CIS 320: Algorithms	December 6-December 8, 2021

Lecture 24-45

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Approximation Algorithms I and II

In studying NP-completeness, we realized that there exist many basic combinatorial optimization problems that (probably) do not have worst-case polynomial time algorithms for exactly solving them. But we don't have to give up — we can relax our goal from finding exact solutions to that of finding *approximate* solutions. Lets remember one of our NP complete problems, Vertex Cover:

Definition 1 *Let $G = (V, E)$ be a graph. A subset $S \subseteq V$ of the vertices is a vertex cover if for each $(u, v) \in E$, either $u \in S$ or $v \in S$. Given an instance of the vertex cover problem, write $OPT = \min |\{S : S \text{ is a vertex cover.}\}|$.*

We proved that finding a vertex cover of size OPT (in fact, the easier problem of determining whether G has a vertex cover of size c for any c) was an NP-complete problem. Lets think about how to approximately solve the problem:

Definition 2 *An algorithm A is a k-approximation algorithm for the vertex cover problem if for every instance G of the vertex cover problem, it returns a solution $S = A(G)$ such that:*

1. *S is a vertex cover for G, and*
2. *$|S| \leq k \cdot OPT$.*

In other words, we insist that it return a feasible solution, but are willing to allow that it return a solution that is boundedly sub-optimal. Its not immediately obvious how to do well. First, we'll consider some simple greedy strategies that do poorly.

1. Proposal 1: Greedily construct a vertex cover S. While there are any uncovered edges e, pick such an edge (u, v), and add u to S. Remove all edges covered by u from the graph. This proposal fails to be a k-approximation for any constant k. Consider a star graph: it has a vertex cover of size 1 (pick the center!), but this algorithm might pick all $n - 1$ of the leaves.

2. Proposal 2: What went wrong in the star graph was we repeatedly picked the wrong end-point of the edges to add to the cover! What if we always select the vertex of max degree amongst the uncovered edges? Greedily construct a vertex cover S. While there are any uncovered edges, pick the vertex u of maximum remaining degree, and add it to S. Remove all edges covered by u.

   Its harder to find a counter-example, but this method also fails to be a k-approximation algorithm for any constant k. Consider the following bipartite graph. L consists of m vertices. R is partitioned into $m - 1$ parts, $R_2, \ldots, R_m$. Part R_i consists of $\lfloor \frac{m}{i} \rfloor$ vertices each of degree i, such that no two vertices in R_i share a neighbor. The greedy algorithm picks the vertices in R_m, then R_{m-1}, etc, until it has selected ever vertex in R, for a total cover of size $\sum_{i=2}^{m} \lfloor \frac{m}{i} \rfloor = \Omega(m \log m)$. But the optimal vertex cover picks all of the vertices in L, which has size only m. For large enough m, $\log m > k$, proving that this cannot be a k-approximation algorithm for any k.

Linear programs (which you should recall) will play a key role in our derivations, together with their computationally intractable cousins, integer linear programs:

Definition 3 *An integer linear program is a linear program in which the decision variables are constrained to take integer values. i.e. they are optimization problems defined over n non-negative* integer valued *decision variables $x_1, \ldots, x_n \geq 0$, a linear objective function, and d linear constraints taking the form:*

$$Maximize \sum_{i=1}^{n} c_i x_i$$

such that for each constraint $j \in [d]$:

$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_j$$

Integer linear programs are very expressive! For example, we can write down an integer linear program for the Vertex-Cover problem. Given a graph $G = (V, E)$ we introduce a binary valued decision variable x_v for each $v \in V$, intuitively representing whether v should be selected as part of a vertex cover S ($x_v = 1$ if $v \in S$). The constraints indicate that every edge must be covered by at least one vertex:

$$\text{Minimize } \sum_{v \in V} x_v$$

$$\text{such that for each } (u, v) \in E : x_u + x_v \geq 1$$

$$\text{and for each } v \in V : x_v \in \{0, 1\}$$

It is straightforward to observe that the optimal value of this integer linear program is equal to the size of the smallest vertex cover (OPT) of G. In making this observation, we have proven:

Theorem 4 *Vertex-Cover* $\leq_P$ *Integer-Programming. In particular, Integer Programming is NP-complete.*

