The Travelling Salesman Problem

Solution, Application and Variants

Darian Fry, Brandon Lee, Erik Harris-Uldall, Harmehar Shergill, Roman Pavlyutin

Abstract

We discuss the NP-Complete version of the Travelling Salesman Problem (TSP). We define what the TSP is, give some background and history and prove why it is NP-Complete and observe when it is NP-Complete, NP-Hard and Co-NP. We cover the methods to solve our version of the TSP and show how our methods for each scenario can be solved in polynomial time. We examine special cases of the TSP and several real life applications of this problem.

Introduction

The definition of the Travelling Salesman Problem (TSP) is as follows: Given a list of n cities and the distances between each pair of cities, what is the shortest possible route that the salesman can take that visits each city once and returns to the starting position? Originating back in the 1800s formulated by the Irish mathematician W.R. Hamilton and the British Mathematician Thomas Kirkman.

The problem ironically originated in W.R. Hamilton's game called the Icosian Game ("The Travelling Salesman Problem", 2019) where you must find a Hamiltonian cycle on a dodecahedron, in other words find a path that visits each vertex once and returns to the original starting place. The motivation of this game was to solve the problem of symmetries of icosahedrons (a platonic solid), where Hamilton invented Icosian calculus which is used to compute symmetries. To give some context, the symmetry of two shapes is when you can rotate, flip or change its position of one of them in some way and it will look exactly like the other. Coincidentally it turns out the solution of the Icosian game contains twenty (in Greek Icosa) edges. The TSP became established as a mathematical problem in the 1930s by the Mathematician Merrill M. Flood who was attempting to solve a school bus routing problem

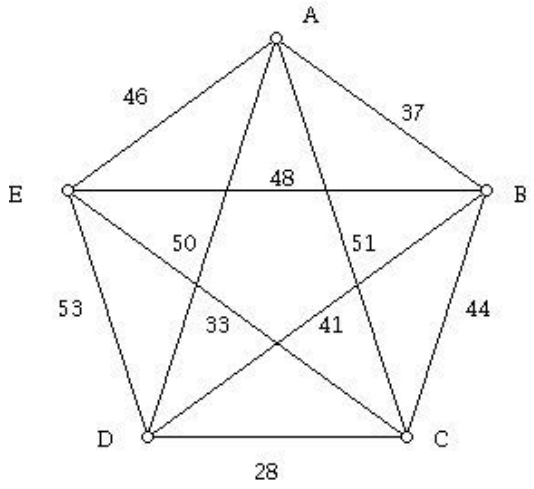
i.e. find the shortest bus route given a set of routes to choose from. In the 1950s and 1960s is when the TSP became a very popular problem, having several scientific articles written about it in Europe and the USA specifically how to find an optimal (minimal) solution for the TSP. Among some of those articles came the method of Branch and Bound, developed by Dantzig, Fulkerson, and Johnson ("The Travelling Salesman Problem", 2019). A few decades later is when a new approach to solve the TSP was made by producing a solution whose length is proven to be a lower bound on the TSP, an example of this kind of method is finding the minimum spanning tree of the vertices of a graph and the lower bound of the solution becomes two times the minimum spanning tree solution. More progress was made in solving the TSP accurately in the 70s and 80s and some of these we will describe in more detail later on.

In Mathematics there are several famous problems out there that are quite similar to that of the TSP. The most obvious one is the Hamiltonian path/cycle problem where you try to identify if there is a path or a cycle in a graph G = (V, E) that visits each vertex at most once and in the cycle version, return back to the starting point. The Hamiltonian path/cycle problem actually plays a very important role in solving the TSP, since we need to be able to find a cycle where all of the nodes or cities get visited, the importance and emphasis for this concept will be observed later on in this paper. Another problem that also plays a role in solving the TSP or at least give us an approximation of the TSP is given n nodes, find the minimum spanning tree of these nodes, in other words find a tree (a graph with no cycles) that connects all n nodes and has the least weight. The minimum spanning tree is used as mentioned earlier as an approximation and a lower bound of the TSP. While it is not the most efficient way to solve the TSP, we will soon discuss and observe how the minimum spanning tree can be used to ensure we are on the right track when solving the TSP. Another famous

problem originating in 1736 that also relates to the TSP is the Seven Bridges of Königsberg by Leonard Euler where you must find a walk through the city that must cross each bridge at most once. This type of walk is also known as an Eulerian path and this concept will also be used and seen in later parts in the paper. As we can see here the concepts mentioned here all play a role to solve or give an approximation of the TSP and we will rely heavily and use these concepts to help solve or give an approximation to the TSP.

The TSP is actually one of the most common and used concepts in real life and is used in several applications to solve several types of problems. The simplest type of problem it is used for is something like logistics or simple planning. Essentially think of any application that involves routes and finding the best route then the TSP will be essential for these types of problems. The applications we are referring to here are apps such as Google maps, arranging bus routes to pick up children in the school district and many more. Now while the TSP is a relatively straightforward type of problem and is used to solve straightforward problems like route planning, the TSP can be used in more interesting and complex type of problems. A classic and well known example is the scheduling of a machine to drill holes in a circuit board. The holes to be drilled are the vertices (cities) and the cost or time to travel for the drill to get to the next hole is the weighted edge. The TSP is used here to optimize the cost when travel time plays a significant role in the manufacturing process. Other interesting applications of the TSP, include DNA sequences, astronomy where we want to minimize the time spent moving the telescope between celestial bodies and is involved in many Computer Science problems and algorithms. More applications will be discussed in greater detail later in this paper.

So now that we have given some history and some brief idea of what the applications of the well known TSP could be, we will give a small example of how to use the TSP on the graph below. Let G = (V, E) be a weighted graph with |V| vertices (cities) and every pair of vertices is adjacent for v and v'. Each edge e in E incident to both v and v' have weight w > 0 and are Graph G = (V, E) is a complete graph where each vertex is connected by an edge. The goal is to find a Hamiltonian cycle in this graph whose sum of weights w is the most minimal. An image of this graph is displayed below (Goncalves, 2016).



