SAINT MARY'S COLLEGE OF CALIFORNIA

DEPARTMENT OF MATHEMATICS

SENIOR ESSAY

Traveling Salesman Problem

Author:

Sofia BURILLE

Mentor:

Dr. Christopher JONES

Supervisor:

Dr. Kathryn PORTER

May 16, 2016


SAINT
MARY'S
COLLEGE
of CALIFORNIA

Abstract

Given a set of cities and corresponding distances between each pair of cities, the Traveling Salesman Problem, or TSP for short, is to find the shortest way to visit all the cities exactly once and return to the starting city. TSP has been around for decades and the algorithms it inspired became tools to study other phenomena, both in mathematics and in the real world. The approximation methods, the Nearest Neighbor Algorithm, the Greedy Algorithm, the Christofides' Algorithm, are discussed. This is followed by the 2-opt and k-opt improvement methods. Finally the most common method for optimal solutions is discussed, using the Simplex Method to solve the Linear Programming form of the TSP.

1 Introduction

1.1 Definition and History of the Traveling Salesman Problem

Given a set of cities along with the cost of travel or distance between each pair of them, the **traveling salesman problem**, or **TSP** for short, is to find the cheapest or shortest way to visit all the cities and return to the starting point.[1] Many variations of this problem exist, but this is one of the most general forms. A predecessor to this formulation was specifically the distances between cities, but this was expanded to include costs. The **path** or order of the visited cities is called a **tour** or **circuit**. A tour is also usually called a **Hamiltonian circuit** in honor of Sir William Rowan Hamilton, who came up with a game that has the rules of TSP but is specifically for the dodecahedron graph. The history of the name the Traveling Salesman Problem is obscure, but we know the term originated sometime in the 1930s or 1940s, most likely at Princeton University since that is where most of the research was taking place. Merrill Flood and Hassler Whitney, both from Princeton, were influential in early TSP research, and the latter possibly coined the name. The Traveling Salesman Problem came about from its original application, which was an actual traveling salesman that needed the best route possible. The problem naturally occurred and many people started studying it in different contexts. Roughly chronological contexts include: circuit courts, circuit-riding Christian preachers, knight's tour, messenger problem (a variant by Karl Menger, where one does not have to go back to the starting point), the traveling farmer, school bus routing, and more. [1]

Figure 1: A map of a set of cities a traveling salesman may have to visit

2 Definitions and Notations

A **graph** G consists of a finite set V of **vertices**, a finite set E of **edges**, and a relationship associating with each edge $e \in E$, a pair of vertices, $u, v \in V$; an edge joins two vertices and is denoted $e(v, u)$ for vertices v and u. [1] (Figure 2a). A Traveling Salesman Problem can be expressed as a graph in which the cities are represented by vertices and the distance or cost between two cities is represented by the weight or scale of the edge.

A graph $H = (W, F)$ is a **subgraph** of $G = (V, E)$ if $W \subseteq V$ and $F \subseteq E$. If $W \subseteq V$ and $F = \{e : e \in E, e \subseteq W\}$, then $H = (W, F)$ is the subgraph **induced** by W. [1]

A graph is a **complete graph** if every pair of vertices is connected by an edge.

A **cycle** is a closed path which consists of a sequence of vertices, starting and ending at the same vertex (Figure 2b). An **Eulerian cycle** is a cycle in which each edge in the graph is visited exactly once. Thus, by definition, an Eulerian cycle may visit a vertex more than once and thus may not be a proper TSP tour. From the definition of a tour in the introduction we know that in general all tours are cycles, but not all cycles are tours. For a cycle to be a tour it must contain all vertices, except the initial vertex, exactly once; the initial vertex which is visited twice.

The **triangle inequality** states that the sum of the lengths of any two sides in a triangle is greater than or equal to the length of the remaining side. There are various versions of TSP, we will only consider instances of TSP that satisfy the triangle inequality.

