Math 482: Linear Programming[1] Mikhail Lavrov

Lecture 36: Approximation Algorithms

May 4, 2020 *University of Illinois at Urbana-Champaign*

1 Approximation algorithms

Integer programming is hard. Sometimes it is prohibitively hard, and we can't hope to find the optimal solution. In this case, we would like to find a pretty good solution.

Approximation algorithms are methods of solving problems which don't claim to find the best solution, but which make guarantees about how close they are to being the best. A common notion of "closeness" is *approximation ratio*. We say that an algorithm has an approximation ratio of ρ if it always finds a solution whose objective value is within a factor of ρ of the optimal objective value. We say that such an algorithm is a *ρ-approximation algorithm*.

In the examples we'll look at today, we'll see several examples of 2-approximation algorithms for minimization problems. In the case of a minimization problem, this guarantee means that if the optimal solution has an objective value of z^*, the algorithm finds a solution with an objective value of z, such that $z^* \leq z \leq 2z^*$.

2 Vertex cover

We're going to consider the vertex cover problem again, but in a more general case than bipartite graphs.

In a general graph, we don't have two designated sides: we just have a set V of vertices, and a set E of edges that are pairs of vertices. The vertex cover problem is still the same: we want to pick a subset $S \subseteq V$ of the vertices that includes at least one endpoint of every edge, and we want $|S|$ to be as small as possible.

In the bipartite case, we know that we can find a minimum vertex cover with a linear program, or by solving a max-flow problem. But for graphs that are not necessarily bipartite, the vertex cover problem is very hard to solve: there's no known efficient algorithm.

2.1 Greedy algorithms

The most natural simple algorithm for vertex cover is the greedy one. Here, we start with $S = \varnothing$, which is *probably* not a vertex cover. Then, as long as S still doesn't cover every edge, we pick some vertex and add it to S.

We can be slightly smart about which vertex we add. For one thing, we can avoid adding vertices that don't help us cover any new edges. The pickiest method of all would be to add the vertex that covers the most uncovered edges.

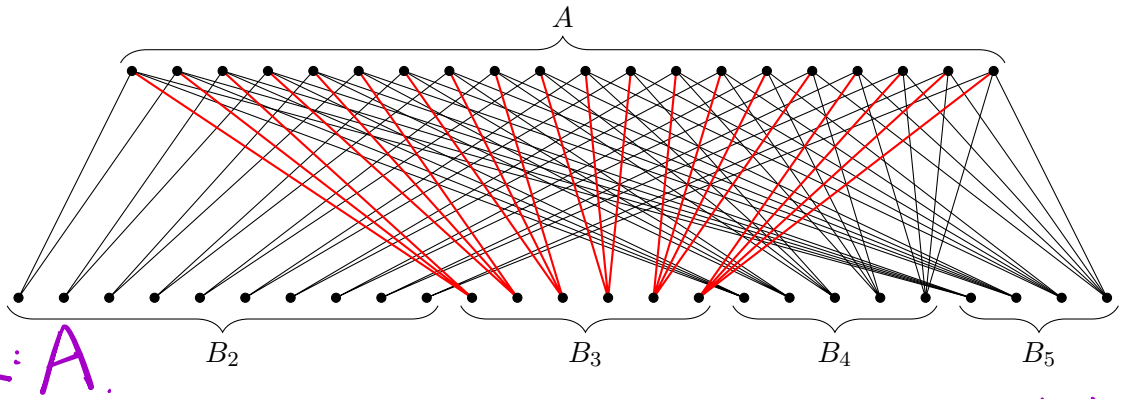
[1] This document comes from the Math 482 course webpage: https://faculty.math.illinois.edu/~mlavrov/courses/482-spring-2020.html

That's the first method we're going to look at. We can guess that it probably won't find the optimal solution. But can it approximate it to within some constant factor?

