Suppose we are given an **NP**-complete problem to solve. Even though (assuming $\mathbf{P} \neq \mathbf{NP}$) we can't hope for a polynomial-time algorithm that always gets the best solution, can we develop polynomial-time algorithms that always produce a "pretty good" solution? In this lecture we consider such *approximation algorithms*, for several important problems. Specific topics in this lecture include:

- 2-approximation for vertex cover via greedy matchings.
- 2-approximation for vertex cover via LP rounding.
- Greedy $O(\log n)$ approximation for set-cover.
- Approximation algorithms for MAX-SAT.

1 Introduction

Suppose we are given a problem for which (perhaps because it is **NP**-complete) we can't hope for a fast algorithm that always gets the best solution. We can't just throw our hands in the air and say "We can't do anything!" We should do something smart.

In particular, can we hope for a fast algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial?

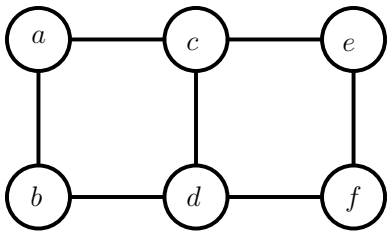
As seen in the last two lectures, the class of **NP**-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in **NP**). However, the difficulty of getting a good approximation to these problems varies quite a bit. In this lecture we will examine several important **NP**-complete problems and look at to what extent we can guarantee to get approximately optimal solutions, and by what algorithms.

2 Vertex Cover

Recall that a *vertex cover* in a graph is a set of vertices such that every edge is incident to (touches) at least one of them. The vertex cover problem is to find the smallest such set of vertices.

Definition 1 VERTEX-COVER*: Given a graph G, find the smallest set of vertices such that every edge is incident to at least one of them. Decision problem: "Given G and integer k, does G contain a vertex cover of size $\leq k$?"*

For instance, this problem is like asking: what is the fewest number of guards we need to place in a museum in order to cover all the corridors.



Exercise: Find a vertex cover in the graph above of size 3. Show that there is no vertex cover of size 2 in this graph.

As we saw last time (via a reduction from INDEPENDENT SET), this problem is **NP**-hard. However, it turns out that for any graph G we can at least get within a factor of 2. That is, if the graph G has a vertex cover of size k^*, we can return a vertex cover of size at most $2k^*$.

Let's start first, though, with some strategies that *don't* work.

Strawman Alg #1: Pick an arbitrary vertex with at least one uncovered edge incident to it, put it into the cover, and repeat.

What would be a bad example for this algorithm? [Answer: how about a star graph]

Strawman Alg #2: How about picking the vertex that covers the *most* uncovered edges. This is very natural, but unfortunately it turns out this doesn't work either, and it can produce a solution $\Omega(\log n)$ times larger than optimal.[1]

How can we get factor of 2? It turns out there are actually several ways. We will discuss here two quite different algorithms. Interestingly, while we have several algorithms for achieving a factor of 2, nobody knows if it is possible to efficiently achieve a factor 1.99.

Algorithm 1: Pick an arbitrary edge. We know any vertex cover must have at least 1 endpoint of it, so let's take *both* endpoints. Then, throw out all edges covered and repeat. Keep going until there are no uncovered edges left.

Theorem 2 *The above algorithm is a factor 2 approximation to* VERTEX-COVER.

Proof: What Algorithm 1 finds in the end is a matching (a set of edges no two of which share an endpoint) that is "maximal" (meaning that you can't add any more edges to it and keep it a matching). This means if we take both endpoints of those edges. we must have a vertex cover. In particular, if the algorithm picked k edges, the vertex cover found has size $2k$. But, *any* vertex cover must have size at least k since it needs to have at least one endpoint of each of these edges, and since these edges don't touch, these are k *different* vertices. So the algorithm is a 2-approximation as desired. ■

Here is now another 2-approximation algorithm for Vertex Cover:

