CS261: A Second Course in Algorithms
Lecture #16: The Traveling Salesman Problem*

Tim Roughgarden†

February 25, 2016

1 The Traveling Salesman Problem (TSP)

In this lecture we study a famous computational problem, the *Traveling Salesman Problem (TSP)*. For roughly 70 years, the TSP has served as the best kind of challenge problem, motivating many different general approaches to coping with NP-hard optimization problems. For example, George Dantzig (who you'll recall from Lecture #10) spent a fair bit of his time in the 1950s figuring out how to use linear programming as a subroutine to solve ever-bigger instances of TSP. Well before the development of NP-completeness in 1971, experts were well aware that the TSP is a "hard" problem in some sense of the word.

So what's the problem? The input is a complete undirected graph $G = (V, E)$, with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$. By a *TSP tour*, we mean a simple cycle that visits each vertex exactly once. (Not to be confused with an Euler tour, which uses each edge exactly once.) The goal is to compute the TSP tour with the minimum total cost. For example, in Figure 1, the optimal objective function value is 13.

The TSP gets its name from a silly story about a salesperson who has to make a number of stops, and wants to visit them all in an optimal order. But the TSP definitely comes up in real-world scenarios. For example, suppose a number of tasks need to get done, and between two tasks there is a setup cost (from, say, setting up different equipment or locating different workers). Choosing the order of operations so that the tasks get done as soon as possible is exactly the TSP. Or think about a scenario where a disk has a number of outstanding read requests; figuring out the optimal order in which to serve them again corresponds to TSP.

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

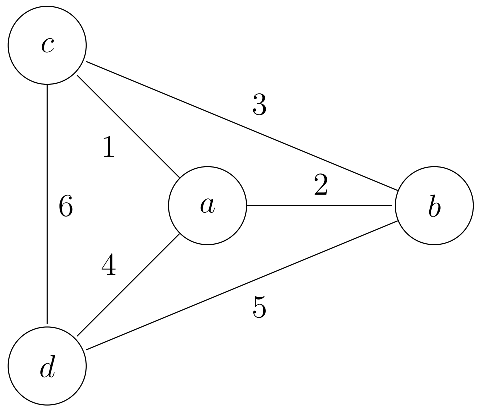

Figure 1: Example TSP graph. Best TSP tour is a-c-b-d-a with cost 13.

The TSP is hard, even to approximate.

Theorem 1.1 *If $P \neq NP$, then there is no α-approximation algorithm for the TSP (for any α).*

Recall that an α-approximation algorithm for a minimization problem runs in polynomial time and always returns a feasible solution with cost at most α times the minimum possible.

Proof of Theorem 1.1: We prove the theorem using a reduction from the Hamiltonian cycle problem. The Hamiltonian cycle problem is: given an undirected graph, does it contain a simple cycle that visits every vertex exactly once? For example, the graph in Figure 2 does not have a Hamiltonian cycle.[1] This problem is NP-complete, and usually one proves it in a course like CS154 (e.g., via a reduction from 3SAT).

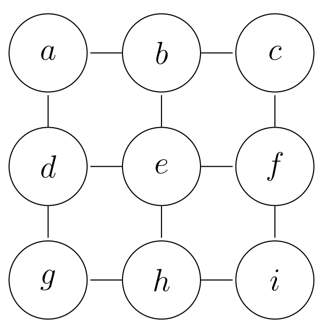

Figure 2: Example graph without Hamiltonian cycle.

[1]While it's generally difficult to convince someone that a graph has no Hamiltonian cycle, in this case there is a slick argument: color the four corners and the center vertex green, and the other four vertices red. Then every closed walk alternates green and red vertices, so a Hamiltonian cycle would have the same number of green and red vertices (impossible, since there are 9 vertices).

For the reduction, we need to show how to use a good TSP approximation algorithm to solve the Hamiltonian cycle problem. Given an instance $G = (V, E)$ of the latter problem, we transform it into an instance $G' = (V', E', c)$ of TSP, where:

- $V' = V$;
- E' is all edges (so (V', E') is the complete graph);
- for each $e \in E'$, set
$$c_e = \begin{cases} 1 & \text{if } e \in E \\ > \alpha \cdot n & \text{if } e \notin E, \end{cases}$$
where n is the number of vertices and α is the approximation factor that we want to rule out.

For example, in Figure 2, all the edges of the grid get a cost of 1, and all the missing edges get a cost greater than αn.

