CSIT 570

IT Fundamentals

Fall 2003

Dr. Rudolf Fleischer

The Hong Kong University of Science and Technology

October 4, 2003

Chapter 4

The Traveling Salesman Problem

5. Lecture

4.1 The Problem

The input to the *Traveling Salesman Problem (TSP)* is an undirected complete graph $G = (V, E)$ with edge lengths $c : E \to \mathbb{N}$. The problem is to find a shortest round trip (called a *tour*) in the graph, i.e., a minimum weight cycle visiting each node at least once. Such a cycle is called a `TSP` *tour*.

Theorem 4.1.1. `TSP` *is NP-complete.* □

4.2 Approximation Algorithms

Since `TSP` is a very important problem (in Operations Research, for example), people have designed good approximation algorithms and powerful heuristics that can even (often) solve rather large `TSP` instances in reasonable time.

In this section, we will only consider `Metric TSP`, where the edge lengths obey the triangle inequality (i.e., no detour can be shorter than a direct connection). This is an important special case of `TSP`; it arises, for example, when we plan shortest tours visiting all cities in some country. `Metric TSP` is still NP-complete. Here are a few (non-)approximability results without proofs.

(a) Let $c \geq 1$. If P $\neq$ NP then there is no polynomial time c-approximation algorithm for general `TSP`.

(b) `TSP` is NPO-complete (NPO is the class of hardest optimization problems, analogously to NP-complete), so there is no 2^{n^ϵ}-approximation for small ϵ (Orponen, Mannila '87).

(c) For `Metric TSP`, an approximation factor of $\frac{5381}{5380} - \epsilon$ for any $\epsilon > 0$ is impossible (Engebretsen '99).

(d) For `Geometric TSP` (cities in the plane) there is a PTAS (Arora '96), even in $\mathbb{R}^d$ for $d \geq 3$ (Arora '97). A PTAS (polynomial time approximation scheme) can compute an $(1+\epsilon)$-approximation for any $\epsilon > 0$ in polynomial time (but time is usually exponential in $\frac{1}{\epsilon}$). □

4.2.1 MST Approximation

We start with a simple approximation algorithm based on computing a MST of G.

Algorithm `Approx-TSP`

- Compute a minimum spanning tree T of G (see Figure 4.1).
- Start at an arbitrary node v of the graph and "walk around the tree T". Let L be the list of nodes encountered on this tour. L is not a simple cycle because interior nodes of T appear at least twice in L. If we remove all but the first appearance of each node from L, then we obtain a simple cycle C (note that G is the complete graph, so all shortcuts exist).

Note that C is just a preorder traversal of T rooted at v.

Theorem 4.2.1. *In metric graphs,* `Approx-TSP` *computes in time $O(n^2)$ a 2-approximation C of the shortest* `TSP`*-tour.*

Proof. The running time is dominated by the time for computing a minimum spanning tree. Prim's algorithm can do this in time $O(m + n \log n)$ which is equal to $O(n^2)$ in a complete graph.

Let C^* be an optimal `TSP`-tour. Removing any edge from C^* gives a spanning tree of G (actually a spanning path). Thus, the cost of C^* is at least the cost of the minimum spanning tree T. Since walking around T will visit every edge of T exactly twice, the cost of L is exactly twice the cost of T, and thus at most twice the cost of C^*. Introducing shortcuts in L to obtain C can only reduce the cost (here we need `Metric TSP`). □

4.2.2 Christofides' Algorithm

In 1976, Christofides refined `Approx-TSP` so that it even computes a $\frac{3}{2}$-approximation. To describe the algorithm, we first need a few definitions and lemmas. A node is called *odd* if it has odd degree.

