600.363 Introduction to Algorithms / 600.463 Algorithms I **Lecturer:** Michael Dinitz
Topic: Approximation algorithms **Date:** 11/18/14

23.1 Introduction

We spent last week proving that for certain problems, we can't expect to find the optimal solution in polynomial time. What do we do in the face of NP-completeness? There are a few options, including just giving up on proving theorems and designing algorithms that we hope give "good-enough" solutions. Let's not give up quite yet, though. Instead, let's try to design *approximation algorithms*: algorithms which run in polynomial time, and give solutions that are *provably* not too far from the optimal solution. We might not be able to find the optimal solution on polynomial time, but maybe we can find a solution that costs at most twice as much.

Definition 23.1.1 *Let $\mathcal{A}$ be some (minimization) problem, and let I be an instance of that problem. Let $OPT(I)$ be the value of the optimal solution on that instance. Let ALG be a polynomial-time algorithm for $\mathcal{A}$, and let $ALG(I)$ denote the value of the solution returned by ALG on instance I. Then we say that ALG is an α-*approximation *if*

$$\frac{ALG(I)}{OPT(I)} \leq \alpha$$

for all instances I of $\mathcal{A}$.

One interesting thing to note is that, as we saw last week, the theory of NP-completeness tells us that all NP-complete problems are in some sense equally hard – f we could solve one of them in polynomial time, then we could solve all of them. However, it is *not* true that they are all equally hard to approximate. Some can be approximated extremely well in polynomial time, and some cannot be approximated at all. So sometimes it is useful to think of approximability as a more "fine-grained" notion of hardness: while all NP-complete problems are hard, problems which can be approximated within 2 are easier then problems which can only be approximated within $\Theta(\log n)$, which are easier than problems which can only be approximated within $\Theta(\sqrt{n})$, etc.

23.2 Vertex Cover

For our first example, let's go back to the vertex cover problem.

Definition 23.2.1 *Let $G = (V, E)$ be a graph. Then $M \subseteq V$ is a* vertex cover *if for every edge $\{u, v\} \in E$, at least one of u, v is in M.*

In other words, a vertex cover is a set of vertices with the property that every edge has at least one endpoint in the set. In the vertex cover problem, we are given a graph $G = (V, E)$ and are asked to find the *smallest* vertex cover. We saw that this problem is NP-complete, so we cannot expect to actually solve it. What can we do instead?

1. Idea 1: Pick an arbitrary vertex with at least one uncovered edge incident on it, add it to the

cover, and repeat. Unfortunately this is arbitrarily far from optimal: see the star graph.

2. Idea 2: Instead of picking arbitrarily, let's try to pick smartly. In particular, an obvious thing to try is the greedy algorithm: pick the vertex with the largest number of uncovered edges incident to it, and add it to the cover. Repeat until all edges are covered.

   While this is a better idea, it's still not very good. Consider a bipartite graph with t nodes on the left side. We build the right side by first dividing it into 2 groups, and creating two right nodes for each group (each right node is adjacent to the $t/2$ left nodes in its group). We then repeat this for groups of size $\lfloor t/3 \rfloor, \lfloor t/4 \rfloor, \dots, 1$. Then it is not hard to see that $OPT(I) = t$, since we can just use the t left nodes as the vertex cover. But if we break ties poorly, the greedy algorithm might first pick the one right node of degree t, then the two right nodes of degree $t/2$, etc. So the algorithm gives a vertex cover if size $t/2 + t/3 + t/4 + \dots 1 = \Theta(t \log t)$. So this algorithm certainly doesn't give an $o(\log n)$-approximation (it does in fact give an $O(\log n)$-approximation, although we won't prove it).

