CS 3510 Algorithms Last updated: November 15, 2023

Lecture 23: Traveling Salesman Problem and Approximate Knapsack

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian*

To summarize this unit, we started with the discussion of NP-Complete problems, a class of problems that we do not know polynomial time algorithms for. To cope with this, we need to compromise on one of two things: efficiency or exactness. We can either find an exact algorithm in exponential time, or we can find an approximate solution in polynomial time. Last lecture we began discussing the latter, where we discussed how to argue about approximate algorithms.

1 Traveling Salesman Problem

Today, we'll begin by talking about the traveling salesman problem, where a salesman must visit n cities exactly once and finish at the city they start at. We describe the cities by some complete graph $G = (V, E), w : V \to \mathbb{R}_{\geq 0}$ where each vertex represents a city, and each edge represents a path between some cities u, v which incurs some cost $w((u, v))$. This problem is hard on it's own, but we have an extra challenge: the salesman wishes to minimize the total cost of the path is minimized. Formally we have some path of edges on the graph π and we'd like to minimize $w(\pi])$.

As we discuss this problem, we'll add one constraint on the edges: the triangle inequality. For any three nodes u, v, w, $w((u, w)) \leq w((u, v)) + w((v, w))$. In many real world problems this holds because the most direct route from anywhere to anywhere else will not include stopping anywhere else. This does not make the problem any less NP-Complete, but it does allow us to develop a simple approximation algorithm.

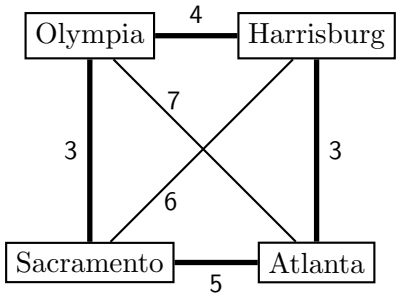

Figure 1: Cities where the shortest tour is bolded.

Take the above graph for example. The shortest tour on this graph is of length 15. Like for most hard problems, we can trivially solve this on small graphs by trying all possibilities, but as the number of possibilities grows, it becomes impractical. To cope with this, we'll use the strategies we discussed last time to develop and argue about an approximation algorithm.

1.1 Minimum Spanning Tree

In the above graph, you may notice something about the solution: it contains the minimum spanning tree. We know that we can find the minimum spanning tree in polynomial time, and it seems like a good start for this case, so let's use the minimum spanning tree as a starting point of our algorithm. To develop a path based on the minimum spanning tree, we can go to each vertex and then go back, as shown below. [1]

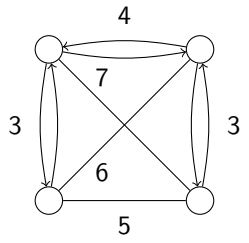

Figure 2: The same graph as above, but with the minimum spanning tree bolded.

Let's call the path we've developed here π_{alg}; then we know that $w(\pi_{\text{alg}}) = 2w(\text{MST})$ because we've used every edge in the minimum spanning tree twice. Now we want to show that this is 2-approximate; so we will consider the optimal tour π_{opt}.

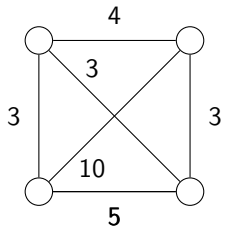

Figure 3: A graph where the minimum spanning tree is not part of the shortest tour.

You'll notice that the minimum spanning tree was contained by the minimum tour in the graph above. This may not always be the case; for instance in the graph above the MST is a Z shape including all edges of weight 3, while the shortest tour is the parameter. You should see that the the MST is a lower bound for the shortest TSP tour, meaning $w(MST) \leq w(\pi_{\text{opt}})$. We can combine this and the equation earlier, $w(\pi_{\text{alg}}) = 2w(\text{MST})$, to see that $w(\pi_{\text{alg}}) \leq 2w(\pi_{\text{opt}})$. This suffices to show that the algorithm is 2-approximate. [2]

[1] You may notice that this does not meet the constraint that every node is visited exactly once. We won't worry about this in the course, but we can actually use the triangle inequality to remove these. For instance in the graph above, we know that the last visited node must go back to the first node, and we also know that the shortest path between these two will be direct, so we can go directly.

[2] A student had a good question about approximation algorithms, "we know that all NP-Complete problems are equally hard", so if we have an approximation algorithm for some hard problem, can we use that approximation algorithm for some other NP-Complete problem. It turns out no, because the reductions

i	v_i	s_i
0	2	1
1	10	10

Figure 4: A situation where the greedy solution is not 2-approximate.

