601.435/635 Approximation Algorithms **Lecturer:** Michael Dinitz
Topic: Intro, Vertex Cover **Date:** 1/29/19
Scribe: Michael Dinitz

1.1 Introduction, Course Info

Welcome to Approximation Algorithms! A quick overview of the syllabus, and a few notes:

- Who am I: I'm an assistant professor in the theory group, where I focus on the theory of algorithms, particularly approximation algorithms and algorithmic problems arising from computer networking and distributed computing (of my last ten papers: about 5 are approximation algorithms, 5 are networking/distributed computing, 2 are extremal graph theory, 1 is algorithmic game theory. Obviously overlap). I particularly like "network design problems", which we'll talk about pretty extensively in this course, starting either today or Thursday.
- TA: Yasamin Nazari, one of my PhD students. She works mostly on approximation algorithms in distributed settings.
- Prerequisite: Intro to Algorithms. This class will be extremely difficult (if not impossible) if you're not familiar with the content from that class.
- This is designed to be a PhD-level class. I'm excited that others are taking it (particularly all the undergrads!), but it's still designed for PhD students. What does this mean? Among other things, the following.
  - The course will not have as strict a timetable/schedule as an undergrad course. The schedule, including lecture topics, homework due dates, etc., will be played a bit by ear.
  - There won't be a lot of handholding. I will give the high level view and some details, but I'll also assume that you can fill in the blanks. But please come to office hours or get in touch with me if there's something you don't understand!
  - Hopefully this class won't be as much work as my undergraduate class (Intro to Algorithms), since PhD students should mostly be focused on research. But it will definitely move faster.
- Textbook: *The Design of Approximation Algorithms*, by Williamson and Shmoys. You can download a copy from `http://www.designofapproxalgs.com/`. We'll be taking a lot of liberties with the book, though – we'll be following it only very loosely.
- Course website: `http://www.cs.jhu.edu/~mdinitz/classes/ApproxAlgorithms/Spring2019/`.
- Online discussion group: `https://piazza.com/jhu/spring2019/601435635/home`.
- Grade breakdown: 50% homework, 30% final exam, 20% participation.

- Homeworks: probably around 5. Can work in groups of up to three people, but must write up solutions separately and turn them in separately (on gradescope).
- Final Exam: will be a normal final exam.
- Participation: participate in lecture and on piazza, come to office hours, etc.

1.2 Approximation Algorithms: What and Why

Recall the following (informal) definitions from undergrad algorithms:

- P: problems decidable / solvable in polynomial time
- NP: problems decidable/solvable in *nondeterministic* polynomial time. Equivalently: solutions can be written down and verified in polynomial time.
- NP-hard: Problems that all problems in NP reduce to in polynomial. Solving one in polynomial time would imply that P = NP
- NP-complete: in NP and NP-hard.

Our main motivation (but not the only possible motivation) for studying approximation algorithms is that we believe that P $\neq$ NP, but we still want to solve NP-hard optimization problems. So what can we do? Let's actually think in a bit more detail about what we "want" that NP-hardness is telling us we can't do. Given an optimization problem, we would like to:

1. find the optimal solution,
2. in polynomial time,
3. for every instance.

Assuming the P $\neq$ NP, we cannot get all three of this. What should we give up on? If we give up on the third condition, we start designing algorithms for special cases (e.g., "random" instances). If we give up on the second condition, we start designing algorithm that are (hopefully) fast in practice but might be slow in theory. This, for example, is where all of the integer programming work in the OR community is. If we give up on the first condition (but don't give up on theory), then we get the topic of this class: approximation algorithms.

Slightly more formally (but still pretty informally), a *problem* consists of three things:

1. An input / instance,
2. Description of the feasible solutions for the input, and
3. An objective function.

Let's see an example.

Definition 1.2.1 *The* Vertex Cover *problem is defined as follows:*

- *Input: Graph* $G = (V, E)$
- *Feasible solutions:* $V' \subseteq V$ *such that* $|e \cap V'| \geq 1$ *for all* $e \in E$
- *Objective: minimize* $|V'|$

Vertex Cover is known to be NP-hard (we proved this in 433/633).