In this paper we will explore the methods to solve the TSP, its NP problems and special cases of the TSP. In particular we will find a way to solve the NP-Complete version of the TSP which is the same as the original definition of the TSP except is has the following constraint: Is it possible to complete the path with cost less than C (or distance less than C)? We will then continue on to prove that our version of the TSP is NP-complete using the Hamiltonian Cycle as the polynomial reduction to the TSP. While this is not the only polynomial reduction to the TSP it is the best one to use and makes more logical sense to do so. The TSP on its own is NP-Hard because the difficulty increases exponentially to solve as

you increase the number of vertices (cities) so we will then discuss and define clearly when the TSP is NP-Complete, NP-Hard and Co-NP complete since the distinction between NP-Complete, NP-Hard, and Co-NP complete cases for the TSP sometimes gets blurred or becomes unclear. We then turn to solve our TSP focusing mainly on solving the NP-Complete and NP-Hard cases. The algorithms used for the NP-Hard cases will be the Brute Force algorithm, and the Nearest Neighbor Algorithm. We will go more in depth for our algorithm to solve the NP-Complete case, the Christofides Algorithm.

Our paper then analyzes to see which algorithm is the most efficient and why one may take less time than the other to determine a solution to the TSP. While we discuss solving the TSP with each algorithm we will examine the importance of the triangle inequality i.e. For three cities N, M, and K: $\text{dist}(N \rightarrow M) \leq \text{dist}(N \rightarrow K) + \text{dist}(K \rightarrow M)$, the direct route must be shorter than indirect route. We will discuss why the triangle inequality is important for polynomial reduction and what happens if the triangle inequality does not hold for a certain graph. It is then important to discuss the polynomial reductions of the TSP as we will be using these for some of the algorithms described since they determine if the solution to the TSP can be solved in polynomial time. We lastly go over some unique or special cases of the TSP where some of the properties of the TSP mentioned above may not apply to these cases. Some special cases that we are including here are the Bottleneck TSP and the Multiple Visits TSP. The goal of this paper is to give and show a better understanding of the TSP as a whole and then focus on our created NP-complete version of the TSP and describe and formulate a method to solve our NP-Complete problem and show why that this is indeed the most efficient way to solve the NP-Complete problem of the TSP.

Travelling Salesman Problem Is NP-Complete

Given a set of cities, or vertices on a graph, we wish to find a path through all the cities/vertices that is as short as possible. If this is considered a weighted graph, then we can say we wish to find a path through all the vertices with the smallest cost possible. The decision problem of the Travelling Salesman Problem is described as, given a set of cities or vertices, is there a path that visits each city or vertex once and has a length or weight less than some given value. This decision version of the problem is NP-Complete. To prove this is NP-Complete we first must show that this problem is in NP, and then that it is NP-Hard.

To see that the Travelling Salesman Problem is in NP, we must show that it can be checked in polynomial time. To see this, consider a set of vertices or cities V, a maximum cost or length K, and a proposed solution path S. If this path is a solution, the path must connect every city once and have a weight less than K. Checking this requires checking each vertex is connected in S, and then checking the weight of the path is less than or equal to our given K. Checking each vertex takes |V| and checking the weight of the path takes |S|, the number of edges in our path. Each of these takes linear time, so we can see that checking the solution to the decision version of the Travelling Salesman Problem takes polynomial time. Thus, the decision version of the Travelling Salesman Problem is in NP.

To show that this is NP-Hard, we must show that every problem within NP can be reduced to the Travelling Salesman Problem in polynomial time. To do this, we consider the Hamiltonian Cycle problem. In the Hamiltonian Cycle problem, we are given a graph G=(V,E) with V vertices and E edges, and the goal is to find a path on the given edges that visits each vertex exactly once. The decision version of this problem is to answer whether or

not there exists such a path on a given graph. This is a well known NP-Complete problem, so we know all problems in NP can be reduced to the HC problem in polynomial time. Therefore, if we can reduce the HC problem to the Travelling Salesman Problem in polynomial time, then all problems in NP can be reduced to the Travelling Salesman Problem in two polynomial functions, which is still polynomial and thus the Travelling Salesman Problem is NP-Hard. To begin, with a given HC problem with a given graph G=(V,E), we can add weights to each edge on the graph. To reduce this to the Travelling Salesman Problem, we first assign each edge the weight 1. Then we ask if there exists a path connecting all vertices with a maximum path weight k, and let our maximum path weight constraint k=|V|. This clearly takes polynomial time, now we show that if there exists a solution to the HC problem, there is a solution to the Travelling Salesman Problem and vice versa.

Observe that if there exists a solution to the HC problem, then the weight of the path will be k, since each vertex is visited exactly once the path will only have |V| vertices and thus have weight |V| = k. Observing that the solution to the HC problem visits each vertex exactly once, we see that a solution to the HC problem satisfies all the requirements for a solution to the Travelling Salesman Problem. Therefore, if there exists a solution to the HC problem, there exists a solution to the Travelling Salesman Problem that was reduced from the HC problem. Now, if there exists a solution to the Travelling Salesman problem that implies that each vertex is visited with a path that has weight less than or equal to k. Since all the edge weights are 1, if the path has a weight less than or equal to k then it must have only k edges. Since all vertices are visited, and there are only k = |V| edges, then each vertex can be visited only once. Therefore, if there is a solution to the Travelling Salesman Problem there must be a solution to the HC problem. Now, if there is no solution to the HC problem,

this means there is no path that connects all vertices exactly once. For contradiction, assume there exists a solution to the Travelling Salesman Problem. This means that there is a path that connects all vertices and has a weight less than or equal to k. But since k = |V| and each edge weight is 1, if there is a solution it must visit each vertex exactly once, a contradiction to the assumption that there is no solution to the HC problem. This shows that the Travelling Salesman Problem can be reduced in polynomial time from the HC problem. Therefore, the Travelling Salesman Problem is NP-Hard.