A graph $G = (V, E)$ is **connected** if for each pair of vertices $u, v \in V$ there exists a path starting at u and ending at v.[1]

A **tree** is a connected graph having no cycles. [1] (Figure 2c)

A **spanning tree** is a minimal set of edges that joins all of the vertices into a single connected component, where "minimal" means that if any edge were removed the graph would no longer be connected. [1]

The **degree** of a vertex v, denoted $deg(v)$, is the number of edges incident with the vertex.

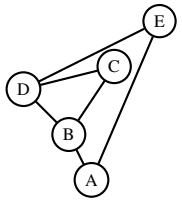

Figure 2a: Example of a graph

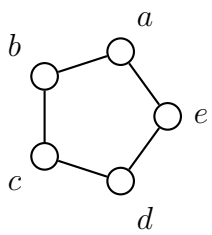

Figure 2b: Example of a cycle

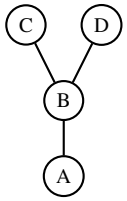

Figure 2c: Example of a tree

3 Describing the Size of the Problem

3.1 Big O Notation

The main strategy to study the Traveling Salesman Problem is through various algorithms. One property of an algorithm is how efficient it is or how quickly it works. Big O notation is the language we use to describe how long it takes an algorithm to run. Big O notation expresses this by how quickly the computational time grows relative to the input, as the input gets arbitrarily large. Some general examples include: $O(1)$ is simply constant meaning time does not depend on input, $O(n)$ is linear meaning time is roughly proportional to amount of input, and $O(n^2)$ is quadratic, meaning that the time is roughly the square of the input. In the case of TSP, the input is the set of cities and distances between each pair of cities. Big O notation only measures the highest order or most dominating part of the function, meaning the lower parts such as constants are simply dropped.

3.2 Difficulty

In general, the problems for which there exist good algorithms are known as the class P, for **polynomial time**. By good algorithm we mean that for a given input there are at most $O(n^k)$ steps where n is the complexity of the input and k is a non-negative integer. A more general class is known as NP, for **non deterministic polynomial time**. A problem is in NP if whenever the answer to a decision is yes, then there exists a means to certify this yes answer in such a way that the

certificate can be checked in polynomial time.

Using this terminology we can explain the difficulty of the Traveling Salesman Problem. TSP falls into the general category of NP problem; TSP can be solved! However, it takes $n!$ possible tours to check. This is because when making a tour we are essentially finding an ordering of the cities and there are $n!$ ways to order n objects (n permutations). There are many methods in which not all possibilities are not checked, saving a lot of time, but each method has varying effects on the accuracy of the tour. A worst-case scenario mathematically defines the absolute worst an algorithm can perform; these are described as the ratio of the worst possibility to the optimal solution. In other words, worst-case scenarios can help measure accuracy, whereas Big O notation helps measure speed. So TSP methods embody the age old battle between accuracy and speed!

4 Methods

Many approaches have been taken to understanding the TSP problem. One simple approach is to use geometry to visualize and solve the problem. This view can be as simple as finding an aesthetically pleasing path, which has been proven to improve path length, or removing acute angles or crossed edges from the tour. More complex methods include: the cutting-plane method, the primal approach, the branch-and-bounding method, Dynamic Programming, the Branch-and-Cut method, linear programming, and finding different ways to bound the problem.[1] We will go through only a selection of the different methods.

4.1 Approximation Methods

Approximation methods are concerned with finding a tour. The tour's level of efficiency, in other words computational time, and accuracy, which may be measured in a worst-case scenario ratio, depend on the algorithm used.

4.1.1 Nearest Neighbor Algorithm

The simplest approximation method is the Nearest Neighbor Algorithm. Essentially a random starting city is selected, and then the nearest unvisited city, or the one with the lowest cost, is chosen until there are no new cities left, in which case the tour goes to the initial city and is complete. This step-by-step process is illustrated in the example below. Early on in the tour this works out nicely, but then later one may have to traverse large distances to reach cities that were carelessly missed in the beginning. However, this algorithm cannot do worse than $1 + \frac{log(n)}{2}$ times the cost of an optimal tour, where n is the number of cities. [2] The computational time of the the nearest neighbor algorithm is $O(n^2)$ [7], since there are $\frac{n(n-1)}{2}$ edges in a complete graph (Traveling Salesman Problems are usually represented as complete graphs). Expanding the number of edges we get $\frac{n^2}{2} - \frac{n}{2}$. Since O notation only regards the most significant terms, the computational time of the nearest neighbor algorithm is $O(n^2)$.

