**CS4234: Optimisation Algorithms** Lecture 4

# TRAVELLING-SALESMAN-PROBLEM (4 variants)

*V1.0: Seth Gilbert, V1.1: Steven Halim* *August 30, 2016*

**Abstract**

The goal of the TRAVELLING-SALESMAN-PROBLEM is to find a tour that connects all the vertices in a graph at a minimal cost. There are several variants, depending on whether repeated visits are allowed, and depending on whether the distances satisfy a metric. We discuss the relationship between these variants, and give a simple 2-approximation algorithm. We then develop a more involved 1.5-approximation algorithm that relates the TSP to Eulerian tours.

## 1 The TRAVELLING-SALESMAN-PROBLEM

Today we consider the TRAVELLING-SALESMAN-PROBLEM[1], often abbreviated TSP. The "TSP"[2] is perhaps one of the most famous[3] (and most studied) Combinatorial Optimization Problem (COP).

### 1.1 Problem Definition

Given a set of cities (i.e., points, or vertices), the goal of the TSP is to find a minimum cost circuit (or cycle, or tour) that visits all the points. More formally, the problem is stated as follows:

**Definition 1** *Given a set $V$ of $n$ points and a distance function $d : V \times V \to \mathbb{R}$, find a cycle $C$ of minimum cost that contains all the points in $V$. The cost of a cycle $C = (e_1, e_2, \ldots, e_n)$ is defined to be $\sum_{e \in C} d(e)$, and we assume that the distance function is non-negative (i.e., $d(x, y) \geq 0$).*

Notice that, unlike the STEINER-TREE problem, there are no Steiner vertices: You have to visit *every* city. The cities/points may be in some geometric space (e.g., the Euclidean plane), or they may not. If the points are in the Euclidean plane, then it is natural to define $d$ to be the Euclidean distance. Otherwise, $d$ can be arbitrary, i.e. non-metric. See Figure 1 for an example of one instance of the TSP.

### 1.2 Variants

As with the STEINER-TREE problem, there are several variants:

- *Metric vs. General:* In the *metric* version of the TSP, the distance function $d$ is a metric, i.e., it satisfies *the triangle inequality*[4]. In the general version, $d$ can assign any arbitrary weight to an edge.
- *Repeated-Visits vs. No-Repeats:* The goal of the TSP is to find a cycle that visits every vertex. Can the cycle contain repeated vertices (or does it have to be a simple cycle)? In the version with No-Repeats, the TSP cycle must visit each vertex *exactly* once. In the version with repeats, it is acceptable to visit each vertex more than once (if that results in a shorter route).

[1] The word 'travelling' (2 'l's) is in British and the word 'traveling' (1 'l') is in American.
[2] As the abbreviation "TSP" already includes the word 'Problem', we do not say "TSP problem" anymore.
[3] We even have a movie for this, see `http://www.travellingsalesmanmovie.com/`.
[4] In the previous lecture about Steiner Tree, we have highlighted that metric version is somewhat easier than the general version.

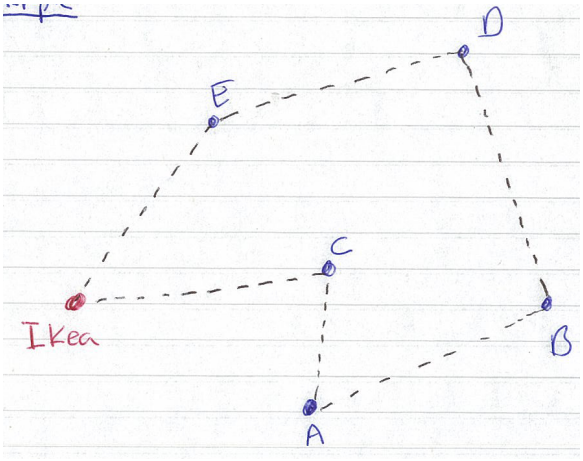


**Figure 1:** Example of the TSP. Here, there are six locations. The problem is to find the shortest delivery circuit starting from Ikea and visiting each of the five houses, and then returning to Ikea.