Since the decision version of the Travelling Salesman Problem is in NP, and is NP-Hard, by definition it is NP-Complete.

The co-NP complement of this problem is the Traveling Salesman Minimum route decision problem. This version asks if, given a path that connects all cities or vertices, if this is the minimal path. This can be verified to be false in polynomial time. Any pure optimization problem to the Traveling Salesman Problem, such as finding the minimum path for a given graph, is NP-hard as it cannot be checked with any polynomial time algorithm.

Algorithms

"The difficulty in solving a combinatorial optimization problem such as the TSP lies not in discovering a single solution, but rather quickly verifying that a given solution is the optimal solution" (Matthew et al.). This problem is associated with the fact that in order to verify that a given solution is indeed optimized, we first need to calculate all other possible solutions and then compare them with each other. This approach is called a Brute-Force method, and it is used to find an exact solution. While this method is very straight-forward, it

quickly becomes very inefficient as the number of cities in the graph grows. “For example, the total number of possible paths for 7 cities is just over 5,000, for 10 cities it is over 3.6 million, and for 13 cities it is over 6 billion” (Matthew et al.). This approach will be discussed in greater detail later in this paper.

Since it is almost impossible to find an exact solution for a graph with more than 20 cities, mathematicians usually are forced to use various approximation algorithms. “In an approximation algorithm, we cannot guarantee that the solution is the optimal one, but we can guarantee that it falls within a certain proportion of the optimal solution” (Matthew et al.). Compared to Brute-Force method, approximating the solution is a rather quick process. In this paper we will focus on two approximation algorithms: the Nearest Neighbour and the Christofide’s approaches.

Triangle Inequality

When talking about approximation algorithms, there is one very important assumption, that we need to make: for a given graph G = (V, E), the cost function d must satisfy the *Triangle Inequality*.

“If for the set of vertices N, M, K $\in$ G, it is true that d (N, K) $\leq$ d (N, M) + d (M, K), where d is the cost function (in this case, the distance between the two cities), we say that d satisfies the Triangle Inequality.”

In other words, it is cheaper (in terms of ticket costs) or it is faster (in terms of distance) to travel from city N directly to city K, rather than taking an indirect route through city M. This assumption enforces the completeness of a graph G, thus allowing us to use Minimal Spanning Trees and other 2-approximation algorithms to approximate the optimal

travelling salesman tour. In fact, there is no heuristic for TSP without the Triangle Inequality that performs well. This is due to the theorem that TSP cannot be approximated within any polynomial time computable function unless P=NP. Therefore, "since we cannot even approximate the general TSP problem, we consider more tractable variants: Metric TSP (where cost function satisfies the Triangle Inequality) or TSP-R (with repetition of vertices is allowed)" (Im, 2009). Although, most applications that we consider are Metric TSPs, notice, however, that there are real-life cases that violate the Triangle Inequality. For example, when we deal with plane ticket fares, it is sometimes cheaper to take a connecting flight through an intermediate city. In this paper, we are not going to discuss such cases, since it limits the use of approximation algorithms.

Brute-Force Method

As mentioned above, one way to find an exact solution is to determine every possible route that a salesman can take and compare them to each other. In order to simplify the process, notice, that for a map of cities A, B, C, D, where A is a starting location, the two tours A>B>C>D>A and A>D>C>B>A are considered to be the same, since the total weight of these two tours are equal. Therefore, for n possible cities, where every city is connected by a path to all other cities, there are $(n-1)!/2$ possible cycles.

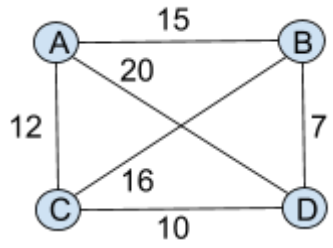


Fig. 1

Consider the graph in Fig. 1, we can use Brute-Force method to find the optimal solution suppose we want to start our tour in city A, then there are (4-1)!/2 = 3!/2 = 6/2 = 3 cycles:

1. A>B>C>D>A = 15+16+10+20 = 61 = A>D>C>B>A
2. A>B>D>C>A = 15+7+10+12 = 44 = A>C>D>B>A
3. A>C>B>D>A = 12+16+7+20 = 55 = A>D>B>C>A

Thus, we can conclude that the route A>B>D>C>A (or A>C>D>B>A) is the optimal solution for this traveling salesman problem. This naive approach is very straightforward and easy, and can be used for a relatively small graph. But major difficulties arise when the number of cities grows: “the problem is with the rapid growth of the factorial function which is involved in the counting of the TSP tours. For even a modest number of cities the best current supercomputers can not enumerate all of the tours” (Malkevitch). The time complexity of this method is O(|V|!), which means Brute-Force cannot be used to solve a problem in polynomial time.

Nearest Neighbour Approach

"Nearest neighbour, was the first greedy algorithm which gave a solution for the travelling salesman problem. The algorithm was introduced by J.G. Skellam and it was continued by the F.C. Evans and P.J Clark" (Arora et al., 2016). While previous method guaranties to find an exact solution, the Nearest Neighbour(NN) approximation algorithm aims to find a significantly good solution in a very short amount of time. The idea behind this approach is very simple: starting in a random city, choose the closest city on a map that has not been visited before, and once you have visited all the cities, return to the starting location. It is easy to notice that the NN algorithm makes a decision on a local level, rather than considering the global picture. "The question is to see whether or not the locally optimal choices lead to a solution which is globally optimal" (Malkevitch).