Definition 1.2.2 *The* optimal solution *is the feasible solution with the best objective value.*

Definition 1.2.3 *Let* $\mathcal{A}$ *be some 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* ALG *always returns a feasible solution, runs in polynomial time, and if*

$$\frac{ALG(I)}{OPT(I)} \leq \alpha \text{ for all instances } I \text{ of } \mathcal{A}, \text{ where } \mathcal{A} \text{ is a minimization problem}$$
$$\frac{ALG(I)}{OPT(I)} \geq \alpha \text{ for all instances } I \text{ of } \mathcal{A}, \text{ where } \mathcal{A} \text{ is a maximization problem}$$

The value α above is called the *approximation ratio* or *approximation factor*.

Note: as defined $\alpha \geq 1$ for minimization problems and $\alpha \leq 1$ for maximization problems. We could change the definition for the maximization case to be $OPT(I)/ALG(I) \leq \alpha$, in which case α would be at least 1 for both cases. Sometimes we'll do this – it should be clear from context. Ask if you're not sure.

So for Vertex Cover, and α-approximation would be an algorithm which always returns a feasible vertex cover with at most α times as many vertices as the optimal solution.

So why study approximation algorithms? There are a few reasons, aside from the fact that it's fun.

- We want to solve NP-hard problems! If we can't solve them, still want to get as close as possible, and don't want to give up on theoretical guarantees.
- "Fine-grained" hardness. By definition, all NP-complete problems can be reduced to each other, so they are in some sense "equally hard". But problems with a 1.01-approximation are "easier" than problems with only an $O(\log n)$-approximation. So approximability let's us make more nuanced distinctions between NP-complete problems.
- Forcing ourselves to still give worst-case guarantees helps us develop new and useful algorithmic techniques.

1.3 Approximating Vertex Cover

Let's try to design our first approximation algorithm. What would be some natural approaches?

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. We'll give a quick overview of the counterexample, but fleshing out the details is a good exercise to do at home. Consider a set U of t nodes. For every $i \in \{2, 3, \ldots, t\}$, divide U into $\lfloor t/i \rfloor$ disjoint groups of size exactly i (if i does not divide t, then there will be less than i nodes which are not in any group). Let the groups for value i be $G^i_1, G^i_2, \ldots, G^i_{\lfloor t/i \rfloor}$. For every $i \in \{2, \ldots, t\}$ and every $j \in [\lfloor t/i \rfloor]$, we create a vertex v^i_j and add edges between v^i_j and every node in G^i_j. We refer to the nodes $\{v^i_j\}_{j \in [\lfloor t/i \rfloor]}$ as *layer* i nodes.

   It is easy to see that in this graph, every node in U has degree at most $t-1$ since it is adjacent to at most one node from each layer. Every node in layer i has degree exactly i. So at the beginning of the algorithm, the maximum degree node is the one node v^t_1 in layer t. After we pick this node, the nodes in U now have degree at most $t-2$, so the algorithm woul next pick the node in layer $t-1$. It is easy to see by induction that this will continue: the algorithm will always choose the nodes in the largest remaining layer rather than the nodes in U.

   Notice that $OPT(G) \leq t$, since U itself is a vertex cover of size t. On the other hand, we just argued that

   $$ALG(G) = \sum_{i=2}^{t} \left\lfloor \frac{t}{i} \right\rfloor \geq \sum_{i=2}^{i} \left(\frac{1}{2} \cdot \frac{t}{i} \right) = \frac{t}{2} \sum_{i=2}^{t} \frac{1}{i} = \Omega(t \log t).$$

   Since $ALG(G)/OPT(G) \geq \log t$ is not a constant, the greedy algorithm is not an α-approximation for any constant α.

OK, so now let's find some algorithms which work better. Here's an algorithm which sounds stupid but is actually pretty good: Pick an arbitrary edge which is not yet covered. Add *both* endpoints to the cover, and repeat. More formally, consider the following algorithm:

- $S = \emptyset$
- While $E \neq \emptyset$:
  - Let $\{u, v\} \in E$ be an arbitrary edge
  - $S \leftarrow S \cup \{u, v\}$
  - Remove u, v, and all their incident edges from G

- Return S.

I claim that this algorithm is a 2-approximation. First, let's show that it runs in polynomial time.