We will summarize these four variants as follows:

| | **Repeats** | **No-Repeats** |
|---|---|---|
| **Metric** | M-R-TSP | M-NR-TSP |
| **General** | G-R-TSP | G-NR-TSP |

### 1.3 NP-hard

All the variants of the TSP are NP-hard [2].

For the No-Repeats variants of the problem, this is easily seen by reduction from the HAMILTONIAN-CYCLE problem: a Hamiltonian cycle is a cycle that contains every vertex exactly once; the goal of the HAMILTONIAN-CYCLE problem is to determine whether a given graph has a Hamiltonian cycle. Deciding whether a graph has a Hamiltonian cycle is NP-hard (and this can be shown by reduction from 3-SAT). Clearly, if we can solve the TSP, then we can solve the HAMILTONIAN-CYCLE problem (by setting the distances properly). There are similar reductions for the other variants.

The general No-Repeats version of TSP (the G-NR-TSP) is NP-hard even to approximate! So we will temporarily ignore this variant in this Lecture.

Even so, we are going to see how to (easily!) approximate the other three variants. In almost all common real-world cases (e.g., where the distance function satisfies the triangle inequality, or where repeats are acceptable), there are good approximation algorithms.

## 2 Equivalence

The first thing we are going to see is that the three other variants are equivalent. That is, M-R-TSP, M-NR-TSP, and G-R-TSP are all equivalent: If we have a $c$-approximation algorithm for any one of these variants, then we also can

construct a $c$-approximation algorithm for the other two.

We first focus on the issue of repeats: As long as the distance function is a metric, it does not matter whether or not we allow repeats. In many ways, this should be unsurprising (given our discussion of STEINER-TREE last week): If we have repeated vertices on the cycle, we can always "short-cut" past them producing a cycle with no repeats at the same cost.

**Lemma 2** *There exists a $c$-approximation algorithm for* M-NR-TSP *if and only if there exists a $c$-approximation algorithm for* M-R-TSP.

**Proof** The proof has three claims. First, we show that both the version with repeats and no-repeats have the same optimal cost. We then show how to transform the solution from the NR variant into the R variant (which is trivial) and from the R variant into the NR variant (which involves skipping repeats).

**Claim 1.** Let $(V, d)$ be an input for Metric TSP (for both NR or R variants). Let $OPT(R)$ be the minimum cost TSP cycle overall (i.e., with repeats), and let $OPT(NR)$ be the minimum cost TSP-cycle with no repeats. Then $OPT(R) = OPT(NR)$.

To show this, first we observe that any cycle with no repetitions is also a legal TSP-cycle overall (i.e., for the version that allows repeats). And so it follows immediately[5] that $OPT(R) \leq OPT(NR)$ — **Part A**.

Next, let $C$ be a cycle with repeats, for example:

$$C = 0, 1, 2, 3, 2, 4(, 0)$$

We can now create cycle $C'$ by skipping the repeated vertices. For example:

$$C' = 0, 1, 2, 3, 4(, 0)$$

By the triangle inequality, we know that $d(C') \leq d(C)$. For example, in constructing $C'$ from $C$, we replace $(3, 2, 4)$ with $(3, 4)$; we know that $d(3, 4) \leq d(3, 2) + d(2, 5)$, by the triangle inequality. We can repeat this inductively for every skipped vertex.

If we assume that cycle $C = OPT(R)$, then we conclude that $OPT(NR) = d(C') \leq d(C) = OPT(R)$ — **Part B**.

Putting together the two inequalities in **Part A** and **Part B**, we have shown that $OPT(NR) \leq OPT(R)$ and $OPT(R) \leq OPT(NR)$, which yields our desired conclusion that $OPT(R) = OPT(NR)$.

**Claim 2.** If $A$ is a $c$-approximation algorithm for M-NR-TSP, then $A$ is a $c$-approximation algorithm for M-R-TSP.

Assume algorithm $A$ outputs cycle $C$. By assumption, $C$ has no repeats, and $d(C) \leq c \cdot OPT(NR)$. As $OPT(R) = OPT(NR)$ from **Claim 1**, we have = $d(C) \leq c \cdot OPT(R)$. Thus we see that $C$ is also a valid solution for M-R-TSP and provides a $c$-approximation.

