Lecture 4

In which we describe a 1.5*-approximate algorithm for the Metric TSP, we introduce the Set Cover problem, observe that it can be seen as a more general version of the Vertex Cover problem, and we devise a logarithmic-factor approximation algorithm.*

1 Better Approximation of the Traveling Salesman Problem

In the last lecture we discussed equivalent formulations of the Traveling Salesman problem, and noted that Metric TSP-R can also be seen as the following problem: given a set of points X and a symmetric distance function $d : X \times X \to \mathbb{R}_{\geq 0}$ that satisfies the triangle inequality, find a multi-set of edges such that (X, E) is a connected multi-graph in which every vertex has even degree and such that $\sum_{(u,v)\in E} d(u, v)$ is minimized.

Our idea will be to construct E by starting from a minimum-cost spanning tree of X, and then adding edges so that every vertex becomes of even degree.

But how do we choose which edges to add to T?

Definition 1 (Perfect Matching) *Recall that graph (V, M) is a matching if no two edges in M have an endpoint in common, that is, if all vertices have degree zero or one. If (V, M) is a matching, we also call the edge set M a matching. A matching is a perfect matching if every vertex has degree one*

Note that a perfect matching can exist only if the number of vertices is even, in which case $|M| = |V|/2$.

Definition 2 (Min Cost Perfect Matching) *The Minimum Cost Perfect Matching Problem is defined as follows: an input of the problem is a an even-size set of*

vertices V and a non-negative symmetric weight function $w : V \times V \to \mathbb{R}_{\geq 0}$; the goal is to find a perfect matching (V, M) such that the cost

$$cost_w(M) := \sum_{(u,v)\in M} w(u, v)$$

of the matching is minimized.

We state, without proof, the following important result about perfect matchings.

Fact 3 *There is a polynomial-time algorithm that solves the Minimum Cost Perfect Matching Problem optimally.*

We will need the following observation.

Fact 4 *In every undirected graph, there is an even number of vertices having odd degree.*

PROOF: Let $G = (V, E)$ be any graph. For every vertex $v \in V$, let $deg(v)$ be the degree of v, and let O be the set of vertices whose degree is odd. We begin by noting that the sum of the degrees of all vertices is even, because it counts every edge twice:

$$\sum_{v\in V} deg(v) = 2 \cdot |E|$$

The sum of the degrees of the vertices in $V - O$ is also even, because it is a sum of even numbers. So we have that the sum of the degrees of the vertices in O is even, because it is a difference of two even numbers:

$$\sum_{v\in O} deg(v) = 2 \cdot |E| - \sum_{v\in V-O} deg(v) \equiv 0 \pmod 2$$

Now it follows from arithmetic modulo 2 that if we sum a collection of odd numbers and we obtain an even result, then it must be because we added an even number of terms. (Because the sum of an even number of odd terms is even.) So we have proved that $|O|$ is even. □

We are now ready to describe our improved polynomial-time approximation algorithm for General TSP-R.

- Input: instance (X, d) of Metric TSP-R
- Find a minimum cost spanning tree $T = (X, E)$ of X relative to the weight function $d(\cdot, \cdot)$

- Let O be the set of points that have odd degree in T
- Find a minimum cost perfect matching (O, M) over the points in O relative to the weight function $d(\cdot, \cdot)$
- Let E' be the multiset of edges obtained by taking the edges of E and the edges of M, with repetitions
- Find a Eulerian cycle C in the graph (X, E')
- Output C

We first note that the algorithm is correct, because (X, E') is a connected multigraph (because it contains the connected graph T) and it is such that all vertices have even degree, so it is possible to find an Eulerian cycle, and the Eulerian cycle is a feasible solution to General TSP-R.

The cost of the solution found by the algorithm is

$$\sum_{(u,v)\in E'} d(u,v) = cost_d(E) + cost_d(M)$$

We have already proved that, if $T = (X, E)$ is an optimal spanning tree, then $cost_d(E) \leq opt_{TSP-R}(X, d)$.

Lemma 5 below shows that $cost_d(M) \leq \frac{1}{2} opt_{TSP-R}(X, d)$, and so we have that the cost of the solution found by the algorithm is $\leq 1.5 \cdot opt_{TSP-R}(X, d)$, and so we have a polynomial time $\frac{3}{2}$-approximate algorithm for Metric TSP-R. (And also General TSP-R and Metric TSP-NR by the equivalence that we proved in the previous lecture.)