Example 4.1. *Below is a small example (6 cities) of a tour produced by the nearest neighbor algorithm starting at vertex A. Each figure (b-g) is a single step in this process. The tour is denoted in red dashed lines and the final circuit is as follows: $A - E - D - F - C - B - A$*

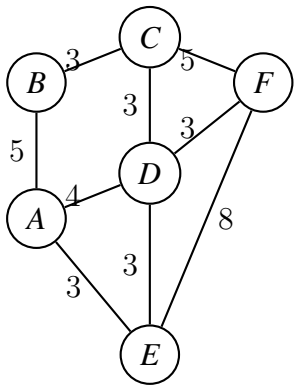

Figure 3a

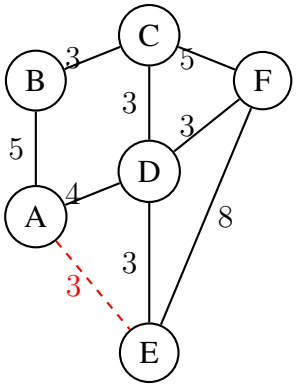

Figure 3b

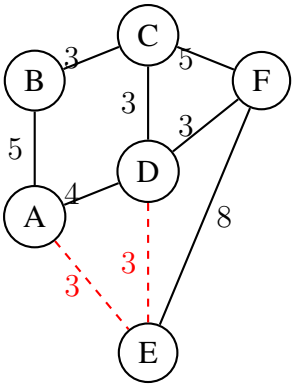

Figure 3c

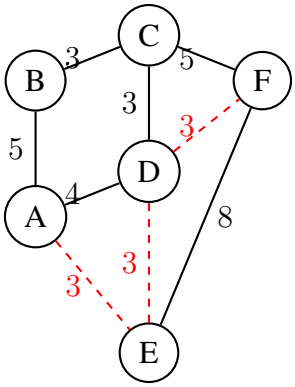

Figure 3d

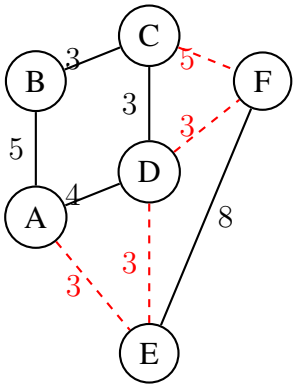

Figure 3e

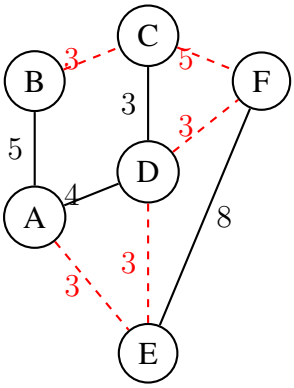

Figure 3f

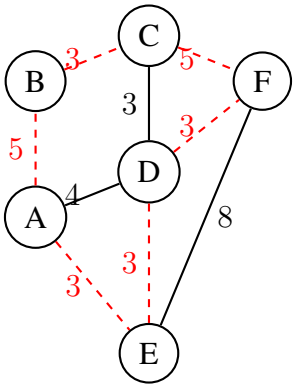

Figure 3g

Notice that there was some choice involved in the algorithm. For instance, after vertex D, the tour could have gone to vertex C next, instead of vertex F. Also, it is not shown in this simple example, but a graph must be a complete graph for the algorithm to be guaranteed to work; otherwise, a path can become stuck at a vertex and not be able to finish the tour. For simplicity only a subset of edges are shown.

4.1.2 The Greedy Algorithm

The next simplest method is the Greedy Algorithm. The Greedy Algorithm grows many subpaths simultaneously by adding the shortest available road segments each time. Essentially, all the shortest edges of the graph that can be added, without visiting a vertex more than once or completing a circuit, are added to the growing tour simultaneously, working up through all the possible edge distances until a tour of all the cities is created.