**Claim 3.** If $A$ is a $c$-approximation algorithm for M-R-TSP, then we can construct an algorithm $A'$ which is a $c$-approximation for M-NR-TSP.

Specifically, construct algorithm $A'$ as follows: Run algorithm $A$ to get cycle $C$ (which may have repeats); then construct $C'$ from $C$ by skipping any repeated vertex. There are two things we have to show.

First, the cost of cycle $C$ is a good approximation of the optimal for M-NR-OPT, i.e. $d(C) \leq c \cdot OPT(R)$. As $OPT(R) = OPT(NR)$ from **Claim 1**, we have = $d(C) \leq c \cdot OPT(NR)$.

[5]Remember that in Lecture 2 about relaxed ILP, when we minimize over a larger space of possible solutions, we always get a solution that is at least as good as the best solution of the smaller subspace.

Second, the cycle we have constructed in $A'$ has cost bounded by the cost of $C$, i.e. $d(C') \leq d(C)$ due to the triangle inequality, similar as in **Claim 1**.

Putting these two facts together, we see that the algorithm $A'$ that produces $C'$ is a $c$-approximation algorithm for M-NR-TSP as $d(C') \leq c \cdot OPT(NR)$. □

We can show exactly the same type of equivalence for the general and metric versions, as long as we allow repeats. In this case, we cannot create "shortcuts" because the triangle inequality does not hold; however, we do not need to create shortcuts because repeats are allowed.

**Lemma 3** *There exists a $c$-approximation algorithm for* M-R-TSP *if and only if there exists a $c$-approximation algorithm for* G-R-TSP.

**Proof** In this case, the proof contains two claims: One transforming $G$ into $M$ (which is trivial), and the other $M$ into $G$ (need some work).

**Claim 1.** If $A$ is a $c$-approximation algorithm for G-R-TSP, then $A$ is a $c$-approximation algorithm for M-R-TSP.

This claim is immediately and obviously true. The two problems are identical, except for the restriction that the input to M-R-TSP must be a metric. However, if a given input *is* a metric, then it clearly satisfies the requirements of G-R-TSP and we can just execute algorithm $A$ on that instance. Moreover, the optimal solution will be the same for G-R-TSP and M-R-TSP (since they are optimizing over the exact same set of possible cycles).

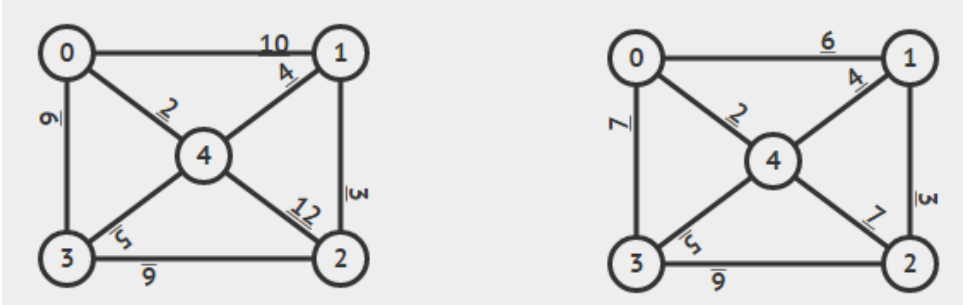


**Figure 2:** Example of constructing $A'$, reducing G-R-TSP to M-R-TSP, e.g. $C = \{0, 4, 1, 2, 3, 0\}$, then $C' = \{0, 4, 1, 2, \underline{3, 4, 0}\}$.

**Claim 2.** If $A$ is a $c$-approximation algorithm for M-R-TSP, then we can construct algorithm $A'$ is a $c$-approximation algorithm for G-R-TSP.

Let $(V, d_g)$ be the input to algorithm $A'$, i.e., an input for G-R-TSP. We construct algorithm $A'$ as follows.