[1]The bad examples for this algorithm are a bit more complicated however. One such example is as follows. Create a bipartite graph with a set S_L of t nodes on the left, and then a collection of sets $S_{R,1}, S_{R,2}, \ldots$ of nodes on the right, where set $S_{R,i}$ has $\lfloor t/i \rfloor$ nodes in it. So, overall there are $n = \Theta(t \log t)$ nodes. We now connect each set $S_{R,i}$ to S_L so that each node $v \in S_{R,i}$ has i neighbors in S_L and no two vertices in $S_{R,i}$ share any neighbors in common (we can do that since $S_{R,i}$ has at most t/i nodes). Now, the optimal vertex cover is simply the set S_L of size t, but this greedy algorithm might first choose $S_{R,t}$ then $S_{R,t-1}$, and so on down to $S_{R,1}$, finding a cover of total size $n - t$. Of course, the fact that the bad cases are complicated means this algorithm might not be so bad in practice.

Algorithm 2: First, solve a *fractional* version of the problem. Have a variable x_i for each vertex with constraint $0 \leq x_i \leq 1$. Think of $x_i = 1$ as picking the vertex, and $x_i = 0$ as not picking it, and in-between as "partially picking it". Then for each edge (i, j), add the constraint that it should be covered in that we require $x_i + x_j \geq 1$. Then our goal is to minimize $\sum_i x_i$.

We can solve this using linear programming. This is called an "LP relaxation" because any true vertex cover is a feasible solution, but we've made the problem easier by allowing fractional solutions too. So, the value of the optimal solution now will be at least as good as the smallest vertex cover, maybe even better (i.e., smaller), but it just might not be legal any more. [Give examples of triangle-graph and star-graph]

Now that we have a super-optimal fractional solution, we need to somehow convert that into a legal integral solution. We can do that here by just picking each vertex i such that $x_i \geq 1/2$. This step is called *rounding* of the linear program (which literally is what we are doing here by rounding the fraction to the nearest integer — for other problems, the "rounding" step might not be so simple).

Theorem 3 *The above algorithm is a factor 2 approximation to* Vertex-Cover.

Proof: Claim 1: the output of the algorithm is a legal vertex cover. Why? [get at least 1 endpt of each edge]

Claim 2: The size of the vertex cover found is at most twice the size of the optimal vertex cover. Why? Let OPT_{frac} be the value of the optimal fractional solution, and let OPT_{VC} be the size of the smallest vertex cover. First, as we noted above, $OPT_{frac} \leq OPT_{VC}$. Second, our solution has cost at most $2 \cdot OPT_{frac}$ since it's no worse than doubling and rounding down. So, put together, our solution has cost at most $2 \cdot OPT_{VC}$. ■

Interesting fact: nobody knows any algorithm with approximation ratio 1.9. Best known is $2 - O(1/\sqrt{\log n})$, which is $2 - o(1)$.

Current best hardness result: Hastad shows 7/6 is NP-hard. Improved to 1.361 by Dinur and Safra. Beating 2-epsilon has been related to some other open problems (it is "unique games hard"), but is not known to be NP-hard.

3 Set Cover

The Set-Cover problem is defined as follows:

Definition 4 Set-Cover*: Given a domain X of n points, and m subsets $S_1, S_2, \ldots, S_m$ of these points. Goal: find the fewest number of these subsets needed to cover all the points. The decision problem also provides a number k and asks whether it is possible to cover all the points using k or fewer sets.*

Set-Cover is NP-Complete. However, there is a simple algorithm that gets an approximation ratio of $\ln n$ (i.e., that finds a cover using at most a factor $\ln n$ more sets than the optimal solution).

Greedy Algorithm (Set-Cover**):** Pick the set that covers the most points. Throw out all the points covered. Repeat.

What's an example where this algorithm *doesn't* find the best solution?