Steps:

1. Put all the edge weights in increasing order.

2. Go down the list adding edges to the tour. Do not add an edge to the tour if violates one of the rules:

   (a) Vertices must have degree of at most 2 in the the tour.

   (b) No cycles in the tour unless the number of edges equals the number of vertices in the graph (the tour is completed).

This process is illustrated in the example below. The greedy algorithm cannot do worse than $\frac{1}{2} + \frac{log(n)}{2}$ times the cost of an optimal tour, where n is the number of cities. [2]. Computational time of the greedy algorithm is $O(n^2 log_2(n))$. [7]

Example 4.2.

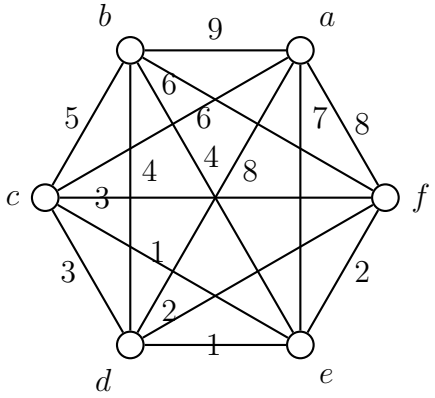

Figure 4a: A complete graph with 6 vertices.

The sorted list of edges is as follows:

edge	weight	edge	weight	edge	weight
(c,e)	1	(c,f)	3	(b,f)	6
(d,e)	1	(b,d)	4	(a,e)	7
(d,f)	2	(b,e)	4	(a,d)	8
(e,f)	2	(b,c)	5	(a,f)	8
(c,d)	3	(a,c)	6	(a,b)	9