First, we need to construct a distance metric $d_m$. For every pair of vertices $(u, v)$, define $d_m(u, v)$ to be the shortest path from $u$ to $v$ according to the distances $d_g$. (Recall that $d_g$ is not a metric, i.e., does not necessarily satisfy the triangle inequality.) We can find $d_m$ by construct the complete graph on $V$, assign the weight of edge $(u, v)$ to be $d_g(u, v)$, and executing an All-Pairs-Shortest-Paths algorithm, e.g. $O(V^3)$ Floyd Warshall's algorithm. You will notice that $d_m$ is now a distance metric, i.e., it satisfies the triangle inequality. Recall that $d_m$ is called the metric completion of the graph[6].

[6]We have done similar thing in Lecture 3b: Steiner Tree, General versus Metric.

Second, execute algorithm $A$ on $(V, d_m)$, producing cycle $C$. We then need to construct a new cycle $C'$ out of $C$. For each edge $(u, v)$, we add the shortest path from $u$ to $v$ according to $d_g$ to our output cycle $C'$. Notice this produces a cycle $C'$ that may contain may repeated vertices[7].

We now argue that the result is a good approximation of the $OPT(V, d_g)$. First, observe that $d(C') = d(C)$. Each edge in $C$ got replaced by a path in $C'$ with the exact same cost, and hence we end up with the same cost cycle.

Second, we know by assumption that $d(C) \leq c \cdot OPT(V, d_m)$, by the assumption that $A$ is a $c$-approximation algorithm.

Finally, we argue that $OPT(V, d_m) \leq OPT(V, d_g)$. In particular, if $R$ is the optimal cycle for $(V, d_g)$, then $R$ is also a cycle in $(V, d_m)$. Moreover, for every pair $(u, v)$, we know that $d_m(u, v) \leq d_g(u, v)$ (because $d_m$ is defined as the shortest path). Thus $d_m(R) \leq d_g(R)$. Since the optimal cycle with metric $d_m$ has to be at least as good as $d_m(R)$, we conclude that $OPT(V, d_m) = d_m(R) \leq d_g(R) = OPT(V, d_g)$.

Putting the pieces together, we conclude that $d(C') = d(C) \leq c \cdot OPT(V, d_m) \leq c \cdot OPT(V, d_g)$, and hence algorithm $A'$ is a $c$-approximation algorithm for G-R-TSP. □

# 3 2-Approximation Algorithm

We now present a 2-approximation algorithm for G-R-TSP. We already know that this will also yield a 2-approximation algorithm for the other two variants. Assume that the input is $(V, d)$ where $V$ is a set of points and $d$ is a distance function (but not necessarily a metric). Consider the following algorithm:

1. Construct the complete graph $G = (V, E)$ with weights $w$ where $E$ contains every pair $(u, v) \in V \times V$ and $w(u, v) = d(u, v)$.
2. Let $T$ be the Minimum Spanning Tree of $G$.
3. Let $C$ be the cycle constructed by performing a Depth-First Search of $T$.

To analyze this algorithm, as usual[8], we start with the optimal solution and work backwards. Let $C^*$ be the optimal (minimum cost) TSP cycle for input $(V, d)$ and let $E^*$ be the edges in $C^*$. Notice that the graph $G^* = (V, E^*)$ is connected, because $C^*$ is a cycle that includes every point. Let $T^*$ be the Minimum Spanning Tree of $G = (V, E^*)$. At this point, we know that:

$$d(T^*) \leq d(C^*) = OPT$$

Since the tree $T$ is a minimum spanning tree of $G = (V, E)$ and $E^* \subseteq E$, we know that:

$$d(T) \leq d(T^*)$$

Finally, since $C$ is construct by a Depth-First Search traversal of $T$, we know that $C$ includes each edge in $T$ exactly twice, and so:

$$d(C) = 2 \times d(T)$$

Putting these inequalities together, we conclude that:

$$d(C) = 2 \times d(T) \leq 2 \times d(T^*) \leq 2 \times d(C^*) \leq 2 \times OPT$$

That is, the cost of $C$ is at most twice the cost of OPT, and hence the algorithm is a 2-approximation algorithm.

[7] Again, this is the same as with Lecture 3b: Steiner Tree, General versus Metric.

[8] We have done similar thing in Lecture 3b: Steiner Tree, General versus Metric.

