While we have good algorithms for many optimization problems, the previous lecture showed that many others we'd like to solve are **NP**-hard. What do we do? Suppose we are given an **NP**-complete problem to solve. Assuming $\mathbf{P} \neq \mathbf{NP}$, we can't hope for a polynomial-time algorithm for these problems. But can we

(a) get faster (though still exponential-time) algorithms than the naïve solutions?
(b) get polynomial-time algorithms that always produce a "pretty good" solution? (A.k.a. *approximation algorithms*)

Today we consider approaches to combat intractability for several problems. Specific topics in this lecture include:

- 2-approximation for vertex cover via greedy matchings.
- 2-approximation for vertex cover via LP rounding.
- A faster-than-$O(n^k)$ time algorithm for exact vertex cover when k is small.
- Greedy $O(\log n)$ approximation for set-cover.

1 Introduction

Given an **NP**-hard problem, we don't hope for a fast algorithm that always gets the optimal solution. But we can't just throw our hands in the air and say "We can't do anything!" Or even, "We can't do anything other than the trivial solution!". There are (at least) two ways of being smarter.

- First approach: find a faster algorithm than the naïve one. There are a bunch of ideas that you can use for this, often related to dynamic programming — today we'll talk about one of them.[1]

- Second approach: find a poly-time 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 faster exact algorithms, or good approximations 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 achieve both goals above.

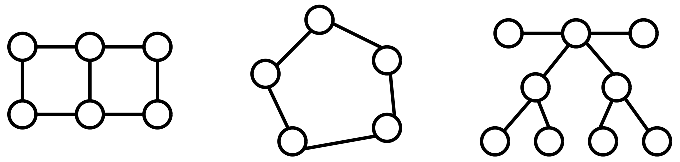
2 Vertex Cover

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.

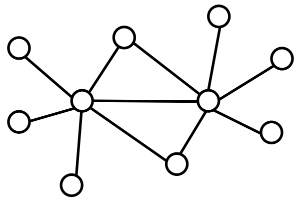
[1]You already saw an example: in Lecture II on Dynamic programming, we gave an algorithm for TSP that ran in time $O(n^2 2^n)$, which is much faster than the trivial $O(n \cdot n!) \geq n^{n/2}$ time algorithm.

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 graphs above of size 3. Show that there is no vertex cover of size 2 in them.



Exercise: Find a vertex cover in the graph above of size 2. Show that there is no vertex cover of size 1 in this graph.

As we saw last time (via a reduction from INDEPENDENT SET), this problem is **NP**-hard.

2.1 Faster Exact Algorithms

To solve the decision version of the vertex cover problem, here's the trivial algorithm taking about n^k time.

```
Iterate over all subsets $S \subseteq V$ of size $k$.
    If $S$ is a vertex cover, output YES and stop.
output NO.
```

How can you do better? Here's one cool way to do it:

Theorem 2 *Given $G = (V, E)$ and $k \leq n = |V|$, we can output (in time poly(n)) another graph H with at most $2k^2$ vertices, and an integer $k_H \leq k$ such that:*

$$(G, k) \textit{ is a YES-instance} \iff (H, k_H) \textit{ is a YES-instance} .$$

Since H is so much smaller, we can use the trivial algorithm above on (H, k') to run in time

$$|V(H)|^{k'} \leq (2k^2)^k \leq 2^{k(2\lg k+1)}.$$

Hence solve the problem on (G, k) in the time it takes to produce H (which is *poly*(n)), plus about $2^{2k \lg k}$ time. Much faster than n^k when $k \ll \sqrt{n}$.

It remains to prove Theorem 2.

Proof: Here's a simple observation: suppose G has an isolated vertex v, i.e., there are no edges hitting v. Then

$$(G, k) \text{ is a YES-instance} \iff (G - \{v\}, k) \text{ is a YES-instance} .$$

The second observation: suppose G has a vertex v of degree at least $k + 1$. Then v *must be* in any vertex cover of size k. Why? If not, these $k + 1$ edges incident to v would have to be covered by their other endpoints, which would require $k + 1$ vertices. So

$$(G, k) \text{ is a YES-instance} \iff (G - \{v\}, k - 1) \text{ is a YES-instance} .$$

(In this step, $G - \{v\}$ removes these edges incident to v as well.)

And the final observation: if G has maximum degree k and has a vertex cover of size at most k, G has at most k^2 edges. Why? Each of the vertices in the cover can cover at most k edges (their degree is at most k).

So now the construction of H is easy: start with G. Keep dropping isolated vertices (without changing k), or dropping high-degree vertices (and decrementing k) until you have some graph G' and parameter k'. By the first two observations,

$$(G, k) \text{ is a YES-instance} \iff (G', k') \text{ is a YES-instance} .$$

Now either G' has more than $(k')^2$ edges, then by the final observation (G', k') is a NO-instance, so then let H be a single edge and $k_H = 0$ (which is also a NO-instance.) Else return $H = G'$ and $k_H = k'$. Since H has $(k')^2 \le k^2$ edges, it has at most $2k^2$ nodes.