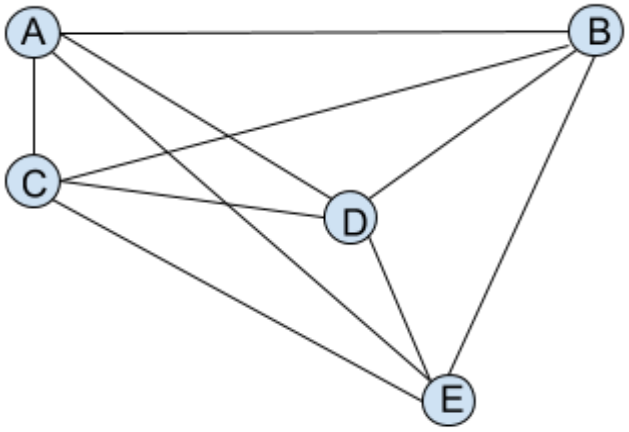


Fig.2

Let's consider the graph in Fig. 2, where we have a map with cities A, B, C, D, and E and ticket prices to travel between the cities. Suppose we start at city A, then according to NN algorithm, we travel to C, since it is the closest city in the map. Repeating this step until we visit all cities, and we end up with a following tour: A>C>D>E>B>A. The runtime of the

NN algorithm is $O(|V|^2)$, which is significantly faster than the previous method and is applied in polynomial time. The tour we have found might not be optimal, but given how fast it is to solve the problem using this method, the following question arises: "Even if NN does not yield an optimal answer to the TSP, does it yield a sufficiently good solution that we can settle for this, given how fast we get a solution?" (Malkevitch). In order to answer this question, we need to take a look at how well this algorithm does on average, and how far off can it be in the worst-case scenarios.

Mathematicians use so called domination number which indicates the performance of various heuristic algorithms. The bigger the domination number, the better heuristic is. Nearest Neighbour for an asymmetric Travelling Salesman problem has a domination number n-1 at most, and n/2 at least, where n is the number of vertices in a graph and $n \geq 4$. "If we evaluate the greedy algorithm and the nearest neighbor algorithm for the TSP, we find that they give good results for the Euclidean TSP but they both give poor results for the asymmetric and the symmetric TSP" (Kounalaki & Kapeloni, 2002). Indeed, for any number of cities, it is almost guaranteed that there will be some city layout that will force NN to produce the worst-case solution. Also, the approximated solution may significantly vary based on the starting point, producing either a worst possible or almost exact optimal solution. The following is also true, for an optimized dataset NN was shown capable of producing an exact solution, however, on average, this algorithm generates a solution that is within 25% range from the optimal solution. Thus, even though, given how fast this method can be applied, it has some fundamental drawbacks to it. "As the algorithm [NN] is a greedy algorithm it misses out some of the shorter routes. These shorter routes can be detected by

human insight easily. So, the nearest neighbour algorithm does not give the feasible solution" (Arora et al., 2016).

Christofide's Algorithm Approach

We will use Christofide's Algorithm to approximate the Travelling Salesman Problem. Christofide's Algorithm is a slightly more complex algorithm that applies multiple different algorithms to the Travelling Salesman Problem to achieve the best approximation ratio of the general metric space problem. It can guarantee a solution with approximation ratio 3/2(Christofides, 1976; Goodrich, n.d.) in the worst case, which is significantly closer than any other algorithm. Like other approximation algorithms, Christofide's Algorithm does require that the graph satisfies the triangle inequality, in other words it must be a metric-TSP problem. The description of the method follows.

First, for our given metric-TSP graph G=(V,E), we construct a minimum spanning tree of G. There are many simple algorithms to find the minimum spanning tree of a graph, we shall use Kruskal's algorithm. This algorithm simply starts with the lowest cost edge on a graph and adds it, then continues to the next lowest cost edge and so on, skipping any edge that would form a cycle on the tree and stopping once there are |V|-1 edges on our tree. This algorithm will take O(|E|log|V|). Let T be the minimum spanning tree we've formed. From T we will take all the vertices that have odd degree, finding these vertices will take O(|V|). Let this set of vertices be W. From the handshake lemma, it is known that W will have an even number of vertices.

Now, we find the minimum cost by using perfect matching, let us call this M, out of W. Since we know W is even, a perfect matching always exists on W. There are a few

algorithms that solve for this perfect matching, such as Blossom Algorithm, which have $O(|V|^3)$. This is the most time intensive part of the algorithm and therefore where the overall growth rate derives from.

Next we combine M and T to make a new graph G'. On this new graph we construct an Eulerian path on G', a path that visits each edge exactly once. Creating this path, let it be called C, will take $O(|E|)$ with E being the edges on G'. The final step is to convert C to a tour that solves the Travelling Salesman Problem by traversing the path C and removing, or shortcutting, any edges that visit a previously visited vertex, let this new tour be C'. C' is the approximated solution to the Travelling Salesman Problem.

To show this is at worst a 3/2 approximation, let S be the true optimal path on our graph. We know that a minimal spanning tree is a lower bound for the Travelling Salesman Problem optimal solution, so for our minimal spanning tree T we know

$$C(T) \leq C(S)$$

Now let R be a solution to the Travelling Salesman Problem on a graph made up of the vertices W, the odd degree vertices of the minimal spanning tree. Since W is a subset of the vertices in G, and by assumption the triangle inequality holds on G,

$$C(R) \leq C(S)$$

This only holds if the triangle inequality holds, otherwise adding more vertices may reduce the overall cost of the path. Now we note that the cost of R is the sum of the cost of the even numbered edges and odd numbered edges. This implies that one of the costs of these sets of edges is at most C(R)/2. We also observe that the set of odd numbered edges and the

set of even numbered edges are both perfect matchings. Since our minimum cost perfect matching M is a subset of these edges, and is a minimum cost perfect matching, the cost of M will be at most the cost of the smallest of these two sets. That is

$$C(M) \leq C(R)/2$$

Now we put the cost of our minimum spanning tree and the cost of our perfect matching together and see:

$$C(T)+C(M) \leq C(S)+C(R)/2$$

Since $C(R) \leq C(S)$ we can expand this to get:

$$C(T)+C(M) \leq C(S)+C(R)/2 \leq 3C(S)/2$$

We observe that our original Eulerian path C had a cost C(T)+C(M), and since the triangle inequality holds, when we created C' by removing any edges that visited a vertex again we made sure the cost of C' was less or equal to the cost of C. That is to say

$$C(C') \leq C(C) \leq C(T)+C(M) \leq C(S)+C(R)/2 \leq 3C(S)/2$$

Simplified we have

$$C(C') \leq 3C(S)/2$$

This completes the proof that the Christofide's algorithm has a worst case approximation ratio of 3/2 so long as the graph satisfies the triangle inequality.