Lemma 5 *Let X be a set of points, $d(\cdot, \cdot)$ be a symmetric distance function that satisfies the triangle inequality, and $O \subseteq X$ be an-even size subset of points. Let M^* be a minimum cost perfect matching for O with respect to the weight function $d(\cdot, \cdot)$. Then*

$$cost_d(M^*) \leq \frac{1}{2} opt_{TSP-R}(X, d)$$

PROOF: Let C be a cycle which is an optimal solution for the Metric TSP-R instance (X, d). Consider the cycle C' which is obtained from C by skipping the elements of $X - O$, and also the elements of O which are repeated more than once, so that exactly once occurrence of every element of O is kept in C'. For example, if $X = \{a, b, c, d, e\}$, $O = \{b, c, d, e\}$ and C is the cycle $a \to c \to b \to d \to e \to a \to b \to a$ then we obtain C' by skipping the occurrences of a and the second occurrence of b, and we have the cycle $c \to b \to d \to e \to c$. Because of the triangle inequality, the operation of

skipping a point (which means replacing the two edges $u \to v \to w$ with the single edge $u \to w$) can only make the cycle shorter, and so

$$cost_d(C') \leq cost_d(C) = opt_{TSP-R}(X, d)$$

Now, C' is a cycle with an even number of vertices and edges, so we can write $C' = v_1 \to v_2 \to \cdots \to v_{2k} \to v_1$, where $v_1, \ldots, v_{2k}$ is some ordering of the vertices and $k := |O|/2$. We note that we can partition the set of edges in C' into two perfect matchings: the perfect matching $\{(v_1, v_2), (v_3, v_4), \ldots, (v_{2k-1}, v_{2k})\}$ and the perfect matching $\{(v_2, v_3), (v_4, v_5), \ldots, (v_{2k}, v_1)\}$. Since C' is made of the union of the edges of M_1 and M_2, we have

$$cost_d(C') = cost_d(M_1) + cost_d(M_2)$$

The perfect matching M^* is the minimum-cost perfect matching for O, and so $cost_d(M_1) \geq cost_d(M^*)$ and $cost_d(M_2) \geq cost_d(M^*)$, so we have

$$cost_d(C') \geq 2cost_d(M^*)$$

and hence

$$opt_{TSP-R}(X, d) \geq cost_d(C') \geq 2 \cdot cost_d(M^*)$$

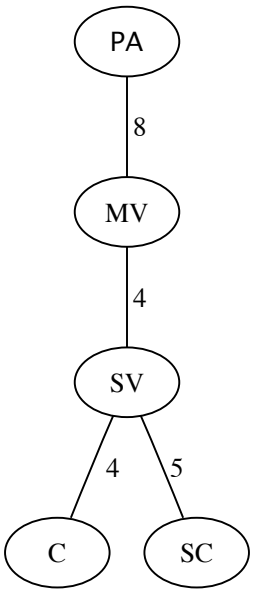
□

An important point is that the algorithm that we just analyzed, like every other approximation algorithm, is always able to provide, together with a feasible solution, a *certificate* that the optimum is greater than or equal to a certain lower bound. In the 2-approximate algorithm TSP algorithm from the previous lecture, the certificate is a minimum spanning tree, and we have that the TSP optimum is at least the cost of the minimum spanning tree. In the improved algorithm of today, the cost of minimum spanning tree gives a lower bound, and twice the cost of the minimum cost perfect matching over O gives another lower bound, and we can take the largest of the two.

Let us work out an example of the algorithm on a concrete instance, and see what kind of solution and what kind of lower bound we derive. Our set of points will be: Cupertino, Mountain View, Palo Alto, Santa Clara, and Sunnyvale. We have the following distances in miles, according to Google map:

	C	MV	PA	SC	SV
C	0	7	12	7	4
MV		0	8	9	4
PA			0	14	10
SC				0	5
SV					0

The reader can verify that the triangle inequality is satisfied. If we run a minimum spanning tree algorithm, we find the following tree of cost 21



This tells us that the optimum is at least 21 miles.

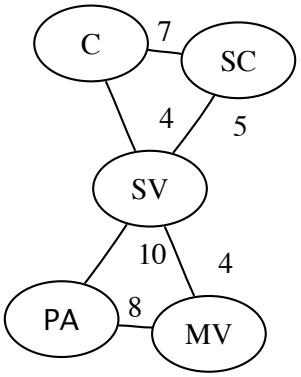
If we employ the algorithm from the last lecture, we perform a DFS which gives us the cycle Palo Alto → Mountain View → Sunnyvale → Cupertino → Sunnyvale → Santa Clara → Sunnyvale → Mountain View → Palo Alto, which has a length of 42 miles. After skipping the places that have already been visited, we get the cycle Palo Alto → Mountain View → Sunnyvale → Cupertino → Santa Clara → Palo Alto, whose length is 37 miles.