Ok — so integer programs are expressive enough to represent vertex cover, but (as a result of this fact!) we can't solve integer programs, so its not clear why this is helpful. But remember that we can solve linear programs, so lets consider the linear program relaxation of the vertex cover problem, which is the same problem but without the integer constraints:

$$\text{Minimize } \sum_{v \in V} x_v$$

$$\text{such that for each } (u, v) \in E : x_u + x_v \geq 1$$

This is a problem we can solve efficiently, but it is not the vertex cover problem, because the solution can take fractional values. Lets call the optimal value of the linear program relaxation OPT_L. We start with a simple observation:

Lemma 5 *For every vertex cover instance* G*:* $OPT_L \leq OPT$

Proof The linear programming relaxation has only fewer constraints than the original integer linear program, so every feasible solution to the ILP is also a feasible solution to the LP — so the optimal objective value cannot be worse. ■

This means that if we start with a fractional optimal solution to the LP relaxation, and can modify it to construct a solution to the original ILP (and hence a valid vertex cover) without increasing the objective value by too much, then we will be to compare the objective value of our solution to OPT_L and therefore to OPT — the size of the smallest vertex cover.

Here is a natural way to take a solution to the fractional problem and try and convert it to an (integer) vertex cover: rounding! For each vertex v, include it in the vertex cover if $x_v \geq 1/2$ and otherwise do not.

Algorithm 1 Approx-Vertex-Cover(G)

Solve the linear programming relaxation of the vertex cover problem on instance G to obtain a fractional solution $\{x_v\}_{v \in V}$.
Construct $S \subseteq V$ such that $v \in S$ if and only if $x_v \geq \frac{1}{2}$
Return S.

We need to argue both that this algorithm returns a feasible vertex cover, and that the size of the vertex cover it returns is only boundedly worse than optimal.

Theorem 6 *Approx-Vertex-Cover is a 2-approximation algorithm for Vertex Cover.*

Proof First we argue that the set S returned by Approx-Vertex-Cover(G) is a vertex cover. To show this, we must argue that for each $(u, v) \in E$, either $u \in S$ or $v \in S$. But this is the case: since x forms a feasible solution to the linear program relaxation for vertex cover, we have that for each such edge, $x_u + x_v \geq 1$. In particular, it must be that either $x_u \geq 1/2$ or $x_v \geq 1/2$. Thus at least one of u and v are in S.

Next, we argue that the size of the vertex cover output is $|S| \leq 2\text{OPT}_L$. If we can show this, we are done, because as we have already observed, $\text{OPT}_L \leq \text{OPT}$, and so we will have $|S| \leq 2\text{OPT}$.

To see this, note that for each $v \in S$, we have $x_v \geq \frac{1}{2}$. Hence,

$$|S| \leq 2 \cdot \sum_{x \in S} x_v \leq 2 \sum_{x \in V} x_v = 2\text{OPT}_L$$

■

That was easy! Lets study one more linear programming based approximation algorithm that is a little more complex. The problem we will study is the *load balancing problem*. It models the problem of assigning jobs i of different sizes to machines j, with the objective of minimizing the load on the maximally loaded machine:

Definition 7 *An instance of the load balancing problem is given by a set J of n jobs, a set M of m machines, and for each job $j \in J$, a size $t_j \in \mathbb{R}$ and a set of machine $M_j \subseteq M$ that job j can be scheduled on. An assignment of jobs j to machines i is feasible if each job j is assigned a machine $i \in M_j$. Given an assignment, let $J_i \subseteq J$ denote the set of jobs assigned to machine i. The load on the machine i is the sum of the sizes of the jobs assigned to it: $L_i = \sum_{j \in J_i} t_j$, and the goal is to find the feasible assignment that minimizes the maximum load: $max_i L_i$. We write OPT to denote the maximum machine load in the optimal solution.*

The load balancing problem is also NP-complete (see if you can prove this as an exercise), but we can attempt the same strategy we used for Vertex Cover to derive an approximation algorithm. We can start off with a straightforward encoding of the load balancing problem as a linear program with decision variables $x_{i,j}$ corresponding to the load of job j assigned to machine i, and L corresponding to the maximum load across machines.:

$$\begin{array}{lll} \text{Minimize} & L & \\ \text{subject to} & \sum_{i \in M} x_{i,j} = t_j, & \forall j \in J \\ & \sum_{j \in J} x_{i,j} \leq L, & \forall i \in M \\ & x_{i,j} = 0, & \forall j \in J, i \notin M_j \\ & x_{i,j} \in \{0, t_j\}, & \forall i \in M, j \in J \end{array}$$