The key point is that there is a one-to-one correspondence between the Hamiltonian cycles of G and the TSP tours of G' that use only unit-cost edges. Thus:

(i) If G has a Hamiltonian cycle, then there is a TSP tour with total cost n.

(ii) If G has no Hamiltonian cycle, then every TSP tour has cost larger than αn.

Now suppose there were an α-approximation algorithm $\mathcal{A}$ for the TSP. We could use $\mathcal{A}$ to solve the Hamiltonian cycle problem: given an instance G of the problem, run the reduction above and then invoke $\mathcal{A}$ on the produced TSP instance. Since there is more than an α factor gap between cases (i) and (ii) and $\mathcal{A}$ is an α-approximation algorithm, the output of $\mathcal{A}$ indicates whether or not G is Hamiltonian. (If yes, then it must return a TSP tour with cost at most αn; if no, then it can only return a TSP tour with cost bigger than αn.) ■

2 Metric TSP

2.1 Toward a Tractable Special Case

Theorem 1.1 indicates that, to prove anything interesting about approximation algorithms for the TSP, we need to restrict to a special case of the problem. In the *metric TSP*, we assume that the edge costs satisfy the triangle inequality (with $c_{uw} \le c_{uv} + c_{vw}$ for all $u, v, w \in V$). We previously saw the triangle inequality when studying the Steiner tree problem (Lectures #13 and #15). The big difference is that in the Steiner tree problem the metric assumption is without loss of generality (see Exercise Set #7) while in the TSP it makes the problem significantly easier.[2]

The metric TSP problem is still NP-hard, as shown by a variant of the proof of Theorem 1.1. We can't use the big edge costs αn because this would violate the triangle inequality.

[2]This is of course what we're hoping for, because the general case is impossible to approximate.

But if we use edge costs of 2 for edges not in the given Hamiltonian cycle instance G, then the triangle inequality holds trivially (why?). The optimal TSP tour still has value at most n when G has a Hamiltonian cycle, and value at least $n+1$ when it does not. This shows that there is no exact polynomial-time algorithm for metric TSP (assuming $P \neq NP$). It does not rule out good approximation algorithms, however. And we'll see next that there are pretty good approximation algorithms for metric TSP.

2.2 The MST Heuristic

Recall that in approximation algorithm design and analysis, the challenge is to relate the solution output by an algorithm to the optimal solution. The optimal solution itself is often hard to get a handle on (its NP-hard to compute, after all), so one usually resorts to bounds on the optimal objective function value — quantities that are "only better than optimal." Here's a simple lower bound for the TSP, with or without the triangle inequality.

Lemma 2.1 *For every instance $G = (V, E, c)$, the minimum-possible cost of a TSP tour is at least the cost of a minimum spanning tree (MST).*

Proof: Removing an edge from the minimum-cost TSP tour yields a spanning tree with only less cost. The minimum spanning tree can only have smaller cost. ■

Lemma 2.1 motivates using the MST as a starting point for building a TSP tour — if we can turn the MST into a tour without suffering too much extra cost, then the tour will be near-optimal. The idea of transforming a tree into a tour should ring some bells — recall our online (Lecture #13) and offline (Lecture #15) algorithms for the Steiner tree problem. We'll reuse the ideas developed for Steiner tree, like doubling and shortcutting, here for the TSP. The main difference is that while these ideas were used only in the *analysis* of our Steiner tree algorithms, to relate the cost of our algorithm's tree to the minimum-possible cost, here we'll use these ideas in the *algorithm* itself. This is because, in TSP, we have to output a tour rather than a tree.

MST Heuristic for Metric TSP

compute the MST T of the input G
construct the graph H by doubling every edge of T
compute an Euler tour C of H
`// every` $v \in V$ `is visited at least once in` C
shortcut repeated occurrences of vertices in C to obtain a TSP tour

When we studied the Steiner tree problem, steps 2–4 were used only in the analysis. But all of these steps, and hence the entire algorithm, are easy to implement in polynomial (even near-linear) time.[3]

[3]Recall from CS161 that there are many fast algorithms for computing a MST, including Kruskal's and Prim's algorithms.

Theorem 2.2 *The MST heuristic is a 2-approximation algorithm for the metric TSP.*

Proof: We have

$$\begin{aligned}\text{cost of our TSP tour} &\leq \text{cost of } C\\ &= \sum_{e\in H} c_e\\ &= 2\sum_{e\in T} c_e\\ &\leq 2\cdot \text{cost of optimal TSP tour,}\end{aligned}$$

where the first inequality holds because the edge costs obey the triangle inequality, the second equation holds because the Euler tour C uses every edge of H exactly once, the third equation follows from the definition of H, and the final inequality follows from Lemma 2.1. ■

The analysis of the MST heuristic in Theorem 2.2 is tight — for every constant $c < 2$, there is a metric TSP instance such that the MST heuristic outputs a tour with cost more than c times that of an optimal tour (Exercise Set #8).