# 4 Eulerian Cycles

We now want to develop a better approximation algorithm, one that will have a better approximation ratio than 2. In order to do that, we are going to need a few additional tools. Let's begin with a famous problem, known as the *Bridges of Konigsberg*.

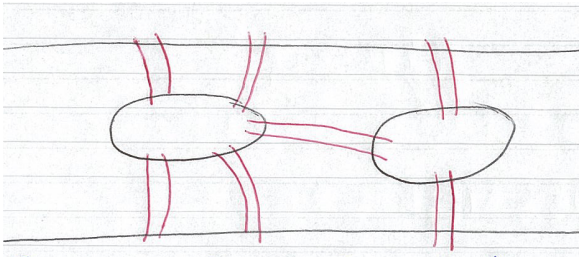

**Figure 3:** Illustration of the Bridges of Konigsberg.

In Konigsberg, there is a river with two islands. Historically, these two islands were connected by seven bridges. The bridges were beautiful and famous, and so the question that everyone wanted to answer was whether it was possible to cross each bridge exactly once—and end up back where you started. With a little bit of thought, it becomes obvious that you cannot. But how would you prove it? Euler solved this problem, and his solution is often seen as marking the beginning of graph theory as a mathematical field. Notably, he realized that the problem could be represented as a graph:

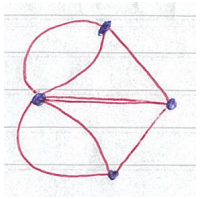

**Figure 4:** Illustration of the Bridges of Konigsberg as a graph.

Once the problem was represented abstractly as a graph, we can apply more powerful techniques to solve the problem.

For the purpose of this section, we will be talking about *multigraphs* rather than simple graphs. A *multigraph* is a graph $G = (V, E)$ where $E$ is a multiset, rather than a set, of edges. This means that each edge can appear in the graph more than once. For example, a given edge $(u, v)$ might appear in graph $G$ four times.

**Definition 4** *A **Eulerian Cycle** in a multigraph $G$ is a cycle that crosses each edge exactly once.*

Notice that, unlike a TSP-cycle, an Eulerian Cycle focuses on edges, rather than vertices. The goal is to cross each edge exactly once—though that may mean visiting a given vertex many times. Clearly the problem of the Bridges of Konigsberg is simply asking whether there is an Eulerian Cycle in the associated graph.

The key claim is to detect when a graph has an Eulerian Cycle. Amazingly, there is a very simple characterization:

**Lemma 5** *A multigraph $G = (V, E)$ has an Eulerian Cycle if and only if:*

- *It is connected (except for vertices of degree 0).*

- *Every vertex has even degree.*

**Proof** Notice that one direction of this claim is easy. Assume graph $G$ has an Eulerian Cycle. In that case, since the cycle cross every edge, clearly the cycle visits every vertex (except those vertices with degree zero) and hence graph is connected—except for vertices with degree zero. In addition, the cycle enters and exits each vertex the same number of times. For example, if the cycle enters a vertex $v$ some $k$ times, then it also exits vertex $v$ the same $k$ times. From this we conclude that every vertex has even degree. (In our example, vertex $v$ has degree $2k$, which is even.)

The surprising result is the opposite direction. Assume graph $G$ is connected (except for vertices of degree 0) and that every vertex has even degree. We need to construct a Eulerian Cycle. We will proceed by induction on the number of edges, proving a stronger invariant as we go: If every vertex in $G$ has even degree, then each connected component with $> 1$ vertices has an Eulerian Cycle.

We will begin (in the base case) with an empty graph containing only the vertices, and add one edge at a time back to the graph. At every step, we will construct an Eulerian Cycle for each connected component.

**Base case:** In this case, $m = 0$, i.e., there are no edges, and hence trivially the claim holds.

**Inductive step:** Assume we have $m$ edges. Let $V_1, V_2, \ldots, V_k$ be the connected components of $G$ that contain $> 1$ vertices. We consider two cases:

*Case 1:* $k \geq 2$: There are at least two connected components. Each connected component has $< m$ edges. Hence, by induction, each component has an Eulerian Cycle.

*Case 2:* $k = 1$: $V_1$ is the only connected component with $> 1$ vertices. Recall that all the vertices in $V_1$ have even degree, and every vertex note in $V_1$ has degree zero.