OK, so now let's find some algorithms which work better. Here's an algorithm which sounds stupid but is actually pretty good:

1. Pick an arbitrary edge which is not yet covered. Add *both* endpoints to the cover, and repeat.

   I claim that this algorithm is a 2-approximation. To see this, suppose that our algorithm took k iterations, and let L be the set of edges that it selected as uncovered and included both of the endpoints (so $|L| = k$). The the algorithm returns a vertex cover of size $2k$. On the other hand, note that these edges form a matching, i.e. they all have distinct endpoints. This means that *any* vertex cover needs to have size at least k, just in order to cover the edges in L!. Thus $ALG \leq 2 \times OPT$, so it is a 2-approximation.

Now let's see a more involved way to get a 2-approximation. At first this will seem unnecessarily complicated, but we'll see later why it is useful.

2. Let $G = (V, E)$ be an instance of vertex cover, and consider the following linear program:

$$\begin{array}{lll} \min & \sum_{u \in V} x_u & \\ \text{subject to} & x_u + x_v \geq 1 & \forall \{u, v\} \in E \\ & 0 \leq x_u \leq 1 & \forall u \in V \end{array}$$

   Our algorithm solves this LP, and then for every vertex v we include v in our cover if $x_v \geq 1/2$.

   We first claim that our algorithm does indeed give a vertex cover. To see this, consider an arbitrary edge $\{u, v\} \in E$. Then the LP has a constraint $x_u + x_v \geq 1$, so clearly either x_u or x_v (or both) is at least $1/2$. Thus at least one endpoint of the edge will be contained in the cover. Since this holds true of all edges, we get a valid vertex cover.

   So now we want to prove the the vertex cover we return is a 2-approximation. Let $LP(G)$ denote the value of the above LP on G, let $OPT(G)$ denote the size of the minimum vertex cover, and let $ALG(G)$ denote the size of the vertex that we return.

First, note that $LP(G) \leq OPT(G)$: this is true because for each vertex cover, there is an LP solution with value exactly equal to the size of the cover (just set $x_u = 1$ if u is in the cover and $x_u = 0$ otherwise). Since the LP is trying to minimize an objective, it does *at least* as well as the smallest vertex cover. On the other hand, we have that $ALG(G) \leq 2 \times LP(G)$. This is because we have at most doubled each LP value: if we choose to include a vertex then it costs the algorithm 1, but by construction it must have cost the LP at least 1/2.

The second algorithm is known as an *LP rounding* algorithm: we first solve an LP which if we could enforce integrality would correspond exactly to the problem we want to solve. Since we can't enforce integrality we instead get back fractional value. Then we somehow want to round these fractional values to integers (usually to 0 or 1). This algorithm is a version of *threshold rounding*, since all we did was set a threshold and round up to 1 any value above the threshold and round down to 0 any value below the threshold. But there are many other types of LP rounding. The most famous is probably *randomized rounding*, where we interpret each LP variable as a probability and set it to 1 with probability equal to its value.

Since we already had a super simple 2-approximation, why did we bother with the LP rounding algorithm? One really nice feature of LP rounding algorithms is that they tend to be extremely flexible to slight changes in the problem. For example, consider the *weighted* vertex cover problem, in which each vertex v also has a positive weight w_v and we try to find the minimum *weight* vertex cover. It is not at all clear how to adapt our first algorithm to this setting. But with the LP algorithm it's trivial – we just change the objective function to $\min \sum_{v \in V} w_v x_v$. Then the rest of the analysis essentially goes through without change! Rounding with a threshold of 1/2 still gives a 2-approximation!

23.3 Set Cover

A fundamental problem which we have not really talked about yet is *set cover*. This is a generalization of vertex cover and is really problem 1a from your last homework in disguise (technically that problem is called Hitting Set, but see if you can convince yourself that Hitting Set and Set Cover are really the same thing).

The input to set cover is a set X of n items, and m subsets $S_1, \ldots, S_m$ of X. The goal is to find the minimum number of these subsets necessary to cover all of the items (in the homework the goal was the inverse: find the minimum number of items necessary to hit all of the sets).

We now know that Set Cover is NP-complete. However, it turns out that the simple greedy algorithm does pretty well. This algorithm picks the set which covers the most items, throws out all of the items covered, and repeats. In other words, we just always pick the set which covers the most uncovered items.