2 Knapsack

Now, we'll return to a problem we've talked about at length: Knapsack. As you recall, you have n jobs where the ith job has a s_i and a value v_i. We want to find a subset $S \subseteq \{1, \ldots, n\}$ s.t. We didn't show that this was NP-Complete in the class, but it turns out to be an NP-Complete problem. [3] As we discussed before, we could only come up with a pseudo-polynomial time solution for it using dynamic programming, but as it turns out, we can get a fast 2-approximate algorithm for it. [4]

2.1 A Greedy Solution

You may recall we discussed a greedy variant of the problem: Greedy($\frac{v_i}{s_i}$). First sort the elements by $\frac{v_i}{s_i}$ and then take as many items that can fit. We showed that this is not optimal, but is it α-approximate?

This is a popular problem though, so we'll work out a 2-approximate algorithm for it. As we've seen before the greedy approach is to order jobs by $\frac{v_i}{s_i}$; now simply using this ordering and adding as many jobs as possible is not optimal, but is it α-approximate? It turns out that this is not the case, in fact greedy can be arbitrarily bad. We can show that this is not the case by making it arbitrarily bad.

Consider the case in the figure above4 with a budget of 10. Here we see that the greedy algorithm will order the items 0, 1 which will lead to the item of value 2 getting picked, even though there is an item of value 10. Then this is not 2-approximate; in fact, we can make this arbitrarily bad by decreasing the size of the picked item. Consider if we made $v_0 = 2\varepsilon, s_0 = \varepsilon$, where ε is some infinitesimal. This item would always get picked, but would be $\frac{10}{\varepsilon}$ or infinitely bad.

2.2 Fractional Knapsack

Actually, this algorithm doesn't fail in all variations of Knapsack. In fractional Knapsack, there's the same optimization and the same constraints, but instead of either doing a job or not doing a job, a job can be partially done, in which case you get a fraction of the reward in a fraction of the time.

In this case, the concern of one item blocking another item out is gone, because you can spend any amount of time for the job (as long as the time you spend is less than the

we've shown only work for the optimal solution but don't maintain the "optimality" of a solution. You can create transformations which maintain the optimality to a certain point, but we will not do this.

[3] I'm not sure the proof can be found in DPV, but the reduction for Knapsack is to a problem called subset sum, which we can reduce to 3-Sat. The reductions nor the fact that subset sum is NP-Complete is important (it's on the practice exam, but we explain the problem there, and we don't ask for a reduction).

[4] This actually serves as a great example of the performance-optimality trade off.

amount of time the job takes). In this case, it becomes optimal to fill your time with the most valuable jobs, where valuable means value per time spent. This will entail doing the most valuable job, until it repeats, and repeating this with the remaining jobs until either no job remains or you run out of time. In the more interesting case, you'll run out of time. This will mean you can assume that you'll be stopped in the middle of a job under this greedy strategy.

Now what is the difference between the fractional greedy solution and the 1-0 variant? It turns out that it is that one job (the job where you get cut off). This means that the job that doesn't fit is the job you need to worry about.

3 A 2-Approximate Greedy Approach

Then let's consider an algorithm for 0-1 knapsack where we either take the greedy solution or the single highest job. This algorithm is clearly polynomial, so let's argue that this is 2-approximate.

Consider the items ordered by their greedy value; we will take the items $1, 2, 3, \ldots, k$, where $k+1$ is the job we can't take in the 0-1 setting but can take in the fractional setting. Our value for the greedy setting will then be $v_1 + v_2 + v_3 + \ldots + v_k + \alpha v_{k+1}$, where α is the fraction of the job that can be taken under the time constraints. We will argue that the following holds $v_1 + v_2 + v_3 + \ldots + v_k + v_{k+1} \geq v_1 + v_2 + v_3 + \ldots + v_k + \alpha v_{k+1} \geq \text{OPT}$ (clearly the first part holds because $\alpha \leq 1$). We also know that the greedy approach is optimal under the fractional setting; if this wasn't true the second part wouldn't hold. Since we know that $v_1 + v_2 + v_3 + \ldots + v_k + v_{k+1} \geq \text{OPT}$ is true, we trivially know that at either $v_1 + v_2 + v_3 + \ldots + v_k$ or v_{k+1} must be greater than $\frac{1}{2}\text{OPT}$. So we know that taking either will give a 2-approximation of the optimal solution.