First, we claim that there must be a cycle in $V_1$. Let $n_1 = |V_1|$. If there are no cycles in $V_1$, then $V_1$ is a tree and contains exactly $n_1 - 1$ edges. However, the degree of every vertex in $V_1$ is at least 2 (since the degree is even and $> 0$). Thus there are at least $2V_1$ endpoints of edges in $V_1$, and hence at least $V_1$ edges in $V_1$. (Notice in general that if $d(v)$ is the degree of vertex $v$, then a graph has exactly $\sum_{u \in V} d(u)/2$ edges.) This contradicts the claim that $V_1$ is a tree.

Let $C$ be any cycle in $V_1$. Let $G'$ be the the graph $G$ where the edges in $C$ are removed. Let $V'_1, V'_2, \ldots, V'_{k'}$ be the new connected components of $G'$. (By removing the cycle $C$, we may have split $V_1$ into several distinct connected components; or $V_1$ may remain connected.) Each of these components now has $< m$ edges. Hence, by induction, each of these components has an Eulerian Cycle. Let $C_1, C_2, \ldots, C_k$ be the Eulerian Cycles for connected components $V'_1, V'_2, \ldots, V'_{k'}$.

It remains to stitch these Eulerian Cycles back together. Notice that these connected components are all connected by the initial cycle $C$. That is, if we add back the cycle $C$, we will have a single connected component. Here's how we do that:

Let $C = (v_1, v_2, \ldots, v_\ell)$. Begin at vertex $v_1$. We know that vertex $v_1$ is part of one of the Eulerian Cycles $V_j$. Follow the Eulerian Cycle $V_j$ until it returns to vertex $v_1$. Then proceed to vertex $v_2$. If $v_2$ is part of a new Eulerian Cycle (i.e., not $V_j$), then follow the new Eulerian Cycle until it returns to $v_2$. Otherwise move on to $v_3$. And so on. At each step, we move to the next vertex $v_i$ in the cycle $C$. If that vertex $v_i$ is part of a new Eulerian Cycle $V_i$ that has not yet been traversed, then we traverse the cycle until it returns to $v_i$. Then we continue to $v_{i+1}$. This continues until we return to vertex $v_1$.

Notice that the new cycle we have constructed traverses every edge in the graph $G$, as it traverses all the edges in the connected components $V'_1, \ldots, V'_{k'}$ as well as all the edges in $C$. Hence we have constructed an Eulerian Cycle for $G$. □

The other interesting fact is that this Eulerian Cycle can be easily constructed in polynomial time, exactly by following the steps of the algorithm:

1. Find a cycle in $G$ (e.g., via DFS). If there is none, then skip the next step.
2. Remove the cycle.
3. Find the connected components in the residual graph.
4. Recurse: Find an Eulerian Cycle in each connected component.
5. If a cycle was found in Step 1, paste the cycles back together.

This can readily be implemented in $O(m^2)$ time. Can you optimize it further? (see [3]).

# 5 A Better TSP Approximation

We consider the M-R-TSP variant again, and as before, if we can have a $c$-approximation algorithm for this variant, then we can also approximate the other two variants: M-NR-TSP and G-R-TSP. Assume $(V, d)$ is the input to our problem.

## 5.1 Basic Idea

Imagine that we built a multigraph $G = (V, E)$ that contained an Eulerian Cycle. Then that cycle would give us a feasible solution to the TSP problem. Conversely, if had a solution to the TSP problem, we could build a multigraph consisting of only the edges in the TSP-cycle, which would result in a graph with a Eulerian Cycle. Hence solving TSP is equivalent to finding a minimum cost set of edges $E$ such that $G = (V, E)$ has an Eulerian Cycle.

For example, consider the following proposed algorithm:

1. Find the MST $T$ of the complete graph on $V$ with weights defined by $d$.
2. Add every edge in $T$ *twice* to the set $E$. This ensures that each vertex in $V$ has even degree.
3. Since every vertex in multigraph $G = (V, E)$ has even degree, we can find a Eulerian Cycle $C$ for $G$.
4. Return $C$ (skipping repeated vertices).