Can we do better with a different algorithm? This is the subject of the next section.

2.3 Christofides's Algorithm

Why were we off by a factor of 2 in the MST heuristic? Because we doubled every edge of the MST T. Why did we double every edge? Because we need an Eulerian graph, to get an Euler tour that we can shortcut down to a TSP tour. But perhaps it's overkill to double every edge of the MST. Can we augment the MST T to get an Eulerian graph without paying the full cost of an optimal solution?

The answer is yes, and the key is the following slick lemma. It gives a second lower bound on the cost of an optimal TSP tour, complementing Lemma 2.1.

Lemma 2.3 *Let $G = (V, E)$ be a metric TSP instance. Let $S \subseteq V$ be an even subset of vertices and M a minimum-cost perfect matching of the (complete) graph induced by S. Then*

$$\sum_{e\in M} c_e \leq \frac{1}{2}\cdot OPT,$$

where OPT denotes the cost of an optimal TSP tour.

Proof: Fix S. Let C^* denote an optimal TSP tour. Since the edges obey the triangle inequality, we can shortcut C^* to get a tour C_S of S that has cost at most OPT. Since $|S|$ is even, C_S is a (simple) cycle of even length (Figure 3). C_S is the union of two disjoint perfect matchings (alternate coloring the edges of C_S red and green). Since the sum of the costs of these matchings is that of C_S (which is at most OPT), the cheaper of these two matchings has cost at most $OPT/2$. The minimum-cost perfect matching of S can only be cheaper. ■

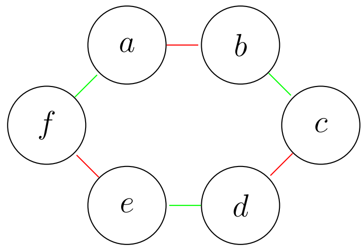

Figure 3: C_S is a simple cycle of even length representing union of two disjoint perfect matchings (red and green).

Lemma 2.3 brings us to *Christofides's algorithm*, which differs from the MST heuristic only in substituting a perfect matching computation in place of the doubling step.

Christofides's Algorithm

compute the MST T of the input G
compute the set W of vertices with odd degree in T
compute a minimum-cost perfect matching M of W
construct the graph H by adding M to T
compute an Euler tour C of H
`// every` $v \in V$ `is visited at least once in` C
shortcut repeated occurrences of vertices in C to obtain a TSP tour

In the second step, the set W always has even size. (The sum of the vertex degrees of a graph is double the number of edges, so there cannot be an odd number of odd-degree vertices.) In the third step, note that the relevant matching instance is the graph induced by W, which is the complete graph on W. Since this is not a bipartite graph (at least if $|W| \geq 4$), this is an instance of *nonbipartite* matching. We haven't covered any algorithms for this problem, but we mentioned in Lecture #6 that the ideas behind the Hungarian algorithm (Lecture #5) can, with additional ideas, be extended to also solve the nonbipartite case in polynomial time. In the fourth step, there may be edges that appear in both T and M. The graph H contains two copies of such edges, which is not a problem for us. The last two steps are the same as in the MST heuristic. Note that the graph H is indeed Eulerian — adding the matching M to T increases the degree of each vertex $v \in W$ by exactly one (and leaves other degrees unaffected), so $T + M$ has all even degrees.[4] This algorithm can be implemented in polynomial time — the overall running time is dominated by the matching computation in the third step.

Theorem 2.4 *Christofides's algorithm is a $\frac{3}{2}$-approximation algorithm for the metric TSP.*

[4]And as usual, H is connected because T is connected.

Proof: We have

$$
\begin{aligned}
\text{cost of our TSP tour} &\le \text{cost of } C \\
&= \sum_{e \in H} c_e \\
&= \underbrace{\sum_{e \in T} c_e}_{\le OPT \text{ (Lem 2.1)}} + \underbrace{\sum_{e \in M} c_e}_{\le OPT/2 \text{ (Lem 2.3)}} \\
&\le \frac{3}{2} \cdot \text{cost of optimal TSP tour},
\end{aligned}
$$

where the first inequality holds because the edge costs obey the triangle inequality, the second equation holds because the Euler tour C uses every edge of H exactly once, the third equation follows from the definition of H, and the final inequality follows from Lemmas 2.1 and 2.3. ■

The analysis of Christofides's algorithm in Theorem 2.4 is tight — for every constant $c < \frac{3}{2}$, there is a metric TSP instance such that the algorithm outputs a tour with cost more than c times that of an optimal tour (Exercise Set #8).

