed to reduce into other such games, until reaching the game $FORK$, and the conclusion that proving a lower bound on its communication complexity will give a matching lower bound on the monotone circuit depth of *st-Connectivity*.

22.1 Introduction

Turing machines are abstract models used to capture our concept of computation. However, we tend to miss some complexity properties of functions when examining them from the Turing machine point of view. One such central property of a function, is how efficiently it can be run in parallel. This property is best observed when we use the circuit model — by the depth of the circuit realizing the function. Another motivation for prefering the circuit model to the Turing machine model, is the hope that using advanced combinatorial methods will more easily give lower bounds to the size of circuits, and hence to the running time of Turing machines.

Recall that we need only examine circuits made up of $NOT(\neg)$ $OR(\vee)$ and $AND(\wedge)$ gates, any other gate can be simulated with constant blowup in the size and depth of the circuit. We may also assume all the NOT gates are at the leaf level because using De-Morgan rewrite rules, we do not increase the depth of the circuit at all, and may increase its size by a constant factor of 2 at most. In this lecture we will only discuss bounded fan-in circuits, and therefore may assume all gates to be of fan-in 2 (except NOT).

As always, our goal is to find (or at least prove the existence of) hard functions. In the context of circuit complexity we measure hardness by two parameters:

Definition 22.1 (Depth, Size): *Given* $f : \{0,1\}^n \to \{0,1\}$*, we define:*

1. *Depth*$(f) \stackrel{\text{def}}{=}$ *The minimum depth of a circuit computing* f, *where the depth of a circuit is the maximum distance from an input leaf to the output (when the circuit is viewed as a directed acyclic graph).*

2. *Size*$(f) \stackrel{\text{def}}{=}$ *The minimum size of a circuit computing* f, *where the size of a circuit is the number of gates it contains.*

Note that these quantities do not necessarily correlate: A circuit that computes f and has size $Size(f)$ and depth $Depth(f)$ may not exist. In other words, it is possible that every circuit of minimal size does not achieve minimal depth.

We will first prove the existence of hard functions:

22.1.1 Hard Functions Exist

There are no explicit families of functions that are proved to need large size circuits, but using counting arguments we can easily prove the existence of functions with size that is exponential in the size of their input.

Proposition 22.1.1 *For large enough* n, *there exists a function* $f : \{0,1\}^n \to \{0,1\}$ *s.t.* $Size(f) > \frac{2^n}{n^2}$.

Proof: First easy observation is that the number of functions $\{f \mid f : \{0,1\}^n \to \{0,1\}\}$ is exactly 2^{2^n}, since each such function can be represented as a $\{0,1\}$ vector of length 2^n.

We will now upper bound the number of circuits of size s. The way we approach the problem is by adding one gate at a time, starting from the inputs. At first we have $2n$ inputs — the variables and their negations. Each gate we add, is either an OR or an AND gate, and its two inputs can be chosen from any of the original inputs or from the outputs of the gates we already have. Therefore, for the first gate we have $\binom{2n}{2}$ choices for the the inputs and another choice between OR and AND. For the second gate we have exactly the same, except now the number of inputs to choose from is increased by one. Thus, the number of circuits of size s is bounded by:

$$\prod_{i=0}^{s-1} 2 \cdot \binom{2n+i}{2} < \prod_{i=0}^{s-1} 2 \cdot \frac{(2n+i)^2}{2} < \prod_{i=0}^{s-1} (2n+s)^2 = (2n+s)^{2s} = 2^{2s \cdot \log(2n+s)}$$

We wish to prove that the number of circuits of size $s = \frac{2^n}{n^2}$ is strictly less than the number of functions on n variables, and hence prove that there are functions that need circuits of size larger than s. For this we need to prove:

$$2^{2s\log(2n+s)} < 2^{2^n}$$
$$\Updownarrow$$
$$2s\log(2n+s) < 2^n$$

Which is obviously true for $s < \frac{2^n}{n^2}$, since for large enough n:

$$2s\log(2n+s) < 2 \cdot \frac{2^n}{n^2}\log(2^n) = 2^n \cdot (\frac{2}{n}) < 2^n$$

■

If we examine the proof carefully, we can see that actually most functions need a large circuit. Thus it would seem that it should be easy to find such a hard function. However, to the shame of all involved, the best known lower bounds for size and depth of "explicitly given" functions (actually families of functions) are:

$$\begin{array}{rcl} Size & \geq & 4n \\ Depth & \geq & 3\log(n) \end{array}$$

We therefore focus on weaker models of computation:

22.1.2 Bounded Depth Circuits

The first model we consider is that of bounded depth circuits. There are two deviations from the standard model. The first is that we artificially bound the depth of the circuit, and only consider the size of the circuit as a parameter for complexity. This immediately implies the other difference from the standard model: We do not bound the fan-in of gates. This is because otherwise, if we bound the depth to be a constant d, we automatically bound the size to be less than 2^d which is also a constant. This makes the model uninteresting, therefore we allow unbounded fan-in. Notice that any function can be computed by a depth 2 circuit (not counting NOT's) by transforming the function's truth table into an OR of many AND's. However, this construction gives exponential size circuits. Several results were reached for this model (see Lecture 20), but we will focus on a different model in this lecture.

22.2 Monotone Circuits

Monotone circuits is the model we consider next, and throughout the rest of this lecture. Monotone circuits are defined in the same way as usual circuits except we do not allow the usage of NOT gates.

It seems intuitive that monotone circuits cannot calculate any function, because there is no way to simulate a NOT gate using AND and OR gates. We will formulate and prove a characterization of the functions that can be computed using monotone circuits:

Definition 22.2 (Monotone Function): $f : \{0,1\}^n \to \{0,1\}$ *is a* monotone function *if for every* $x, y \in \{0,1\}^n$, $x \geq y$ *implies* $f(x) \geq f(y)$. *Where the partial order* $\geq$ *on* $\{0,1\}^n$ *is the hamming order, i.e.,* $(x_1, \ldots, x_n) \geq (y_1, \ldots, y_n)$ *if and only if for every* $1 \leq i \leq n$ *we have* $x_i \geq y_i$.

Remark: The hamming partial order can be thought of as the containment order between sets, where a vector $x \in \{0,1\}^n$ corresponds to the set $S_x = \{i \mid x_i = 1\}$. Then: $x \leq y$ if and only if $S_x \subseteq S_y$.

An example of a monotone function is $CLIQUE_{n,k} : \{0,1\}^{\binom{n}{2}} \to \{0,1\}$. The domain of the function $CLIQUE_{n,k}$ is the set of graphs on n vertices $\{1, \ldots n\}$. A graph is represented by assignments to the $\binom{n}{2}$ variables $x_{i,j}$, where for every pair $i, j \in \{1, \ldots n\}$, $x_{i,j} = 1$ iff (i, j) is an edge in the graph.

$CLIQUE_{n,k}$ is 1 on a graph if and only if the graph has a clique of size k. Clearly, $CLIQUE_{n,k}$ is a monotone function, because when our ordering is interpreted as the containment ordering between the sets of edges in a graph, then if a graph G contains a clique of size k, any other graph containing the edges of G will also contain the same clique.

Theorem 22.3 *A function* $f : \{0,1\}^n \to \{0,1\}$ *is monotone if and only if it can be computed by a monotone circuit.*

Proof:

- ($\Longrightarrow$) We will build a monotone circuit that computes f: For every α s.t. $f(\alpha) = 1$ we define:

  $$\phi_\alpha(x) = \bigwedge_{\alpha_i=1} x_i$$

  We also define:

  $$\phi(x) = \bigvee_{f(\alpha)=1} \phi_\alpha(x)$$

  It is clear that ϕ can be realized as a monotone circuit. Now we claim that $\phi = f$.

  1. For every α s.t. $f(\alpha) = 1$, we have $\phi_\alpha(\alpha) = 1$ and therefore $\phi(\alpha) = 1$.
  2. If $\phi(x) = 1$, then there is an α s.t., $\phi_\alpha(x) = 1$ and thereby $f(\alpha) = 1$. The fact that $\phi_\alpha(x) = 1$ means that $x \geq \alpha$ by the definition of ϕ_α. Now, from the monotonicity of f we conclude that $f(x) \geq f(\alpha) = 1$, meaning $f(x) = 1$.

- ($\Longleftarrow$) The functions AND and OR and the projection function $p_i(x_1, \ldots, x_n) = x_i$ are all monotone. We will now show that composition of monotone functions forms a monotone function, and therefore conclude that every monotone circuit computes a monotone function.

  Let $g : \{0,1\}^n \to \{0,1\}$ be a monotone function. Let $f_1, \ldots, f_n : \{0,1\}^N \to \{0,1\}$ be also monotone. We claim that $G : \{0,1\}^N \to \{0,1\}$ define by:

  $$G(x) = g(f_1(x), \ldots, f_n(x))$$

  is also monotone.

  If $x \geq y$ then from the monotonicity of $f_1, \ldots, f_n$, we have that for all i: $f_i(x) \geq f_i(y)$. In other words:

  $$(f_1(x), \ldots f_n(x)) \geq (f_1(y), \ldots f_n(y))$$

  Now, from the monotonicity of g, we have:

  $$G(x) = g(f_1(x), \ldots, f_n(x)) \geq g(f_1(y), \ldots, f_n(y)) = G(y)$$

■

We make analogous definitions for complexity in monotone circuits:

Definition 22.4 (*Mon-Size*, *Mon-Depth*): *Given a monotone function $f : \{0,1\}^n \to \{0,1\}$, we define:*

1. *Mon-Size*$(f) \stackrel{\text{def}}{=}$ *The minimum size of a monotone circuit computing f.*
2. *Mon-Depth*$(f) \stackrel{\text{def}}{=}$ *The minimum depth of a monotone circuit computing f.*

Obviously for every monotone function f, *Mon-Size*$(f) \geq$ *Size*(f), and *Mon-Depth*$(f) \geq$ *Depth*(f). In fact there are functions for which these inequalities are strict. We will not prove this result here.

Unlike the general circuit model, several lower bounds were proved for the monotone case. For example, it is known that for large enough n and specific k (depending on n):

$$\begin{aligned} \textit{Mon-Size}(CLIQUE_{n,k}) &= \Omega(2^{\frac{1}{3}n}) \\ \textit{Mon-Depth}(CLIQUE_{n,k}) &= \Omega(n) \end{aligned}$$

¿From now on we shall concentrate on proving a lower bound on *st-Connectivity*:

Definition 22.5 (*st-Connectivity*): *Given a directed graph G on n nodes, two of which are marked as s and t, st-Connectivity$(G) = 1$ if and only if there is a directed path from s to t in G.*

Obviously *st-Connectivity* is a monotone function since if we add edges we cannot disconnect an existing path from s to t.

Theorem 22.6 *Mon-Depth(st-Connectivity)* $= \Theta(\log^2(n))$

In a previous lecture, we proved that *st-Connectivity* is in NC_2. This we proved by constructing a circuit that performs $O(\log(n))$ successive boolean matrix multiplications. Notice that the operation of multiplying boolean matrices is a monotone operation (it uses only AND and OR gates). Therefore, the circuit constructed for *st-Connectivity* is actually monotone. If we define $Mon\text{-}NC_i$ to be the natural monotone analog of NC_i, then *st-Connectivity* is in $Mon\text{-}NC_2$. Also, from the above theorem *st-Connectivity* is not in $Mon\text{-}NC_1$. This gives us:

Corollary 22.7 $Mon\text{-}NC_1 \neq Mon\text{-}NC_2$

An analogous result in the non-monotone case is believed to be true, yet no proof is known.

We will proceed by reducing the question of monotone depth to a question in communication complexity.

22.3 Communication Complexity and Circuit Depth

There is an interesting connection between circuit depth and communication complexity which will assist us when proving our main theorem. Since the connection itself is interesting, we will prove it for general circuits. First some definitions:

Definition 22.8 *Given $f : \{0,1\}^n \rightarrow \{0,1\}$ we define a communication game G_f:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. $f(x) = 1$.*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. $f(y) = 0$.*

Their goal is to find a coordinate i s.t. $x_i \neq y_i$.

Notice that this game is not exactly a communication game in the sense we defined in the previous lecture, since the two players do not compute a function, but rather a relation.

We denote the communication complexity of a game G by $CC(G)$. The connection between our complexity measures is:

Lemma 22.3.1 $CC(G_f) = Depth(f)$

Proof:

1. First we'll show $CC(G_f) \leq Depth(f)$. Given a circuit C that calculates f, we will describe a protocol for the game G_f. The proof will proceed by induction on the depth of the circuit C.

   - **base case:** $Depth(f) = 0$. In this case, f is simply the function x_i or $\neg x_i$, for some i. Therefore there is no need for communication, since i is a coordinate in which x and y always differ.

- **Induction step:** We look at the top gate of C: Assume $C = C_1 \wedge C_2$, then

$$Depth(C) = 1 + \max\{Depth(C_1), Depth(C_2)\}$$
$$\Downarrow$$
$$Depth(C_1), Depth(C_2) \leq Depth(C) - 1$$

Denote by f_1 and f_2 the functions that C_1 and C_2 calculate respectively. By the induction hypothesis:

$$CC(G_{f_1}), CC(G_{f_2}) \leq Depth(C) - 1$$

We know that $f(x) = 1$ and $f(y) = 0$, therefore:

$$f_1(x) = f_2(x) = 1$$
$$f_1(y) = 0 \text{ or } f_2(y) = 0$$

Now, as the first step in the protocol, player 2 sends a bit specifying which of the functions f_1 or f_2 is zero on y. Assume player 2 sent 1. In this case they both know:

$$f_1(y) = 0$$
$$f_1(x) = 1$$

And now the game has turned into the game G_{f_1}. This we can solve (using our induction hypothesis) with communication complexity $CC(G_{f_1}) \leq Depth(f_1)$. If player 2 sent 2 we would use the protocol for G_{f_2}. We needed just one more bit of communication. Therefore our protocol will have communication complexity of:

$$\begin{aligned} CC(G_f) &\leq 1 + \max\{CC(G_{f_1}), CC(G_{f_2})\} \\ &\leq 1 + \max\{Depth(f_1), Depth(f_2)\} \\ &= 1 + (Depth(f) - 1) \qquad = \quad Depth(f) \end{aligned}$$

We proved this for the case where $C = C_1 \wedge C_2$. The case where $C = C_1 \vee C_2$ is proved in the same way, expect player 1 is the one to send the first bit (indicating if $f_1(x) = 1$ or $f_2(x) = 1$).

2. Now we'll show the other direction: $CC(G_f) \geq Depth(f)$. For this we'll define a more general sort of communication game based on two non-intersecting sets: $A, B \subseteq \{0,1\}^n$:

   - Player 1 gets $x \in A$
   - Player 2 gets $y \in B$
   - Their goal is to find a coordinate i s.t. $x_i \neq y_i$.

   We'll denote this game by $G_{A,B}$. Using the new definition G_f equals $G_{f^{-1}(1), f^{-1}(0)}$. We will prove the following claim:

   Claim 22.3.2 *If $CC(G_{A,B}) = d$ then there is a function $f : \{0,1\}^n \rightarrow \{0,1\}$ that satisfies:*

   - *$f(A) = 1$ (i.e., $f(x) = 1$ for every $x \in A$).*
   - *$f(B) = 0$*
   - *$Depth(f) \leq d$*

In the case of G_f, the function we get by the claim must be f itself, and we get that it satisfies $Depth(f) \leq CC(G_f)$, proving our lemma.

Proof: *(claim)* By induction on $d = CC(G_{A,B})$

- **Base case:** $d = 0$, meaning there is no communication, so there is a coordinate i in which all of A is different then all of B, and so the function $f(\alpha) = \alpha_i$ or the function $f(\alpha) = \neg\alpha_i$ will satisfy the requirements depending on whether the coordinate i is 1 or 0 in A.
- **Induction step:** We have a protocol for the game $G_{A,B}$ of communication complexity d. First assume player 1 sends the first bit in the protocol. This bit partitions the set A into two disjoint sets $A = A_0 \cup A_1$, or in other words, this bit turns our game into one of the following games (depending on the bit sent): $G_{A_0,B}$ or $G_{A_1,B}$. Each one of these has communication complexity of at most $d-1$ simply by continuing the protocol of $G_{A,B}$ after the first bit is already sent. Now, by the induction hypothesis we have two functions f_0 and f_1 that satisfy:
  - $f_0(A_0) = 1$ and $f_1(A_1) = 1$.
  - $f_0(B) = f_1(B) = 0$
  - $Depth(f_0), Depth(f_1) \leq d-1$

  We define $f = f_0 \vee f_1$. Then:
  - $f(A) = f_0(A) \vee f_1(A) = 1$, because f_0 is 1 on A_0 and f_1 is 1 on A_1.
  - $f(B) = f_0(B) \vee f_1(B) = 0$
  - $Depth(f) = 1 + \max\{Depth(f_0), Depth(f_1)\} \leq d$

  So f is exactly what we wanted.

  If player 2 sends the first bit, he partitions B into two disjoint sets $B = B_0 \cup B_1$, and turns the game into G_{A,B_0} or G_{A,B_1}. By the induction hypothesis we have two functions corresponding to the two games, g_0 and g_1, so that:

$$g_0(A) = g_1(A) = 1$$
$$g_0(B_0) = g_1(B_1) = 0$$

  We define $g \stackrel{\text{def}}{=} g_0 \wedge g_1$. This g satisfies:
  - $g(A) = g_0(A) \wedge g_1(A) = 1$.
  - $g(B) = g_0(B) \wedge g_1(B) = 0$ (because g_0 is 0 on B_0, and g_1 is 0 on B_1).

■

■

22.4 The Monotone Case

Let us remember that our goal was to prove tight bounds on the monotone depth of *st-Connectivity*. Therefore we will define an analogue game for monotone functions, that will give us a lemma of the same flavor as the last.

22.4.1 The Analogous Game and Connection

Definition 22.9 (Monotone game): *Given a monotone $f : \{0,1\}^n \to \{0,1\}$ we define a communication game M_f:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. $f(x) = 1$.*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. $f(y) = 0$.*

Their goal is to find a coordinate i s.t. $x_i > y_i$, i.e. $x_i = 1$ and $y_i = 0$ We denote this kind of a game a monotone game.

The game is exactly the same as G_f, except f is monotone, and the goal is more specific; i.e., the goal is to a find a coordinate i where not only $x_i \neq y_i$ but also $x_i > y_i$. Notice that the goal is always achievable, because if there is no such i, then y is at least as large as x in every coordinate. This means that $y \geq x$, but this contradicts the fact that f is monotone and $f(x) = 1$, $f(y) = 0$.

Our corresponding lemma for the monotone case is:

Lemma 22.4.1

$$CC(M_f) = Mon\text{-}Depth(f)$$

Proof: The proof is similar to the non-monotone case:

1. When building the protocol from a given circuit:
   - **Base case:** since f is monotone, if the depth is 0, we have that $f(\alpha) = \alpha_i$ and therefore it must be the case that $x_i = 1$ and $y_i = 0$. Hence, again there is no need for communication, and the answer is i (after all $x_i > y_i$).
   - **Induction step:** In the induction step, the top gate separates our circuit into two subcircuits. The protocol then uses one communication bit to decide which of the two games corresponding to the two sub-circuits to solve. Since the sub-circuits are monotone, by the induction hypothesis they each have a protocol to solve their matching monotone game. This solves the monotone game corresponding to the whole circuit, since the sub-games are monotone, and therefore the coordinate i found satisfies $x_i > y_i$.
2. When building the circuit from a given protocol:
   - **Base case:** if there is no communication, both players already know a coordinate i in which $x_i > y_i$, hence our circuit would simply be $f(\alpha) = \alpha_i$, which is monotone and of depth 0.
   - **Induction step:** Each communication bit splits our game into two sub-games of smaller communication. Notice that if the original game was a monotone game, so are the two sub-games. By the induction hypothesis, the circuits for these games are monotone. Now, since we only add AND and OR gates, the circuit built is monotone.