Convince yourself that this integer linear program captures the load balancing problem. Of course — we can't solve it because integer programming is NP-complete. But once again, we can solve the *linear program* that results from removing the integer constraint:

$$\begin{array}{lll} \text{Minimize} & L & \\ \text{subject to} & \sum_{i \in M} x_{i,j} = t_j, & \forall j \in J \\ & \sum_{j \in J} x_{i,j} \leq L, & \forall i \in M \\ & x_{i,j} = 0, & \forall j \in J, i \notin M_j \end{array}$$

Just as with Vertex Cover, if we denote by OPT the optimal solution to the load balancing problem (i.e. the original integer program), and we let OPT_L denote the optimal solution to the linear program relaxation, it must be that:

Lemma 8 *For every instance of the load balancing problem, $OPT_L \leq OPT$*

This follows because every feasible solution to the integer program is also feasible for the linear program, but not necessarily vice versa. Our plan of attack will be the same as it was for Vertex Cover at a high level: we will solve the linear program relaxation of the load balancing problem, and then use the resulting fractional solution to "round" to an actual, feasible solution to the load balancing problem. We'll argue that in doing so, we have not increased the objective value by too much, in comparison to OPT_L — and hence also in comparison to OPT. The details will be more involved this time though.

We start by observing something extremely simple about OPT:

Lemma 9 *For every instance: $OPT \geq \max_j t_j$.*

This holds because the largest job must be placed *somewhere*.

To convert a fractional solution x to a real assignment of jobs to machines, we'll consider the following bipartite graph $G(x) = (V(x), E(x))$ defined as follows. The vertices are $V(x) = J \cup M$ — i.e. we have one vertex for each job on the left, and one vertex for each machine on the right. There is an edge $(i, j) \in E(x)$ if and only if $x_{i,j} > 0$ — i.e. if the fractional solution puts *any fraction* of job i onto machine j. Our proof will consist of two steps:

1. If our graph $G(x)$ has no cycles, then we can efficiently convert a fractional optimal solution x to the linear program into a solution to the load balancing problem that has objective value at most $\text{OPT} + \text{OPT}_L \leq 2\text{OPT}$.

2. Given any fractional optimal solution x to the linear program, we can efficiently convert it into a new solution x' that is also optimal, such that $G(x')$ has no cycles.

If we can do both of these things, the result follows. Lets do them both in order.

Theorem 10 *Suppose $G(x)$ is acyclic. Then there is an efficient algorithm that from $G(x)$ obtains a feasible assignment of jobs to machines with maximum load at most $OPT + OPT_L \leq 2OPT$.*

Proof Since $G(x)$ is acyclic, each of its connected components forms a tree, in which layers alternate between jobs and machines. Consider any such component tree, and pick a node corresponding to a job. There are two cases:

1. The job j is a leaf of the tree. In this case, it must have an edge to only a single machine i (its parent in the tree), which means that in the fractional solution, we have $x_{i,j} = t_j$. In other words, the "fractional" solution did not in fact split job j at all — it assigned it entirely to machine i. In this case, we'll do the same and assign job j to machine i.

2. In the remaining case, job j is not a leaf — we will assign it to an arbitrary child machine i in the tree.

This results in a feasible assignment of jobs to machines. We only need to argue that the load on any machine is at most $\text{OPT} + \text{OPT}_L$. Consider any machine i, and let J_i be the set of jobs assigned to it. J_i contains any children of machine i that are leaves, as well as (possibly) its parent. But the children of machine i that were leaves were already fully assigned to machine i in the fractional solution, so they contribute load at most OPT_L. The parent j (if assigned) contributes load at most $t_j \leq \text{OPT}$. Thus the total load is at most $\text{OPT}_L + \text{OPT}$ as claimed. ■

It remains to show that without loss of generality, we can work with optimal solutions x^* such that $G(x^*)$ is acyclic. To prove this, we will assume that we have an optimal solution x to the linear program such that $G(x)$ contains a cycle. We will show how to modify x into a new solution x' so that x' is still optimal, but $G(x')$ contains at least one fewer edge and one fewer cycle. If we can do this, we are done, since we can just iterate this operation until we obtain an optimal solution x^* such that $G(x^*)$ is acyclic.