Solving TSP with Christofide's Algorithm

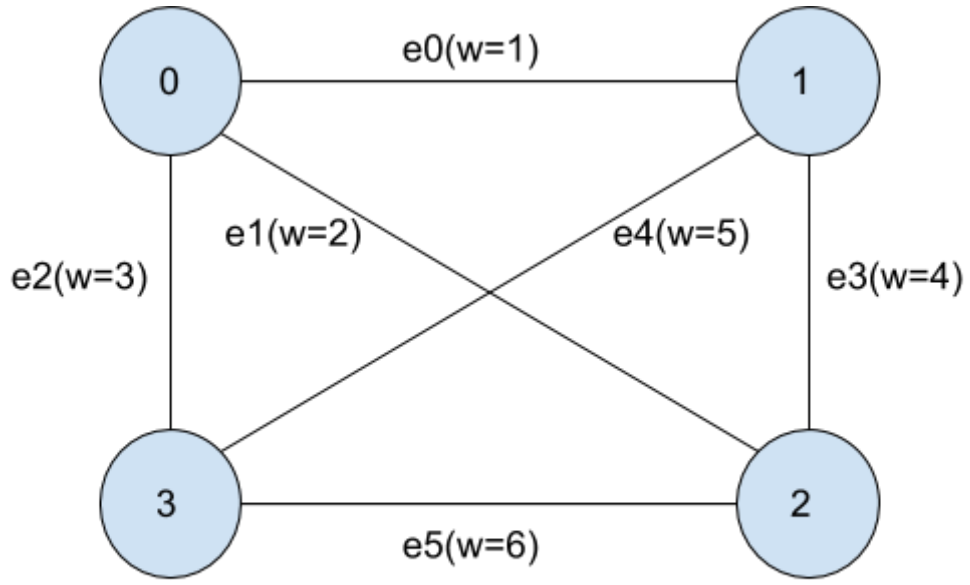


Fig. 3

We consider Fig.3 graph above to solve TSP by implementing the Christofide's algorithm using programming with C++ (see A1-A8 in Appendix for code snippets).

First, we construct a completed graph with 4 vertices and 6 edges, where (|V|*|V-1|)/2 computes the number of edges for given number of vertices, |V|. (see A1, lines 3-7). Then, we manually add the weights to all of the edges of the completed graph that we have constructed (see A1, lines 9-15). As emphasized previously, we must ensure that the constructed graph, our input, satisfies the triangle inequality so we check if the graph after allocating weights on every edge satisfies the triangle inequality (see A1, lines 17-18). To check for satisfiability of the triangle inequality, we find all possible triangles in the graph by finding all possible triangular cycles formed by three vertices (see A3). Then, we check if the weight of one edge is less than or equal to the sum of the weights of the other two edges.

After the verification of the triangle inequality (or else, the program will terminate), we build the minimum spanning tree of the graph using Kruskal's algorithm (see A4).

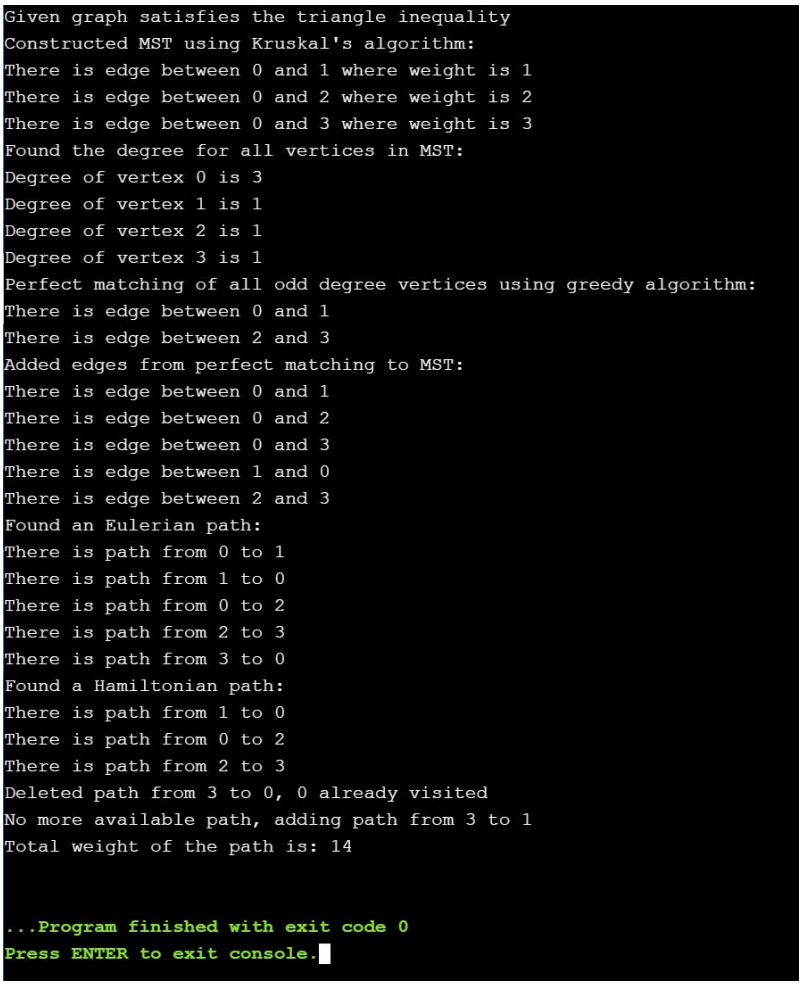


Fig.4 Console print statements after running the main function (see A1 for input)

As we can see above from Fig 4. after compiling the C++ file, we can see that we are able to find a minimum spanning tree of our input. We then go ahead and compute the degree for every vertex in the MST by counting the number of edges a vertex has, and find the vertices with odd degrees. With these vertices, we apply our perfect matching algorithm, which matches each pair of vertices with a single edge. Because we have implemented the perfect matching algorithm using the greedy algorithm, we cannot ensure that the resulting

perfect matching is optimal, meaning that the total weight of these edges may not be minimal; however, the greedy algorithm gives a good approximation of the optimal perfect matching.