This is a correct algorithm for solving the TSP problem. Unfortunately, it will not yield any improvement over the earlier 2-approximation algorithm because we are still adding each edge in the MST twice, leading to a cost that is at most twice optimal.

Notice that it is quite inefficient to add each edge twice! Our only goal is to ensure that each vertex has even degree—if a vertex already has even degree, why are we doubling all of its outgoing edges, and thus increasing our cost? Consider the example in Figure 5.

Thus the question we must answer is: How do we add the minimal number of edges to a multigraph to ensure that each vertex has even degree? Thus, in summary, the resulting algorithm should look like:

1. Find the MST $T$ of the complete graph on $V$ with weights defined by $d$.
2. Add every edge in $T$ to the set $E$.

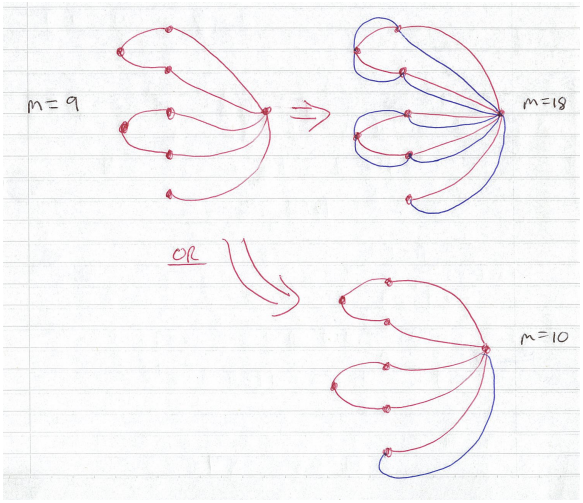


**Figure 5:** Example of finding a TSP by creating vertices with even degree. Initially, we have a graph with nine edges. If we double every edges, we have a graph with 18 edges. However, if we want to ensure that eachvertex has even degree, it suffices to add just one edge to the graph, yielding a graph with only 10 edges.

3. Add the minimum cost set of edges $E'$ to $E$ such that in $E \cup E'$, every vertex has even degree.
4. Since every vertex in multigraph $G = (V, E \cup E')$ has even degree, we can find a Eulerian Cycle $C$ for $G$.
5. Return $C$ (skipping repeated vertices).

We already know that $cost(T) \leq cost(OPT)$, since the optimal TSP-cycle is also a spanning tree (once you have removed a single edge). The key is to prove that $cost(E') \leq OPT/2$.

## 5.2 Matchings

The last tool we need is a perfect matching:

**Definition 6** *We say that $(V, M)$ is a **matching** if no two edges in $M$ share an endpoint, i.e., all vertices in $V$ have degree $\leq 1$. We say that $(V, M)$ is a perfect matching if every vertex in $V$ has exactly degree $1$, i.e., every vertex is matched.*

Notice that a perfect matching only exists if $|V|$ is even: if there are an odd number of vertices, it is clearly impossible to match them all.

If each edge has a weight, then we may want to find the minimum cost perfect matching. One of the most beautiful results in combinatorial optimization yields exactly that:

**Theorem 7** *There is a polynomial time algorithm for finding the minimum cost perfect matching.*

This finds the optimal solution, and it does it efficiently! Unfortunately, this algorithm is beyond the scope of today's class. (See Edmonds's algorithm [1]. The solution involves his famous *Blossom Algorithm* for finding matchings, along with the primal-dual method and linear programming.) For now, we are just going to assume that we can find a minimum cost perfect matching.

## 5.3 Evens and Odds

We want to use the idea of perfect matchings to pair up the vertices in the multigraph that have odd degree. If we can assign each of them a partner, then they will have even degree. But there is one potential problem: what if there are an odd number of vertices that need a partern? Luckily, that is impossible due to a nice counting argument:

**Lemma 8** *In every graph $G = (V, E)$, there are an even number of vertices with odd degree.*

**Proof** Let $O$ be the set of vertices with odd degree, and $V - O$ the set of vertices with even degree. Let $deg(u)$ be the degree of $u$.

Recall that we can count the number of edges in the graph by summing up the total degree and diving by 2, i.e.:

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

That is, the sum of the degrees of all the vertices is even.