Theorem 5 *If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $k \ln n$ sets.*

Proof: Since the optimal solution uses k sets, there must some set that covers at least a $1/k$ fraction of the points. The algorithm chooses the set that covers the most points, so it covers at least that many. Therefore, after the first iteration of the algorithm, there are at most $n(1-1/k)$ points left. Again, since the optimal solution uses k sets, there must some set that covers at least a $1/k$fraction of the remainder (if we got lucky we might have chosen one of the sets used by the optimal solution and so there are actually $k-1$ sets covering the remainder, but we can't count on that necessarily happening). So, again, since we choose the set that covers the most points remaining, after the second iteration, there are at most $n(1-1/k)^2$ points left. More generally, after t rounds, there are at most $n(1-1/k)^t$ points left. After $t = k \ln n$ rounds, there are at most $n(1-1/k)^{k \ln n} < n(1/e)^{\ln n} = 1$ points left, which means we must be done. ■

Notice how the above analysis is similar to the analysis we used of Edmonds-Karp #1. Also, you can get a slightly better bound by using the fact that after $k \ln(n/k)$ rounds, there are at most $n(1/e)^{\ln(n/k)} = k$ points left, and (since each new set covers at least one point) you only need to go k more steps. This gives the somewhat better bound of $k \ln(n/k) + k$. So, we have:

Theorem 6 *If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $k \ln(n/k) + k$ sets.*

In fact, it's been proven that unless everything in **NP** can be solved in time $n^{O(\log \log n)}$, then you can't get better than $(1-\epsilon)\ln(n)$ for any constant $\epsilon > 0$ [Feige].

4 Max-SAT

The MAX-SAT problem is defined as follows:

Definition 7 MAX-SAT*: Given a CNF formula (like in SAT), try to maximize the number of clauses satisfied.*

To make things cleaner, let's assume we have reduced each clause [so, $(x \vee x \vee y)$ would become just $(x \vee y)$, and $(x \vee \neg x)$ would be removed]

Theorem 8 *If every clause has size exactly 3 (this is sometimes called the* MAX-EXACTLY-3-SAT PROBLEM*), then there is a simple randomized algorithm can satisfy at least a 7/8 fraction of clauses. So, this is for sure at least a 7/8-approximation.*

Proof: Just try a random assignment to the variables. Each clause has a 7/8 chance of being satisfied. So if there are m clauses total, the expected number satisfied is $(7/8)m$. If the assignment satisfies less, just repeat. Since the number of clauses satisfied is bounded (it's an integer between 0 and m), with high probability it won't take too many tries before you do at least as well as the expectation. ■

How about a deterministic algorithm? Here's a nice way we can do that. First, let's generalize the above statement to talk about general CNF formulas.

Claim 9 *Suppose we have a CNF formula of m clauses, with m_1 clauses of size 1, m_2 of size 2, etc. ($m = m_1 + m_2 + \ldots$). Then a random assignment satisfies $\sum_k m_k(1-1/2^k)$ clauses in expectation.*

Proof: linearity of expectation. ■

Theorem 10 *There is an efficient deterministic algorithm that given a CNF formula of m clauses, with m_1 clauses of size 1, m_2 of size 2, etc. ($m = m_1 + m_2 + \ldots$) will find a solution satisfying at least $\sum_k m_k(1 - 1/2^k)$ clauses.*

Proof: Here is the deterministic algorithm. Look at x_1: for each of the two possible settings (0 or 1) we can calculate the expected number of clauses satisfied if we were to go with that setting, and then set the rest of the variables randomly. (It is just the number of clauses already satisfied plus $\sum_k m_k(1 - 1/2^k)$, where m_k is the number of clauses of size k in the "formula to go".) Fix x_1 to the setting that gives us a larger expectation. Now go on to x_2 and do the same thing, setting it to the value with the highest expectation-to-go, and then x_3 and so on. The point is that since we always pick the setting whose expectation-to-go is larger, this expectation-to-go never decreases (since our current expectation is the average of the ones we get by setting the next variable to 0 or 1). ■

This is called the "conditional expectation" method. The algorithm itself is completely deterministic — in fact we could rewrite it to get rid of any hint of randomization by just viewing $\sum_k m_k(1 - 1/2^k)$ as a way of weighting the clauses to favor the small ones, but our motivation was based on the randomized method.

Interesting fact: getting a $7/8 + \epsilon$ approximation for any constant $\epsilon > 0$ (like .001) for MAX-exactly-3-SAT is NP-hard.

In general, the area of approximation algorithms and approximation hardness is a major area of algorithms research.