Today's algorithm, instead, looks for a minimum cost perfect matching of the points that have odd degree in the spanning tree, that is all the places except Mountain View. A minimum cost perfect matching (there are two optimal solutions) is $\{(PA, SV), (C, SC)\}$ whose cost is 17 miles, 10 for the connection between Palo Alto and Sunnyvale, and 7 for the one between Cupertino and Santa Clara.

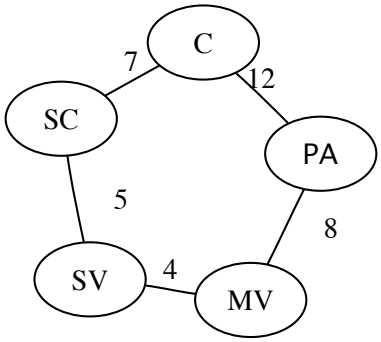
This tells us that the TSP optimum must be at least 34, a stronger lower bound than the one coming from the minimum spanning tree.

When we add the edges of the perfect matching to the edges of the spanning tree we

get the following graph, which is connected and is such that every vertex has even degree:



We can find an Eulerian cycle in the graph, and we find the cycle Palo Alto → Mountain View → Sunnyvale → Santa Clara → Cupertino → Sunnyvale → Palo Alto, whose length is 38 miles. After skipping Sunnyvale the second time, we have the cycle Palo Alto → Mountain View → Sunnyvale → Santa Clara → Cupertino → Palo Alto whose length is 36 miles.



In summary, yesterday's algorithm finds a solution of 37 miles, and a certificate that the optimum is at least 21. Today's algorithm finds a solution of 36 miles, and a certificate that the optimum is at least 34.

2 The Set Cover Problem

Definition 6 *The Minimum Set Cover problem is defined as follows: an input of the problem is a finite set X and a collection of subsets $S_1, \ldots, S_m$, where $S_i \subseteq X$ and $\bigcup_{i=1}^{m} S_i = X$.*

The goal of the problem is to find a smallest subcollection of sets whose union is X, that is we want to find $I \subseteq \{1, \ldots, m\}$ such that $\cup_{i \in I} S_i = X$ and $|I|$ is minimized.

For example, suppose that we want to assemble a team to work on a project, and each of the person that we can choose to be on the team has a certain set of skills; we want to find the smallest group of people that, among themselves, have all the skills that we need. Say, concretely, that we want to form a team of programmers and that we want to make sure that, among the team members, there are programmers who can code in C, C++, Ruby, Python, and Java. The available people are Andrea, who knows C and C++, Ben, who knows C++ and Java, Lisa, who knows C++, Ruby and Python, and Mark who knows C and Java. Selecting the smallest team is the same as a Minimum Set Cove problem in which we have the instance

$$X = \{\mathrm{C}, \mathrm{C}++, \mathrm{Ruby}, \mathrm{Python}, \mathrm{Java}\}$$
$$S_1 = \{\mathrm{C}, \mathrm{C}++\}, S_2 = \{\mathrm{C}++, \mathrm{Java}\},$$
$$S_3 = \{\mathrm{C}++, \mathrm{Ruby}, \mathrm{Python}\}, S_4 = \{\mathrm{C}, \mathrm{Java}\}$$

In which the optimal solution is to pick S_3, S_4, that is Lisa and Mark.

Although this is an easy problem on very small instances, it is an NP-hard problem and so it is unlikely to be solvable exactly in polynomial time. In fact, there are bad news also about approximation.

Theorem 7 *Suppose that, for some constant $\epsilon > 0$, there is an algorithm that, on input an instance of Set Cover finds a solution whose cost is at most $(1 - \epsilon) \cdot \ln |X|$ times the optimum; then every problem in* **NP** *admits a randomized algorithm running in time $n^{O(\log \log n)}$, where n is the size of the input.*

If, for some constant c, there is a polynomial time c-approximate algorithm, then $\mathbf{P} = \mathbf{NP}$.

The possibility of nearly-polynomial time randomized algorithms is about as unlikely as $\mathbf{P} = \mathbf{NP}$, so the best that we can hope for is an algorithm providing a $\ln |X|$ factor approximation.

A simple greedy approximation provides such an approximation.