■

22.4.2 An Equivalent Restricted Game

Let us define a more restricted game than the one in Definition 22.9, that will be easier to work with. First some definitions regarding monotone functions:

Definition 22.10 (minterm, maxterm): *Let $f : \{0,1\}^n \to \{0,1\}$ be a monotone function.*

- *A* minterm *of f is $x \in \{0,1\}^n$ s.t. $f(x) = 1$ and for every $x' < x$ we have $f(x') = 0$.*
- *A* maxterm *of f is $y \in \{0,1\}^n$ s.t. $f(y) = 0$ and for every $y' > y$ we have $f(y') = 1$.*

For example, for *st-Connectivity*:

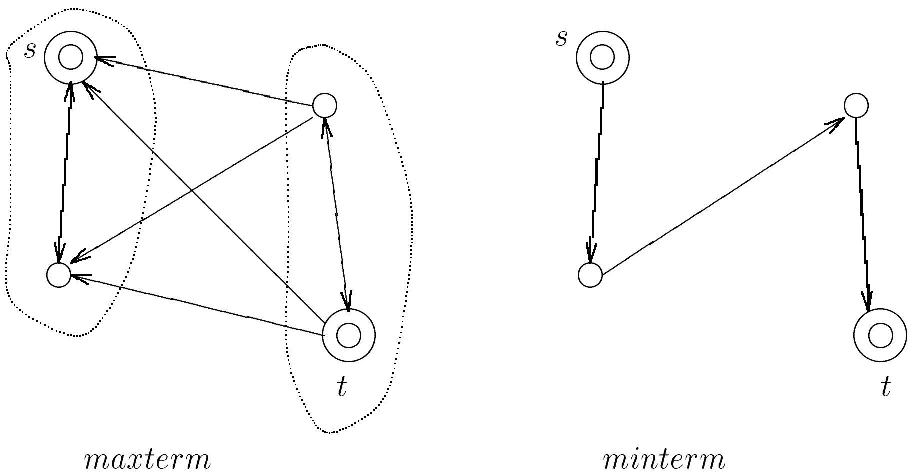


Figure 22.1: A maxterm and minterm example for *st-Connectivity*.

- The set of minterms is the set of graphs that contain only a simple (does not intersect itself) directed path from s to t:
  1. If a graph G is a minterm, then it must contain a simple path from s to t, and it cannot contain any other edge. This is because *st-Connectivity*$(G) = 1$, therefore there is a simple path P from s to t in G. G cannot contain any other edges, because then $P < G$ (in the edge containment order), but *st-Connectivity*$(P) = 1$, contradicting the fact that G is a minterm.
  2. Every G that is a simple path from s to t is a minterm, because *st-Connectivity*$(G) = 1$ and every edge we drop will disconnect s from t, therefore it is minimal.
- The set of maxterms for *st-Connectivity* is the set of graphs G s.t. G's set of vertices can be partitioned into two disjoint parts S and T that satisfy:
  1. $s \in S$ and $t \in T$.
  2. G contains all possible directed edges except those from S to T

  This is indeed the set of maxterms for *st-Connectivity*:

  1. If G is a maxterm then let S be the set of vertices that are reachable from s in G. Set T to be all other vertices. $t \in T$ because one cannot reach t from s in G, since

st-$Connectivity(G) = 0$. Also, G must contain all edges except those from S to T, otherwise we can add the missing edges and still leave t unconnected from s. There are no edges from S to T by the definition of S as the connected component reachable from s.

2. If G satisfies both criteria, then every path starting from s in G will remain in S and therefore will not reach t so st-$Connectivity(G) = 0$. Every edge we add to G will connect S to T and since S and T are strongly connected it will create a path between s and t.

Another way to view a maxterm of *st-Connectivity*, is that the partition is defined by a coloring of the vertices by two colors 0 and 1, where s is colored 0 and t is colored 1. The set of vertices colored 0 is S, and those colored 1 are T.

We will now use maxterms and minterms to define a new communication game:

Definition 22.11 ($\hat{M}_f$): *Given a monotone $f : \{0,1\}^n \rightarrow \{0,1\}$ we define a communication game $\hat{M}_f$:*

- *Player 1 gets $x \in \{0,1\}^n$, s.t. x is a minterm of f (in particular $f(x) = 1$).*
- *Player 2 gets $y \in \{0,1\}^n$, s.t. y is a maxterm of f (in particular $f(y) = 0$).*

Their goal is to find a coordinate i s.t. $x_i > y_i$, i.e. $x_i = 1$ and $y_i = 0$.

Notice that $\hat{M}_f$ is a restriction of M_f to a smaller set of inputs, therefore the protocol that will solve M_f will also solve $\hat{M}_f$. Hence $CC(\hat{M}_f) \leq CC(M_f)$. In fact, the communication complexity of the two games is exactly the same:

Proposition 22.4.2 $CC(\hat{M}_f) = CC(M_f)$

Proof: What is left to prove is that: $CC(\hat{M}_f) \geq CC(M_f)$. Given a protocol for $\hat{M}_f$ we construct a protocol for M_f of the same communication complexity.

1. Player 1 has x s.t. $f(x) = 1$. He now finds a minimal x' s.t. $x' \leq x$ but $f(x') = 1$. This is done by successively changing coordinates in x from 1 to 0, while checking that $f(x')$ still equals 1. This way, eventually, he will get x' that is a minterm.

2. In the same manner player 2 finds a maxterm $y' \geq y$.

The players now proceed according to the protocol for $\hat{M}_f$ on inputs x' and y'. Since x' is a minterm, and y' is a maxterm, the protocol will give a coordinate i in which:

$$\begin{array}{lll} {x'}_i = 1 & \Longrightarrow \quad x_i = 1 & \text{because } x' \leq x \\ {y'}_i = 0 & \Longrightarrow \quad y_i = 0 & \text{because } y' \geq y \end{array}$$

The communication complexity is exactly the same, since we used the same protocol except for a preprocessing stage that does not cost us in communication bits. ■

Combining our last results we get:

Corollary 22.12 *Given a monotone function $f : \{0,1\}^n \rightarrow \{0,1\}$:*

$$Mon\text{-}Depth(f) = CC(\hat{M}_f)$$

22.5 Two More Games

As we have seen, when examining bounds on the monotone depth of *st-Connectivity*, we need only examine the communication complexity of the following game denoted KW (for Karchmer and Wigderson) which is simply a different formulation of $\hat{M}_{st\text{-}Connectivity}$:

Given n nodes and two special nodes s and t,

- Player 1 gets a directed path from s to t.
- Player 2 gets a coloring C of the nodes by 0 and 1, s.t. $C(s) = 0$ and $C(t) = 1$.
- The goal is to find an edge (v, w) on player 1's path s.t. $C(v) = 0$ and $C(w) = 1$.

First we will use this formulation to show an $O(\log^2(n))$ upper bound on $Mon\text{-}Depth(st\text{-}Connectivity)$ using a protocol for KW with communication complexity $O(\log^2(n))$:

Proposition 22.5.1 $CC(KW) = O(\log^2(n))$

Proof: The protocol will simulate binary search on the input path of player 1. In each step, we reduce the length of the path by a factor of 2, while keeping the invariant that the color of the first vertex in the path is 0, and the color of the last is 1. This is of course true in the beginning since $C(s) = 0$ and $C(t) = 1$.

The base case, is that the path has only one edge, and in this case we are done, since our invariant guarantees that this edge is colored as we want. Now, player 1 sends player 2 this edge. The communication cost is $O(\log(n))$.

If the path is longer, player 1 asks player 2 the color of the middle vertex in the path. This costs $\log(n) + 1$ bits of communication — the name of the middle vertex sent from player 1 to player 2 takes $\log(n)$ bits, and player 2's answer costs one more bit. If the color is 1, the first half of the path satisfies our invariant, since the first vertex is colored 0, and now the last will be colored 1. If the color is 0, we take the second half of the path. In any case, we cut the length of the path by 2 with communication cost $O(\log(n))$.

Since the length of the original path is at most n, we need $O(\log(n))$ steps until we reach a path of length 1. All in all, we have communication complexity of $O(\log^2(n))$. ■

We will now direct our efforts towards proving a lower bound of $\Omega(\log^2(n))$ for $Depth(st\text{-}Connectivity)$ via a lower bound for KW. For this we will continue to yet another reduction into a different communication game called $FORK$:

Definition 22.13 ($FORK$): *Given $n = l \cdot w$ vertices and three special vertices s, t_1, and t_2, where the n vertices are partitioned into l layers $L_1, \ldots, L_l$, and each layer contains w vertices:*

- *Player 1 gets a sequence of vertices $(x_0, x_1, \ldots, x_l, x_{l+1})$, where for all $1 \le i \le l$: $x_i \in L_i$, and $x_0 = s$, $x_{l+1} = t_1$.*
- *Player 2 gets a sequence of vertices $(y_0, y_1, \ldots, y_l, y_{l+1})$, where for all $1 \le i \le l$: $y_i \in L_i$, and $y_0 = s$, $y_{l+1} = t_2$.*
- *Their goal is to find an i such that $x_i = y_i$ and $x_{i+1} \neq y_{i+1}$.*

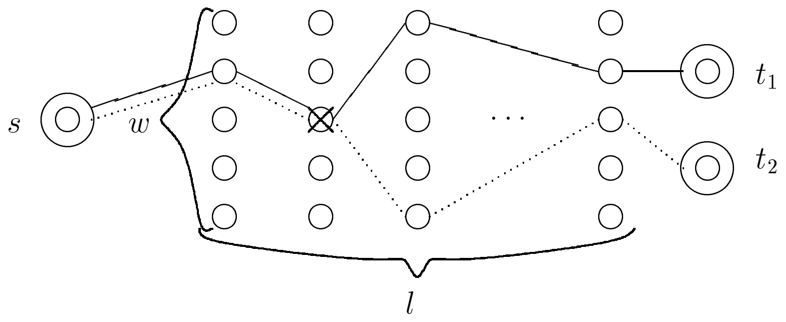


Figure 22.2: Player 1's sequence is solid, player 2's is dotted, and the fork point is marked with an x.

Obviously, such an i always exists, since the sequences start at the same vertex (s), and end in different vertices ($t_1 \neq t_2$), therefore there must be a fork point.

Note: The sequences the players get can be thought of as an element in $\{1, \ldots, w\}^l$. Since the start vertex is set to be s, and the end vertices for both players are also set to be t_1 and t_2 (depending on the player).

This game is somewhat easier to deal with then KW, because of the symmetry between the players. We will show that this new game needs no more communication than KW , and therefore, proving a lower bound on its communication complexity suffices.

Proposition 22.5.2 $CC(FORK) \leq CC(KW)$

Proof: Assuming we have a protocol for KW, we will show a protocol for $FORK$ which uses the same amount of communication. Actually, as in the proof of Proposition 22.4.2, all that the players have to do is some preprocessing, and the protocol itself does not change.

Recall that in the game KW player 1 has a directed path between two special vertices s and t, that goes through a set of regular vertices. Player 2 has a coloring of all vertices by 0 and 1, where s is colored 0, and t is colored 1.

To use the protocol for KW, the players need to turn their instance of $FORK$ into an instance of KW.

- We define the vertex s in $FORK$ to be s in KW.
- We define the vertex t_1 to be t.
- All other vertices are regular vertices.
- The path of player 1 remains exactly the same — it is indeed between s and $t(= t_1)$.
- The coloring of player 2 is: a vertex is colored 0 if and only if it is in his input path of vertices — note that s is colored 0, since it is the first vertex in his sequence, and $t(= t_1)$ is colored 1 because it is not on this path (which goes from s to t_2).

After this preprocessing we use the protocol for KW to get an edge (u, v) that is on player 1's path, where u is colored 0, and v is colored 1. This means, that u is on player 2's path, because it is colored 0, and v is not, because it is colored 1.

Hence, u is exactly the kind of fork point we were looking for, since it's in both players path and its successor is different in the two paths. ■

In the next lecture we will prove that

$$CC(FORK) = \Omega(\log(l)\log(w))$$

Setting $l = w = \sqrt{n}$, our main theorem follows:

$$CC(st\text{-}Connectivity) = \Theta(\log^2(n))$$

Bibliographic Notes

This lecture is based mainly on [2]. Specifically, the connection between circuit depth and the corresponding communication complexity problem was made there. The current analysis of the communication complexity problem corresponding to s-t-Connectivity, via reduction to the game FORK, is due to [1] and is simpler than the original analysis (as in [2]).

1. M. Gringi and M. Sipser. Monotone Separation of Logarithmic Space from Logarithmic Depth. *JCSS*, Vol. 50, pages 433–437, 1995.

2. M. Karchmer and A. Wigderson. Monotone Circuits for Connectivity Require Super-Logarithmic Depth. *SIAM J. on Disc. Math.*, Vol. 3, No. 2, pages 255–265, 1990.

Lecture 23

The FORK Game

Lecture given by Ran Raz
Notes taken by Dana Fisman and Nir Piterman

Summary: We analyze the game FORK that was introduced in the previous lecture. We give tight lower and upper bounds on the communication needed in a protocol solving FORK. This completes the proof of the lower bound on the depth of monotone circuits computing the function st-Connectivity.

23.1 Introduction

We have seen in the previous lecture the connection between circuit depth and communication complexity: We compared the depth of a circuit computing a function f to the number of bits transferred between two players playing a communication game G_f. We saw that given a communication protocol solving G_f using communication of c-bits, we can construc t a circuit computing f whose depth is c. On the other hand, given a c-depth circuit computing f we can plan a communication protocol for G_f which uses c bits. Thus, we established that the best communication protocol for solving G_f uses t he same amount of communication bits as the depth of the best circuit computing f. tion protocol. The plan was to use communication complexity to prove upper and lower bounds on the depths of circuits. Failing to reach satisfactory results using this model, the weaker model of monotone functions and *monotone* circuits was introduced. Since a mono tone circuit is in particular a circuit but not the other way around, proving lower bounds on monotone circuits does not prove the same lower bounds in the unrestricted model. All the same, our goal was to show a (tight) lower bound for the depth of monot one circuits computing st-Connectivity. The plan was to achieve the result by conducting a series of reductions. The first used the connection between circuit depth and communication complexity to reduce the question to a communication protocol. Then, various reductions between several kinds of games led us to the FORK game (see Definition 23.1). Based on a lower bound for the communication needed in FORK, the lower bound for st-Connectivity of $\Omega(log^2 n)$ was proven. The purpose of this lecture is to prove the lower bound of the FORK game. We will give a complete analysis of the FORK game showing that the communication between the two players is $\Theta(log_2 w \cdot log_2 l)$, thus supplying the missing link in the proof of the lower bound on the depth of st-Connectivity.

23.2 The FORK game – recalling the definition

FORK is a game between two players. Each player gets a path from a predefined set of possible paths. Both paths start at the same point but have different end points. Hence, at least one fork point in which the two paths separate, must exist. The players' goal is to find such a fork point. More formally, we recall the definition of the FORK game given in previous lecture.

Definition 23.1 (FORK): *Given $n = l \cdot w$ vertices and three special vertices s, t_1, and t_2, where the n vertices are divided into l layers $l_1, \ldots, l_l$, and each layer contains w vertices:*

- Player I *gets a sequence of vertices $(x_1, x_2, \ldots, x_l)$, where for all $1 \leq i \leq l$: $x_i \in l_i$. For simplicity of notation we assume that* Player I *is given two more coordinates: $x_0 = s$, $x_{l+1} = t_1$.*
- Player II *gets a sequence of vertices $(y_1, y_2, \ldots, y_l)$, where for all $1 \leq i \leq l$: $y_i \in l_i$. Again we consider $y_0 = s$, $y_{l+1} = t_2$.*
- *Their goal is to find a coordinate i such that $x_i = y_i$ and $x_{i+1} \neq y_{i+1}$.*

In order to stress the fact that the inputs are elements of $[w]^l$, where $[w] = \{1, ..., w\}$, we slightly modified the definition, excluding the constant points s, t_1 and t_2 from the input sequences given to the players.

The following theorem states the main result of this lecture, giving the bounds on the communication complexity of solving the FORK game.

Theorem 23.2 *The communication complexity of the* FORK *game is $\Theta(log_2 w \cdot log_2 l)$.*

We first show an upper bound on the communication complexity of the game. The upper bound is given here only for the completeness of the discussion concerning the FORK problem.

23.3 An upper bound for the FORK game

The proof given here is very similar to the proof of the upper bound for the KW game (see previous lecture). We basically perform a binary search on the path to find the FORK point.

Proposition 23.3.1 *The communication complexity of the* FORK *game is $O(log_2 w \cdot log_2 l)$.*

Proof: For the sake of simplicity, the following notation is introduced. We denote $F_{a,b}$ as the FORK game where the inputs are of the form $\overline{x} = (x_a, x_{a+1}, ..., x_b)$ and $\overline{y} = (y_a, y_{a+1}, ..., y_b)$. Like in the general FORK gam e, we consider $\overline{x}$ and $\overline{y}$ as having two more coordinates. An $(a-1)$-coordinate such that $x_{a-1} = y_{a-1} = s$ is the origin point of the paths. A $(b+1)$-coordinate such that $x_{b+1} = t_1$ and $y_{b+1} = t_2$ are the endpoints of the paths.

First notice that if the length of the paths is only one, i.e., $a = b$, then the problem can be solved using $log_2 w$ bits, obeying the following protocol:

- Player I sends its input (this requires $log_2 w$ bits).
- Player II replies with 1 if he has the same coordinate and with 0 otherwise (one bit).

If they have the same coordinate, the fork point is found in that coordinate and then the paths separate to t_1 and t_2. Otherwise the fork is the point of origin $s = x_{a-1} = y_{a-1}$.

If the length of the paths is larger than 1, i.e., $a < b$, then the problem can be reduced to half using $log_2 w$ bits. (For simplicity, we assume that $\frac{a+b}{2}$ is an integer):

- Player I sends its middle layer node: $x_{\frac{(a+b)}{2}}$ (this requires $log_2 w$ bits).
- Player II checks if they have the same middle node: $y_{\frac{a+b}{2}} = x_{\frac{a+b}{2}}$.

  If so he sends 1 otherwise he sends 0 (one bit).

If the players have the same middle point (i.e., Player I sent 1), then there has to be a fork point between the middle and the end. Thus the game is reduced to $F_{\frac{a+b}{2}+1,b}$. On the other hand, if the middle points differ, then there has to be a fork point between the start and the middle. Thus the game is reduced to $F_{a,\frac{a+b}{2}-1}$. Note that there is no point in including the middle layer itself in the range of the reduced game. The reason being that in the first case, the mutual point in layer $\frac{a+b}{2}$ is actually the new origin point, while in the second case, the two points in layer $\frac{a+b}{2}$ are actually the new end points.

Therefore FORK (i.e. $F_{1,l}$ using this notation) is solved in $log_2 l$ iterations of the protocol, requiring transmission of $O(log_2 w \cdot log_2 l)$ bits. ■

Our goal now is to show that this upper bound is tight.

23.4 A lower bound for the FORK game

In order to show the lower bound we consider games that work only for a subset of the possible inputs. We perform some inductive process in which we establish the connection between games with different sets. Two kinds of transformations are considered:

1. Given a protocol that works for some set, we devise a protocol that works for smaller density sets with one less bit of communication.
2. Given a protocol that works for some set, we convert it into a protocol that works for sets of higher density but shorter paths using the same amount of communication.

When adapting the given protocol into a new one, some heavy computations are involved. However, recall that the only parameter considered during the application of the protocol is the number of bits transmitted. Thus, any computation done in the preparati on of the protocol and any computation done locally by either side is not taken into account.

23.4.1 Definitions

We first define a subgame of FORK that works only on a subset of the possible inputs. Given a subset $S \subseteq [w]^l$ we consider the game FORK^S where Player I gets an input $\overline{x} \in S$ and Player II gets an input $\overline{erliney} \in S$ and they have the same goal of finding the fork point between the two paths.

We will work only with subsets of $[w]^j$, where the power j (the length of the path) will change from time to time. We call w the width of the game. Throughout the discussion the width w of the game will be constant and we frequently ignore i t.

As explained above, the density of the set will be an important parameter in the transformations:

Definition 23.3 (density): *The* density *of a set $S \subseteq [w]^l$ is defined as $\alpha(S) = \frac{|S|}{w^l}$.*

Given a protocol solving a partial FORK game, we ask ourselves what is the density of the sets that this protocol works on. We are interested in the minimal protocol that will work for some set of density α:

Definition 23.4 ($(\alpha, l) - protocol$): *A communication protocol will be called an* (α, l)-protocol *if it works for some set S of paths of length l with density $\alpha(S) = \alpha$.*

Definition 23.5 ($CC(\alpha, l)$): *Let $CC(\alpha, l)$ denote the smallest communication complexity of an $(\alpha, l) - protocol$.*

Using this terminology, since there is just one set of density 1, a protocol for FORK is just a $(1, l) - protocol$ and so we are interested in $CC(1, l)$.

23.4.2 Reducing the density

The following lemma enables the first kind of transformation discussed above. Given a protocol that works for a set of a certain density we adapt it to a protocol that works for a subset of half that density. The new protocol will work with one less bit o f communication than the original protocol.

Lemma 23.4.1 *If there exists an $(\alpha, l) - protocol$ which works with c bits and $c > 0$, then there is also an $(\frac{\alpha}{2}, l) - protocol$ that works with $c - 1$ bits.*

By the above lemma, the best protocol for an α density set will require at least one more bit than the best protocol for an $\frac{\alpha}{2}$ density set. Thus, we have

Corollary 23.6 *If $CC(\alpha, l)$ is non-zero then $CC(\alpha, l)$ is greater than $CC(\frac{\alpha}{2}, l)$ by at least one;*

$$CC(\alpha, l) \geq CC(\tfrac{\alpha}{2}, l) + 1$$

Proof: The proof can be generalized for any communication protocol. As explained in detail in Lecture 21, we use the fact that any communication protocol can be viewed as the two parties observing a matrix. The matrix' rows correspond to the possible inputs of Player I and its columns correspond to the possible inputs of Player II. The entries of the matrix are the desired results of the protocol. Passing one bit between the two players partitions this matrix into two rectangles (horizontally if the first bit is sent by Player I and vertically if it is sent by Player II).