Turns out that it doesn't, even for bipartite graphs. Here's a construction of a family of bipartite graphs for which this algorithm is arbitrarily bad:

- On one side of the graph, put n vertices, for some (large) n. Call the set of these vertices A.
- On the other side of the graph, put $k-1$ "blocks" of vertices, numbered $B_2, B_3, \ldots, B_k$. Block B_i will have $\lfloor \frac{n}{i} \rfloor$ vertices.
- Each vertex in A gets one edge to a vertex in B_i, and they are distributed as evenly as possible: each vertex in B_i ends up with either i or $i+1$ neighbors in A.

Here's a picture of this graph for $n = 20$ and $k = 5$. The edges from block B_3 are highlighted to illustrate the pattern.



标准答案：A.

The best vertex cover in this graph is the set A, and has n vertices. cover $n(k-1)$ edges.

What will the greedy algorithm do? Well, the vertices with the most neighbors here are the vertices in B_k: they all have at least k neighbors. (Vertices in A, and most vertices in B_{k-1}, only have $k-1$ neighbors.) So it will begin by adding every vertex in B_k to the vertex cover. B_i, $i=2,\ldots k$.

But now, all vertices in A have one fewer uncovered edge out of them, so they would only cover $k-2$ uncovered edges. Vertices in B_{k-1} are better: each of them covers at least $k-1$ uncovered edges. So the greedy algorithm will add every vertex in B_{k-1} to the vertex cover.

This will continue. At each step, the last of the B_i blocks will look like it's the most efficient to add. So the greedy algorithm will end up adding all the vertices in $B_2 \cup B_3 \cup \cdots \cup B_{k-1} \cup B_k$.

Approximately (ignoring the rounding we did), the vertex cover found by the greedy algorithm has $\frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{k-1} + \frac{n}{k}$ vertices, which achieves an approximation ratio of $\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k-1} + \frac{1}{k}$. This can be arbitrarily bad: the infinite series $\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots$ diverges, so taking the first $k-1$ terms of this series can get us arbitrarily large numbers.

2.2 2-approximating vertex cover

There is, however, a 2-approximation algorithm for vertex cover. Again, start with $S = \varnothing$. Then, as long as S doesn't cover every edge, we pick an uncovered edge and add *both* endpoints to S.

Use greedy algorithm:

① $S = \phi$

② S += B_k cover n edges.

加入A中的一个 vertex 会多 cover $k-2$ edges，但加入 B_{k-1} 中

③ S += B_{k-1} cover $n+(k-1)\frac{n}{k-1}$ edges. 的点会多 cover $k-1$ edges.

$\Rightarrow \cdots \quad S = B_2 \cup B_2 \cup \cdots \cup B_k$

Cover $\frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{k-1} + \frac{n}{k}$ vertices.

Adding both endpoints seems wasteful: it is definitely more than we need. But it makes the algorithm perform much better in the worst case, because it guarantees we don't add the wrong endpoint to S and miss the right one.

Let's prove that this algorithm finds a 2-approximation. Call the uncovered edges we took in the algorithm were $e_1, e_2, \ldots, e_k$. (So the vertex cover we found has $2k$ vertices in it: both endpoints of each of these edges.) Let S^* be the optimal vertex cover, whatever it is.

Then for each edge e_i, S^* has to include at least one endpoint. Moreover, two different edges e_i and e_j can't share any endpoints: if they did, and we looked at e_i first, we'd have added both endpoints of e_i to S, which would have covered e_j as well. So S^* has to include at least k vertices: at least one endpoint of each of $e_1, e_2, \ldots, e_k$, which are all different.

Since $|S| = 2k$ and $|S^*| \geq k$, we have $|S| \leq 2|S^*|$, so we have a 2-approximation algorithm.

2.3 Weighted vertex cover

The vertex cover problem has a weighted variant. Here, we give vertex i a weight w_i. Instead of finding a vertex cover S with the smallest size $|S|$, our goal is to find a vertex cover S with the smallest total weight: the sum of w_i over all $i \in S$.

Our previous strategy for 2-approximation won't work. Maybe there is an edge with an endpoint of weight 1 and an endpoint of weight 100. The optimal vertex cover might cover it by taking the weight-1 endpoint. But taking both endpoints will have weight 101, which is much more than twice as bad.