Consider the following greedy approach to finding a set cover:

- Input: A set X and a collection of sets $S_1, \ldots, S_m$
- $I := \emptyset$
- while there is an *uncovered* element, that is an $x \in X$ such that $\forall i \in I . x \notin S_i$

  - Let S_i be a set with the largest number of uncovered elements
  - $I := I \cup \{i\}$
- return I

To work out an example, suppose that our input is

$$X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$$
$$S_1 = \{x_1, x_2, x_7, x_8\}$$
$$S_2 = \{x_1, x_3, x_4, x_8, x_{10}\}$$
$$S_3 = \{x_6, x_3, x_9, x_{10}\}$$
$$S_4 = \{x_1, x_5, x_7, x_8\}$$
$$S_5 = \{x_2, x_3, x_4, x_8, x_9\}$$

The algorithm will pick four sets as follows:

- At the first step, all the elements of X are uncovered, and the algorithm picks S_2, which is the set that covers the most elements (five);
- At the second step, there are five remaining uncovered elements, and the best that we can do is to cover two of them, for example picking S_1;
- At the third step there remain three uncovered elements, and again the best we can do is to cover two of them, by picking S_3;
- At the fourth step only x_5 remains uncovered, and we can cover it by picking S_4.

As with the other algorithms that we have analyzed, it is important to find ways to prove lower bounds to the optimum. Here we can make the following easy observations: at the beginning, we have 10 items to cover, and no set can cover more than 5 of them, so it is clear that we need at least two sets. At the second step, we see that there are five uncovered items, and that there is no set in our input that contains more than two of those uncovered items; this means that even the optimum solution must use at least $5/2$ sets to cover those five items, and so at least $5/2$ sets, that is at least 3 sets, to cover all the items.

In general, if we see that at some point there are k items left to cover, and that every set in our input contains at most t of those items, it follows that the optimum contains at least k/t sets. These simple observations are already sufficient to prove that the algorithm is $(\ln |X| + O(1))$-approximate.

We reason as follows. Let $X, S_1, \ldots, S_m$ be the input to the algorithm, and let $x_1, \ldots, x_n$ be an ordering of the elements of X in the order in which they are covered by the algorithm. Let c_i be the number of elements that become covered at the same time step in which x_i is covered. Let *opt* be the number of sets used by an optimal solution and *apx* be the number of sets used by the algorithm.

For every i, define

$$cost(x_i) := \frac{1}{c_i}$$

The intuition for this definition is that, at the step in which we covered x_i, we had to "pay" for one set in order to cover c_i elements that were previously uncovered. Thus, we can think of each element that we covered at that step as having cost us $\frac{1}{c_i}$ times the cost of a set. In particular, we have that the total number of sets used by the algorithm is the sum of the costs:

$$apx = \sum_{i=1}^{n} cost(x_i)$$

Now, consider the items $x_i, \ldots, x_n$ and let us reason about how the optimum solution manages to cover them. Every set in our input covers at most c_i of those $n - i + 1$ items, and it is possible, using the optimal solution, to cover all the items, including the items $x_i, \ldots, x_n$ with *opt* sets. So it must be the case that

$$opt \geq \frac{n - i + 1}{c_i} = (n - i + 1) \cdot cost(x_i)$$

from which we get

$$apx \leq opt \cdot \left(\sum_{i=1}^{n} \frac{1}{n - i + 1} \right)$$

The quantity

$$\sum_{i=1}^{n} \frac{1}{n - i + 1} = \sum_{i=1}^{n} \frac{1}{n}$$

is known to be at most $\ln n + O(1)$, and so we have

$$apx \leq (\ln n + O(1)) \cdot opt$$

It is easy to prove the weaker bound $\sum_{i=1}^{n} \frac{1}{n} \leq \lceil \log_2 n + 1 \rceil$, which suffices to prove that our algorithm is $O(\log n)$-approximate: just divide the sum into terms of the form $\sum_{i=2^k}^{2^{k+1}-1} \frac{1}{i}$, that is

$$1 + \left(\frac{1}{2} + \frac{1}{3}\right) + \left(\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}\right) + \cdots$$

and notice that each term is at most 1 (because each term is itself the sum of 2^k terms, each $\leq 2^{-k}$) and that the whole sum contains at most $\lceil \log_2 n + 1 \rceil$ such terms.

3 Set Cover versus Vertex Cover

The Vertex Cover problem can be seen as the special case of Set Cover in which every item in X appears in precisely two sets.

If $G = (V, E)$ is an instance of Vertex Cover, construct the instance of Set Cover in which $X = E$, and in which we have one set S_v for every vertex v, defined so that S_v is the set of all edges that have v as an endpoint. Then finding a subcollection of sets that covers all of X is precisely the same problem as finding a subset of vertices that cover all the edges.

The greedy algorithm for Set Cover that we have discussed, when applied to the instances obtained from Vertex Cover via the above transformation, is precisely the greedy algorithm for Vertex Cover: the algorithm starts from an empty set of vertices, and then, while there are uncovered edges, adds the vertex incident to the largest number of uncovered edges. By the above analysis, the greedy algorithm for Vertex Cover finds a solution that is no worse than $(\ln n + O(1))$ times the optimum, a fact that we mentioned without proof a couple of lectures ago.