Let P be a non-zero communication $(\alpha, l) - protocol$ for the set S. Assume without loss of generality that Player I sends the first bit in the protocol P. Let $S_0 \subseteq S$ be those inputs in S for which Player I sends 0 as the f irst bit, and similarly let $S_1 \subseteq S$ be those inputs in S for which Player I sends 1 as the first bit. Since $S_0 \cup S_1 = S$, either S_0 or S_1 contains at least half the inputs in S. Assuming that this is S_0 then $\alpha(S_0) \geq \frac{\alpha(S)}{2}$. Let P' be a protocol that works like P but without sending the first bit, while the players assume that the value of this bit is 0. Obviously P' works if Player I gets inputs from S_0 and Player II gets inputs fr om S. In particular it works when both inputs are from the subset S_0. We conclude that P' is an $(\frac{\alpha}{2}, l) - protocol$ that works with one bit less than P.
The case where S_1 is larger than S_0 is identical. ■

In order to apply Lemma 23.4.1 the protocol must cause at least one bit to be sent by some player. Therefore, we would like to identify the sets for which a protocol consumes some communication. We show that for large enough sets, whose density is more than $\frac{1}{w}$, any protocol indeed uses non-zero communication.

Lemma 23.4.2 *Any protocol for a set whose density is larger than $\frac{1}{w}$ requires communication of at least one bit.*

Corollary 23.7 *For every l and every $\alpha > \frac{1}{w}$,* $CC(\alpha, l) > 0$.

Proof: We will show that if P is a protocol that works with no communication at all and solves FORK^S then the density of S is at most $\frac{1}{w}$.

Since no information is passed between the players, any of the players must establish his result depending only on his input. It could be the case where both players get the same path $\overline{x} = \overline{y}$. In this case the only fork point is in the last layer $x_l = y_l$ (with $x_{l+1} = t_1 \neq t_2 = y_{l+1}$). So Player I must always give the last layer as the fork point. If some $\overline{y} \in S$ has a different point in the last layer (i.e. $y_l \neq x_l$), the protocol will give a wrong answer. Hence for all the paths in S the last point in the path has to be j for some j between 1 and w. So the set S is in fact a subset of $[w]^{l-1} \times \{j\}$ and therefore its density is not greater than $\frac{w^{l-1}}{w^l} = \frac{1}{w}$. ■

Note that using these two lemmas we can get a lower bound of $\Omega(log_2 w)$ bits for FORK. As long as the density is greater than $\frac{1}{w}$, Lemma 23.4.2 guarantees that we can apply Lemma 23.4.1. Repeating Lemma 23.4.1 less than $log_2 w$ times we know that the density has not decreased beyond $\frac{1}{w}$ and so Lemma 23.4.1 can be applied again.

$$CC(1, l) \geq CC(\tfrac{1}{2}, l) + 1 \geq CC(\tfrac{1}{4}, l) + 2 \geq ... \geq CC(\tfrac{1}{w}, l) + log_2 w$$

Considering our aim is to use this bound in connection with boolean circuits, the bound $CC(1, l) = \Omega(log_2 w)$ is insignificant. This is the case, since in order to read an input of length w we must use a circuit of at least $log_2 w$ depth anyhow.

23.4.3 Reducing the length

In order to reach the desired bound we need to manipulate the length of the paths as well. The main tool will be the following 'amplification' lemma that allows us, using an $(\alpha, l) - protocol$, to construct another protocol that works on a set of short er paths (of length $\frac{l}{2}$) but whose density is larger than α.

Lemma 23.4.3 *Let $\alpha \geq \frac{12}{w}$. If there exists an $(\alpha, l) - protocol$ for FORK^S, that uses c bits of communication, then there is also an $(\frac{\sqrt{\alpha}}{2}, \frac{l}{2}) - protocol$ that uses the same number of bits.*

For α's in the range $\frac{12}{w} < \alpha < \frac{1}{4}$ using this lemma increases the density of the set. Since:

$$\frac{12}{w} < \alpha < \frac{1}{4} \implies \frac{\sqrt{\alpha}}{2} > \alpha$$

The proof of the lemma uses the following technical claim:

Claim 23.4.4 *Consider an $n \times n$ matrix of 0-1. Denote by α the fraction of one-entries in the matrix and by α_i the fraction of one-entries in the i^{th} row. We say that a row i is* dense *if $\alpha_i \geq \frac{\alpha}{2}$. One of the following two cases hold:*

1. *there is some row i with $\alpha_i \geq \sqrt{\frac{\alpha}{2}}$*
2. *the number of dense rows is at least $\sqrt{\frac{\alpha}{2}} \cdot n$*

Proof: Intuitively, the claim says that either one of the rows is 'very dense' or there are a lot of dense rows.

Suppose by contradiction that the two cases do not hold. Let us calculate the density of one-entries in the entire matrix. Since Case 2 does not hold there are less than $\sqrt{\frac{\alpha}{2}} \cdot n$ dense rows. Since

Case 1 does not hold each of them has less than $\sqrt{\frac{\alpha}{2}} \cdot n$ one-entries. Hence the fraction of one-entries in all the dense rows is less than $\sqrt{\frac{\alpha}{2}} \cdot \sqrt{\frac{\alpha}{2}} = \frac{\alpha}{2}$. Non-dense rows contain less than $\frac{\alpha}{2} \cdot n$ one-entries and there are at most n non-dense rows, hence the fraction of one-entries in the non-dense rows is less than $\frac{\alpha}{2}$. Thus, the total fraction of one-entries is less than $\frac{\alpha}{2} + \frac{\alpha}{2} = \alpha$, contradicting the assumed fraction of one-entries in the matrix. ■

Proof: (Of Lemma 23.4.3)
Given an $(\alpha, l) - protocol$ we would like to show an $(\frac{\sqrt{\alpha}}{2}, \frac{l}{2}) - protocol$.
Let P be an $(\alpha, l) - protocol$. Assume that it works for the set S of paths in $[w]^l$ whose density is $\alpha(S) = \alpha$. We view the paths in S as two concatenated paths of half the length. Given $(s_1, ..., s_l)$ a path in S we denote $(s_1, ..., s_l)$ by $a \circ b$ where $a = (s_1, ..., s_{\frac{l}{2}})$, $b = (s_{\frac{l}{2}+1}, ..., s_l)$.

For any $a \in [w]^{\frac{l}{2}}$ we denote by $Suffix(a)$ the set of possible suffixes b for a, that form a path in S:

$$Suffix(a) = \{b \in [w]^{\frac{l}{2}} \mid a \circ b \in S\}$$

Consider a matrix whose rows and columns correspond to paths in $[w]^{\frac{l}{2}}$. An entry of the matrix (a, b) is 1 if the path $a \circ b$ is in S and 0 otherwise. Thus, the fraction of one-entries in the matrix is α (the density of S). Applying Claim 23.4.4 to the matrix, we get that this matrix satisfies either (1) or (2). Either there exists a prefix of a path in S that has a lot of suffixes, or there exist many prefixes of paths in S that have quite a lot of suffixes. In the first case we use the set of suffixes as the new set for which we build a new protocol, while in the second case we use the set of 'heavy' prefixes as the new set. In both cases we adapt the protocol P to work for the new set of half length paths. Details follow:

1. In case there is a prefix of a path a with at least $\sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$ suffixes, we let $S' = Suffix(a)$ be the set of suffixes of that prefix, and the new protocol P' will work as following:

   Player I gets an input x in S', he concatenates it to a forming the path $a \circ x$ in S. In a similar way Player II forms the path $a \circ y$. Now they have paths in S and can apply the protocol P to find the fork point. Since the paths coincide in their first halves, the fork point must be in the second half.

   Note that if the first coordinate in x and y is different, the fork in the whole path can be found in the last coordinate of a. This is the case where for S' the fork was found in the point of origin of the paths, s.

2. In case there are many (i.e. at least $\sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$) prefixes of paths with at least $\frac{\alpha}{2}$ suffixes we take S' to be the set of all the prefixes of 'dense' paths; that is, $S' = \{x : |Suffix(x)| \geq \frac{\alpha}{2} \cdot w^{\frac{l}{2}}\}$. We have $|S'| \geq \sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}}$. For each possible input x in S' we will try and build two suffixes $b_1(x)$ and $b_2(x)$ such that for any two inputs x and y the suffixes $b_1(x)$ and $b_2(y)$ will not coincide. In this case a fork found between $x \circ b_1(x)$ and $y \circ b_2(y)$ must be in the first half of the path, since the second half is ensured not to coincide in any point.

   We suggest a method for building the suffixes (for simplicity we assume that w is even):

   For each layer $\frac{l}{2} + 1, ..., l$ we color half the nodes in the layer in orange and the other half in purple. If for every x, the suffix $b_1(x)$ will be colored orange and the suffix $b_2(x)$ will be colored purple the goal will be fulfilled. We wi ll show that such a coloring exists by showing that the probability for such a coloring over all random colorings is positive.

Claim 23.4.5 *There exists a coloring of all nodes such that for most $a \in S'$ there are suffixes $b_1(a)$ and $b_2(a)$ such that:*

- $a \circ b_1(a) \in S$ *and* $a \circ b_2(a) \in S$
- *all nodes in* $b_1(a)$ *are colored orange*
- *all nodes in* $b_2(a)$ *are colored purple*

We propose the following way to color the vertices in layers $\frac{l}{2}+1, ..., l$:

- Choose randomly $\frac{w}{2}$ paths $r_1, ..., r_{\frac{w}{2}}$ in $[w]^{\frac{l}{2}}$ and color all the vertices appearing in them in orange.
- In layers where the paths covered less than $\frac{w}{2}$ of the vertices choose randomly orange vertices to achieve $\frac{w}{2}$ vertices colored in orange.
- Color the rest of the vertices in purple.

By symmetry it is apparent that this method for coloring produces the same distribution as uniformly choosing $\frac{w}{2}$ nodes out of the w possible nodes in each layer (coloring them in orange and the rest in purple).

We shall show that for any a in S' there is a high probability that two such suffixes (i.e., one colored orange and one colored purple) can be found. Define S'' as the set of all prefixes a in S' that have two such suffixes. The expected size of S'' will be close to the size of S'. Therefore exists a coloring that induces such a set S'', whose size is very close to the size of S'.

For a path $a \in S'$, since the density of suffixes of a is at least $\frac{\alpha}{2}$, the probability that for some i, the random path r_i (chosen in the above process) is not a suffix of a is at most $(1-\frac{\alpha}{2})$. Since these paths are chosen independently and $\alpha > \frac{12}{w}$, the probability that none of the $\frac{w}{2}$ paths is a suffix of a is at most 0.05. This is because

$$Prob[r_i \notin Suffix(a)] \leq (1-\tfrac{\alpha}{2})$$

$$Prob[\forall i,\ r_i \notin Suffix(a)] \leq (1-\tfrac{\alpha}{2})^{\frac{w}{2}} < (1-\tfrac{6}{w})^{\frac{w}{2}} \approx e^{-3} \approx 0.049$$

There is no difference between the probability that the first suffix (i.e. the one colored orange) does not exist and the probability that the second suffix (i.e. the one colored purple) does not exist. Hence, we can use union bound to determine that the probability of either $b_1(a)$ or $b_2(a)$ not to exist, is smaller than 0.1.

Therefore, the expected size of S'' is at least 0.9 of the size of S'. Formally:

Define for every element a in S' a random variable $X_a = \begin{cases} 1 & \text{if } a \text{ has two such suffixes} \\ 0 & \text{otherwise} \end{cases}$

Obviously $|S''| = \Sigma_{a \in S'} X_a$ hence:

$$E(|S''|) = E(\Sigma_{a \in S'} X_a) = \Sigma_{a \in S'} E(X_a) = |S'| \cdot E(X_a) \geq 0.9 \cdot |S'|$$

So we have a set S'' of density at least $0.9 \cdot \sqrt{\frac{\alpha}{2}} \cdot w^{\frac{l}{2}} > \frac{\sqrt{\alpha}}{2} \cdot w^{\frac{l}{2}}$ such that for any path a of length $\frac{l}{2}$ in S'' we have two suffixes $b_1(a)$ and $b_2(a)$ colored orange and purple, respectively.

Player I will get an input x in $[w]^{\frac{l}{2}}$ and concatenate $b_1(x)$ to it, creating $x \circ b_1(x)$, a path in S. Player II, getting y, will create in a similar way $y \circ b_2(y)$. The two players will run the protocol P and find the desired fork point.

Note that if the last coordinate in x and y is equal, the fork can be found there. This is legitimate in S'' because the end points of the paths t_1 and t_2 are different.

We have seen that in both cases we were able to find subsets of $[w]^{\frac{l}{2}}$ of density at least $\frac{\sqrt{\alpha}}{2}$ for which we adapted the protocol P. Therefore we have proven:

$$CC(\alpha, l) \geq CC(\tfrac{\sqrt{\alpha}}{2}, \tfrac{l}{2})$$

■

23.4.4 Applying the lemmas to get the lower bound

We show that any protocol for solving the FORK game uses at least $\Omega(log_2 w \cdot log_2 l)$ bits of communication by employing Lemmas 23.4.1 and 23.4.3. From Lemma 23.4.2 we know that during all the transformat ions, we do not find a zero-bit protocol. For the simplicity of calculations we assume that l is a power of 2.

We start by reducing FORK to any subset of density $\frac{2}{\sqrt{w}}$.

$$CC(1, l) \geq CC(\tfrac{2}{\sqrt{w}}, l)$$

Using Lemma 23.4.1 by a series of $\Omega(log_2 w)$ transformations (all allowed by Lemma 23.4.2) we get:

$$CC(\tfrac{2}{\sqrt{w}}, l) \geq CC(\tfrac{16}{w}, l) + \Omega(log_2 w)$$

Using Lemma 23.4.3, we have

$$CC(\frac{16}{w}, l) \geq CC(\frac{2}{\sqrt{w}}, \frac{l}{2})$$

and so

$$CC(\frac{2}{\sqrt{w}}, l) \geq CC(\frac{2}{\sqrt{w}}, \frac{l}{2}) + \omega(log_2 w)$$

Iterate the last two steps $log_2 l$ times, getting

$$CC(1, l) = \Omega(log_2 l \cdot log_2 w)$$

We conclude that the communication complexity of the FORK game is $\Theta(log_2 l \cdot log_2 w)$.

Bibliographic Notes

This lecture is based on [1].

1. M. Gringi and M. Sipser. Monotone Separation of Logarithmic Space from Logarithmic Depth. *JCSS*, Vol. 50, pages 433–437, 1995.

Lecture 24

Average Case Complexity

Notes taken by Tzvika Hartman and Hillel Kugler

Summary: We introduce a theory of average case complexity. We define the notion of a distribution function and the classes P-computable and P-samplable of distributions. We prove that P-computable $\subset$ P-samplable (strict containment). The class DistNP, which is the distributional analogue of NP, is defined. We introduce the definition of polynomial on the average and discuss the weaknesses of an alternative definition. The notion of reductions between distributional problems is presented. Finally, we prove the existence of a problem that is complete for DistNP.

24.1 Introduction

Traditionally, in theoretical computer science, the emphasis of the research is on the worst case complexity of problems. However, one may think that the more natural (and practical) way to measure the complexity of a problem is by considering its average case complexity. Many important problems were found to be NP-complete and there is little hope to solve them efficiently in the worst case. In these cases it would be useful if we could develop algorithms that solve them efficiently on the average. This is the main motivation for the theory of average case complexity.

When discussing the average case complexity of a problem we must specify the distribution from which the instances of the problem are taken. It is possible that the same problem is efficiently solvable on the average with respect to one distribution, but hard on the average with respect to another. One may think that it is enough to consider the most natural distribution - the uniform one. However, in many cases it is more realistic to assume settings in which some instances are more probable than others (e.g. some graph problems are most interesting on dense graphs, hence, the uniform distribution is not relevant).

It is interesting to compare the average case complexity theory with cryptography theory, since they deal with similar issues. One difference between the two theories is that in cryptography we deal with problems that are difficult on the average, while in the average case theory we try to find problems that are easy on the average. Another difference is that in cryptography we need problems for which it is possible to generate efficiently instance-solution pairs such that solving the problem given only the instance is hard. In contrast, this property is not required in average case complexity theory.

24.2 Definitions

24.2.1 Distributions

We now introduce the notion of a distribution function. We assume a canonical order of the binary strings (e.g. the standard lexicographic order). The notation $x < y$ means that the string x precedes y in this order and $x - 1$ denotes the immediate predecessor of x.

Definition 24.1 (Probability Distribution Function) A *distribution function* $\mu : \{0,1\}^* \to [0,1]$ is a non-negative and non-decreasing function (i.e., $\mu(0) \geq 0, \mu(x) \leq \mu(y)$ for each $x < y$) from strings to the unit interval $[0,1]$ which converges to one (i.e., $lim_{x\to\infty}\mu(x) = 1$). The *density function* associated with the distribution function μ is denoted by μ' and defined by $\mu'(0) = \mu(0)$ and $\mu'(x) = \mu(x) - \mu(x-1)$ for every $x > 0$.

Clearly, $\mu(x) = \sum_{y\leq x} \mu'(y)$. Notice that we defined a single distribution on all inputs of all sizes, rather than ensembles of finite distributions (each ranging over fixed length strings). This makes the definition robust under different representations. An important example is the uniform distribution function defined by $\mu'_u \stackrel{\text{def}}{=} \frac{1}{|x|^2} \cdot 2^{-|x|}$. This density function converges to some constant different than 1. A minor modification, defining $\mu'_u \stackrel{\text{def}}{=} \frac{1}{|x|\cdot(|x+1|)} \cdot 2^{-|x|}$, settles this problem:

$$\sum_{x\in\{0,1\}^*} \mu'_u(x) = \sum_{x\in\{0,1\}^*} \frac{1}{|x|\cdot|x|+1} \cdot 2^{-|x|} = \sum_{n\in N} \sum_{x\in\{0,1\}^n} (\frac{1}{n} - \frac{1}{n+1}) \cdot 2^{-n}$$

$$= \sum_{n\in N} (\frac{1}{n} - \frac{1}{n+1}) = 1$$

We will use a notation for the probability mass of a string relative to all strings of equal size: $\mu'_n(x) \stackrel{\text{def}}{=} \frac{\mu'(x)}{\sum_{|y|=|x|} \mu'(y)}$.

24.2.2 Distributional Problems

Average case complexity is meaningful only if we associate a problem with a specific distribution of its instances. We will consider only decision problems. Similar formulations for search problems can be easily derived.

Definition 24.2 (Distributional Problem) A *distributional decision problem* is a pair (D, μ), where $D : \{0,1\}^* \to \{0,1\}$ and $\mu : \{0,1\}^* \to [0,1]$ is a distribution function.

24.2.3 Distributional Classes

Before defining classes of distributional problems we should consider classes of distributions. It is important to restrict the distributions, otherwise, the whole theory collapses to the worst case complexity theory (by choosing distributions that put all the probability mass on the worst cases). We will consider only "simple" distributions in a computational sense.

Definition 24.3 (P-samplable) A distribution μ is in the class *P-samplable* if there is a probabilistic Turing machine that gets no input and outputs a binary string x with probability $\mu'(x)$, while running in time polynomial in $|x|$.

Definition 24.4 (P-computable) A distribution μ is in the class *P-computable* if there is a deterministic polynomial time Turing machine that on input x outputs the binary expansion of $\mu(x)$.

Interesting distributions must put noticeable probability mass on long strings (i.e., at least $\frac{1}{poly(n)}$ on strings of length n). Consider to the contrary the density function $\mu'(x) \stackrel{\text{def}}{=} 2^{-3|x|}$. An algorithm of exponential running time, $t(x) = 2^{|x|}$, will be considered to have constant on the average running time with respect to this distribution (since $\sum_x \mu'(x) \cdot t(|x|) = \sum_n 2^{-n} = 1$). Intuitively, this distribution does not make sense since usually the long instances are the difficult ones. By assigning negligible probability to these long instances, we can artificially make the average running time of the algorithm small, even though the algorithm is not efficient. Consider, for example, an extreme case in which all instances of size greater than some constant have zero probability. In this case, every algorithm has average constant running time.

We now show that the uniform distribution is P-computable. For every x,

$$\mu_u(x) = \sum_{y \leq x} \mu'_u(y) = \sum_{\substack{y \leq x \\ |y|<|x|}} \mu'_u(y) + \sum_{\substack{y \leq x \\ |y|=|x|}} \mu'_u(y) = \left(1 - \frac{1}{|x|}\right) + \left(N_x \cdot \frac{1}{|x|(|x|+1)} \cdot 2^{-|x|}\right)$$

where $N_x \stackrel{\text{def}}{=} |\{y \in \{0,1\}^{|x|} : y \leq x\}| = 1 + \sum_{i=1}^{|x|} 2^{i-1} \cdot x_i$, where $x = x_n \cdots x_1$. Obviously, this expression can be computed in time polynomial in the length of x.