Lemma 4.2.2. *Any graph G has an even number of odd nodes.*

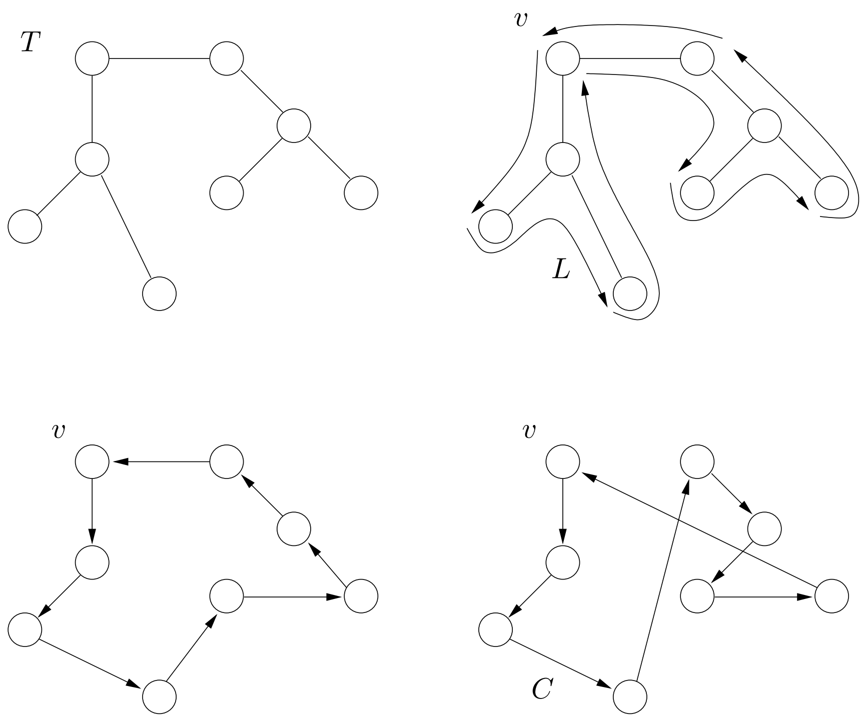

Figure 4.1: A minimum spanning tree T for eight cities in the plane (top left), the walk L around T starting at v (top right), the simple cycle C derived from L by introducing shortcuts (bottom right), and the minimum tour (bottom left).

Proof. If m is the number of edges in G, then $\sum_v deg(v) = 2m$. Splitting the sum into two sums, S_e for the nodes of even degree and S_o for nodes of odd degree, we see that S_o must be even. Since all terms in S_o are odd, there must be an even number of terms in S_o, i.e., there must be an even number of odd nodes. □

A *matching* M is a set of pairwise disjoint edges, i.e., each node is hit by at most one edge in M. In a *perfect matching*, all nodes are hit exactly once. Of course, only graphs with an even number of nodes can have a perfect matching. A *minimum weight perfect matching* is a perfect matching of least cost. We will learn later in the course that a minimum weight perfect matching can be computed in polynomial time.

A graph is called *Eulerian* if there exists a cycle that visits every edge exactly once. Such a cycle is called an *Eulerian tour*.

Lemma 4.2.3. *A graph is Eulerian if and only if it is connected and all nodes have even degree. An Eulerian tour can be computed in linear time.*

Proof. An Eulerian tour may visit a node several times, but each time it enters and leaves the node on two different edges. Thus, in an Eulerian graph all nodes have even degree.

If all nodes have even degree, then we can greedily compute an Eulerian tour as follows. Starting at an arbitrary node v, we always leave a node on some arbitrary unused edge. Since nodes have even degree, there must always be an unused edge when we have just entered a node. Except, if we enter v later again there may be no unused edge, in which case we have closed a cycle C. If C contains all edges, we are done. Otherwise, there is a node w on C with unused adjacent edges (because the graph is connected). We compute a second cycle C', starting in w (the edges of C are still classified as "used"). Then we combine C and C' into a single larger cycle by splicing C' into C at w, i.e., we follow C until we reach w for the first time, then follow C', then continue on C from w. □

In algorithm `Approx-TSP`, we walked around the tree T. In a more formal way, we could say we take T twice to define a multi-graph F, i.e., the edges of T appear exactly twice in F (in a *multi-graph* edges can appear several times). F is Eulerian (each edge exists twice). Walking around T is one possible Eulerian tour in F (there can also be others). Therefore, `Approx-TSP` is also sometimes called `Double-Tree Algorithm`. The idea of Christofides' algorithm is to make T Eulerian by overlaying it with some smaller graph than T itself, thus reducing the total cost of the overlay multi-graph.

Christofides' Algorithm

- Compute a minimum spanning tree T of G (see Figure 4.2).
- Compute a minimum weight perfect matching M of the odd nodes of T (by Lemma 4.2.2, there is an even number of odd nodes).
- Consider the multi-graph F that is the overlay of T and M. If M also contains edges of T, these edges will appear as multiple edges in F. Since all odd nodes in T are adjacent to exactly one edge in M, F is Eulerian (see Lemma 4.2.3) and we can compute an Eulerian tour C'.
- Introduce shortcuts in C' to obtain a TSP tour C in G.

Theorem 4.2.4. *In metric graphs, Christofides' algorithm computes a $\frac{3}{2}$-approximation of the shortest* `TSP`*-tour in polynomial time.*

Proof. Let c^* be the cost of an optimal `TSP` tour C^*. Again, c^* is at least as large as the cost of T.

If we connect the odd nodes of T in the order of their cyclic appearance on C^* we obtain a cycle of cost at most c^* (because of the triangle inequality).