Now, consider just the vertices in $V - O$: since each vertex has even degree, the sum of their degrees must also be even:

$$\sum_{u \in V-O} deg(u) = 2k$$

Summing everything, we conclude that:

$$\begin{aligned} \sum_{u \in O} deg(u) + \sum_{u \in V-O} deg(u) &= \sum_{u \in V} deg(u) \\ \sum_{u \in O} deg(u) + 2k &= 2|E| \end{aligned}$$

Since the right-hand-side of the expression is even, the left-hand side of the expression must also be even. That is:

$$\sum_{u \in O} deg(u) = 2\ell$$

But each vertex in $O$ has odd degree. Hence the only way that the sum of their degrees can be even is if there are an even number of odd vertices, i.e., $|O|$ is even. □

## 5.4 Christofides Algorithm

Putting the pieces together, we now have the following algorithm:

1. Find the MST $T$ of the complete graph on $V$ with weights defined by $d$.
2. Add every edge in $T$ to the set $E$.
3. Let $O$ be the vertices in $T$ with odd degree. Notice that $|O|$ is even.
4. Let $M$ be the minimum cost perfect matching for $O$.
5. Construct the multigraph $G = (V, E \cup M)$.
6. Since every vertex in multigraph $G = (V, E \cup M)$ has even degree, we can find a Eulerian Cycle $E$ for $G$.

7. Return $E$ (skipping repeated vertices).

To analyze this, we notice that the cost consists of the following:

$$cost(E) = \sum_{e \in E} d(e) + \sum_{e \in M} d(e)$$

The first part of the expression comes from the MST, and hence we already know that:

$$\sum_{e \in E} d(e) \leq OPT$$

Since OPT forms a cycle, we can remove any edge from the cycle to get a spanning tree; the MST $T$ must have cost no greater than this cycle.

We now need to argue that $2 \times \sum_{e \in M} d(e) \leq OPT$. Let $C$ be the cycle for OPT. Let $C'$ be the same cycle as $C$ where we skip all the vertices in $V - O$, and we skip repeats. That is, $C'$ is a cycle on the odd vertices with no repeats. Notice that $cost(C') \leq cost(C)$, since we have only skipped vertices (and the triangle inequality holds).

Also, notice that cycle $C'$ has an even number of vertices (because there are an even number of odd vertices) and an even number of edges. Assume that $C' = (v_1, v_2, \ldots, v_{2k})$.

We can now construct two different perfect matchings $M_1$ and $M_2$. We define them as follows:

$$\begin{aligned} M_1 &= (v_1, v_2), (v_3, v_4), (v_5, v_6), \ldots, (v_{2k-1}, v_{2k}) \\ M_2 &= (v_2, v_3), (v_4, v_5), (v_6, v_7), \ldots, (v_{2k}, v_1) \end{aligned}$$

Notice that each of these perfect matchings has $k = |O|/2$ edges, and both are valid perfect matchings for the set $O$. Since $M$ is the minimum cost perfect matching, we conclude that:

$$\begin{aligned} cost(M) &\leq cost(M_1) \\ cost(M) &\leq cost(M_2) \end{aligned}$$

Notice, though, that $cost(M_1) + cost(M_2) = cost(C') \leq cost(C)$. So, if we sum the matchings, we get:

$$2 \times cost(M) \leq cost(M_1) + cost(M_2) \leq cost(C) = OPT$$

Thus we have shown that $cost(M) \leq OPT/2$.

Putting the pieces together, we see that the cost of the cycle output by the algorithm is:

$$\begin{aligned} cost(E) = \sum_{e \in E} d(e) + \sum_{e \in M} d(e) &\leq cost(T) + cost(M) \\ &\leq OPT + OPT/2 \\ &\leq 1.5 \cdot OPT \end{aligned}$$

Thus, we have discovered a 1.5-approximation algorithm for the M-R-TSP.

# References


[1] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal on Maths*, 17:449–467, 1965.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] Steven Halim and Felix Halim. *Competitive Programming: The New Lower Bound of Programming Contests*. Lulu, 3rd edition, 2013.

# Index