As explained before, we search for the Eulerian path from a new graph created by adding edges from perfect matching to the MST. For the implementation of finding the Eulerian path, we have used the Fleury's algorithm (see A6). The result of the path can be seen in Fig.4 as well; we can observe that there is a double edge between vertices 0 and 1, for the graph that we used, so we remove one of these edges to satisfy the Eulerian path. With a new graph formed by the Eulerian path, we traverse through the given path with random initialization and remove any duplicating vertices by checking if the next vertex has already been visited. If the next vertex has been visited, we remove the edge that connects the current vertex to the next; we look for a different path to vertex that has not been visited. And before the termination of our program, we check if we have no other vertices to visit. After verifying that there are no more vertices to visit, we create a path from our current vertex to the initial vertex, and thus creating a Hamiltonian cycle (see A7). Finally, we calculate the total weight of the Hamiltonian cycle by traversal. When we run the main function with the graph seen in Fig.4, our program computed an optimal path with a total weight of 14.

Special Cases & Variants

Being a rich optimization problem, the TSP can pertain to many practical applications in real life which will be discussed in the next section. However, due to various constraints, it’s not always possible to apply this problem to all of its real-world applications. Therefore, the following section proposes several simple variants of the TSP to manage these application-specific constraints.

The Maximum Travelling Salesman Problem, also known informally as the "taxicab rip-off problem", is a specific case where the objective is to find a tour *G* where the total cost of edges in the tour is maximum. Thus, unlike the TSP, in this problem, the salesman visits each city exactly once and returns to the starting city with maximum possible distance travelled. This problem can be stated as: : "Given an *n x n* real matrix $c = (c_{ij})$, called a weight matrix, find a Hamiltonian cycle $i_1 \mapsto i_2 \mapsto \ldots \mapsto i_n \mapsto i_1$, for which the maximum value of $c_{i_1 i_2} + c_{i_2 i_3} + \ldots + c_{i_{n-1} i_n} + c_{i_n i_1}$ is attained. Here $(i_1, \ldots, i_n)$ is a permutation of the set $\{1, \ldots, n\}$" (Barvinok, Gimadi & Serdyukov, 2007). The MAX TSP can be solved by reformulating the problem as a TSP by replacing each edge cost with its additive inverse. The bottleneck TSP (BTSP), is another variant of the TSP where the objective is to find a minimum cost tour *G* such that the largest cost of edges in *G* is as minimal as possible. This problem can be stated as: "Let *G* = (*N, E*) be a directed or undirected graph on node set *N* = {1, 2,...,*n*} and let F be the family of all Hamiltonian cycles in *G*. For each edge e $\in E$, a cost c_e is prescribed and for any $H \in F$, let $c_{max}(H) = max\{c_e : e \in H\}$. Then the BTSP is to find a Hamiltonian cycle $H \in F$ such that $c_{max}(H)$ is as small as possible" (Gutin & Punnen, 2007, p. 697). The bottleneck TSP is a variation of the TSP, that only differs from the original problem in the objective function. It can be solved by reconstructing the problem as a TSP but with exponentially large edge costs, such that an optimal solution to the TSP gives an optimal solution to BTSP.

Another simple variant of TSP is the TSPM, also known as high multiplicity TSP, which is a generalization of the TSP where the objective is to find a route for a salesman such that each node is visited at least once, and the total distance travelled is minimized. In this problem, the travel costs can be asymmetric and visiting a city twice in a row can result in a

non-zero cost (Berger et al., 2018). To solve this problem, one can reformulate it into a TSP by substituting the edge costs in the original problem with the shortest distances of paths in *G* (Gutin et al, 2007). Another compelling variation of the TSP is the Messenger Problem. Also known as the wandering salesman problem, this problem aims to find a least-cost Hamiltonian path in *G* that starts at a specified node *u* and ends at another specified node *v*. According to Gutin and Punnen (2007), in order to solve this problem as a TSP, a cost of -*M* for the edge (*u,v*) can be chosen, where M is a large number (p. 7). He further states that this can also be achieved in the case where no nodes are specified by creating a new node in *G* and connecting it with all other nodes by edges of cost -*M* (Gutin et al, 2007). Therefore, the reconstruction of this problem into a TSP entails the incorporation of a large negative cost for the edge (*u,v*). As a result, an optimal solution to the Messenger Problem can be retrieved from the optimal solution to the TSP on the aforementioned modified graph.