Finally, the time to produce H? You can do all the above in linear time in G. ■

2.2 Poly-time Algorithms that Output Approximate Solutions

OK, we don't expect to find the optimal solution in poly-time. However, any graph G we *can* get within a factor of 2 (or better). 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.[2]

[2] 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.

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 3 *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:

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 4 *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 $2 - o(1)$.

2.3 Hardness of Approximation

There are results showing that a good-enough approximation algorithm will end up showing that P=NP. Clearly, a 1-approximation would find the exact vertex cover, and show this. Håstad showed that if you get a 7/6-approximation, you would prove P=NP. This 7/6 was improved to 1.361 by Dinur and Safra. Beating $2 - \varepsilon$ 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 5 Set-Cover*: Given a domain X of n points, and m subsets $S_1, S_2, \dots, S_m$ 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 6 *If the optimal solution uses k sets, the greedy algorithm finds a solution with at most $O(k \ln n)$ 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.[3] ■

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

For set cover, this may be about the best you can do. Dinur and Steurer (improving on results of Feige) showed that if you get a $(1 - \epsilon) \ln(n)$-approximation algorithm for any constant $\epsilon > 0$, you will prove that P=NP.

[3]Notice how the above analysis is similar to the analysis we used of Edmonds-Karp #1.

4 Scheduling Jobs to Minimize Load

Here's a different problem. You have m identical machines on which you want to schedule some n jobs. Each job $j \in \{1, 2, \ldots, n\}$ has a processing time $p_j > 0$. You want to partition the jobs among the machines to minimize the load of the most-loaded machine. In other words, if S_i is the set of jobs assigned to machine i, define the *makespan* of the solution to be $\max_{\text{machines } i}(\sum_{j \in S_i} p_j)$. You want to minimize the makespan of the solution you output.

Approach #1: Greedy. Pick any unassigned job, and assign it to the machine with the least current load.

Theorem 8 *The greedy approach outputs a solution with makespan at most* 2 *times the optimum.*

Proof: Let us first get some bounds on the optimal value OPT. Clearly, the optimal makespan cannot be lower than the average load $\frac{1}{m}\sum_j p_j$. And also, the optimal makespan must be at least the largest job $\max_j p_j$.

Now look at our makespan. Suppose the most-loaded machine is i^*. Look at the last job we assigned on i^*, call this job j^*. If the load on i^* was L just before j^* was assigned to it, the makespan of our solution is $L + p_{j^*}$. By the greedy rule, i^* had the least load among all machines at this point, so L must be at most $\frac{1}{m}\sum_{j \text{ assigned before } j^*} p_j$. Hence, also $L \leq \frac{1}{m}\sum_j p_j \leq OPT$. Also, $p_{j^*} \leq OPT$. So our solution has makespan $L + p_{j^*} \leq 2OPT$.

Being careful, we notice that $L \leq \frac{1}{m}\sum_{j \neq j^*} p_j \leq OPT - \frac{p_j^*}{m}$, and hence our makespan is at most $L + p_{j^*} \leq (2 - \frac{1}{m})OPT$, which is slightly better. ■

Is this analysis tight? Sadly, yes. Suppose we have $m(m-1)$ jobs of size 1, and 1 job of size m, and we schedule the small jobs before the large jobs. The greedy algorithm will spread the small jobs equally over all the machines, and then the large job will stick out, giving a makespan of $(m-1) + m$, whereas the right thing to do is to spread the small jobs over $m-1$ machines and get a makespan of m. The gap is $\frac{2m-1}{m} = (2 - \frac{1}{m})$, hence this analysis is tight.

Hmm. Can we get a better algorithm? The bad example suggests one such algorithm.

Approach #2: Sorted Greedy. Pick the *largest* unassigned job, and assign it to the machine with the least current load.

Theorem 9 *The sorted greedy approach outputs a solution with makespan at most* 1.5 *times the optimum.*

Proof: Again, suppose the most-loaded machine is i^*, the last job assigned to it is j^*, and the load on i^* just before j^* was assigned to it is L. So $OPT = L + p_{j^*}$.

Suppose $L = 0$, then we are optimal, since $p_{j^*} \leq OPT$, so assume that at least one job was already on i^* before j^* was assigned. This means that each machine had at least one job on it when we assigned j^*. But by the sorted property, each such job has size at least p_{j^*}. So there are $m+1$ jobs of size at least p_{j^*}, and by the pigeonhole principle, OPT is at least $2p_{j^*}$. In other words, $p_{j^*} \leq \frac{OPT}{2}$.

By the same arguments as above, we know that our makespan is $L + p_{j^*} \leq OPT + \frac{OPT}{2} = 1.5\,OPT$. (And you can show a slightly better bound of $(1.5 - \frac{1}{2m})OPT$.) ■

It is possible to show that the makespan of Sorted Greedy is at most $\frac{4}{3}OPT$. (Try it!)