Table 1: Edges in increasing order as one progresses down then to the right

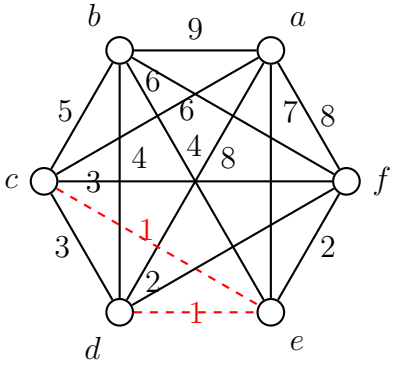

Figure 4b

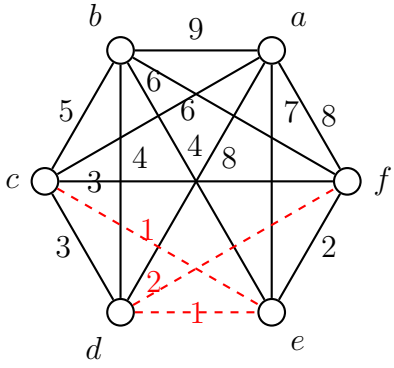

Figure 4c

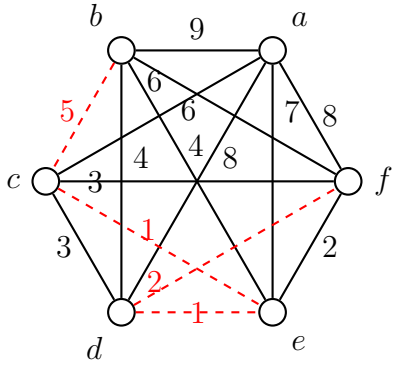

Figure 4d

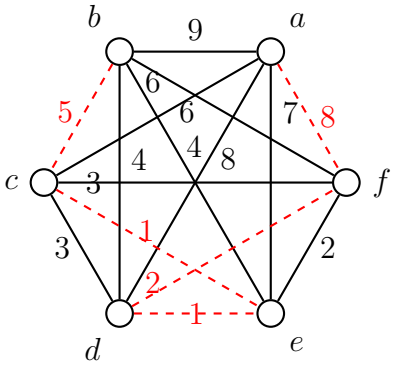

Figure 4e

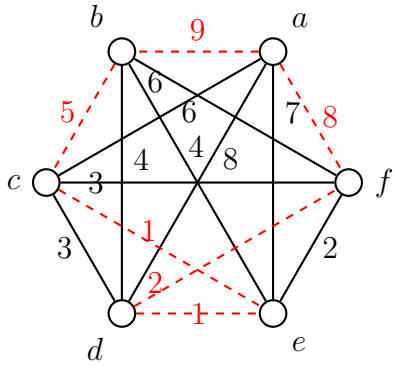

Figure 4f

Progressing down and then right in Table 1, we add acceptable edges to the tour. The first edge is $e(c, e)$, which can be added because our tour is currently empty. Next, $e(d, e)$ and $e(d, f)$ are added because they do not violate either rule. However, $e(e, f)$ cannot be added because it violates both rules: $deg(e) = 3$ and the cycle (e, d, f) would be formed. Similarly, $e(c, d)$ cannot be added ($deg(d) = 3$ and cycle (c, e, d) would be formed). Proceeding in this manner we find the tour (a, b, c, e, d, f, a).

The greedy algorithm must also be applied to a complete graph for it to be guaranteed to work.

4.1.3 The Christofides' Algorithm

Nicos Christofides, a professor at Imperial College London, combined a minimum cost **spanning tree**, a tree that spans the entire set of cities, with a **perfect matching**, a set of edges that

meet each of the odd vertices exactly once. An odd vertex is a vertex that has an odd degree in the spanning tree. [2] The steps are as follows:

1. Find a minimum spanning tree that covers the graph.

2. Connect each odd vertex in the tree to exactly one other odd vertex to create a perfect matching using Edmonds' matching algorithm. (For a details of the algorithm see [3])

3. Make a Eulerian cycle from the resulting graph, then traverse the cycle, taking shortcuts.

Finding a minimum spanning tree (step 1) is actually a very similar process to the greedy algorithm. First, all the edge weights are put in increasing order. Then edges are added in order from the list to the tree as long as they do not form cycles, until all the vertices are reached. For step 2, looking at the initial complete graph, induce a subgraph on only the odd vertices in the minimum spanning tree, put the edge weights of this induced subgraph in order, and match the vertices by going down the list simultaneously removing edges with any vertices that have already been matched. Combining the minimum spanning tree with the minimum weight matching, a Eulerian cycle results. Traverse this cycle (start at a vertex and go around to all the vertices), but in traversing this graph, if a vertex has already been visited (and all vertices have not been visited yet), skip that vertex (delete the corresponding edge), and create (use from original graph) an edge that visits the next city, thus shortcutting the Eulerian cycle. We know that this is a shortcut because all the graphs we are considering satisfy the triangle inequality, so the weights of the edges must satisfy $e(v, u) + e(v, w) \geq e(u, w)$, for all combinations of pairs of vertices v, u, and w. The resulting cycle is the TSP tour.

In comparison to the other methods thus far discussed, the Christofides' algorithm has the best worst case scenario currently known: a guarantee of at most 1.5 times that of an optimal solution. The algorithm's computational time is $O(n^3)$.

Below is an example showing each step of the Christofides' Algorithm, starting with the minimum spanning tree in black and ending with the final TSP tour in the dotted red lines.

Example 4.3.