Instead, let's look at the integer linear program for weighted vertex cover:

$$\begin{aligned} \underset{\mathbf{x} \in \mathbb{Z}^{|V|}}{\text{maximize}} \quad & \sum_{i \in V} w_i x_i \\ \text{subject to} \quad & x_i + x_j \geq 1 \quad \text{for all } ij \in E \\ & \mathbf{0} \leq \mathbf{x} \leq \mathbf{1} \end{aligned}$$

Solving the linear programming relaxation gets us a point $\mathbf{x}$ which is at least as good as the optimal integer solution (corresponding to the minimum-weight vertex cover). Unfortunately, $\mathbf{x}$ is probably not an integer point, so that's not directly useful.

But we can get somewhere useful by rounding. Obtain $\mathbf{x}'$ from $\mathbf{x}$ as follows: when $x_i \geq \frac{1}{2}$, set $x'_i = 1$, and when $x_i < \frac{1}{2}$, set $x'_i = 0$. Then:

- $\mathbf{x}'$ still satisfies all the constraints. For each edge ij, since $x_i + x_j \geq 1$, we must have either $x_i \geq \frac{1}{2}$ or $x_j \geq \frac{1}{2}$ (or both). Then either $x'_i = 1$ or $x'_j = 1$ (or both), so we have $x'_i + x'_j \geq 1$ as well.
- $\mathbf{x}'$ is an integer point. Together with the previous observation, this makes $S = \{i \in V : x_i = 1\}$ a vertex cover.
- For every i, we have $x'_i \leq 2x_i$. (If $x_i \geq \frac{1}{2}$, then $2x_i \geq 1$, and $x'_i = 1$; if $x_i < \frac{1}{2}$, then $x'_i = 0 \leq 2x_i$.) Therefore the total weight of $\mathbf{x}'$ is at most twice the weight of $\mathbf{x}$.

So we've found a vertex cover whose total weight is at most twice the weight of $\mathbf{x}$. The minimum-weight vertex cover has weight at least the weight of $\mathbf{x}$. So the vertex cover we found has at most twice the weight of the minimum-weight vertex cover, and once again we have a 2-approximation algorithm.

Rounding the solution to a linear programming relaxation is a common technique in approximation algorithms. It doesn't always work: here, the lucky thing is that all the constraints remain feasible, which wouldn't work in general.

3 Traveling salesman problem

Another important approximation algorithm exists for the traveling salesman problem in the previous lecture. That is, it exists for the *metric* traveling salesman problem: where we assume that costs are symmetric ($c_{ij} = c_{ji}$) and obey the triangle inequality ($c_{ij} + c_{jk} \geq c_{ik}$).

An algorithm called the Christofides algorithm, which we won't cover, is a $\frac{3}{2}$-approximation algorithm. We'll talk about an earlier algorithm which is a 2-approximation algorithm.

In this algorithm, we begin by finding a minimum-cost spanning tree in the traveling salesman graph. We've already seen spanning trees when discussing minimum-cost flow: this means picking $n-1$ edges that connect all n cities without creating cycles. There are several efficient methods for finding a minimum cost spanning tree. In particular, the greedy algorithm "pick the lowest-cost edge that doesn't create a cycle, and add it to the tree" always works.

The cost of the minimum-cost spanning tree is always at most the cost of the optimal traveling salesman tour. The tour is an n-edge cycle through all n cities. If you delete any edge in the tour, you get a spanning tree (a very particular kind with no branches). Its cost is at least the cost of the minimum-cost spanning tree.

However, given any spanning tree, we can find a tour with at most *double* the cost of the tree, giving us a 2-approximation algorithm. If the tree is drawn as a picture, this tour is easy to describe: just walk clockwise around the perimeter of the tree. But this "perimeter walk" can also be described recursively just given the structure of the tree.

That's actually not a tour yet; it visits many cities more than once. But because of the triangle inequality, we can fix this by repeatedly taking "shortcuts": whenever the perimeter walk tells us to revisit cities we've been to, skip them and go directly to the next city that's new.

Below is an example (we start from the top left vertex going clockwise; shortcuts are in red):