Theorem 23.3.1 *The greedy algorithm is a* $(\ln n)$*-approximation algorithm for Set Cover.*

Proof: Suppose that the optimal set cover has size k. Then there must be some set which covers at least a $1/k$ fraction of the items. Since we always pick the best remaining set, this means that after the first iteration there are at most $(1 - \frac{1}{k})n$ items still uncovered. But now the logic continues to hold! No matter what items have been covered already, the original optimal solution certainly covers all of the remaining items (since it covers all items). So there must be some set from that solution which covers at least a $1/k$ fraction of the remaining items.

Thus after i iterations, the number of uncovered items is at most $(1-\frac{1}{k})^i n$. So after $k \ln n$ iterations, the number of uncovered items is at most $(1-\frac{1}{k})^{k \ln n} n < e^{-\ln n} n = 1$. Thus we are finished after $k \ln n$ iterations and so we chose at most $k \ln n$ sets, which immediately implies that it's the greedy algorithm is a $(\ln n)$-approximation ■

In fact, we know that this is the best possible approximation for Set Cover: it is NP-hard to give an $o(\log n)$-approximation ratio, and under a stronger complexity assumption (that problems in NP cannot be solved in $n^{O(\log \log n)}$ time) the lower bound can be improved to $(1-\epsilon)\ln n$ for arbitrarily small constant $\epsilon > 0$ (this is due to Uri Feige).

23.4 Max-3SAT

In the 3SAT problem we are given a 3-CNF boolean formula (i.e. a formula which was the AND of a collection of clauses, and each clause was the OR of at most 3 literals) and asked to determine if there exists a satisfying assignment. A natural optimization version of this problem is Max-3SAT, in which we are again given a collection of clauses, each of which is the OR of at most 3 literals, but where our goal is to find an assignment which maximizes the number of satisfied clauses. So if there is a fully satisfying assignment then we hope to find it, but if not then we try to find an assignment which satisfies as many clauses as possible. We'll actually spend most of our time discussing Max-E3SAT (or maximum *exact* 3SAT) in which every clause has *exactly* three literals.

First, one technicality: since this is a maximization problem, we want to redefine an α-approximation to mean that $ALG(I)/OPT(I) \geq \alpha$ for all instances I (rather than $\leq \alpha$).

Here's an easy randomized algorithm: simply use a random assignment! It turns out that this is a pretty good approximation.

Theorem 23.4.1 *A random assignment is (in expectation) a 7/8-approximation for Max-E3SAT.*

Proof: Consider a clause. Since there are exactly 3 literals in it, there are exactly $2^3 = 8$ possible assignments to it, of which 7 are satisfying. So the probability that a random assignment satisfies it is exactly 7/8. Now linearity of expectations lets us apply this to all of the clauses: the expected number of clauses satisfied is exactly $(7/8)m$, where m is the total number of clauses. Since $OPT(I) \leq m$, this implies that a random assignment is in expectation a 7/8-approximation. ■

What if we want a *deterministic* algorithm? This turns out to be reasonably simple based on a derandomization of the randomized algorithm using what is known as the "method of conditional expectations".

Consider the first variable x_1. If we set it to 0, then some clauses become satisfied (those with $\bar{x}_1$ as a literal) and some others can no longer be satisfied by x_1 (those with x_1 as a literal). So we can calculate the number of expected clauses that would be satisfied if we set x_1 to 0 and set all of the other variables randomly. Similarly, we can calculate the number of expected clauses that would be satisfied if we set x_1 to 1 and set all of the other variables randomly. Then we'll just set x_1

according to whichever of these expectations is larger. And now that x_1 is set we can do the same thing with x_2 (where now x_1 is fixed), x_3, etc., until in the end we have fixed an assignment. Since we always pick whichever of the two expectations is larger, it never goes down, and thus stays at least 7/8. So in the end we have satisfied at least 7/8 of the clauses.