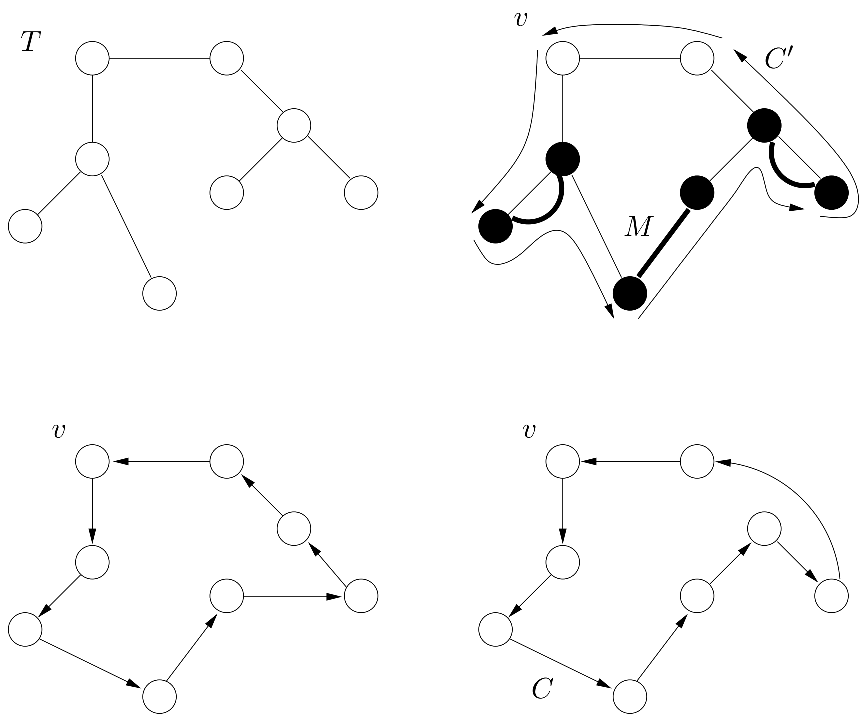

Figure 4.2: A minimum spanning tree T for eight cities in the plane (top left), an Eulerian tour C' in $T + M$, where M is a minimum weight perfect matching of the (black) odd nodes of T (top right), the simple cycle C derived from C' by introducing shortcuts (bottom right), and the minimum tour (bottom left).

The cycle has an even number of nodes and edges. If we choose every second edge on the cycle, we partition its edges into two disjoint sets. Each set is a perfect matching of the odd nodes of T. Thus, one of the two sets has cost at most $\frac{c^*}{2}$. Therefore, the cost of the minimum weight perfect matching M is at most $\frac{c^*}{2}$.

F is the overlay of T and M, so the Eulerian tour C' has cost $cost(T) + cost(M) \leq c^* + \frac{c^*}{2}$. Introducing shortcuts to obtain a TSP tour can only reduce the cost. □

The running time of Christofides' algorithm is actually $O(n^3)$. Without the triangle inequality, there is no constant factor approximation for TSP.

4.2.3 Nearest Neighbour Rule

In the *Nearest Neighbor Rule (NNR)* algorithm, we form a path by starting with an arbitrary node and proceeding by always joining the previously added node to its nearest neighbour node that has not yet been selected, until we have visited all nodes. Connecting the last node selected to the first one closes the tour. Rosenkrantz, Stern and Lewis (1974) have shown the following theorem.

Theorem 4.2.5. *NNR computes a* $\frac{\log n+1}{2}$*-approximation to the optimal* `TSP` *tour.*

4.2.4 Nearest Insertion Rule

In the *Nearest Insertion Rule (NIR)* algorithm, we start with a single node tour and then proceed by always choosing the node that is closest to the current tour and adding it to the tour in a way that minimizes the cost. Rosenkrantz, Stern and Lewis (1974) have shown the following theorem.

Theorem 4.2.6. *NIR computes a* 2*-approximation to the optimal* `TSP` *tour.*

4.2.5 Local Search Heuristics

Many heuristics start with some tour and then do local changes improving the tour (thus the name *local search heuristics*). Unfortunately, local improvements do not always lead to the overall optimum solution (thus the name *heuristic*).

4.2.5.1 k-Opt Exchange

The first heuristic is called k-`Opt Exchange`, where $k \geq 2$ is some fixed parameter. A k-opt move consists of deleting some arbitrary k edges from the current tour C and reconnecting the (at most k) paths in the best possible way. Figure 4.3 shows a 2-opt move.

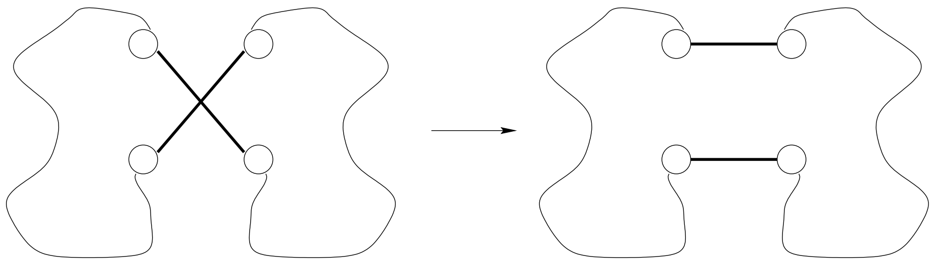

Figure 4.3: A 2-opt move. In the left tour, we remove the two thick edges and then reconnect the two pieces of the tour to obtain a shorter tour.