A Clustered TSP, another common variant of the TSP, includes the node set of *G* partitioned into clusters, where the objective is to find a least-cost tour in *G* given that a group of cities in the same cluster is visited consecutively in an optimal, unspecified order before moving on to another cluster. According to Ahmed (2014), in the CTSP, "the problem is to simultaneously determine the optimal cluster order as well as the routing within and between clusters" (p. 1). This problem can be converted to a TSP by adding *M*, a large cost, to the cost of each inter-cluster edge (Gutin et al, 2007). The Generalized TSP, another variant of TSP, can be seen as a continuation of the CTSP mentioned above. Given the partition of nodes into clusters $V_1, V_2, \ldots, V_k$,the goal of this problem is to identify the shortest cycle in *G* that passes through exactly one node in each cluster (Fischetti et al, 2002). The GTSP can be reduced to a TSP if $|V_i| = 1 \quad \forall \quad i$, and thus can be solved as TSP by

modifying the cost matrix (Gutin et al, 2007). Lastly, another widely used variant of the TSP is the m-salesmen TSP which, as the name suggests, involves multiple salesmen for the solution. This problem can be defined as: "Given a set of nodes, let there be m-salesmen located at a single depot node. The remaining nodes that are to be visited are called intermediate nodes. Then, the mTSP consists of finding tours for all m-salesmen, who all start and end at the depot, such that each intermediate node is visited exactly once and the total cost of visiting all nodes is minimized" (Bektas, 2006). As a natural generalization of the TSP, the mTSP captures many more real-world problems and additionally, all solution approaches for the mTSP are valid for the TSP.

Applications

Applications of the TSP can be noted in various fields including mathematics, computer science, engineering, operations research, genetics etc. and go further than its relevance to salesmen routing problems. The most common and well researched application of this problem is that of machine sequencing and scheduling. According to Gutin (2007), an example of this scheduling can be described as: "There are n jobs $\{1, 2,... , n\}$ to be processed sequentially on a machine. Let c_{ij} be the set-up cost required for processing job j immediately after job i. When all the jobs are processed, the machine is reset to its initial state at a cost of c_{j1}, where j is the last job processed" (p. 9). Evidently, the resulting scheduling problem is that of finding which jobs get processed so that the overall set up cost in minimized. Another popular application of the TSP is the vehicle routing problem. According to the VRP, there are a certain number of mailboxes in a city, that need to be unloaded within a certain period of time. The goal here is to collect mail from the boxes in the shortest amount of time using the least number of vehicles (Matai et al, 2010). There are

also many real world scenarios of the VRP where time and capacity constraints are combined.

Despite transportation problems being the most natural setting for the TSP, it also has many interesting applications in prominent areas like technology. An example of this would be computer wiring where a special case of connecting components on a computer board is considered with a requirement that no more than two wires are attached to a pin (Lenstra & Kan, 1975). The aim here is to minimize the total wire length to improve wiring in the system. Another classic application of the TSP pertains to the bottleneck TSP mentioned in the previous section. According to Behzad (2002), in circuit manufacturing, scheduling a route for the drill machine that drills holes in the circuit is a variation of the problem where the holes act as 'nodes' and the objective is to minimize the time taken to retool the drill machine. Lastly, a rather intriguing application of the TSP is DNA sequencing where a certain number of DNA substrings which are the 'nodes' are embedded in one universal string and the problem aims to minimize the length of this universal string (Caserta, 2014).

Bibliography

Ahmed Z. A., (2014) The Ordered Clustered Travelling Salesman Problem: A Hybrid Genetic Algorithm, *The Scientific World Journal*, Article ID 258207, 13 pages, 2014. https://doi.org/10.1155/2014/258207

Arora, K., Agarwal, S., Tanwar, R. (2016). *Solving TSP using Genetic Algorithm and Nearest Neighbour Algorithm and their Comparison*. Retrieved from International Journal of Scientific & Engineering Research, Volume 7, Issue 1, 1014 ISSN 2229-5518.

https://pdfs.semanticscholar.org/9fc2/8fac2603cfda11da21033a58bbb5c7b75e09.pdf

Barvinok A., Gimadi E.K., Serdyukov A.I. (2007) The Maximum TSP. In: Gutin G., Punnen A.P. (eds) *The Traveling Salesman Problem and Its Variations. Combinatorial Optimization*, vol 12. Springer, Boston, MA

Behzad, A., Modarres, M., (2002) New Efficient Transformation of the Generalized Traveling Salesman Problem into Traveling Salesman Problem", *Proceedings of the 15th International Conference of Systems Engineering* (Las Vegas).

Bektas, T. (2006). The multiple traveling salesman problem: an overview of formulations and solution procedures. *OMEGA: The International Journal of Management Science* 34(3), 209-219.

Berger, A., Kozma, L., Mnich, M., & Vincze, R. (2018). A time- and space-optimal algorithm

for the many-visits TSP. *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1770–1782. doi: 10.1137/1.9781611975482.106

Caserta, M., & Voß, S. (2014). A hybrid algorithm for the DNA sequencing problem. *Discrete Applied Mathematics*, *163*, 87–99. doi: 10.1016/j.dam.2012.08.025

Christofides, N. (1976). *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem* (No. RR-388). Retrieved from CARNEGIE-MELLON UNIV PITTSBURGH PA MANAGEMENT SCIENCES RESEARCH GROUP website: https://apps.dtic.mil/docs/citations/ADA025602

Fischetti M., Salazar-Gonzalez JJ., Toth P. (2007) The Generalized Traveling Salesman and Orienteering Problems. In: Gutin G., Punnen A.P. (eds) *The Traveling Salesman Problem and Its Variations. Combinatorial Optimization*, vol 12. Springer, Boston, MA

Goodrich, M. T. (2015). *Algorithm Design and Applications*. Retrieved from http://canvas.projekti.info/ebooks/Algorithm%20Design%20and%20Applications%5BA4%5D.pdf

Goncalves, E. (2016). *Algorithm for classification of vertices in a graph based on their weight.* Retrieved from https://stackoverflow.com/questions/34600199/algorithm-for-classification-of-vertices-in-a-graph-based-on-their-weight

Gutin, G., Punnen, A. P., & SpringerLink ebooks - Mathematics and Statistics. (2007;2002;). *The traveling salesman problem and its variations*. New York: Springer

Im, S. (2009). *Traveling Salesman Problem*. Retrieved from

https://courses.engr.illinois.edu/cs598csc/sp2011/Lectures/lecture_2.pdf

Kounalaki, E. & Kapeloni, K. (2002). *Traveling Salesman Problem*. Retrieved from https://www.csd.uoc.gr/~hy583/papers/ch11.pdf