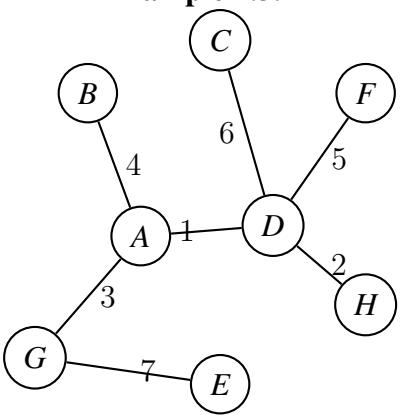

Figure 5a-Minmum spanning tree

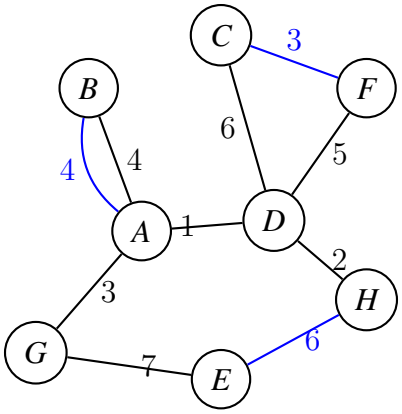

Figure 5b-minimum weight matching by connecting odd vertices

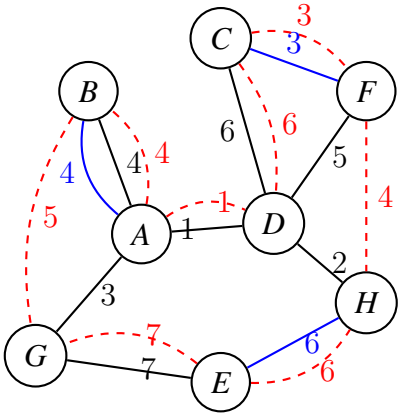

Figure 5c- short-cutted Eulerian cycle

Note: Figure 5a is the minimum spanning tree graph, Figure 5b is the union of the spanning tree graph and the matching of the odd vertices (in blue), and Figure 5c is the union of the Figure 3b graph with the outputted TSP tour (in red).

We start with a minimum spanning tree that was found from a graph. (Figure 5a) Then we locate the odd vertices in the tree, which are vertices $A, B, C, F, E,$ and H. For an optimal perfect matching one would use Edmonds algorithm, but that is outside the scope of this paper so we will arbitrarily create a matching, $(A, B), (C, F), (E, H)$. (Figure 5b) By matching all the odd vertices in the tree we have guaranteed that all vertices have even degree. Thus, the resulting graph is guaranteed to contain an Eulerian cycle. So then we traverse an Eulerian cycle, $(A, D, C, F, D, H, E, G, A, B, A)$, (Figure 5c- black blue solid lines) which by definition uses each edge exactly once. However, having visited vertices A and D more than once. By the triangle inequality, we can shortcut the path by skipping these vertices to obtain the tour $(A, D, C, F, H, E, G, B, A)$. (Figure 5c-red dashed line)

4.1.4 Summary of Algorithms

The 3 above algorithms each have their pros and cons. The nearest neighbor is the least accurate, with a worst-case ratio of $1 + \frac{log(n)}{2}$, but is also the quickest, with a computational time of $O(n^2)$.

Whereas, for large n, the Christofides' Algorithm is the most accurate of the algorithms discussed thus far, with a worst-case ratio of 1.5, but is the slowest, with a computational time of $O(n^3)$. (Table 2)

Algorithm	Worst-Case Scenario	Computational Time
Nearest Neighbor	$1 + \frac{log(n)}{2}$	$O(n^2)$
Greedy	$\frac{1}{2} + \frac{log(n)}{2}$	$O(n^2 log_2(n))$
Christofides'	1.5	$O(n^3)$

Table 2: Comparison of Algorithms: n is the number of vertices

4.2 Tour Improvement Methods

Instead of trying to make a good tour, another strategy for TSP is to start with a tour, for example from one of the algorithms discussed thus far, and make changes to make it better.

4.2.1 2-opt moves

One major way of tour improvement is to switch edges to make a more optimal tour. One example of this is a 2-opt improvement method which consists of choosing 2 edges that do not share a vertex, and then replacing the edges if there is more optimal way to connect the graph that still allows for a proper tour. By more optimal way we mean two other edges, incident with the same 4 vertices, that have a sum of weights less than the sum of weights of the original edges. This process is then done for every acceptable pair of edges. Note that for a 2-opt move, there is only one other way to connect the graph that results in a tour of all the vertices. Following is an example of one iteration of this tour-improvement process.

Example 4.4.