Lemma 1.3.1 *The algorithm runs in polynomial time*

Proof: In every iteration two vertices are deleted from G. Thus there are at most $n/2$ iterations. Clearly each iteration takes polynomial time, and thus the total running time is polynomial. ■

Now let's show that it returns a feasible solution.

Lemma 1.3.2 *S is a vertex cover (a feasible solution).*

Proof: Consider an arbitrary edge $\{u, v\} \in E$. It was deleted in some iteration, since the algorithm terminates with no remaining edges. In the iteration in which it was deleted, either u or v (or both) was added to S. Thus $|S \cap \{u, v\}| \geq 1$. ■

Now we'll start to prove that approximation ratio. A simple definition and lemma will help us.

Definition 1.3.3 *$M \subseteq E$ is a* matching *if no two edges of M share an endpoint, i.e., if $e \cap e' = \emptyset$ for all $e, e' \in M$,*

Lemma 1.3.4 *Let M be a matching and let S be a vertex cover. Then $|S| \geq |M|$.*

Proof: Every $e \in M$ must have at least one endpoint in S. Since no two edges in M share an endpoint, $|S| \geq |M|$. ■

Theorem 1.3.5 *The algorithm is a 2-approximation for* Vertex Cover

Proof: Lemma 1.3.1 implies that the algorithm runs in polynomial time, and Lemma 1.3.2 implies that it always returns a feasible solution. So we just need to prove that it has approximation ratio of at most 2.

Let S^* denote the optimal vertex cover, and let M be the set of all edges selected in the first line of the while loop. Then by construction, $|S| = 2|M|$, since for every edge in M we include both endpoints in S. Moreover, M is a matching, since after we choose an edge we delete both of its endpoints, so no future edges we choose can share an endpoint with it. So by Lemma 1.3.4 we know that $|S^*| \geq |M|$. Hence $|S| \leq 2|M| \leq 2|S^*|$. ■

So we have our first approximation algorithm!

One more note about this: what was the key step? If you think about it, the key step in the analysis was Lemma 1.3.4, which let us claim that $|S^*| \geq |M|$. We wanted to show that $|S| \leq 2|S^*|$, and since we designed the algorithm we can usually say something about $|S|$. But relating that to $|S^*|$ is the tricky bit. This step, of lower bounding the optimal solution (or for a maximization problem of upper bounding it), is often the trickiest part of designing approximation algorithms. Later in the semester we'll talk about techniques that make this step a little easier / more automatic.

Whenever we design an approximation algorithm, we should keep two questions in mind.

1. Question: Is the *analysis* tight? That is, is 2 the best possible approximation ratio we could prove *for this algorithm*?

Answer: Yes. Many easy examples, such as $K_{n,n}$, where the algorithm picks exactly twice as many vertices as is necessary.

2. Question: Is the *algorithm* tight? That is, is there a different algorithm with a better approximation ratio? This is usually a *much* tougher question.

   Answer: Yes and no. The best known algorithm is a $(2 - \frac{1}{\sqrt{\log n}})$-approximation due to Karakostas [Kar09]. Assuming $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial-time algorithm which gives better than a $10\sqrt{5} - 21 \approx 1.3606$ approximation [DS05] (in other words, it is NP-hard not just to solve Vertex Cover, but to give a better than $10\sqrt{5} - 21$ approximation). If we make an assumption stronger than $\mathbf{P} \neq \mathbf{NP}$ known as the *Unique Games Conjecture*, then for any constant $\epsilon > 0$ there is no polynomial-time algorithm which is better than a $(2 - \epsilon)$-approximation [KR08].

References

[BGRS13] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. Steiner tree approximation via iterative randomized rounding. *J. ACM*, 60(1):6:1–6:33, 2013.

[DS05] Irit Dinur and Samuel Safra. On the hardness of approximating minimum vertex cover. *Ann. of Math. (2)*, 162(1):439–485, 2005.

[Kar09] George Karakostas. A better approximation ratio for the vertex cover problem. *ACM Trans. Algorithms*, 5(4):41:1–41:8, November 2009.

[KR08] Subhash Khot and Oded Regev. Vertex cover might be hard to approximate to within $2-\varepsilon$. *Journal of Computer and System Sciences*, 74(3):335 – 349, 2008. Computational Complexity 2003.