Theorem 11 *Let x be an optimal solution to the load balancing linear program such that $G(x)$ is not acyclic. There is an efficient algorithm that can transform x into a new solution x' such that x' remains an optimal solution to the LP, such that $G(x')$ has fewer cycles and edges than $G(x)$.*

Proof Consider any cycle C in $G(x)$. Since $G(x)$ is a bipartite graph, C alternates between machines and jobs: $C = (i_1, j_1, i_2, j_2, \ldots, i_k, j_k)$. By definition of $G(x)$, we have that $x_{i_t,j_t}, x_{i_{t+1},j_t} > 0$, for all $t \leq k$ where here i_{k+1} denotes i_1. The idea will be to "shift" some fractional assignment from each job j_t from machine i_t to machine i_{t+1} so as not to violate any of the feasibility constraints of the linear program. Let:

$$\Delta = \min_t x_{i_t,j_t} > 0$$

be the minimum extent to which any job j_t is assigned to machine i_t. We will construct x' as follows:

For each $t \leq k$ we will set

$$x'_{i_t,j_t} = x_{i_t,j_t} - \Delta \quad \text{and} \quad x'_{i_{t+1},j_t} = x_{i_{t+1},j_t} + \Delta$$

For all other pairs (i, j), we have $x'_{i,j} = x_{i.j}$.

Observe that since x was a feasible solution, x' remains feasible. By definition of Δ, we have that for each t, $x'_{i_t,j_t} \geq 0$. Observe also that our modification has left $\sum_i x_{i.j}$ invariant for each job j, and so $\sum_j x'_{i,j} = \sum_j x_{i,j} = t_j$. Finally, for each job j we have only moved weight onto machines i that were feasible (in M_j), since $x_{i_{t+1},j_t} > 0$ for all t.

Next, observe that we have not changed the load on any machine, and hence have not modified the objective value of the LP: For each machine i_t, we have added Δ load from job j_{t-1}, but have subtracted Δ load from job j_t. Thus if x was optimal, so is x'.

Finally, observe that by definition of Δ, we have for at least one value t $x'_{i_t,j_t} = 0$, and hence cycle C does not appear in $G(x')$. We have not introduced any new edges, so $G(x')$ has at least one fewer cycle than $G(x)$, as desired. ■

Together, these two theorems prove the following:

Theorem 12 *There is a polynomial time algorithm that given any instance of the load balancing problem, obtains a solution with objective value at most $2 \cdot OPT$.*

Proof The algorithm is as follows: First, we write down and solve the linear programming relaxation of the load balancing problem, to obtain an optimal solution x. Then we modify this optimal solution into an optimal solution x^* that contains no cycles, via Theorem 11. Finally we use this acyclic fractional solution to obtain a feasible assignment of jobs to machines with objective value at most $2 \cdot \text{OPT}$ via Theorem 10. ■

Tada!

Finally, as we wrap up CIS-320, lets take a moment to reflect on what we have learned, and how these last two lectures tie it together.

1. We've learned many algorithmic techniques that let us solve *some* problems efficiently: divide and conquer, dynamic programming, greedy algorithms, flow based algorithms, and linear programming. Of these techniques, linear programming is among the most powerful, allowing us to encode max flow and matching problems as special cases. These were some of our first examples of "polynomial time reductions", allowing us to solve one problem via the ability to solve another.

2. But not every problem has an efficient algorithmic solution: in particular, the NP-complete problems do not (unless $P = NP$). It turns out that polynomial time reductions, which are really just algorithms themselves, allow us to prove that many problems are NP complete, by showing that if we could solve them efficiently, we could also solve other problems (that we already know to be NP complete) efficiently.

3. Finally, just because a problem is NP-complete does not mean we need to give up on finding efficient algorithms: it is unlikely we can find efficient algorithms to *exactly* solve NP-complete problems, but as we have seen, we can often find efficient algorithms to *approximately* solve them. And these approximation algorithms can make crucial use of the powerful techniques we learned for solving other problems in this class, like linear programming.