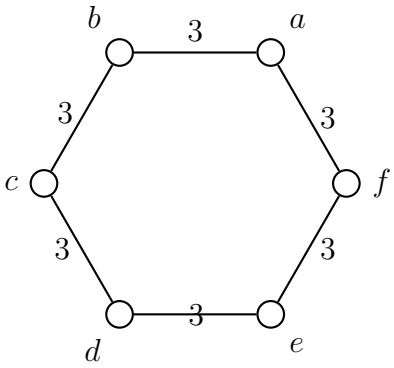

Figure 6a: A tour with total cost 18

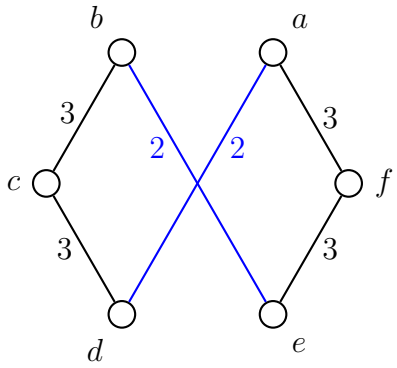

Figure 6b: The tour after the move with cost 16

Note: The entire graph is not shown. The edges (a, b) and (d, e) have cost $3 + 3 = 6$. Once the are removed (Figure 4b without the blue edges), the only way to properly reconnect the tour is to add the edges (a, d) and (b, e). Connecting vertex b to vertex d, would force vertex a to connect to vertex e which would result in two disconnected cycles and therefore is not a tour. Thus, the only reconfiguration left is the one shown above, adding edges (a, d) and(b, e) with cost $2 + 2 = 4$. Thus, an improved tour is achieved. We could proceed to check all possible 2-opt moves, making changes when a move decreases the cost, to further improve the tour.

4.2.2 Lin, Kernighan, and Helsgaun

Instead of only replacing 2 edges at a time, more edges could be considered at once, so then in a k-opt move, k edges are replaced at a time. However, the larger the k the more difficult, in terms of polynomial time, it is to check all the possible k-tuples. Furthermore, unlike Example 4.4, $k > 2$ results in more possible ways to reconnect the tour. Since checking one 4-opt move is already difficult, iterating this multiple times for all edges can become increasingly difficult. So Shen Lin and Brian Kernighan, who is a Canadian computer scientist at Princeton University, came up with a variable k-opt algorithm in which for each iteration a value for k is picked. [5] Essentially, the Lin-Kernighan algorithm considers ascending values for k for each iteration, given that a specific k is being studied, the algorithm checks to see if the next value should even be deliberated. To make

this possible Lin and Kernighan had to make lots of restrictions to which possibilities were even checked to help reduce running time. [5]

However, these changes sometimes made it so the optimal solution could not be reached, so then Keld Helsgaun at Roskible University managed to make some modifications to the algorithm loosening some criteria, such as considering more and larger (5-opt moves) edge exchanges. This caused there to be many possibilities in picking replacement edges and Helsgaun accounted for this by essentially coding for each of the possible cases. [2] This improvement, now called the Lin-Kernighan-Helsgaun method, or LKH for short, has become one of the best performance TSP algorithms, managing great accuracy and speed. However, both of the algorithms are very complex and therefore we will not discussed in further details.

4.3 Optimal Solutions

4.3.1 Linear Programming and the Simplex Algorithm

The best tool to find optimal tours is Linear Programming. **Linear programming** combines a set of simple rules or constraints to optimize a linear function, referred to as the **objective function**. From this definition we can see that linear programming reaches far into many other applications besides TSP, so long as the function is linear. In TSP all of these rules (which form a set of equations and inequalities), must be satisfied by all the tours in order to obtain a feasible solution (possible solution, given the constraints). In order to solve the linear program model, the simplex algorithm is used. First the problem must be separated into decision and slack variables into **independent** (having value zero) and **dependent** (having a nonzero value) sets. A description and example of **simplex algorithm** is described below from [4].

1. Put the linear program in Tableau Format.

2. Begin with a known extreme point, usually the origin (0,0).

3. Determine whether an adjacent intersection point improves the value of the objective function. If not, the current extreme point is optimal. If yes, perform the optimality test to

determine which variable in the independent set should enter the dependent set and become nonzero.

4. In order to find a new intersection point. a variable in the dependent set must exit to allow the variable from Step 3 to enter the dependent set. Perform the feasibility test to decide which variable should exit such that the feasibility is still possible.

5. Form a new, equivalent system of equations by eliminating the new dependent variable from the equations that do not contain the variable that exited in Step 4. Then set the new independent variable to zero in the new system to find the values of the new dependent variables, thereby determining an intersection point.