Christofides's algorithm is from 1976. Amazingly, to this day we still don't know whether or not there is an approximation algorithm for metric TSP better than Christofides's algorithm. It's possible that no such algorithm exists (assuming $P \neq NP$, since if $P = NP$ the problem can be solved optimally in polynomial time), but it is widely conjecture that $\frac{4}{3}$ (if not better) is possible. This is one of the biggest open questions in the field of approximation algorithms.

3 Asymmetric TSP

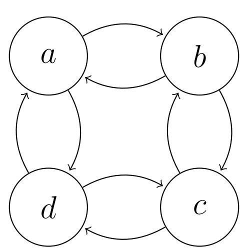

Figure 4: Example ATSP graph. Note that edges going in opposite directions need not have the same cost.

We conclude with an approximation algorithm for the *asymmetric TSP (ATSP)* problem, the directed version of TSP. That is, the input is a complete directed graph, with an edge in each direction between each pair of vertices, and a nonnegative cost $c_e \geq 0$ for each edge (Figure 4). The edges going in opposite directions between a pair of vertices need not have the same cost.[5] The "normal" TSP is equivalent to the special case in which opposite edges (between the same pair of vertices) have the same cost. The goal is to compute the directed TSP tour — a simple directed cycle, visiting each vertex exactly once — with minimum-possible cost. Since the ATSP includes the TSP as a special case, it can only harder (and appears to be strictly harder). Thus we'll continue to assume that the edge costs obey the triangle inequality ($c_{uw} \leq c_{uv} + c_{vw}$ for every $u, v, w \in V$) — note that this assumption makes perfect sense in directed graphs as well as undirected graphs.

Our high-level strategy mirrors that in our metric TSP approximation algorithms.

1. Construct a not-too-expensive Eulerian directed graph H.
2. Shortcut H to get a directed TSP tour; by the triangle inequality, the cost of this tour is at most $\sum_{e \in H} c_e$.

Recall that a directed graph H is *Eulerian* if (i) it is strongly connected (i.e., for every v, w there is a directed path from v to w and also a directed path from w to v); and (ii) for every vertex v, the in-degree of v in H equals its out-degree in H. Every directed Eulerian graph admits a directed Euler tour — a directed closed walk that uses every (directed) edge exactly once. Assumptions (i) and (ii) are clearly necessary for a graph to have a directed Euler tour (since one enters and exists a vertex the same number of times). The proof of sufficiency is basically the same as in the undirected case (cf., Exercise Set #7).

The big question is how to implement the first step of constructing a low-cost Eulerian graph. In the metric case, we used the minimum spanning tree as a starting point. In the directed case, we'll use a different subroutine, for computing a *minimum-cost cycle cover*.

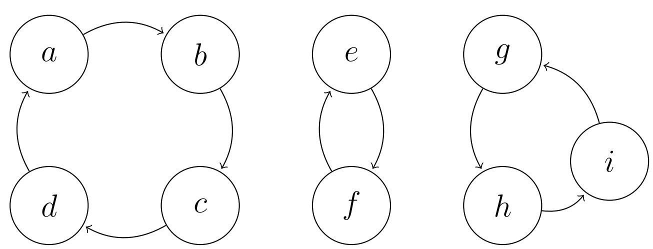

Figure 5: Example cycle cover of vertices.

A *cycle cover* of a directed graph is a collection of $C_1, \ldots, C_k$ of directed cycles, each with at least two vertices, such that each vertex $v \in V$ appears in exactly one of the cycles. (This is, the cycles partition the vertex set.) See Figure 5. Note that directed TSP tours

[5]Recalling the motivating scenario of scheduling the order of operations to minimize the overall setup time, it's easy to think of cases where the setup time between task i and task j is not the same as if the order of i and j are reversed.

are exactly the cycle covers with $k = 1$. Thus, the minimum-cost cycle cover can only be cheaper than the minimum-cost TSP tour.