Proposition 24.2.1 *P-computable $\subset$ P-samplable (strict containment assuming $\#P \neq P$)*

Proof: We prove the proposition in two steps:

1. **Claim 24.2.2** *For every distribution μ, if $\mu \in$ P-computable then $\mu \in$ P-samplable.*

   Proof: Let μ be a distribution that is P-computable. We describe an algorithm that samples strings according to the distribution μ, assuming that we can compute μ in polynomial time. Intuitively, we can view the algorithm as picking a random real number r in $[0,1)$ and checking which string $x \in \{0,1\}^*$ satisfies $\mu(x-1) < r \leq \mu(x)$. Unfortunately, this is not possible since r has infinite precision. To overcome this problem, we select randomly in each step one bit of the expansion (of r) and stop when we are guaranteed that there is a unique x satisfying the requirement above (i.e., every possible extension will yield the same x).

   The algorithm is an iterative procedure with a stopping condition. In each iteration we select one bit. We view the bits selected so far as the binary expansion of a truncated real number, denoted by t. Now we find the smallest n such that $\mu(1^n) \geq t$. By performing a binary search over all binary strings of length n we find the greatest $x \in \{0,1\}^n$ such that $\mu(x) < t$. At this point we check if $\mu(x+1) \geq t + 2^{-l}$, where l is the length of the binary expansion of t. If so, we halt and output $x+1$. Otherwise, we continue to the next iteration. Obviously, this can be implemented in polynomial time. ■

2. **Claim 24.2.3** *There exists a distribution which is P-samplable but not P-computable, under the assumption $\#P \neq P$.*

   Proof: Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$ be an NP relation (we assume for simplicity that R contains only pairs of strings with the equal lengths). We define a distribution function:

$$\mu'(x \cdot \sigma \cdot y) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } R(x,y) \neq \sigma \\ \frac{1}{|x|^2 \cdot |y|^2 \cdot 2^{|x|} \cdot 2^{|y|}} & \text{otherwise} \end{cases}$$

for every $x, y \in \{0,1\}^*$ of equal length and $\sigma \in \{0,1\}$. The fact that this distribution function converges to a constant can be easily verified using the fact that for every x, y exactly one of the possible values of σ gives a non-zero probability.

We now show that μ is P-samplable. The algorithm samples uniformly a string of length $2n$, denoted $x \cdot y$, where $|x| = |y| = n$ (recall that the uniform distribution is P-computable and thus P-samplable). Next, σ is defined as $R(x, y)$, which is an NP-relation and therefore can be computed efficiently. The algorithm outputs $x \cdot \sigma \cdot y$.

The next step is to show that if μ is P-computable then #P problems can be solved in polynomial time. The number of NP witnesses for x is given by the following expression:

$$\frac{\mu(x \cdot 1 \cdot 1^n) - \mu(x \cdot 0 \cdot 1^n)}{\frac{1}{n^4 \cdot 2^{2n}}} = (\mu(x \cdot 1 \cdot 1^n) - \mu(x \cdot 0 \cdot 1^n)) \cdot n^4 \cdot 2^{2n}$$

The numerator is the probability mass of all strings that end with an NPwitness for x. By normalizing we get the actual number of witnesses, thus solving the #P problem. ■

The proposition follows directly from the two claims. ■

In the sequel we will focus on the P-computable class.

24.2.4 Distributional-NP

We now define the average-case analogue to the class NP:

Definition 24.5 (The class DistNP) A distributional problem (D, μ) belongs to the class *DistNP* if D is an NP-predicate and μ is P-computable. *DistNP* is also denoted $\langle \mathcal{NP}, \mathcal{P} - computable \rangle$.

The class $\langle \mathcal{NP}, \mathcal{P} - samplable \rangle$ is defined similarly.

24.2.5 Average Polynomial Time

The following definition may seem obscure at first glance. In the appendix we discuss the weaknesses of alternative naive formulation.

Definition 24.6 (Polynomial on the Average) A problem D is *polynomial on the average* with respect to a distribution μ if there exists an algorithm A that solves D in time $t_A(\cdot)$ and there exists a constant $\epsilon > 0$ such that

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{t_A(x)^\epsilon}{|x|} < \infty$$

A necessary property that a valid definition should have is that a function that is polynomial in the worst case should be polynomial on the average. Assume that x^d bounds the running-time of the problem and let $\epsilon = \frac{1}{d+1}$. This function is polynomial on the average (with respect to any μ) according to Definition 6:

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{t_A(x)^\epsilon}{|x|} \leq \sum_{x \in \{0,1\}^*} \mu'(x) \cdot \frac{|x|^{\frac{d}{d+1}}}{|x|} \leq \sum_{x \in \{0,1\}^*} \mu'(x) = 1 < \infty$$

We will now try to give some intuition for Definition 6. A natural definition for the notion of a function $f(\cdot)$ that is "constant on the average" with respect to the distribution μ is requiring

$$\sum_{x \in \{0,1\}^*} \mu'(x) \cdot f(x) < \infty$$

Using this definition, $g(\cdot)$ is called "linear on the average" if $g(x) = O(f(x) \cdot |x|)$ where $f(\cdot)$ is constant on the average. This implies

$$\sum_{x \in \{0,1\}^*} \mu'(x) \frac{g(x)}{|x|} < \infty$$

A natural extension of this definition for the case of polynomial on the average yields Definition 6.

24.2.6 Reductions

We now introduce the definition of a reduction of one distributional problem to another. In the worst case reductions, the two requirements are efficiency and validity. In the distributional case we also require that the reduction "preserve" the probability distribution. The purpose of the last requirement is to ensure that the reduction does not map very likely instances of the first problem to rare instances of the second problem. Otherwise, having a polynomial time on the average algorithm for the second distributional problem does not necessary yield such an algorithm for the first distributional problem. This requirement is captured by the domination condition.

Definition 24.7 (Average Case Reduction) We say that the distributional problem (D_1, μ_1) reduces to (D_2, μ_2) (denote $(D_1, \mu_1) \propto (D_2, \mu_2)$) if there exists a polynomial time computable function f such that

1. *validity*: $x \in D_1$ iff $f(x) \in D_2$.
2. *domination*: There exists a constant $c > 0$ such that for every $y \in \{0,1\}^*$,

$$\sum_{x \in f^{-1}(y)} \mu_1'(x) \le |y|^c \cdot \mu_2'(y)$$

The following proposition shows that the reduction defined above is adequate:

Proposition 24.2.4 *If $(D_1, \mu_1) \propto (D_2, \mu_2)$ and (D_2, μ_2) is polynomial on the average then so is (D_1, μ_1).*

See proof in Appendix B.

24.3 DistNP-completeness

In this section we state two theorems regarding DistNP-complete problems and prove the first one.

Theorem 24.8 *There exists a DistNP-complete problem.*

Theorem 24.9 *Every problem complete for DistNP is also complete for $\langle \mathcal{NP}, \mathcal{P} - sampable \rangle$.*

Proof: (of first theorem) We first define a distributional version of the *Bounded Halting* problem (for a discussion on the *Bounded Halting* problem see Lecture 2.3). We then show that it is in DistNP-complete.

Definition 24.10 (Distributional Bounded Halting)

1. *Decision:* $BH(M, x, 1^k) = 1$ iff there exists a computation of the non-deterministic machine M on input x which halts within k steps.
2. *Distribution:*
$$\mu'_{BH}(M, x, 1^k) \stackrel{\text{def}}{=} \frac{1}{|M|^2 \cdot 2^{|M|}} \cdot \frac{1}{|x|^2 \cdot 2^{|x|}} \cdot \frac{1}{k^2}$$

Proving completeness results for distributional problems is more complicated than usual. The difficulty is that we have to reduce all DistNP problems with different distributions to one single distributional problem with a specific distribution. In the worst case version we used the reduction $x \to (M_D, x, 1^{P_D(|x|)})$, where D is the NP problem we want to reduce and M_D is the non-deterministic machine that solves D in time $P_D(n)$ on inputs of length n (see Lecture 2.3). A first attempt is to use exactly this reduction. This reduction is valid for every DistNP problem, but for some distributions it violates the domination condition. Consider for example distributional problems in which the distribution of (infinitely many) strings is much higher than the distribution assigned to them by the uniform distribution. In such cases, the standard reduction maps an instance x having probability mass $\mu'(x) \gg 2^{-|x|}$ to a triple $(M_D, x, 1^{P_D(|x|)})$ with much lighter probability mass (recall that $\mu'_{BH}(M_D, x, 1^{P_D(|x|)}) < 2^{-|x|}$). Thus the domination condition is not satisfied.

The essence of the problem is that μ'_{BH} gives low probability to long strings, whereas an arbitrary distribution can give them high probability. To overcome this problem, we will map long strings with high probability to short strings, which get high probability in μ'_{BH}. We will use an encoding of strings which maps a string x into a code of length bounded above by $\log_2 \frac{1}{\mu'(x)}$. We will use the following technical coding lemma:

Lemma 24.3.1 (Coding Lemma): *Let μ be a polynomial-time computable distribution function (i.e., $\mu \in \mathcal{P} - computable$). Then there exists a coding function C_μ satisfying the following three conditions:*

1. Efficient encoding: *The function C_μ is computable in polynomial time.*
2. Unique decoding: *The function C_μ is one-to-one.*
3. Compression: *For every x*
$$|C_\mu(x)| \leq 1 + min\{|x|, \log_2 \frac{1}{\mu'(x)}\}$$

Proof: The function C_μ is defined as follows:

$$C_\mu(x) \stackrel{\text{def}}{=} \begin{cases} 0 \cdot x & \text{if } \mu'(x) \leq 2^{-|x|} \\ 1 \cdot z & \text{otherwise} \end{cases}$$

where z is the longest common prefix of the binary expansions of $\mu(x-1)$ and $\mu(x)$ (e.g., if $\mu(1010) = 0.10000$ and $\mu(1011) = 0.10101$ then $C_\mu(1011) = 110$). The intuition behind this definition is that we want to find uniquely a point with certain precision in the interval between $\mu(x-1)$ and $\mu(x)$. As the length of the interval grows, the precision needed is lower. Recall that the length of the interval is exactly the value of $\mu'(x)$, implying the short encoding of high probability strings.

We now show that $C_\mu(x)$ satisfies the conditions of the lemma:

1. *Efficient encoding:* The efficiency of the encoding follows from the fact that μ is a polynomial time computable function.

2. *Unique decoding:* In the first case, in which $C_\mu(x) = 0 \cdot x$, the unique decoding is obvious. In the second case, in which $C_\mu(x) = 1 \cdot z$, since the intervals are disjoint and $\mu(x-1) < 0.z1 \leq \mu(x)$, every z determines uniquely the encoded x, and the unique decoding conditions follows.

3. *Compression:* In the first case, in which $\mu'(x) \leq 2^{-|x|}$, $|C_\mu(x)| = 1 + |x| \leq 1 + \log_2 \frac{1}{\mu'(x)}$. In the second case, in which $\mu'(x) > 2^{-|x|}$, let $l = |z|$ and $z_1 \cdots z_l$ be the binary representation of z. Then,

$$\mu'(x) = \mu(x) - \mu(x-1) \leq \left(\sum_{i=1}^{l} 2^{-i} z_i + \sum_{i=l+1}^{poly(|x|)} 2^{-i} \right) - \sum_{i=1}^{l} 2^{-i} z_i < 2^{-|z|}$$

   So $|z| \leq \log_2 \frac{1}{\mu'(x)}$, and the compression condition follows.

■

Now we use the Coding Lemma to complete the proof of the theorem. We define the following reduction of (D, μ) to (BH, μ_{BH}):

$$x \rightarrow (M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)})$$

where $M_{D,\mu}$ is a non-deterministic machine that on input y guesses non-deterministically x such that $C_\mu(x) = y$ (notice that the non-determinism allows us not to require efficient decoding), and then runs M_D on x. The polynomial $P_{D,\mu}(n)$ is defined as $P_D(n) + P_C(n) + n$, where $P_D(n)$ is a polynomial bounding the running time of M_D on acceptable inputs of length n and $P_C(n)$ is a polynomial bounding the running time of the encoding algorithm.

It remains to show that this reduction satisfies the three requirements.

1. *Efficiency:* The description of $M_{D,\mu}$ is of fixed length and by the coding lemma C_μ is computable by polynomial time. Therefore, the reduction is efficient.

2. *Validity:* By construction of $M_{D,\mu}$ it follows that $D(x) = 1$ if and only if there exists a computation of machine $M_{D,\mu}$ that on input $C_\mu(x)$ halts with output 1 within $P_{D,\mu}(|x|)$ steps.

3. *Domination:* Notice that it suffices to consider instances of *Bounded Halting* which have a preimage under the reduction. Since the coding is one-to-one, each such image has a unique preimage. By the definition of μ_{BH},

$$\mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)}) = c \cdot \frac{1}{P_{D,\mu}(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2 \cdot 2^{|c_\mu(x)|}}$$

where $c = \frac{1}{|M_{D,\mu}|^2 \cdot 2^{|M_{D,\mu}|}}$ is a constant independent of x. By the compression requirement of the coding lemma,

$$\mu'(x) \leq 2 \cdot 2^{-|C_\mu(x)|}$$

Hence,

$$\mu'_{BH}(M_{D,\mu}, C_\mu(x), 1^{P_{D,\mu}(|x|)}) \geq c \cdot \frac{1}{P_{D,\mu}(|x|)^2} \cdot \frac{1}{|C_\mu(x)|^2} \cdot \frac{\mu'(x)}{2} > \frac{c}{2 \cdot P_{D,\mu}(|x|)^2 \cdot |C_\mu(x)|^2} \cdot \mu'(x)$$

Therefore, the reduction satisfies the requirements and the distributional version of the *Bounded Halting* problem is DistNP-complete.

■

Bibliographic Notes

The theory of average-case complexity was initiated by Levin [4]. Theorem 24.9 is due to [3]. The lecture was based on the exposition in [2]. For further investigations see [1].

1. S. Ben-David, B. Chor, O. Goldreich, and M. Luby. On the Theory of Average Case Complexity. *Journal of Computer and system Sciences*, Vol. 44, No. 2, April 1992, pp. 193–219.

2. O. Goldreich. Notes on Levin's Theory of Average-Case Complexity. *ECCC*, TR97-058, Dec. 1997.

3. R. Impagliazzo and L.A. Levin. No Better Ways to Generate Hard NP Instances than Picking Uniformly at Random. In *Proc. of the 31st FOCS*, 1990, pp. 812–821.

4. L.A. Levin. Average Case Complete Problems. *SIAM Jour. of Computing*, Vol. 15, pages 285–286, 1986.

Appendix A : Failure of a naive formulation

We now discuss an alternative definition of the notion of polynomial on the average which seems more natural:

Definition 24.11 (Naive Formulation of Polynomial on the Average) A problem D is *polynomial on the average* with respect to a distribution μ if there exists an algorithm A that solves D in time $t_A(\cdot)$ and there exists a constant $c > 0$ such that for every n

$$\sum_{x \in \{0,1\}^n} \mu'_n(x) \cdot t_A(x) < n^c$$

There are three main problems with this naive definition:

1. This definition is very dependent on the particular encoding of the problem instance. In this definition, the average is taken over all instances of equal length. Changing the encoding of the problem instances does not preserve the partition of instances according to their length and hence does not preserve the average running time of the algorithm.

2. This definition is not robust under functional composition of algorithms. Namely, if distributional problem A can be solved in average polynomial time given access to an oracle for distributional problem B and B can be solved in average polynomial time, then it does not follow that A can be solved in average polynomial time.

3. This definition is not machine independent, i.e., an algorithm can be polynomial on the average in one reasonable computational model, but hard on the average in another (e.g. the simulation of a two-tape Turing machine on a one-tape Turing machine).

The two last problems stem from the fact that the definition is not closed under application of polynomials. For example, consider a function $t(x)$ defined as follows:

$$t(x) \stackrel{\text{def}}{=} \begin{cases} 2^n & \text{if } x = 1^n \\ n^2 & \text{otherwise} \end{cases}$$

This function is clearly polynomial on the average with respect to the uniform distribution (i.e., for every $x \in \{0,1\}^n, \mu'_n(x) = 2^{-n}$).

This is true since

$$\sum_{x\in\{0,1\}^n} \mu'_n(x) \cdot t(x) = 2^{-n} \cdot 2^n + (1-2^{-n}) \cdot n^2 < n^2 + 1$$

On the other hand,

$$\sum_{x\in\{0,1\}^n} \mu'_n(x) \cdot t^2(x) = 2^{-n} \cdot 2^{2n} + (1-2^{-n}) \cdot n^4 > 2^n$$

which implies that the function $t^2(x)$ is not polynomial on the average. This problem does not occur in Definition 6 since if $t(x)$ is polynomial on the average with the constant ϵ then $t^2(x)$ is polynomial on the average with the constant $\frac{\epsilon}{2}$.

Appendix B : Proof Sketch of Proposition 24.2.4

In this appendix we sketch the proof of proposition 2.4. We first restate the proposition:

Proposition 2.4 : If $(D_1, \mu_1) \propto (D_2, \mu_2)$ and (D_2, μ_2) is polynomial on the average then so is (D_1, μ_1).

The formal proof of the proposition has many technical details, so it will be only sketched. As a warm-up, we will first prove the proposition under some simplifying assumptions. To our belief, this gives intuition to the full proof. The simplifying assumptions, regarding the definition of a reduction, are:

1. There exists a constant c_1 such that for every x, $|f(x)| \leq c_1 \cdot |x|$, where f is the reduction function.

2. Strong domination condition: There exists a constant c_2 such that for every $y \in \{0,1\}^*$,

$$\sum_{x\in f^{-1}(y)} \mu'_1(x) \leq c_2 \cdot \mu'_2(y)$$

Proof: (*of simplified version*) The distributional problem (D_2, μ_2) is polynomial on the average implies that there exists an algorithm A_2 and a constant $\epsilon_2 > 0$ such that

$$(1) \qquad \sum_{x\in\{0,1\}^*} \mu_2'(x) \cdot \frac{t_{A_2}(x)^{\epsilon_2}}{|x|} < \infty$$

We need to prove that (D_1, μ_1) is polynomial on the average, i.e. that there exists an algorithm A_1 and a constant $\epsilon_1 > 0$ such that

$$(2) \qquad \sum_{x\in\{0,1\}^*} \mu_1'(x) \cdot \frac{t_{A_1}(x)^{\epsilon_1}}{|x|} < \infty$$

The algorithm A_1, when given x, applies the reduction f on x and then applies A_2 on $f(x)$. Therefore, $t_{A_1}(x) = t_f(x) + t_{A_2}(f(x))$, where t_f denotes the time required to compute f. For the sake of simplicity we ignore $t_f(x)$ and assume $t_{A_1}(x) = t_{A_2}(f(x))$. Taking $\epsilon_1 \stackrel{\text{def}}{=} \epsilon_2$ we obtain:

$$\begin{aligned}
\sum_{x\in\{0,1\}^*} \frac{\mu_1'(x)\cdot t_{A_1}(x)^{\epsilon_1}}{|x|} &= \sum_{y\in\{0,1\}^*}\sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} \\
&\le c_1 \cdot \sum_{y\in\{0,1\}^*} \frac{t_{A_2}(y)^{\epsilon_1}}{|y|} \sum_{x\in f^{-1}(y)} \mu_1'(x) \\
&\le c_1 \cdot c_2 \cdot \sum_{y\in\{0,1\}^*} \frac{t_{A_2}(y)^{\epsilon_2}\cdot \mu_2'(y)}{|y|} < \infty
\end{aligned}$$

The first inequality uses Assumption 1, and the second uses the simplified domination condition (Assumption 2). ■

Sketch of full proof: First we explain how to deal with the technical problem that arises when considering the running time of the reduction, $t_f(x)$, in the running time of algorithm A_1. This technical problem arises many times in the proof and is solved as described below. The fact that for every $\epsilon \le 1$, $(a+b)^\epsilon \le a^\epsilon + b^\epsilon$ is used to bound expression (2) by two sums. The first sum contains the factor $t_f(x)$ and can be easily bounded by choosing the appropriate ϵ. The second sum converges as shown in the proof of the simplified version.