Algorithm k-`Opt Exchange`

- Let C be any tour in G.
- As long as there is a k-opt move that reduces the cost of the current tour, do the k-opt move.

Obviously, each iteration of k-`Opt` needs time $O(n^k)$, since there are $\binom{n}{k}$ possibilities to remove k edges from the current tour. For each set of k edges, we must try all possible ways to reconnect the tour pieces. Therefore, we would usually only choose $k = 2$ or $k = 3$.

We say a tour is *k-opt* if it cannot be improved by further k-opt moves. The following theorem shows that k-`Opt` is a heuristic that can sometimes fail to compute optimal `TSP` tours.

Theorem 4.2.7. *For any $k \geq 2$ there are* `TSP` *instances and k-opt tours that are not $(k+1)$-opt.* □

4.2.5.2 Lin-Kernighan

The *Lin-Kernighan (LK)* heuristic does not limit its search in each iteration to a 2-opt or 3-opt move that immediately improves the current tour. Instead, it tries a 3-opt move followed by several 2-opt moves, until a tour has been found that is better than the one the iteration had started with. To bound the running time, the heuristic limits the search depth and uses many other tricks to avoid unnecessary re-computations of “bad” moves. Since these tricks allow many degrees of freedom in the implementation, LK is actually a generic name for a great number of rather different heuristics. Usually, these heuristics perform much better than for example `3-Opt`. Since the heuristic often cannot find 4-opt moves that would improve the solution, one can improve the performance of LK by trying to find a good 4-opt move whenever LK stops, and then continuing with LK on that better tour.

4.3 Exact Solutions

There are two main methods to solve TSP exactly. The first one uses dynamic programming, the second one uses an integer linear program (ILP) formulation. Both methods have exponential running time. ILP based solution methods can be much more effective than the dynamic programming approach.

4.3.1 Dynamic Programming

The dynamic programming solution to TSP is due to Held and Karp (1962). Let the nodes be numbered $N = \{1, \ldots, n\}$. For $A \subseteq \{2, \ldots, n\}$ and $x \in A$, denote by $TSP(A, x)$ the minimum cost path from 1 to x only using nodes in $A + \{1\}$. We can compute these values recursively as follows:

$$TSP(A,x) = \begin{cases} cost(A,x) & \text{if} A = \{x\} \\ \min_{x \neq y \in A}(TSP(A-x,y) + cost(y,x)) & \text{otherwise} \end{cases}$$

The optimal `TSP` tour can then be obtained from

$$\min_x (TSP(N,x) + cost(x,1)) \, .$$

4.3.2 Integer Linear Program

A *Linear Program (LP)* consists of a set of variables $x_1, \dots, x_n$, a set of *constraints* $C_1, \dots, C_m$, where each C_j is either a linear equation in the x_i or a linear inequality, and a linear function of the variables, the *objective function.* The *dimension* of the LP is n, the number of variables. A *feasible solution* to the LP is a vector of values for the variables fulfilling all constraints. An *optimal solution* is a feasible solution that minimizes (maximizes) the objective function.

Example 4.3.1. Consider the following LP in two variables

$$\begin{array}{rll} \max & x_1 + 2x_2 & \\ \text{subject to} & x_1 - x_2 & \le 0 \\ & x_1 + x_2 & \le 4 \\ & x_1, x_2 & \ge 0. \end{array}$$

A feasible solution is $x_1 = x_2 = 1$. An optimal solution is $x_1 = 0$ and $x_2 = 4$. □

An *Integer Linear Program (ILP)* is an LP with the additional constraint that all variables must have integer values. The most famous algorithm to solve LPs is the `Simplex` algorithm. Although it has an exponential running time, it often runs very fast in practice. A rather unpractical algorithm to solve LPs in polynomial time is the *Ellipsoid Method* due to Karmakar and Khatchian.

On the other hand, solving ILPs is NP-complete. Unfortunately, many problems in real life, like for example `TSP`, can only be formulated as ILP, but not as LP. Doing this does not seem to be advantageous compared with, let's say, the exponential time dynamic programming approach. However, there is a rich theory on heuristics to solve ILPs in general, and these can be used to solve practical instances of `TSP`, for example. These heuristics include *LP relaxations* (solve the ILP as an LP and then somehow round the optimal solution to obtain an integer solution) and *cutting plane* algorithms (add more constraints to the ILP to make the LP relaxation more successful). Currently, the best commercial LP and ILP solver is the *CPLEX* system.