Lemma 3.1 *For every instance $G = (V, E, c)$ of ATSP, the minimum-possible cost of a directed TSP tour is at least that of a minimum-cost cycle cover.*

The minimum-cost cycle cover of a directed graph can be computed in polynomial time. This is not obvious, but as a student in CS261 you're well-equipped to prove it (via a reduction to minimum-cost bipartite perfect matching, see Problem Set #4).

Approximation Algorithm for ATSP

initialize $F = \emptyset$
initialize G to the input graph
while G has at least 2 vertices **do**
  compute a minimum-cost cycle cover $C_1, \ldots, C_k$ of the current G
  add to F the edges in $C_1, \ldots, C_k$
  for $i = 1, 2, \ldots, k$ **do**
    delete from G all but one vertex from C_i
compute a directed Euler tour C of $H = (V, F)$
`//` H `is Eulerian, see discussion below`
shortcut repeated occurrences of vertices on C to obtain a TSP tour

For the last two steps of the algorithm to make sense, we need the following claim.

Claim: The graph $H = (V, F)$ constructed by the algorithm is Eulerian.

Proof: Note that $H = (V, F)$ is the union of all the cycle covers computed over all iterations of the while loop. We prove two invariants of (V, F) over these iterations.

First, the in-degree and out-degree of a vertex are always the same in (V, F). This is trivial at the beginning, when $F = \emptyset$. When we add in the first cycle cover to F, every vertex then has in-degree and out-degree equal to 1. The vertices that get deleted never receive any more incoming or outgoing edges, so they have the same in-degree and out-degree at the conclusion of the while loop. The undeleted vertices participate in the cycle cover computed in the second iteration; when this cycle cover is added to H, the in-degree and out-degree of each vertex in (V, F) increases by 1 (from 1 to 2). And so on. At the end, the in- and out-degree of a vertex v is exactly the number of while loop iterations in which it participated (before getting deleted).

Second, at all times, for all vertices v that have been deleted so far, there is a vertex w that has not yet been deleted such that (V, F) contains both a directed path from v to w and from w to v. That is, in (V, F), every deleted vertex can reach and be reached by some undeleted vertex.

To see why this second invariant holds, consider the first iteration. Every deleted vertex v belongs to some cycle C_i of the cycle cover, and some vertex w on C_i was left undeleted. C_i

contains a directed path from v to w and vice versa, and F contains all of C_i. By the same reasoning, every vertex v that was deleted in the second iteration has a path in (V, F) to and from some vertex w that was not deleted. A vertex u that was deleted in the first iteration has, at worst, paths in (V, F) to and from a vertex v deleted in the second iteration; stitching these paths together with the paths from v to an undeleted vertex w, we see that (V, F) contains a path from u to this undeleted vertex w, and vice versa. In the final iteration of the while loop, the cycle cover contains only one cycle C. (Otherwise, at least 2 vertices would not be deleted and the while loop would continue.) The edges of C allow every vertex remaining in the final iteration to reach every other such vertex. Since every deleted vertex can reach and be reached by the vertices remaining in the final iteration, the while loops concludes with a graph (V, F) where everybody can reach everybody (i.e., which is strongly connected). ■

The claim implies that our ATSP algorithm is well defined. We now give the easy argument bounding the cost of the tour it produces.

Lemma 3.2 *In every iteration of the algorithm's main while loop, there exists a directed TSP tour of the current graph G with cost at most OPT, the minimum cost of a TSP tour in the original input graph.*

Proof: Shortcutting the optimal TSP tour for the original graph down to one on the current graph G yields a TSP tour with cost at most OPT (using the triangle inequality). ■

By Lemmas 3.1 and 3.2:

Corollary 3.3 *In every iteration of the algorithm's main while loop, the cost of the edges added to F is at most OPT.*

Lemma 3.4 *There are at most $\log_2 n$ iterations of the algorithm's main while loop.*

Proof: Recall that every cycle in a cycle cover has, by definition, at least two vertices. The algorithm deletes all but one vertex from each cycle in each iteration, so it deletes at least one vertex for each vertex that remains. Since the number of remaining vertices drops by a factor of at least 2 in each iteration, there can only be $\log_2 n$ iterations. ■

Corollary 3.3 and Lemma 3.4 immediately give the following.

Theorem 3.5 *The ATSP algorithm above is a $\log_2 n$-approximation algorithm.*

This algorithm is from the early 1980s, and progress since then has been modest. The best-known approximation algorithm for ATSP has an approximation ratio of $O(\log n / \log \log n)$, and even this improvement is only from 2010! Another of the biggest open questions in all of approximation algorithms is: is there a constant-factor approximation algorithm for ATSP?