In order to prove that expression (2) converges we partition the sum in expression (2) into two sums and show separately that each sum converges. Formally, $y \in \{0,1\}^*$ is called a *bad* y if $\mu_2'(y) \cdot t_{A_2}(y)^{\epsilon_2} \ge |y|$ and is called a *good* y otherwise. We partition the x's according to the "goodness" of their images under f. This induces the following partition into two sums.
Denote

$$B = \sum_{bad\ y}\sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} \qquad G = \sum_{good\ y}\sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|}$$

Then

$$\sum_{x\in\{0,1\}^*} \frac{\mu_1'(x)\cdot t_{A_1}(x)^{\epsilon_1}}{|x|} = \sum_{y}\sum_{x\in f^{-1}(y)} \frac{\mu_1'(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} = B + G$$

The intuition behind this partition is that each bad y contributes large weight (at least one) to the sum in expression (1). The fact that expression (1) converges implies that there is a finite number of bad y's. This is used to show that B converges. For the second sum we again partition it into two sums, G_1 and G_2. The first sum, G_1, consists of good y's for which $t_{A_2}(y)$ is bounded by $p(|x|)$ for every $x \in f^{-1}(x)$ and some polynomial p (that depends on ϵ_2 and the domination constant c). This sum can be bounded by choosing an ϵ_1 so that $t_{A_2}(y)^{\epsilon_1} < |x|$ for any $x \in f^{-1}(y)$. That is,

$$\sum_{y:t_{A_2}(y)<\min_{x\in f^{-1}(y)}\{p(|x|)\}} \sum_{x\in f^{-1}(y)} \frac{\mu'_1(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} < \sum_x \mu'_1(x) = 1$$

The second sum, G_2, consists of the rest of the good y's. For each y in this sum, we have $t_{A_2}(y) \geq q(y) \stackrel{\text{def}}{=} \min_{x\in f^{-1}(y)}\{p(|x|)\}$. Note that q grows at least as a power of p (depending on the relation of $|f(x)|$ to $|x|$); that is for some $\epsilon > 0$ we have $q(y) \geq p(|y|^{\epsilon})$. By a suitable choice of p and ϵ_1, we have

$$|y|^c \cdot t_{A_2}(y)^{\epsilon_1} = \frac{|y|^c \cdot t_{A_2}(y)^{\epsilon_2}}{p(|y|^{\epsilon})^{\epsilon_2-\epsilon_1}} < \frac{t_{A_2}(y)^{\epsilon_2}}{|y|}$$

Thus,

$$\begin{aligned}\sum_{y:t_{A_2}(y)\geq q(y)} \sum_{x\in f^{-1}(y)} \frac{\mu'_1(x)\cdot t_{A_2}(y)^{\epsilon_1}}{|x|} &\leq \sum_{y:t_{A_2}(y)\geq q(y)} |y|^c \mu'_2(y)\cdot t_{A_2}(y)^{\epsilon_1} \\ &\leq \sum_{y:t_{A_2}(y)\geq q(y)} \frac{\mu'_2(y)\cdot t_{A_2}(y)^{\epsilon_2}}{|y|}\end{aligned}$$

which converges being a partial sum of (1).

Finally, we choose ϵ_1 to be the minimum of all the ϵ_1's used in the proof to obtain the convergence of the original sum. ■

Lecture 25

Computational Learning Theory

Lecture given by Dana Ron
Notes taken by Oded Lachish and Eli Porat

Summary: We define a model of learning known as probably approximately correct (PAC) learning. We define efficient PAC learning, and present several efficient PAC learning algorithms. We prove the Occam's Razor Theorem, which reduces the PAC learning problem to the problem of finding a succinct representation for the values of a large number of given labeled examples.

25.1 Towards a definition of Computational learning

Learning is a notion we are familiar with from every day life. When embarking on the task of importing this notion into computer science, the first natural step is to open a dictionary and find an exact meaning for it. A natural meaning that can be found is "gaining knowledge through experience". With this meaning in mind, we set on the task of defining a formal computer science model for learning. In order to get a clue to what the model should look like, it is worthwhile to examine a real life setting.

Learning to diagnose a new disease. A medical doctor learns the symptoms for diagnosing a new disease by drawing a number of files, from a file archive. Each file contains a list of the patient's parameters such as weight, body temperature, age, etc., and a *label* indicating the diagnosis; that is, whether the person has this specific disease. The number of examples the doctor has drawn depends on how accurate he wanted his list of symptoms to be. Using these files he concludes a list of symptoms for diagnosing the new disease. In order to check the accuracy of symptoms he concluded, he draws a labeled file, uses the list of symptoms to reach a diagnosis, and finally whether the diagnosis obtained by him matches the true diagnosis provided in the file (i.e., the label of the file).

Using this setting we can phrase the process of learning as follows: In order to learn an unknown subject (disease) from a family of subjects (family of diseases) the learner (doctor) receives a measure of accuracy, with this measure in mind he draws a number of labeled examples (files) from the set of all possible examples (archive of files), labeled with respect to subject. Using the examples he reaches a rule (list of symptoms) for labeling examples. He checks his rule accuracy, by drawing an example, and comparing its label to the label computed by the rule for this example.

But something is still lacking in the above model. We demonstate this by using the above setting (of a doctor learning to diagnose a new disease): Can a doctor learn a new disease that occurs in a certain group of people, when he does not see a file of a person from this specific group? Of course we cannot hope for this to happen. Thus, the missing component in the above model is that the doctor's ability to diagnose will be tested against the same distribution of examples (files) based on which he has learned to diagnose. Thus, if in the learning stage the doctor didn't draw any file of a person from a specific group, then chances are that he won't be asked to diagnose such a person later (i.e., the the accuracy of symptoms he concluded will not be checked for such a person). We stress that we do not require the learner (doctor) to have any knowledge regarding the distribution of examples.

The components of the learning process: We call the object that conducts the learning process, the *learner*. In this lecture the learner is an algorithm. The objects on which the learning is done are called *instances*. The specifics of the instances do not interest us. We are only interested in an abstract representation of the parameters characterizing it. We represent an instance by a vector defined as follows: a value is assigned to each parameter of the instance, this vector is the representation of these values.

- $X = \bigcup_{n \geq 1} X_n$ Instance space – the set of all possible values of instance representation.
- $D : X_n \longrightarrow [0,1]$ Underlying distribution – the (unknown) probability distribution over example space.

where n is the number of parameters characterizing an instance. In this lecture the instance space will be either $\{0,1\}^n$ or R^n. The learning is done with respect to the distribution D, from which we obtain independent samples. We denote by $x \sim D$ an example x drawn from X_n according to distribution D.

The subject and aim of our learning is a *labeling* of the instance space, where by labeling we mean a correlation between the instance space and the a set values. In this lecture we use the set $\{0,1\}$ (or any set isomorphic to it) as possible labeling values. Each instance is labeled by such a value, and so the labeling (of all instances) is a function from X to $\{0,1\}$. Such a labeling is called a *concept*. We use the following notations

- $F = \bigcup_{n \geq 1} F_n$ concept class – family of concepts we may need to learn. This family is a subset of the set of all Boolean functions defined over X.
- $f \in F_n$ target concept – the concept which is the subject of the learning.

where n appears in notations since a concept is a function over the instance space X_n.

The rule that is the output of the learner is also a labeling of the instance space, therefore it is also a Boolean function. We call this function the *hypothesis*.

The final component missing is a measure of accuracy of the hypothesis. We actually consider the complementary measure; that is the error of the hypothesis. The latter is merely the probability that for an instance drawn according to D, which is the very distribution used in the learning process, the hypothesis agrees with the target concept. That is,

Definition 25.1 (hypothesis error): *$err_{f,D}(h)$ is the probability that the hypothesis h disagrees with the target function f on an instance x drawn from X_n according to D. That is,*

$$err_{f,D}(h) \stackrel{\text{def}}{=} \Pr_{x \sim D}[h(x) \neq f(x)]$$

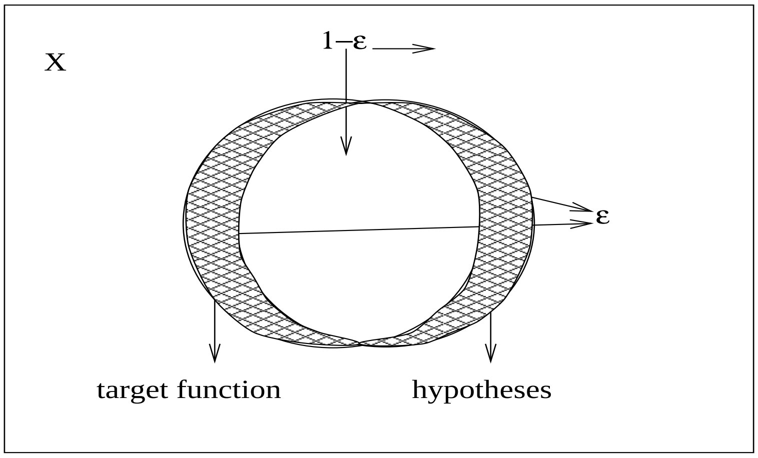


Figure 25.1: The target function and hypothesis with their difference shaded

25.2 Probably Approximately Correct (PAC) Learning

In this model the algorithm is only given partial information on the target concept, that is, a "small" number of examples drawn independently by a given distribution. Since the examples are chosen independently by a distribution, it could be the case that the algorithm was given the same example all the time. Therefore it is unconceivable to expect the algorithm to give an hypothesis h which is fully equivalent to the target function f. A realistic view to the problem would be to expect with a given probability that the algorithm will supply an hypothesis h that is an approximation of the target function f with respect to the underlying distribution. The following definitions capture these qualities:

Definition 25.2 (PAC learning algorithm): *An algorithm A is called a PAC* learning algorithm for a concept class *F if the following holds: For every n, for every function $f \in F_n$, for every distribution $D : X_n \longrightarrow [0,1]$, for every error parameter $0 < \epsilon < 1$ and for every confidence parameter $0 < \delta < 1$, given parameters n, ϵ, δ and a set of f-labeled examples, $\{< x^i, f(x^i) >\}$, where x^i is drawn independently under the distribution D, the algorithm outputs a hypothesis h such that:*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where the probability is taken over the choice of examples, as well as over the internal coin tosses of the algorithm. Note $h = A(n, \epsilon, \delta, \{< x^i, f(x^i) >\})$.

Once we established a formal model, a natural step would be to inquire upon following questions:

1. What is an efficient PAC learning algorithm?

2. How many examples do we need for PAC learning a concept from a concept class?

In order to deal with these questions we define complexity measures for the PAC learning model. A natural complexity measure is the number of examples needed:

Definition 25.3 (sample complexity): *The* sample complexity *of a PAC algorithm is the number of examples it utilizes, as a function of n, ϵ and δ.*

The classical measure of running time of an algorithm is also applicable for this model. The running time complexity is necessarily larger then the sample complexity. In some cases it appears to be substaintially larger.

> Oded's Note: *That is, under reasonable complexity assumptions (e.g., existence of one-way functions), there are concept classes that can be PAC learned using* $\mathrm{poly}(n, \frac{1}{\epsilon}, \log\frac{1}{\delta})$ *examples, but cannot be PAC learned in* $\mathrm{poly}(n, \frac{1}{\epsilon}, \frac{1}{\delta})$*-time. (In analogy to other contexts, we should expect complexities to be logarithmic in* $1/\delta$*.)*

Using these complexity measures we can answer the first question. We say that a PAC learning algorithm is efficient if it runs in time polynomial in n, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and $size(f)$, where the size of $f \in F_n$ is usually polynomial in n. In the rest of this lecture, we focus on the second question (i.e., consider only the sample complexity of PAC algorithms).

Learning axis-aligned rectangles over $[0,1]^2$**.** We wish to design an efficient PAC learning algorithm for axis-aligned rectangles over $[0,1]^2$.

Let us first explicitly cast the problem according to the PAC learning model.

- An instance is a point in $[0,1]^2$, we represent it by its x,y axis coordinates, that is $< x, y >$.
- The instance space is $X = \{\text{all } < x, y > \text{ representations of points in } [0,1]^2\}$.
- A concept is an axis-aligned rectangle in $[0,1]^2$, represented by four points $< x_{min}, y_{min}, x_{min}, y_{max} >$ such that:
$$x_{min}, x_{max}, y_{min}, y_{max} \in [0,1]$$
$$x_{min} \le x_{max} \text{ and } y_{min} \le y_{max}$$
An instance $< x, y >$ is in the axis aligned rectangle f represented by $< x_{min}, y_{min}, x_{min}, y_{max} >$ if:
$$x_{min} \le x \le x_{max} \text{ and } y_{min} \le y \le y_{max}$$
If $< x, y >$ is in this f, we label it by a "+" (i.e., $f(< x, y >) = +$), otherwise we label it by a "-".
- The concept class is $F = \{\text{all axis-aligned rectangle in } [0,1]^2\ \}$. The target concept is an axis aligned rectangle $f \in F$.
- Finally, as usual, the underlying distribution is denoted D, the error parameter is $0 < \epsilon < 1$ and the confidence parameter is $0 < \delta < 1$.

For the sake of clarity we will refer to the target concept by target rectangle, and to axis-aligned rectangles as rectangles. We use the following notations:

- $W_D(g)$ is the weight assigned by distribution D to the rectangle g. That is,
$$W_D(g) \stackrel{\text{def}}{=} \Pr_{s\sim D}[g(s) = +]$$
- S is a random variable representing the set of example points. Each element in S is drawn according to D. We let S^+ denote the subset of S labeled "+".

We stress that all probabilities are taken over the choice of examples.

Claim 25.2.1 *The following algorithm is an efficient PAC learning algorithm for axis-aligned rectangles over* $[0,1]^2$.

If S^+ *is empty then output the empty rectangle as hypothesis.*
Otherwise, output the rectangle represented by $< \tilde{x}_{min}, \tilde{y}_{min}, \tilde{x}_{max}, \tilde{y}_{max} >$, *where*
$\tilde{x}_{min}$ = *the minimal x-axis coordinate in* S^+.
$\tilde{y}_{min}$ = *the minimal y-axis coordinate in* S^+.
$\tilde{x}_{max}$ = *the maximal x-axis coordinate in* S^+.
$\tilde{y}_{max}$ = *the maximal y-axis coordinate in* S^+.

Proof: Let f be the target rectangle and g be the hypothesis output by the algorith. Note that the hypothesis rectangle is always contained in the target rectangle (since the borders of the former are determined by positive examples).

We partition the proof into two cases:

1. $W_D(f) > \epsilon$
2. $W_D(f) \leq \epsilon$

We start with the proof of the first case. Given a target rectangle let us draw the following auxiliary construction: We cover the upper side of the target rectangle with a line, we push this line towards the opposite side, until we get a rectangle A_1, such that $W_D(A_1) = \frac{\epsilon}{4}$.

> Oded's Note: *We assume that an adequate stopping point exists; that is, that the distribution is such that some rectangle has weight smaller than* $\epsilon/4$ *whereas a slightly bigger rectangle has weight* $\epsilon/4$. *Clearly, an approximation of this assumption is good enough, but even such an approximation is not guaranteed to exist. This issue is dealt with in an appendex.*

We repeat this process for all other sides of the residual target rectangle (see Fig. 25.2). We get rectangle A_2,A_3,A_4 such that $W_D(A_2) = W_D(A_3) = W_D(A_4) = \frac{\epsilon}{4}$. (It should be stressed, that we assumed that this process can be done, which is not necessarily the case. We will deal with this problem in appendix.) Let us look at the part of the f, that is not covered by A_1,A_2,A_3 and A_4, it is a rectangle we call it B. According to auxiliary construction:

$$\begin{aligned} W_D(f) - W_D(B) &= W_D(A_1) + W_D(A_2) + W_D(A_3) + W_D(A_4) \\ &= 4 \cdot \frac{\epsilon}{4} = \epsilon \end{aligned}$$

If hypothesis rectangle h contains B (i.e., $h \supseteq B$) then (using $h \subseteq f$)

$$\begin{aligned} err_{f,D}(h) &= \Pr_{s \sim D}[s \in f \text{ and } s \notin h] \\ &\leq \Pr_{s \sim D}[s \in f \text{ and } s \notin B] \\ &= W_D(f) - W_D(B) = \epsilon \end{aligned}$$

Furthermore, according to algorithm, if there is an example in each of the rectangle A_1, A_2, A_3 and A_4, then $B \subseteq h$. Thus, we merely bound the number of examples, denoted m, such that the probability for this event not to occur, is less then δ. For any $i \in \{1,2,3,4\}$, the probability that no example resides in A_i is $(1 - W_D(A_i))^m = (1 - \frac{\epsilon}{4})^m$. Thus,

$$\begin{aligned} \Pr[err_{f,D}(h) > \epsilon] &\leq \Pr[\exists i \text{ no sample in } A_i] \\ &\leq 4 \cdot \left(1 - \frac{\epsilon}{4}\right)^m \end{aligned}$$

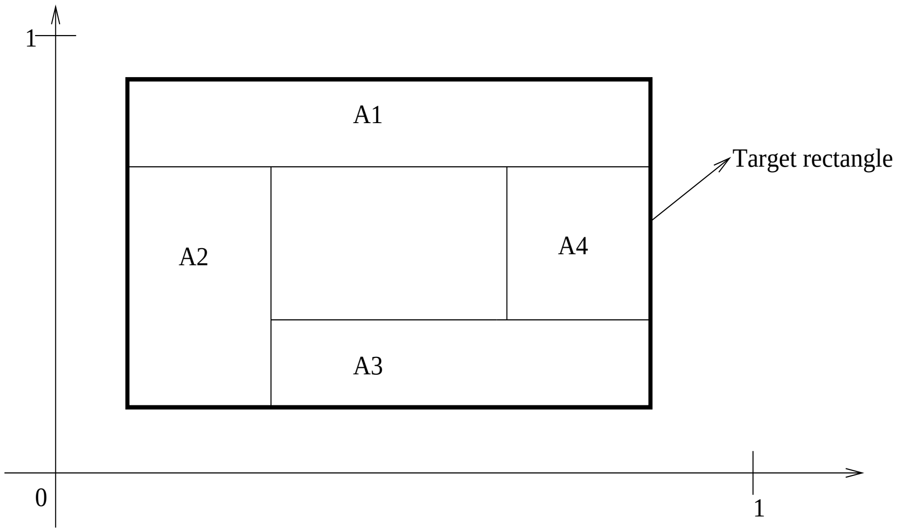


Figure 25.2: Target rectangle in bold, and rectangle A_1, A_2, A_3, A_4 each of weight $\frac{\epsilon}{4}$

So we need to set m so that $4 \cdot (1 - (\epsilon/4))^m \leq \delta$. Using the inequality $1 - \frac{\epsilon}{4} \leq e^{-\frac{\epsilon}{4}}$, the condition simplifies to $4(e^{-\frac{\epsilon}{4}})^m \leq \delta$, which solves to $m \geq (4/\epsilon) \cdot \ln(4/\delta)$.

We now turn to the second case, where $W_D(f) \leq \epsilon$: Using $h \subseteq f$, we have (for every sequence of examples used by the algorithm):

$$\begin{aligned} err_{f,D}(h) &= \Pr_{s \sim D}[s \in f \text{ and } s \notin h] \\ &\leq \Pr_{s \sim D}[s \in f] \;=\; W_D(f) \;\leq\; \epsilon \end{aligned}$$

Thus, in this case, $\Pr[err_{f,D}(h) \leq \epsilon] = 1$.

Note all operations in algorithm depend linearly on the size of the sample, which in turn depends linearly on $\frac{1}{\epsilon}$ and $ln\frac{4}{\delta}$. Therefore the algorithm is an efficient PAC learning algorithm. ■

25.3 Occam's Razor

Occam's Razor is based on a principle stated by William of Occam. We interpret Occam's principle as follows: learning can be achieved by finding a succinct representation for the labels of large number of examples. By a succinct representation, we mean that the size of the representation is sublinear in the number of examples. This reduces the problem of learning to the problem of finding a hypothesis, consistent with every given example. We say an algorithm is an *Occam's Algorithm*, if it outputs a succinct hypothesis consistent with every given example.

Theorem 25.4 (Occam's Razor – basic version): *Let F_n be a finite concept class. Let A be an algorithm such that for every n, for every $f \in F_n$ and for every number of examples labeled by f, the algorithm outputs an hypothesis $h \in F_n$, that is consistent with all the given examples. Then for any distribution D, for any error parameter $0 < \epsilon < 1$ and for any confidence parameter $0 < \delta < 1$ if the number of examples drawn independently by D is larger or equal $\frac{1}{\epsilon}(\log \frac{1}{\delta} + \log |F_n|)$, then*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where h is the hypothesis output by the algorithm, and the probability is taken over the choice of examples as well as over the internal coin tosses of the algorithm.