6. Repeat steps 3-5 until an optimal extreme point is found.

Note: In general a computer will solve this so we will not go through these steps in more detail.

Unfortunately, the Simplex Algorithm has a very high worst-case computational time, many problems are exponential. However, in practice many times the solution is found in only a few steps. So the simplex algorithm works really well for some classes of problems, but really poorly for others.

The general form of LP problems is:

$$\text{maximize } c^T x$$

$$\text{subject to}$$

$$Ax \leq b$$

$$l \leq x \leq u$$

where c^T is the transpose of the $c - vector$ or cost vector. Matrix A is called the **constraint matrix**; vector b is the *right-hand-side vector*; vectors l and u are the **lower and upper bounds**; $c^T x$ is the **objective function** [1]

An example construction of a general Linear Program from [4] is:

$$\text{Minimize } 25x_1 + 30x_2$$

Subject to

$$20x_1 + 30x_2 \leq 690$$

$$5x_1 + 4x_2 \leq 120$$

$$x_1, x_2 \leq 0$$

4.3.2 Linear Programming for TSP

In more specific terms the TSP format given in [6] is as follows

Maximize:

$$\sum_{i=0}^{n} \sum_{j \neq i, j=0}^{n} c_{ij} x_{ij}$$

Subject to:

$\sum_{j=1}^{n} x_{ij} = 1$	$0 \leq i \leq n$	out degrees $= 1$
$\sum_{i=1}^{n} x_{ij} = 1$	$0 \leq j \leq n$	in degrees $= 1$
$u_i - u_j + n x_{ij} \leq n - 1$	$0 \leq i \neq j \leq n$	guarantees no disjoint cycles

The notation is defined as the following:

- c_{ij} is the cost of the edge going from i to j.
- $x_{ij} = 1$ if the tour travels from vertex i to vertex j, and 0 otherwise. The tour must have exactly one edge coming out from vertex i (out degree = 1) and one edge coming into vertex i (in degree = 1). This is because a tour is a cycle.

- u_i, u_j are dummy variables for which we must solve.

  In other words, we want to minimize the sum of all the edge costs in our tour. We know we will only add edges in our tour because if $e(i, j)$ is not in our tour, $x_{ij} = 0$, which insures we do not add the cost of that edge. Since we have a linear function we wish to optimize and a set of constraints we can apply the Simplex Method to solve this form of a Linear Program.

5 Conclusion

Although the concept of the Traveling Salesman Problem came from very real-world problems, eventually a whole sub-area of mathematics became dedicated to it, and through extensive mathematical research, it bore its own applications. Some of apparent applications are logistics, or transporting people, vehicles (from cars to buses to delivery systems), or things, in the best way possible. In genome sequencing, distance functions for marker locations are aimed at creating TSP solutions that are approximate to the actual ordering of a genome. In scan chains, TSP is used to determine the ordering of scan points, to make the chain as short as possible. Other applications include, but are not limited to: drilling problems, aiming telescopes and X-rays, data clustering, minimizing wall paper waste, picking items in a warehouse, and studying problems in evolutionary change.[1]

References

[1] David Applegate. *The Traveling Salesman Problem: A Computational Study* Princeton University Press 2006.

[2] William Cook. *In Pursuit of the Traveling Salesman* Princeton University Press. 2012.

[3] Jack Edmonds. "Maximum matching and a polyhedron with 0,1-vertices". Journal of Research of the National Bureau of Standards. 1965.

[4] Frank Giordano, William Fox, Steven Horton, Maurice Weir. *A First Course in Mathematical Modeling*. Brooks/Cole, Cengage Learning. 2009.

[5] Keld Helsgaun. "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic". *European Journal of Operational Research* 2000.

[6] C. Miller, A. Tucker, R. Zemlin "Integer Programming Formulation of Traveling Salesman Problems". *Journal of the ACM (JACM)* 1960.

[7] Christian Nilsson. "Heuristics for the Traveling Salesman Problem". Linköping University.

[8] Kenneth Rosen. *Discrete Mathematics and Its Applications* McGraw-Hill 2011.