Proof: We use the following notations:

- $S = < a_j, f(a_j) >_{j=1...m}$ is the set of labeled examples.
- $C_F(S)$ is the set of hypotheses in F that are consistent on all examples in S. That is,

$$C_F(S) \stackrel{\text{def}}{=} \{h \in F : h(a_i) = f(a_i) \text{for } i = 1, ..., m \ \}$$

- $B_{D,\epsilon}(f)$ is the set of hypotheses in F that have a probability of error larger than ϵ. That is,

$$B_{D,\epsilon}(f) \stackrel{\text{def}}{=} \{h \in F : err_{f,D}(h) > \epsilon\}$$

Note that the algorithm always outputs an hypothesis $h \in C_F(S)$. We will upper bound the probability that this hypothesis is in $B_{D,\epsilon}(f)$. Actually, we will upper bound the probability that any hypothesis in $C_F(S)$ remains in $B_{D,\epsilon}(f)$.

$$\Pr_S[\text{there exists } h \in B_{D,\epsilon}(f) \text{ such that } h \in C_F(S)] \leq \sum_{h \in B_{D,\epsilon}(f)} \Pr_S[h \in C_F(S)] \tag{25.1}$$

Proposition 25.3.1 *For any $h \in B_{D,\epsilon}(f)$,* $\quad \Pr_S[h \in C_F(S)] \leq \frac{\delta}{|F|}$

Proof: For any hypothesis in $B_{D,\epsilon}$ the probability that the hypothesis will be in $C_F(S)$ is the probability that all given example points landed in the agreement area. By using definition of $B_{D,\epsilon}(f)$ the probability that a single example point landed in the agreement area is at most $1 - \epsilon$. Therefore for any fixed $h \in B_{D,\epsilon}(f)$:

$$\Pr_S[h \in C_F(S)] \leq (1 - \epsilon)^{|S|} \leq e^{-\epsilon|S|}$$

Using the assumption on $|S|$:

$$\epsilon|S| \geq \log(\tfrac{1}{\delta}|F|)$$

Thus,

$$e^{-\epsilon|S|} \leq \frac{\delta}{|F|}$$

and the proposition follows. ■

Combining Eq. (25.1) and Proposition 25.3.1, the theorem follows. ■

It is easy to observe, that an algorithm satisfying this theorem, is a PAC learning algorithm. Also note that since the hypothesis is taken from the concept class it is necessarily succinct.

Learning monomials. The concept class F_n of monomials is a family of boolean expressions, over literals corresponding to n variables $x_1, x_2, ..., x_n$, defined as follows:

$$F_n = \{f : f = l_1 \wedge l_2 \wedge ... \wedge l_t\}$$

Where for each $1 \leq i \leq t$

$$l_i \in \{x_j, \overline{x_j}\}_{j=1}^n$$

We seek an efficient PAC learning algorithm for the concept class of monomials. A instance $a \in X_n$ is interpreted as an assignment to the variables $x_1, x_2, ..., x_n$. We us the notation a_i for the value of the i'th bit of a.
We denote the set of given examples by:

$$S = \{< a^j, f(a^j) >\}_{j=1,...,m}$$

where each $a^j = a_1^j, ..., a_n^j$.

We call an example *negative* if it is labeled by 0, otherwise we call it *positive*. Before stating the algorithm, let us see what information can we conclude about the target concept, from a given example. A positive example consists of an assignment to the target concept, such that every the literal in it evaluates to 1. A negative example consists of an assignment to the target concept, such that there exists a literal in it that evaluates to 0. Thus negative examples convey much less information then positive examples, and the information they convey is not trivial to use. Our algorithm uses only positive examples.

Claim 25.3.2 *Given at least $\frac{1}{\epsilon}(2n + \log\frac{1}{\delta})$ examples, the following algorithm is a PAC learning algorithm.*

1. *initialize $h = x_1 \wedge \overline{x_1} \wedge x_2 \wedge \overline{x_2} \wedge ... \wedge x_n \wedge \overline{x_n}$*
2. *for $j = 1...m$, if $f(a^j) =$ '$+$' then do*
   - *for each $i \in \{1...n\}$ if $a_i^j = 1$ remove $\overline{x_i}$ from h*
   - *for each $i \in \{1...n\}$ if $a_i^j = 0$ remove x_i from h*

Proof: We use the following notations: By $S = \{< a^j, f(a^j) >\}_{j=1,...,m}$ we denote the set of all samples. By h^j we denote the expression h after j iterations; that is, after processing the examples $a^1, ..., a^j$.

We first show that the final hypothesis h is consistent with all examples. Using induction on j, we show that h^j is consistent with the first j examples and that h^j includes all literals in f (i.e., $h^j \supseteq f$). The induction basis ($j = 0$) holds trivially (as h^0 is as initialized in Step 1). In the induction step, suppose that $h^j \supseteq f$ is consistent with the first j examples, and consider what happens in the $j + 1^{\text{st}}$ iteration. We consider two cases

Case 1: the $j + 1^{\text{st}}$ example is positive. In this case $h^{j+1} \subseteq h^j$ and $h^{j+1} \supseteq f$ (since the only literals omitted from h^j are those that cannot be in f).

- Using $h^{j+1} \subseteq h^j$ (i.e., $h^j(a) = 1$ implies $h^{j+1}(a) = 1$), we have $h^{j+1}(a^i) = h^j(a^i) = f(a^i)$ for every $i = 1, ..., j$ satisfying $f(a^i) = 1$.
- Using $h^{j+1} \supseteq f$ (i.e., $f(a) = 0$ implies $h^{j+1}(a) = 0$), we have $h^{j+1}(a^i) = f(a^i)$ for every $i = 1, ..., j$ satisfying $f(a^i) = 0$.
- Finally, by the operation of the current interaction, $h^{j+1} \subseteq \wedge_{i=1}^n l_i$, where $l_i = x_i$ if $a_i^{j+1} = 1$, and $l_i = \overline{x}_i$ otherwise. Thus, $h^{j+1}(a^{j+1}) = 1$.

It follows that $h^{j+1}(a^i) = f(a^i)$ holds for $i = 1, ..., j, j + 1$.

Case 2: the $j + 1^{\text{st}}$ example is negative. In this case $h^{j+1} = h^j$ and so $h^{j+1} \supseteq f$. Also, since $f(a^{j+1}) = 0$, it follows that $h^{j+1}(a^{j+1}) = 0$. Thus, $h^{j+1}(a^i) = f(a^i)$ holds for $i = 1, ..., j, j+1$.

Let us compute the cardinality of F_n. The cardinality of F_n is bounded by 2^{2n}, since each literal can either not appear in the monomial or appear in monomial, and the number of literals is $2n$. Thus, $\log_2 |F_n| \leq 2n$, and the claim follows by applying Occam's Razor. ■

Is this version of Occam's Razor powerful enough? The following example shows that this current version is somewhat limited.

Learning a 3-term DNF**.** The concept class F_n of 3-term DNF's is a family of boolean expressions, over variables $x_1, x_2, ..., x_n$, defined as follows:

$$F_n = \{f : f = M1 \wedge M2 \wedge M3 \text{ where } M1, M2, M3 \text{ are monomials over } x_1, x_2, ..., x_n\}$$

The learning task seems similar to the previous task of learning monomials. However, the problem of finding a consistent 3-term DNF seems intractable. Specifically:

Claim 25.3.3 *The problem of finding a 3-term* DNF *that is consistent with a given set of examples is* $\mathcal{NP}$*-complete.*

The proof is via a reduction to 3-colorability and is omitted. Actually, the difficulty is not due to Occam algorithms only. It rather holds with respect to any PAC learning algorithm that always outputs hypotheses in the concept class (in our case a 3-term DNF). Recall that in our definition of PAC algorithms we did not insist that when learning a target concept from F_n the algorithm must output a hypothesis in F_n. However, we did make this condition when defining an Occam algorithm.

Claim 25.3.4 *If* $\mathcal{NP}$ *is not contained in* $\mathcal{BPP}$ *then no probabilistic polynomial-time that outputs a hypothesis in 3-term DNF can learn the class of 3-term* DNF *formulae.*

Proof: We show a randomized polynomial-time reduction of the problem of finding a 3-term DNF that is consistent with a given set of examples to the problem of PAC learning 3-term DNF's via such hypotheses.

Let L be a PAC learning algorithm of the latter form, and suppose we are given a set of m instances, denoted S, labeled by some 3-term DNF, denoted f. We invoke algorithm L on input parameters $\epsilon = 1/2m$ and $\delta = 1/3$, and feed it with a sequence of (labeled) examples uniformly distributed in S. (This sequence may contain repetirtions.) Thus, L is running with distribution D which is uniform on S, and outputs a 3-term DNF hypothesis h satisfying

$$\Pr\left[err_{f,D}(h) \geq \frac{1}{2m}\right] \leq \frac{1}{3}$$

Since each $s \in S$ has probability mass $1/m$, it follows that

$$\Pr[\exists s \in S \ \text{ s.t. } \ h(s) \neq f(s)] \leq \frac{1}{3}$$

Thus, with probability $2/3$ the hypothesis h is consistent with f on S. Invoking Claim 25.3.3, the current claim follows. ■

Discussion: What the last claim says is that if we insist that the learning algorithm outputs a hypothesis in the concept class being learned (as we do in case of Occam's Razor) then we cannot learn 3-term DNF formulae. In the next section, we shall see that the latter class can be learned if we allow the hypothesis class to be different.

25.4 Generalized definition of PAC learning algorithm

Oded's Note: *This section was drastically revised by me.*

In accordance with the above discussion we define explicitly the notion of learning one concept class with a possibly different class of hypotheses, which typically is a superset of the concept class.

Definition 25.5 (PAC learning, revisited): *Let $F = \cup_n F_n$ and $H = \cup_n H_n$ so that $F_n \subseteq H_n$ are classes of functions mapping X_n to $\{0,1\}$. We say that algorithm A PAC* learns the concept class *F* using the hypothesis class *H if the following holds: For every n, for every function $f \in F_n$, for every distribution $D : X_n \longrightarrow [0,1]$, for every error parameter $0 < \epsilon < 1$ and for every confidence parameter $0 < \delta < 1$, given parameters n, ϵ, δ and a set of f-labeled examples, $\{< x^i, f(x^i) >\}$, where x^i is drawn independently under the distribution D, the algorithm outputs a hypothesis $h \in H_n$ such that:*

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

where the probability is taken over the choice of examples, as well as over the internal coin tosses of the algorithm.

That is, the only change relative to Definition 25.2 is the condition that the output hypothesis h belongs to H_n. In case $H = F$ we say that the algorithm is a proper learning algorithm for F.

In contrast to the negative results in the previous section, we show that the class 3-term DNF can be efficiently learned using the hypothesis class 3CNF. This statement is proven via a reduction of this learning task to the task of learning monomials (already solved efficiently above). To present this reduction, we first define what we mean in general by reduction among learning tasks.

25.4.1 Reductions among learning tasks

Reductions are a powerful tool common in computer science models. It is only natural to define such notion for the model of PAC learning.

Definition 25.6 *We say that the concept class F over the instance space X, is PAC-reducible to the concept class F' over the instance X' if for some polynomial p*

- *there exists a polynomial-time computable mapping G from X to X' such that for every n and for every $x \in X_n$, $\quad G(x) \in X_{p(n)}$.*
- *there exists a polynomial $q()$ such that for every concept $f \in F_n$, there exists a concept $f' \in F'_{p(n)}$ such that $size(c') \leq q(size(c))$ and for every $x \in X_n$, $\quad f(x) = f'(G(x))$.*

Note that the second item does not require an efficient transformation between f and f'. In fact, for proper learning (by efficient algorithms), an efficient transformation from F' to F must be required.

Theorem 25.7 *Let F and F' be concept classes. If F is PAC-reducible to F', and F' is efficiently PAC learnable, then F is efficiently PAC learnable.*

Proof: Given an efficient PAC learning algorithm L'. We use L' to learn a an unknown target concept $f \in F_n$ as follows: Given an example labeled $< x, f(x) >$, where $x \in X_n$, we compute an example labeled $< G(x), f(x) >$, where $G(x) \in X'_n$, and supply it to L'. The original examples are chosen by a distribution D on X, and so the reduced examples $G(x)$ (computed by us) are drawn by distribution D' on X' induced by ditribution D. Also, let f' be the function associated by Item 2 to f (i.e., $f'(G(x)) = f(x)$). Algorithm L' will output an hypothesis h' such that:

$$\Pr[err_{f',D'}(h') \leq \epsilon] \geq 1 - \delta$$

where the probability depends on a sample of D'. We take the composition of h' and G to be our hypothesis h for f (i.e., $h(x) = h'(G(x))$).

We need to evaluate $\Pr[err_{f,D}(h) \leq \epsilon]$, where $h = h' \circ G$ and h' is the output of L'. We first observe that

$$\begin{aligned} err_{f,D}(h) &= \Pr_{x \sim D}[h(x) \neq f(x)] \\ &= \Pr_{x \sim D}[h'(G(x)) \neq f'(G(x))] \\ &= \Pr_{x' \sim D'}[h'(x') \neq f'(x')] \\ &= err_{f',D'}(h') \end{aligned}$$

and so $\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$, as required. ■

Learning a 3-term *DNF*, revisited. Recall that, assuming $\mathcal{NP}$ is not contained in $\mathcal{BPP}$, it is infeasible to *properly* learn 3-term DNF's (i.e., learn this class by hypotheses in it). In contrast, we now show that it is feasible to learn 3-term DNF's by 3CNF hypotheses. Actually, we show that it is feasible to (properly) learn the class of 3CNF (which contain via a easy reduction all 3-term DNF's).[1] This is shown by reducting the learning of 3-CNF's to the learning of monomials.

Claim 25.4.1 *Learning 3-CNF's is reducible to learning monomials. Furthermore, there exists an efficient algorithm for* properly *learning 3-CNF.*

Proof: We define the following transformation G from 3-CNF instance space X_n, to the monomial instance space X'_k, where $k = 8 \cdot \binom{n^3}{3}$. For each of the possible k clauses, we associate a distinct variable, and the transformation from 3CNF to monomials just maps the set of clauses in the 3CNF into the set of variables. (Indeed, we reduce to learning monotone monomials.) The transformation G does the analogous thing; that is, it maps truth assignments to the n 3NCF variables onto truth assignments to the k monomial varaibles in the natural way (i.e., a monomial variable representing a possible clause is assigned the value to which this clause evaluates under the assignment to the 3CNF variables). This transformation satisfies the conditions of Definition 25.6.

To show the furthermore-part, observe that the hypothesis constructed by the reduction-algorithm (given in the proof of Theorem 25.7) can be readily put in 3CNF with respect to the space X_n. ■

25.4.2 Generalized forms of Occam's Razor

We have seen that it is worthwhile to use a hypothesis from a wider class of functions (than merely from the concept class). A staightfoward generalization of Occam's Razor to a case where the algorithm outputs a hypothesis from a predetermined hypothesis class follows.

Theorem 25.8 (Occam's Razor – generalization to predetermined hypothesis class): *Let $F_n \subseteq H_n$ be finite concept classes. Let A be an algorithm such that for every n, for every $f \in F_n$ and for every*

[1] Applying the distributional law to a 3-term DNF over n variables, we obtain a 3CNF with at most $(2n)^3$ clauses.

number of examples labeled by f, the algorithm outputs an hypothesis $h \in H_n$, that is consistent with all the given examples. Then for any distribution D, for any error parameter $0 < \epsilon < 1$ and for any confidence parameter $0 < \delta < 1$ if the number of examples drawn independently by D is larger or equal $\frac{1}{\epsilon}(\log \frac{1}{\delta} + \log |H_n|)$, then

$$\Pr[err_{f,D}(h) \leq \epsilon] \geq 1 - \delta$$

The proof is by a straightforward adaptation of the proof given to Theorem 25.4 (which is indeed a special case obtained by setting $H_n = F_n$).

A wider generalization is obtained by not determining a priori the hypothesis class. In such case, we need some other way to bound the hypothesis from merely recording all examples. This is done by requiring that the hypothesis's length is strictly shorter than the number of examples seen. This leads to the following formulation.

Definition 25.9 *Let α and $\beta < 1$ be constants. We say that L is an (α, β)-*Occam's algorithm *for F if given m examples, L outputs a hypothesis h such that h is consistent with every given example and*

$$size(h) \leq (n \cdot size(f))^{\alpha} m^{\beta}$$

Theorem 25.10 *For every α and $\beta < 1$, an (α, β)-Occam's algorithm can be turned into a PAC learning algorithm by running it on $O(\epsilon^{-1} \cdot (\text{poly}(n \cdot size(f)) + \log(1/\delta))$ examples.*

Proof: Firstly, we generalize Theorem 25.8 to a setting in which the Occam algorithm may uses different hypothesis classes for different number of examples. Specifically, suppose that when seeing m examples labelled by $f \in F_n$, the algorithm outputs a hypothesis in $H_{n,m}$. Then such an algorithm is a PAC learner provided

$$m \geq \epsilon^{-1} \cdot (\log(1/\delta) + \log |H_{n,m}|),$$

where ϵ and δ are the error and confidence parameters (given to the PAC version).

Now, the conditions of the current theorem provide such a "dynamic" hypothesis class, $H_{n,m}$, and the upper bound on the length of hypothesis guarantees that

$$\log |H_{n,m}| \leq (n \cdot size(f))^{\alpha} m^{\beta}$$

So, since $\beta < 1$, for sufficiently large $m = \text{poly}(n \cdot size(f))/\epsilon$, we have $\epsilon m \ll (n \cdot size(f))^{\alpha} m^{\beta}$. The theorem follows. ■

25.5 The (VC) Vapnik-Chervonenkis Dimension

In all versions of Occam's Razor discussed above, we assumed that the hypothesis class is finite. In general this is not necessarily the case. For example, the natural hypothesis class for learning axis-aligned rectangles over $[0,1]^2$ is the infinite set of all possible axis-aligned rectangles. Therefore we can not apply Occam's Razor to this problem. We would like a tool with similar flavor for the case of infinite hypothesis classes. The first step for achieving this goal, is finding a parameter of finite value that characterizing also infinite classes.

Definition 25.11 (shattering a set): *A finite subset S of instance space X_n is* shattered by a family of functions *F_n if for every $S' \subseteq S$ there exists a function $f \in F_n$ such that*

$$f(x) = \begin{cases} 1 & x \in S' \\ 0 & x \in S \backslash S' \end{cases}$$

That is for $S = \{x^1, ..., x^m\}$ and for every $\alpha \in \{0,1\}^m$ there exists a function $f \in F$, such that for any $i = \{1...m\}$, $f(x^i) = \alpha^i$, where α^i is the i^{th} bit of α.

Definition 25.12 (VC dimention): *The* VC dimention of a set of functions *F, denoted VC-dim(F), is the maximal integer d such that there exists a set S of cardinality d that is shattered by F.*

25.5.1 An example: VC dimension of axis aligned rectangles

The following example demonstrates the computation of VC dimension of a family of functions.

Proposition 25.5.1 *VC-dim(axis-aligned rectangles)* $= 4$

Proof: We start by exhibiting a set of four points in $[0,1]^2$ that can be shattered. For every labeling of the set of four points exhibited in Fig. 25.3, we can find a rectangle that induces such labeling.

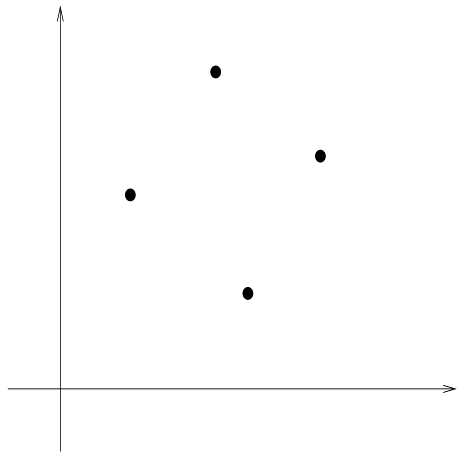


Figure 25.3: Four point that in $[0,1]^2$, that can be shattered

1. In case all points are labeled "+", we take the rectangle $[0,1]^2$ itself.
2. In case all points are labeled "-", we take the empty rectangle.
3. For the case three points are labeled "-" and one point "+", take a "small" rectangle that covers only the point labeled "+".
4. For the case three points are labeled "+" and one point"-", we take rectangles as can be seen in Fig. 25.4.
5. For the case two points are labeled "-" and two points "+", we take rectangles as can be seen in Fig. 25.5. (The figure shows only 4 out of the 6 subcases; the other two are easier.)

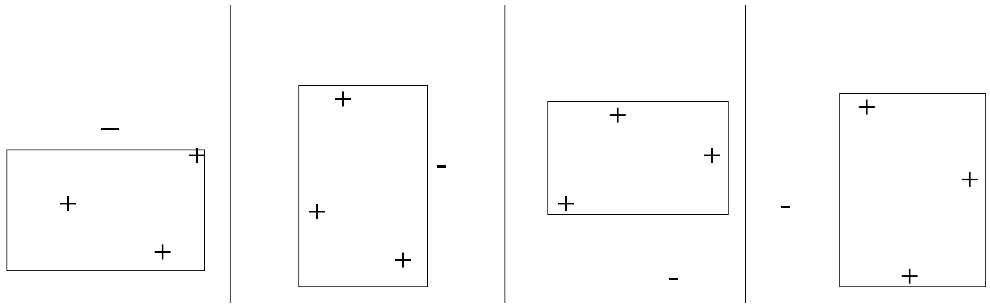


Figure 25.4: rectangle covering the three points labeled "+"

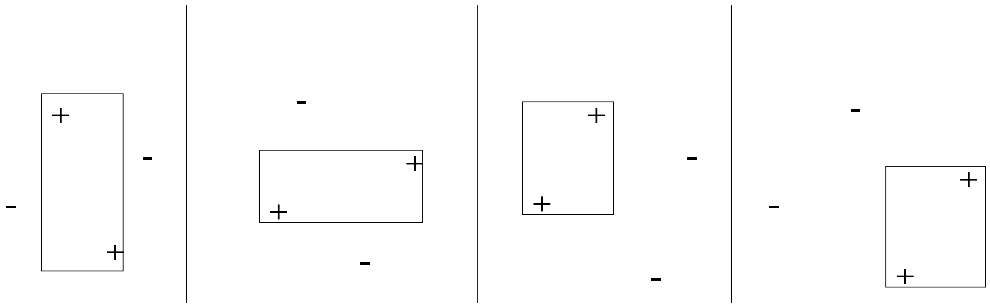


Figure 25.5: rectangle covering the two points labeled "+"

It remains to prove that for every set of five points in $[0,1]^2$, there exists is a labeling, such that it can not be induced by any rectangle in $[0,1]^2$.

By the proof of Claim 25.2.1 (the learning algorithm for axis aligned rectangles), for every set of five points in $[0,1]^2$, at most four of them determine the minimal rectangle that contains the whole set. Then no rectangle is consistent with the labeling that assigns these 4 boundary points '+' and assigns the remaining point (which must reside inside any rectangle coving these points) a '-'. ■

25.5.2 General bounds

VC dimension is linked to the sample complexity of PAC learning via the following two theorems:

Theorem 25.13 (upper bound on sample complexity)*: Let H, F be function classes such that $F \subseteq H$. Let A be a algorithm that given a labeled sample always outputs a hypothesis $h \in H$ consistent with the sample. Then there exists a constant c_0, such that for any target function $f \in F$, for any underlying distribution D, any error parameter $0 < \epsilon < 1$ and any confidence parameter $0 < \delta < 1$, if the number of examples is greater or equal:*

$$\frac{c_0}{\epsilon} \cdot \left(VC\text{-}dim(H) \cdot \log \frac{1}{\epsilon} + \log \frac{1}{\delta} \right)$$

then with probability greater then $1 - \delta$ we get

$$err_{D,f}(h) \leq \epsilon$$

where h is the hypothesis output by A.

Theorem 25.14 (lower bound on sample complexity)*: Any PAC learning algorithm for learning a concept class F such that $d =$ VC-dim(F) requires $m = \Omega(\frac{1}{\epsilon} \cdot (\log \frac{1}{\delta} + d))$ examples.*

We note that there is a gap of factor $\log \frac{1}{\epsilon}$ between the bounds. The proof of the these theorems is complex. Instead we prove a slightly weaker lower bound theorem:

Theorem 25.15 (lower bound, slightly weaker form): *Any PAC learning algorithm for learning a concept class F such that $d =$ VC-dim(F) requires requires $\Omega(\frac{d}{\epsilon})$ examples in the worst case.*

Oded's Note: *The proof was revised by me.*

Proof: Since the VC dimension is d, there exists a set of d points shattered by F. We denote this set by $S = \{e^1, ..., e^d\}$. Since S is shattered by F, for each possible labeling of S there exists a function $f \in F$ which is consistent with this labeling. Let us denote by $f_\alpha \in F$ a function consistent with the labeling $\alpha = \alpha_1 \cdots \alpha_d \in \{0,1\}^d$; i.e., $f_\alpha(e^i) = \alpha_i$ for each $i = 1, ..., d$. Let $F' = \{f_\alpha : \alpha \in \{0,1\}^d\} \subseteq F$.

We consider error parameter $\epsilon \leq \frac{1}{8}$, and an arbitrary confidence parameter $0 < \delta < \frac{1}{2}$.

We start by proving a bound of $\Omega(d)$. Towards this goal, we define the underlying distribution D to be uniform over S; that is, every point in S is assigned probability $\frac{1}{d}$, and all other instances are assigned zero probability.

Let us assume, in contrary to the claimed bound, that $\frac{d}{2}$ examples suffice. Under this assumption in the best case, $\frac{d}{2}$ different examples where drawn. We denote this set by S_1, and let $S_2 \stackrel{\text{def}}{=} S \setminus S_1$. Intutively, the algforithm only got the labeling of S_1 and so there is no way it can distinguish between the $2^{d-|S_1|}$ functions in F' consistent with S_1.

Formally, we consider a target function f chosen uniformly in F'. The algorithm then obtain a sample S_1 labeled by f, and outputs a hypothesis h. We are interested in the behavior of the random variable $err_{f,D}(h)$. Recall that this random varaible is defined over the probability space defined by the uniform choice of $f \in F'$, the uniform choice of $d/2$ points in S (yielding S_1), and additional coin tosses the learning algorithm may do. Let us reverse the "natural" order of the randomization, and consider what happens when S_1 is selected first, and $f \in F'$ is selected next. Furthermore, we select f in two phases: first we select at random the value of f on S_1, and we postpone for later the random choice of the value of f on S_2. (Together, the values of f on S_1 and S_2 will determine a unique $f \in F'$, which will be uniformly distributed.) However, the learning algorithm is oblivious of the latter choices (i.e., of the values of f on S_2), and so the hypothesis h is stochastically independent of the latter. So the full order of events we consider is:

1. d points are selected uniformly in S, determining S_1 and S_2.
2. one assigns uniformly Boolean values to the points in S_1 and presents these to the learning algorithm.
3. the learning algorithm outputs a hypothesis h.
4. one assigns uniformly Boolean values to the points in S_2, thus determining a unique function $f \in F'$.

Since f is determined on S_2 only after h is fixed, and its values on S_2 are uniformly distributed in $\{0,1\}$, the expected value of $err_{f,D}(h)$ is at least

$$\frac{1}{2} \cdot \frac{|S_2|}{|S|} \geq \frac{1}{2} \cdot \frac{d/2}{d} = \frac{1}{4}$$

Thus, there exists $f \in F' \subseteq F$ so that with probability at least 1/2,

$$err_{f,D}(h) \geq \frac{1}{8}$$

This establishes a lower bound of $d/2$ on the sample complexity, for any $\epsilon \leq 1/8$ (and $\delta \leq 1/2$).

In order to prove the stronger bound, stated in the theorem, we modify the distribution D as follows. Fixing an arbitrary element $e^1 \in S$, we let D assign it probability $1 - 8\epsilon$, and assign each other element of S probability $8\epsilon/(d-1)$. (Again, only points in S are assigned non-zero probability.) If we take a sample of m points from D, we expect only $8\epsilon m$ points to be different than e^1. Thus, if $m < (d-1)/(20\epsilon)$ then with very high probability only $10\epsilon m < (d-1)/2$ points will be different than e^1. Applying an argument as above, we conclude that our error with respect to D is expected to be the probability assigned to points not seen by the algorithm times one half; that is,

$$\left(\frac{d-1}{2} \cdot \frac{8\epsilon}{d-1}\right) \cdot \frac{1}{2} = 2 \cdot \epsilon$$

Thus, there exists $f \in F' \subseteq F$ so that with probability at least 1/2, $err_{f,D}(h) \geq \epsilon$. This establishes a lower bound of $(d-1)/(20\epsilon)$ on the sample complexity, for any $\epsilon \leq 1/8$ (and $\delta \leq 1/2$). ■

Bibliographic Notes

The PAC learning model was introduced by Valiant [3]. Occam's Razor Theorem is due to [1]. The interested reader is referred to a textbook by Kearns and Vazirani [2].

1. A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Occam's Razor. *Information Processing Letters*, Vol. 24 (6), pages 377–380, April 1987.

2. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

3. L.G. Valiant. A Theory of the Learnable. *Communications of the ACM*, Vol. 27 (11), pages 1134–1142, November 1984.

Appendix: Filling-up gaps for the proof of Claim 25.2.1

Oded's Note: *The gap left open in the proof was the assumption that we can slides the border of the A_i's so that the weight of each of them is exactly $\epsilon/4$. Firstly, we repeat the comment by which it is not essential to have these rectanges have weight exactly $\epsilon/4$, and it suffices to have each of the A_i's have weight $\Theta(\epsilon)$. But still it may be that there is a probability mass of $\Omega(\epsilon)$ residing on one single axis-allinged line. This problem can be resolved is several ways. For example, one may pertub all points at random, and argue that the performance of the algorithm cannot be improved (a proof is indeed called for!). Alternatively, one may argue seperately for these patalogical cases of single lines having probability mass of $\Omega(\epsilon)$.*

Lecture 26

Relativization

Notes taken by Leia Passoni

Summary: In this lecture we deal with relativization of complexity classes. In particular, we discuss the role of relativization with respect to the $\mathcal{P} \stackrel{?}{=} NP$ question; that is, we shall see that for some oracle A, $\mathcal{P}^A = \mathcal{NP}^A$ whereas for another A $\mathcal{P}^A \neq \mathcal{NP}^A$. However, it also holds that $\mathcal{IP}^A \neq \mathcal{PSPACE}^A$ for a random A, whereas $\mathcal{IP} = \mathcal{PSPACE}$

Oded's Note: *The study of relativization is motivated by the belief that relativized results indicate limitations of proof techniques which may be applied in the real (unrelativized) world. In the conclusion section we explain why we do not share this belief. In a nutshell, whereas it is useful to refer to proof techniques when discussing and unifying known results, it is misleading to refer to a "proof technique" as if it were a domain with well-defined boundaries (and speculate on which results are beyond such boundaries).*

In contrast, we see benefit in an attempt to define frameworks for proving certain results, and discuss properties of proofs within such well-defined frameworks (e.g., proofs of certain well-defined properties cannot prove a particular result or results of certain well-defined form). In fact, our original intention was to present such a framework, called Natural Proofs, in this lecture. This intention was abandoned since we did not see circuit size lower bounds in the course, and such proofs are the context of Natural Proofs.

26.1 Relativization of Complexity Classes

We have already mentioned the use of polynomial time machines that have access to some oracle in order to define the Polynomial Hierarchy (PH).

Given any two compexity classes C_1 and C_2, were C_2 is the class of oracles, it is not always possible to have a natural notion of what is the class $C_1^{C_2}$ – it is not necessarily the case that for every complexity class C_1 we can define its *relativization* to C_2.

There are some conditions under which such a relativization can be done. One of those conditions seems to be that the complexity class C_1 has to be defined in terms of some type of machines – that is, for any language L in C_1 there exists a machine M_L such that $L = L(M_L)$. Furthermore, the definition of M_L has to be extendable to the definition of *oracle machine.*

Oracle Machines. We consider three types of oracle machines:

Definition 26.1 *A (deterministic/nondeterministic/probabilistic) polynomial time* oracle machine *is a machine with a distinguished work tape – the query tape, where the machine may make oracle queries. These queries are answered by a function called the* oracle.

Some notations: $M^f(x)$ denotes a computation of M on input x when given access to oracle f. When the machine writes on the oracle tape a query q (and invokes the oracle), then the tape's contents q is replaced by $f(q)$.

We also write $M^A(x)$, where the oracle A is a language. In this case M is given access to the characteristic function of the language, χ_A.

26.2 The $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question Relativized

Trying to solve the $\mathcal{P} \neq \mathcal{NP}$ conjecture, the corresponding relativized conjecture has been investigated, that is, determining whether $\mathcal{P}^A \neq \mathcal{NP}^A$ or $\mathcal{P}^A = \mathcal{NP}^A$ for some oracle A. Thus, we ask which of the following two possibilities holds:

- There exists an oracle A such that $\mathcal{P}^A = \mathcal{NP}^A$
- There exists an oracle A such that $\mathcal{P}^A \neq \mathcal{NP}^A$

Recall that

$$\mathcal{P}^A \stackrel{\text{def}}{=} \{L : \exists \ \text{ deterministic } \ \text{poly} - \text{time } \ M : L(M^A) = L\}$$

$\mathcal{P}^A$ is then a class of languages: from a countable set of machines M we get a countable set of oracle machines M^A that run in polynomial time, each having oracle access to A.

Note that if $A = \emptyset$ or $A = \{0,1\}^*$, then $\mathcal{P}^A = \mathcal{P}$. In fact, if the answers given to queries to the oracle are all "yes" or all "no", then any language in $\mathcal{P}^A$ can be accepted by a deterministic polynomial time machine, just committing the oracle questins and aswers.

Also, $A \in \mathcal{P} \Rightarrow \mathcal{P}^A = \mathcal{P}$: in this case a P machine can simulate the access to the oracle in polynomial time.

Therefore, it is worthwhile to use oracles machines when the oracle A defines a language which is "complex".

In the same way $\mathcal{NP}^A$ can be defined:

$$\mathcal{NP}^A \stackrel{\text{def}}{=} \{L : \exists \ \text{ nondeterministic } \ \text{poly} - \text{time } \ M : L(M^A) = L\}$$

It could seem that if $\exists A : \mathcal{P}^A \neq \mathcal{NP}^A$ then $\mathcal{P} \neq \mathcal{NP}$, but this is not the case. Of course it would be correct to assert that if $\forall A \ \ \mathcal{P}^A \neq \mathcal{NP}^A$ then $\mathcal{P} \neq \mathcal{NP}$, because in this case it would be possible to consider a trivial oracle like $A = \emptyset$ or $A = \{0,1\}^*$.

Going back to the $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ question, a possible reason for investigating its relativized version is the hope that a result obtained for a class of oracles would shed light on the unrelativized question. The two following theorems show that this hope fails.

Theorem 26.2 *An oracle A exists such that $\mathcal{P}^A = \mathcal{NP}^A$*

Proof: It is obvious that $\forall A \;\; \mathcal{P}^A \subseteq \mathcal{NP}^A$. It is left to show that $\exists A : \mathcal{NP}^A \subseteq \mathcal{P}^A$.

The idea of the proof is to choose as oracle A a $\mathcal{PSPACE}$-complete language, that is, a language so "powerful" that no more advantage is left for a nondeterministic machine over a deterministic one if they both have access to oracle A.

We first show that $\mathcal{NP}^{\mathcal{PSPACE}} \subseteq \mathcal{PSPACE}$. Let A be a $\mathcal{PSPACE}$ language and let $L \in \mathcal{NP}^A$. Then there exists a nondeterministic poly-time machine M such that $L = L(M^A)$. Equivalently, there exists a deterministic poly-time machine $\bar{M}$ such that $\forall x \in L, \exists y$ and polynomial $p_1 : |y| \leq p_1(|x|)$ and such that $\bar{M}^A(x, y)$ accepts. It is possible then to derive a deterministic machine M' such that $M'(x)$ accepts using at most $p_1(|x|)$ cells of space. Machine M' cycles throughout all possible y's of length $\leq p_1(|x|)$, simulating $\bar{M}^A(x, y)$ for each such y. In this simulation, when $\bar{M}$ queries the oracle on a word w, machine M' simulates the oracle by determining whether $w \in A$. Note that $|w| \leq p_2(|x|)$, where $p_2(|x|)$ is the polynomial bound on the number of steps in which $\bar{M}$ accepts the input x. Since $A \in \mathcal{PSPACE}$ it follows that determining $w \in A$ can be done in space $poly(|w|) = poly(|x|)$. So, M' can decide $L \in \mathcal{NP}^A$ using in all $p(|x|)$ space cells for some polynomial p. We conclude that

$$\text{for any } A \in \mathcal{PSPACE}, \quad \mathcal{NP}^A \subseteq \mathcal{PSPACE}. \tag{26.1}$$

We now show that for some $A \in \mathcal{PSPACE}$ (specifically, any $\mathcal{PSPACE}$-complete A will do), $\mathcal{PSPACE} \subseteq \mathcal{P}^A$. Consider a $\mathcal{PSPACE}$-complete language, A. For any $\bar{L} \in \mathcal{PSPACE}$, consider the Cook-reduction of $\bar{L}$ to A; that is, a polynomial-time oracle machine R that given access to oracle A decides $\bar{L}$. But this means that $\bar{L} = L(R^A) \in \mathcal{P}^A$ (which also equals $\mathcal{P}^{\mathcal{PSPACE}}$). Thus,

$$\text{for some } A \in \mathcal{PSPACE}, \quad \mathcal{PSPACE} \subseteq \mathcal{P}^A. \tag{26.2}$$

Combining Equations (26.1) and (26.2), we conclude that there exists an oracle A such that

$$\mathcal{NP}^A \subseteq \mathcal{PSPACE} \subseteq \mathcal{P}^A$$

and the theorem follows. ∎

In spite of the result of Theorem 26.2, the oracle A can be chosen in such a way that the two relativized classes $\mathcal{P}^A$ and $\mathcal{NP}^A$ are different.

Theorem 26.3 *An oracle A exists such that $\mathcal{P}^A \neq \mathcal{NP}^A$*

Proof: Since $\mathcal{P}^A \subseteq \mathcal{NP}^A$, we want to find a language L such that $L \in \mathcal{NP}^A \setminus \mathcal{P}^A$, that is, a language that separates the two classes.
In this case the idea-technique for getting the separation is to define the language using the oracle A. Thus, for every oracle A we define

$$L_A \stackrel{\text{def}}{=} \{1^n : \exists w \in \{0,1\}^n : w \in A\}$$

What is peculiar about language L_A is that it describes whether $A \cap \{0,1\}^n$ is a nonempty set. It can be seen that $L_A \in \mathcal{NP}^A$, by showing a nondeterministic machine with access to A that accepts L_A:

<u>On input $x \in \{0,1\}^n$:</u>

1. if $x \neq 1^n$, then reject

2. otherwise guess $w \in \{0,1\}^n$
3. accept iff $w \in A$

If $x = 1^n \in L_A$, there exists a computation of the machine that accepts – the computation that succeeds in guessing a proper w which is checked to be in A by an oracle query to A. Otherwise, if $1^n \notin L_A$, there is no $w \in \{0,1\}^n$ so that $w \in A$, and therefore the machine can never accept; so in this case, no matter what guess it makes, the machine always rejects.

Now we have to show the existence of an oracle A so that no deterministic poly-time machine can accept L_A. We first show the following

Claim 26.2.1 *For every deterministic polynomial time M, there exists A so that $L_A \neq L(M^A)$*

Proof: We are given machine M and $p(\cdot)$, the polynomial upper bound on the running time of M and on the number of possible queries M can ask to the oracle.

We then choose n_1 such that $n_1 = min\{m : 2^m > p(m)\}$, and we consider the computation of M^A on input 1^{n_1}. The purpose is to define A in such a way that M will err in its decision whether $1^{n_1} \in A$ or not.

Definition of A: we consider three cases regarding the length $n = |q|$ of a query q: $n < n_1$, $n = n_1$, $n > n_1$.

- If $n < n_1$, we can for instance assume we have already answered "0" to all 2^n possible queries during computations with inputs of length less than n_1. In this case, on input 1^{n_1} we answer consistently "0".
- If $n = n_1$, then we use the fact $2^n > p(n)$, which implies that not all n-bit strings are queried. To all (up to $p(n)$) queries on input 1^{n_1} answer "0". What about the queries which M didn't ask? We will define A in such a way that
  - If $M^A(1^{n_1})$ accepts then let $A \cup \{0,1\}^{n_1} = \emptyset$. So we don't put in A any word of length n_1, i.e. we answer "0" to all remaining queries;
  - If $M^A(1^{n_1})$ rejects then let $A \cup \{0,1\}^{n_1} = \{\bar{w}\}$, where $\bar{w}$ is one of the words of length n_1 that have not been queried by M.

  We stress that $M^A(1^{n_1})$ does not depend on whether $\bar{w} \in A$ (since $\bar{w}$ is not queried in this compuation), but $1^n \in L_A$ does depend on whether $\bar{w} \in A$.
- If $n > n_1$. Like before, answer "0". But in this case we are committing the oracle answers on longer strings.

  Anyway, this is not a problem because any $n > n_1$ has to be $n \leq p(n_1)$, otherwise M wouldn't have the time to write the query. Thus, given n_1 there is a finite number of possible lengths of queries $n : n > n_1$, so we can carry on at this stage the construction for these longer n's. (This extra discussion is only important for the extension below.)

Oracle A is thus defined in such a way that, on input 1^{n_1} machine M^A cannot decide whether 1^{n_1} is in L_A or not. ■

Anyway, Claim 26.2.1 is not enough to prove Theorem 26.3, since the oracle A in the claim is possibly different for any M. We need to state a stronger claim. That is

$$\exists A : \forall \text{ deterministic polynomial-time } M,\ L_A \neq L(M^A)$$

This claim is provable by extending the same argument used in Claim 26.2.1. The idea is as follows: first we enumerate all deterministic poly-time machines

$$\begin{array}{ccccc} M_1, & M_2, & \ldots, & M_i, & \ldots \\ \downarrow & \downarrow & & \downarrow & \\ n_1, & n_2, & \ldots, & n_i, & \ldots \end{array}$$

and to each machine M_i we associate a "large enough" integer n_i. The oracle A is built in stages in such a way that, at any step we deal with a portion $A(i-1) \subseteq \{0,1\}^{n_{i-1}}$ of the oracle. Then $A = \cup_{i>0} A(i)$ and $M_i^A(1^{n_i}) = M_i^{A(i-1)}(1^{n_i})$. The oracle A is such that, at any step i, M_i^A errs in decision on input 1^{n_i}.

Define $A(i) = \{\text{words that have been placed in } A \text{ after step } i\}$. We let $A(0) = \emptyset$ and $n_0 = 0$. Consider machine M_1 and its polynomial bound $p_1(\cdot)$. Choose n_1 such that

$$n_1 = min\{m : 2^m > p_1(m)\}$$

At this point $A = A(0) = \emptyset$. To build $A(1)$ we follow the same procedure used in Claim 26.2.1; in particular

- if $M_1^A(1^{n_1})$ accepts then $A(1) = A(0)$;
- if $M_1^A(1^{n_1})$ rejects then $A(1) = A(0) \cup \{w_{n_1}\}$, where w_{n_1} is one of the words of length n_1 that have not been queried by M_1.

At stage i, consider machine M_i and $p_i(\cdot)$. Choose n_i such that

$$n_i = min\{m : 2^m > p_i(m) \text{ and } (\forall j < i),\ m > p_j(n_j)\}$$

Currently A equals $A(i-1)$; so when on input 1^{n_i} machine M_i asks queries of length less than n_i, the anwers have to be consistent with what has previously defined to be in the oracle. Like before, we then build $A(i)$ such that

- if $M_i^A(1^{n_i})$ accepts then $A(i) = A(i-1)$;
- if $M_i^A(1^{n_i})$ rejects then $A(i) = A(i-1) \cup \{w_{n_i}\}$, where w_{n_i} is one of the words of length n_i that have not been queried by M_i.

The word w_{n_i} possibly added to A at this stage is one of those not queried by M_i on input 1^{n_i}; moreover, words possibly added in successive stages have length greater than $p_j(n_j)$ for every $j < i$. This ensures that $M_j^{A(i)}(1^{n_j}) = M_j^{A(j-1)}(1^{n_j})$ for every $j < i$.

The oracle A is then built in such a way that no polynomial time machine M with oracle A exists such that $L(M^A) = L_A$ – every M_i^A fails deciding on input 1^{n_i}.

It is important to note that this construction works only on a specific portion of the oracle A at every stage, keeping all other portions of A untouched. ■

Considering the results of Theorem 26.2 and Theorem 26.3, we can understand that the investigation of the relativized $\mathcal{P} \neq \mathcal{NP}$ conjecture cannot possibly help in proving or disproving the unrelativized one. What instead should be taken from these results is the fact that the $\mathcal{P} \neq \mathcal{NP}$ conjecture should be investigated with *proof techniques that do not relativize*, that is, proof thechniques that cannot be extended to oracles – so the simulation and diagonalization thechniques used in the above theorems do not seem to be adequate for investigating the conjecture.

26.3 Relativization with a Random Oracle

Theorem 26.3 can be somehow generalized; in fact it is possible to prove that

$$Prob_A[\mathcal{P}^A \neq \mathcal{NP}^A] = 1 \tag{26.3}$$

The probabilily is taken over all oracles A; Eq. (26.3) means that for almost all oracles A, $\mathcal{P} \neq \mathcal{NP}$ holds. Since $\mathcal{P}^A \subseteq \mathcal{BPP}^A$ for all A, Eq. (26.3) is a consequence of the following

Theorem 26.4 *For a random oracle A, $\mathcal{NP}^A \not\subseteq \mathcal{BPP}^A$, that is, $Prob_A[\mathcal{NP}^A \not\subseteq \mathcal{BPP}^A] = 1$*

Proof: We have to find a language L such that $L \in \mathcal{NP}^A$ but $L \notin BPP^A$ with probability 1. Again, the idea is to define L_A depending on the oracle A:

$$L_A \stackrel{\text{def}}{=} \{1^n : \exists w \in \{0,1\}^n : \forall u \in C_n, wu \in A\}$$

Here C_n is defined to be any canonical set of n strings of length n. For instance, we can take C_n to be the set of all strings of Hamming weight 1: $C_n \stackrel{\text{def}}{=} \{10\ldots0, 010\ldots0, \ldots, 0\ldots01\}$.

Oded's Note: *The idea behind the use of C_n is to associate with each $w \in \{0,1\}^n$ a coin with odds 2^{-n} of being 1 (and being zero otherwise). If our oracle were to be chosen so that each string $w \in \{0,1\}^n$ is in it with probability 2^{-n} then the extra complication of using C_n would not have been needed. However, the oracle is chosen so that each string is equally likely to be or not be in it, and so we need to implement a coin with odds 2^{-n} of being 1 by using an unbiased coin. That's exactly what the above construction does. The aim of all this is to get to a situation that $1^n \notin L_A$ and "$1^n \in L_A$ via a single witness w" are about as likely, and are also quite likely.*

Again, it can be seen that $L_A \in \mathcal{NP}^A$ by showing a nondeterministic machine with access to A that generates L_A:

On input $x \in \{0,1\}^n$:

1. if $x \neq 1^n$, then reject
2. otherwise guess $w \in \{0,1\}^n$
3. accept iff $\forall u \in C_n, wu \in A$

Here the machine makes n queries to A and accepts iff all of them are anwered "yes" by the oracle.

Now for a randomly chosen A it can be shown that

$$Prob_A[1^n \notin L_A] = Prob_A[\nexists w : \forall u \in C_n, wu \in A] = 1/\mathrm{e} \;\; = \text{ constant} \tag{26.4}$$

In fact, if w and u are fixed, then $Prob_A[wu \in A] = 1/2$ because A is chosen at random, thus every query has probability $1/2$ to be in A or not.

If only w is fixed then $Prob_A[\forall u \in C_n, wu \in A] = 2^{-n}$ since all n events $[wu \in A]$ are independent.

So for this fixed w, $Prob_A[\neg\forall u \in C_n, wu \in A] = 1 - Prob_A[\forall u \in C_n, wu \notin A] = 1 - 2^{-n}$.

Again, there are 2^n possible w and all events are independent, thus $Prob_A[1^n \notin L_A] = Prob_A[\forall w, \neg\forall u \in C_n, wu \in A] = (1-2^{-n})^{2^n} = 1/\mathrm{e} = \text{constant}$.

In a similar way it can also be shown that

$$Prob_A[\exists! w \in \{0,1\}^n : \forall u \in C_n, wu \in A] \approx 1/\mathrm{e} \quad = \quad \text{constant} \tag{26.5}$$

Language L_A was then defined in such a way that

- with constant probability, $1^n \notin A$
- with constant probability, there exists a unique witness for $1^n \in A$ i.e. $\exists! w : \forall u \in C_n, wu \in A$

Intuitively, there is therefore a very small probability for any probabilistic polynomial time machine to come across the unique w so that $\forall u \in C_n, wu \in A$. In fact the following can be proven:

Claim 26.3.1 *Let M be a probabilistic polynomial time oracle machine. Then, for sufficiently large n*

$$Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] < 0.8$$

Remarks: What we actually want to achieve is $Prob_A[L(M^A) = L_A] = 0$; by Claim 3.1 we can make the probability that M^A decides correctly on all inputs 1^n (i.e. for all n) vanish exponentially with the number of n's we consider.
Moreover, we want Claim 3.1 to hold for any possible M, that is $Prob_A[\mathcal{BPP}^A \ni L_A] = 0$. This extension can be carried out similarly to what was done in Theorem 26.3.

Proof: We choose n large enough so that the running time of $M^A(1^n)$ which equals $poly(n) \ll 2^n$. Now consider the case in which on input 1^n machine M makes queries of length $2n$ (other cases can be carried as it has been done in Theorem 26.3). We evaluate

$$\begin{aligned}
Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] &= P_{\text{no}} + P_{\text{uni}} + P_{\text{rest}} \text{ , where} \\
P_{\text{no}} &\stackrel{\text{def}}{=} Prob_A[\chi_{L_A}(1^n) = 0] \cdot Prob_A[M^A(1^n) = 0 | \chi_{L_A}(1^n) = 0] \\
P_{\text{uni}} &\stackrel{\text{def}}{=} Prob_A[\exists! w : \forall u \in C_n, wu \in A] \\
&\qquad\qquad \cdot Prob_A[M^A(1^n) = 1 | \exists! w : \forall u \in C_n, wu \in A] \\
P_{\text{rest}} &\stackrel{\text{def}}{=} Prob_A[\chi_{L_A}(1^n) = 1 \text{ and } \exists w' \neq w : \forall u \in C_n,\ wu \in A \,\&\, w'u \in A]
\end{aligned}$$

(That is, P_{no} represents the probability that M is correct on an input not in L_A, P_{uni} the probability M is correct on an input with a unique witness, and P_{rest} the that M is correct on an input with several witnesses.) Looking at the various probabilities, we have:

1. By Eq. (26.4), $\quad Prob_A[\chi_{L_A}(1^n) = 0] = 1/\mathrm{e}$,
2. By Eq. (26.5), $\quad Prob_A[\exists! w : \forall u \in C_n, wu \in A] \approx 1/\mathrm{e}$.

So

$$P_{\text{rest}} \ \leq \ Prob_A[\exists w' \neq w : \forall u \in C_n,\ wu \in A \,\&\, w'u \in A] \approx 1 - 2/\mathrm{e} \tag{26.6}$$

It is possible to see that the difference between the two probabilities

a) $p_1 \stackrel{\text{def}}{=} Prob_A[M^A(1^n) = 1 | \chi_{L_A}(1^n) = 0]$

b) $p_2 \stackrel{\text{def}}{=} Prob_A[M^A(1^n) = 1 | \exists! w : \forall u \in C_n, wu \in A]$

is very small. In fact, we can define the following oracle:

$A^{(w_0)} \stackrel{\text{def}}{=}$
- randomly choose A so that $1^n \notin L_A$ meaning $\forall w \in \{0,1\}^n \;\; \exists u \in C_n : wu \notin A$
- only on w_0 modify A so that $\forall u \in C_n, w_0 u \in A$

Then it can be seen that the probability – taken over a random A and a random selceted w of length n

b') $Prob_{A, w \in \{0,1\}^n}[M^{A^{(w)}}(1^n) = 1] = p_2$

In fact, consider the process of selecting A at random such that there exists no w for which $\forall u \in C_n, wu \in A$. Then select w_0 at random and modify A so that only in w_0 $\forall u \in C_n, w_0 u \in A$. This process is the same as selecting a random A conditioned to the existence of a unique w such that $\forall u \in C_n, wu \in A$. So the set of oracles $A^{(w)}$ considered in **b')** has the same distribution as the set of random oracles conditioned to the existence of a unique w for which $\forall u \in C_n, wu \in A$, that is considered in **b)**.

Probabilities **a)** and **b')** are then very close; in fact in **b')**, when the machine asks the first query, there is a chance only of 2^{-n} for the machine to come across the unique w such that $\forall u \in C_n, wu \in A$. (On all other w's the two oracles behave the same.) Asking the second query, there is a chance only of $1/(2^n - 1)$ for the machine to get different answers to the query, and so on. Since the total number of possible queries in n is $p(n)$ for some polynomial p, the difference $p_1 - p_2$ is approximately $\frac{p(n)}{2^n} \ll 1$ for the appropriate n that has been chosen.
The fact that these two probabilities are so close means that there is a very small chance that the machine can notice the modification that has been introduced.

Thus, since $Prob_A[M^A(1^n) = 0 | \chi_{L_A}(1^n) = 0] = 1 - Prob_A[M^A(1^n) = 1 | \chi_{L_A}(1^n) = 0] = 1 - p_1$, we see that

$$\begin{aligned}
Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] &= P_{\text{no}} + P_{\text{uni}} + P_{\text{rest}} \\
&\leq Prob_A[\chi_{L_A}(1^n) = 0] \cdot (1 - p_1) \\
&\quad + Prob_A[\exists! w : \forall u \in C_n, wu \in A] \cdot p_2 \; + \; (1 - 2\mathrm{e}^{-1}) \\
&\approx \mathrm{e}^{-1} \cdot (1 - p_1) + \mathrm{e}^{-1} \cdot p_2 + 1 - 2\mathrm{e}^{-1} \\
&= \mathrm{e}^{-1} \cdot (p_2 - p_1) + 1 - \mathrm{e}^{-1}
\end{aligned}$$

Since the term $\mathrm{e}^{-1}(p_2 - p_1) < p(n)/2^n$ is negligible, we get

$$Prob_A[M^A(1^n) = \chi_{L_A}(1^n)] \leq 1 - \mathrm{e}^{-1} < 0.8$$

and the claim follows. ■

This also completes the proof of Theorem 26.4. ■

The reason for investigating how $\mathcal{P}$ and $\mathcal{NP}$ behave if relativized to a random oracle is the following. The statements in Theorems 26.2 and 26.3 only mean that both $\mathcal{P} = \mathcal{NP}$ and $\mathcal{P} \neq \mathcal{NP}$ are supported by *at least* one oracle. This cannot possibly give evidence towards, say, $\mathcal{P} \neq \mathcal{NP}$. But then, what Theorem 26.4 says is that, considering all possible oracles, only a negligible fraction of them supports $\mathcal{P} = \mathcal{NP}$. So it seem plausible that such an assertion supports the $\mathcal{P} \neq \mathcal{NP}$ conjecture. The underlying reasoning, if valid, should extend to any pair of complexity classes. It is known as the *Random Oracle Hypothesis* and asserts that two complexity classes are different if and only if they differ under relativization to a random oracle. Unfortunately, the Random Oracle Hypothesis turns out to be false. For example, the following can be proven:

Theorem 26.5 *For a random oracle A, $\text{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A$, that is, $Prob_A[\text{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A] = 1$*

Since it is known that $\text{co}\mathcal{NP} \subseteq \mathcal{IP}$, Theorem 26.5 indicates that the Random Oracle Hypothesis fails. Thus, the random-relativization separation result of Theorem 26.4 cannot say anything about separation of the unrelativized classes.

26.4 Conclusions

> Oded's Note: *This section was written by me. Although my main motivation for giving this lecture was to present these views, and although I did express them in class, little of them found their way to the above notes. (Unfortunately, in general, students tend to focus on technical material and attach less importance to conceptual discussions. At times even high-level discussions of proof ideas are gives less attention than some very low-level details.)*

The study of relativized complexity classes was initiated with the hope that it may shed some light on the unrelativized classes. Our own opinion is that this hope is highly unjustified.

Relativized results as predictors of non-relativized results. The most naive hope regarding relativized complexity classes is that the relations which hold in *some* relativized world are exactly those holding in the real (unrelativized) world. As shown above, this cannot possibly be true since different oracles may lead to different relativized results. That is, there may be a relativized world (i.e., an oracle) in which some relation holds and another in which the relation does not hold; whereas the same situation can not possibly hold in the (single) real world.

Conflicting relativizations cannot occur when one considers relativization results which hold for almost all orcales (i.e., relative to a random oracle). A new hope postulated by the "Random Oracle Conjecture" was that relations which hold in *almost all* relativized world are exactly those holding in the real (unrelativized) world. However, this conjecture has been refuted; we have mentioned above one such refutation (i.e., $\text{co}\mathcal{NP} \subseteq \mathcal{IP}$ and yet $\text{co}\mathcal{NP}^A \not\subseteq \mathcal{IP}^A$ for almost all oracles A).

We mention that not only that we have results which contradict the above naive hopes, but also the way these results are proven seems to indicate that the structure of computation of an oracle machine has little to do with the computation of an analogous regular machine. Add to this our initial comment by which not any class can be relativized (i.e., relativization requires that the class be defined in terms of some type of machines and that these machine extend naturally – at least in a syntaxtic sense – to orcale machines). We stress again that such extensions, even when they seem syntaxtically natural, may totally disrupt the "semantics" associated with the bare machine.

Our conclusion is that relativized results are poor predictors of non-relativized results. The advocates of relativization are certainly aware of the above, and their current advocacy is based on the belief that relativized results indicate limitations of proof techniques which may be applied in the real (unrelativized) world. That is –

Relativized results as indicating limitations of proof techniques. The thesis (of the above advocates) is that a relativized result asserting some relation indicates that the contrary relation *cannot be proven* in the real (unrelativized) world *via* "a proof which relativizes". This thesis is based on the hidden assumption that there is a natural (generic) way of extending a real (unrelativized) proof into a version which refers to the corresponding oracle classes. The thesis suggests that one should consider the question of whether such a natural extention is indeed valid in the current case. Suppose that some result fails relative to some oracle, then the thesis asserts that any proof for which the natural relativization remains valid will fail to prove the real (unrelativized) result (since it would have proven the same result also relative to any oracle, whereas the relativized result asserts the existence of an oracle for which the result does not hold).

Our objection to the above thesis is based on the objection to the assumption that such a natural (generic) way of extending a real (unrelativized) proof into a version which refers to the "corresponding oracle classes" exists. Indeed certain known proofs (e.g., the time hierarchy) extend naturally to the relativized world, but we don't see a generic procedure of transforming real (unrelativized) proofs into relativized ones. (Recall our warning that not all complexity classes have a relativized version.) Without such generic procedure of transforming unrelativized proofs into relativized ones, it is not clear what is suggested by the above thesis. Furthermore, one can transform any proof which is known to "relativize" into a proof (of the same statement) that does not "relativize" (via the process witnessing the former "relativization"). Of course this transformation can be undone by a new transformation, but the latter can not be considered generic in any natural sense.

What is left is a suggestion to use relativized results as a tool for testing the viability of certain approaches for proving real (unrelativized) results. That is –

Relativized results as a debugging tool. Suppose one has a vague idea on how to prove some result regarding the real world. Further suppose that one is not sure whether this idea can be carried out. Then the suggestion is to ask whether the same idea seems to apply also to the relativized world and whether a contrary relativization result is known. The thesis is that if the answer to both questions is possitive then one should give-up hope that the idea can work (in the real world).

We object even to this minimalistic suggestion. If one has a vague idea which may or may not work in the real work, possibly depending on unspecified details, then why should one believe in the validity of statements inferred from applying a vague procedure (i.e., "relativization") to this vague idea? Indeed, in some cases the (relativized) indication obtained (as suggested above) may save time which could have been wasted pursuing an unfruitful idea, but in some cases it may cause to abandon a fruitful idea. We see no reason to believe that the probability of the first event is greater than that of the second, and point out that the pay-off (benefit/damage) seems to be much greater in the second.

Bibliographic Notes

The study of relativized complexity classes was initiated in [1], which shows that the P vs NP question has contradicting relativizations. This came as a surprise since in recursive function theory, the origin of the relativization paradigm, all standard results do hold in all relativized worlds. Thus, relativization was taken as an indication that the main technique of recursive function theory –

that is diagonalization – can not settle questions such as P vs NP.[1] The "curse of contradicting relativizations" was eliminated in [2] that considers the measure of oracles (or relativized worlds) for which certain relations hold. The authors also postulated the Random Oracle Hypothesis that essentially states that structural relationships which hold in almost all oracle worlds also hold in the unrelativized case (i.e., the real world). Several refutations of the Random Oracle Hypothesis are known: For example, co$\mathcal{NP} \subseteq \mathcal{IP}$ (see Lecture 11 or [4]), whereas this fails in almost all oracle worlds (see [3]).

A different approach towards investigation of the limitation of certain proofs was taken in [5]. The latter work defines a framework for proving circuit size lower bounds and certain features, called *natural*, of proofs within this framework. It then shows that the existence of natural proofs contradicts the existence of corresponding pseudorandom functions, which in turn implies that the existence of pseudorandom functions cannot be proven via natural proofs.[2]

1. T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SIAM Journal on Computing*, Vol. 4 (4), pages 431–442, December 1975.

2. C. Bennett and J. Gill. Relative to a random oracle A, $\mathcal{P}^A \neq \mathcal{NP}^A \neq \text{co}\mathcal{NP}^A$ with probability 1. *SIAM Journal on Computing*, Vol. 10 (1), pages 96–113, February 1981.

3. R. Chang, B. Chor, O. Goldreich, J. Hartmanis, J. Håstad, D. Ranjan, and P. Rohatgi. The Random Oracle Hypothesis is False. *JCSS*, Vol. 49 (1), pages 24–39, 1994.

4. C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic Methods for Interactive Proof Systems. *JACM*, Vol. 39 (4), pages 859–868, 1992.

5. A.R. Razborov and S. Rudich. Natural Proofs. *JCSS*, Vol. 55 (1), pages 24–35, 1997.

[1] We consider the latter sentence to be on the verge on non-sense: It is not clear what is meant by saying that a proof technique cannot settle a certain question. In contrast, saying that a proof with certain well-defined properties cannot settle questions of certain properties may be clear and useful. That is, whereas the notion of a proof technique (i.e., a proof which uses a certain technique) is undefined, one may define properties of proofs. An inspiring case in which the latter was done is the formulation and study of Natural Proofs [5].

[2] Our original intention was to discuss also Natural Proofs in the lecture. However, since the students did not see circuit size lower bounds in this course, the former plan was abandoned.