Lenstra, J. K., & Kan, A. H. G. R. (1975). Some Simple Applications of the Travelling Salesman Problem. *Operational Research Quarterly (1970-1977)*, *26*(4), 717. doi: 10.2307/3008306

Malkevitch, J. (2019). *Traveling Salesman Problem*. Retrieved from American Mathematical Society http://www.ams.org/publicoutreach/feature-column/fcarc-tsp

Matai, R., Singh, S., & Lal, M. (2010). Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. *Traveling Salesman Problem, Theory and Applications*. doi: 10.5772/12909

Matthew, R., Zhao, K., Cherukupallu, D. & Pusich, K. *Traveling Salesman Algorithms: From Naive to Cristofide*. Retrieved from https://cse442-17f.github.io/Traveling-Salesman-Algorithms/?fbclid=IwAR0AWxx4xigPHssp93AeO5ZEo0gAhizHEhjCVyzUSiU_B2rNs1k2pwoA_bk

The Travelling Salesman Problem. (2019) Retrieved from https://simple.wikipedia.org/wiki/Travelling_salesman_problem

Appendix

```
int main()
{
    int V = 4;
    int E = V * (V - 1) / 2;

    Graph *graph = createGraph(V, E);
    completeGraph(graph);

    // Add weights to all edges
    graph->edge[0].weight = 1;
    graph->edge[1].weight = 2;
    graph->edge[2].weight = 3;
    graph->edge[3].weight = 4;
    graph->edge[4].weight = 5;
    graph->edge[5].weight = 6;

    // Checks if the given graph satisfies the triangle inequality
    triangleInequality(graph);

    // Build MST using Kruskal's algorithm
    Graph *tree = buildMST(graph);

    // Find the degree for all vertices in MST
    findDegree(tree, V, tree->V);

    // Perfect matching of all odd degree vertices using the greedy algorithm
    Graph *match = perfectMatching(tree);

    // Add edges from perfect matching to MST
    Graph *newGraph = addEdges(match);

    // Finds the Eulerian path of given graph
    Graph *eulerianPath = findEPath(newGraph);

    // Finds the Hamiltonian path of given Eulieran path
    Graph *hamiltonianPath = findHPath(eulerianPath);

    // Calculates the total weight of given Hamiltonian path
    getWeight(hamiltonianPath);
}
```

A1. Code snippet of the main function (input)

```
Graph *createGraph(int V, int E)
{
    Graph *graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    graph->dir = new int *[V];

    for (int i = 0; i < V; i++)
    {
        graph->dir[i] = new int[V];
    }

    return graph;
}

void completeGraph(Graph *graph)
{
    int count = 0;
    for (int i = 0; i < graph->E; i++)
    {
        for (int j = i + 1; j < graph->V; j++)
        {
            graph->edge[count].from = i;
            graph->edge[count].to = j;
            graph->dir[i][j] = 1;
            count++;
        }
    }
}
```

A2. Code snippet of constructing a graph

```
int triangleInequality(Graph *graph)
{
    for (int i = 0; i < graph->E; i++)
    {
        for (int j = i + 1; j < graph->E; j++)
        {
            for (int k = j + 1; k < graph->E; k++)
            {
                if (graph->edge[i].to == graph->edge[j].from && graph->edge[j].to == graph->edge[k].from && graph->edge[k].to == graph->edge[i].from)
                {
                    if ((graph->edge[i].weight) <= (graph->edge[j].weight + graph->edge[k].weight))
                    {
                    }
                    else
                    {

                        cout << "Given graph does not satisfy the triangle inequality\n";
                        return 0;
                    }
                }
            }
        }
    }

    cout << "Given graph satisfies the triangle inequality\n";
}
```

A3. Code snippet of checking for the triangle inequality

```
Graph *buildMST(Graph *graph)
{
    int V = graph->V;
    Edge result[V];
    int e = 0;
    int i = 0;

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    subset *subsets = new subset[(V * sizeof(subset))];

    for (int v = 0; v < V; ++v)
    {...
    }

    while (e < V - 1 && i < graph->E)
    {...
    }

    Graph *tree = createGraph(graph->V, graph->E);

    cout << "Constructed MST using Kruskal's algorithm:\n";

    for (i = 0; i < e; i++)
    {
        tree->edge[result[i].from].from = result[i].from;
        tree->edge[result[i].from].from = result[i].to;
        tree->edge[result[i].from].weight = graph->edge[result[i].from].weight;
        tree->dir[result[i].from][result[i].to] = 1;
        tree->dir[result[i].to][result[i].from] = 1;

        cout << "There is edge between " << result[i].from << " and " << result[i].to << " where weight is " << result[i].weight << endl;
    }

    return tree;
}
```

A4. Code snippet of building MST (implemented with reference to Kruskal's algorithm from https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)

```
int findDegreeHelper(Graph *graph, int ver)
{
    int degree = 0;
    for (int i = 0; i < graph->V; i++)
    {

        if (graph->dir[ver][i] == 1)
        {
            degree++;
        }
    }

    return degree;
}

void findDegree(Graph *tree, int V, int ver)
{
    cout << "Found the degree for all vertices in MST:\n";

    int degrees[ver];
    for (int i = 0; i < ver; i++)
    {
        int degree = findDegreeHelper(tree, i);

        cout << "Degree of vertex " << i << " is " << degree << endl;
    }
}
```

A5. Code snippet of finding degree of all vertices in MST

```
Graph *findEPath(Graph *graph)
{
    cout << "Found an Eulerian path:" << endl;

    int ver = 0;
    for (int i = 0; i < V; i++)
        if (adj[i].size() & 1)
        {
            ver = i;
        }

    return findEPathHelper(graph, ver);
}

Graph *findEPathHelper(Graph *graph, int ver)
{
    Graph *eulerianPath = createGraph(graph->V, graph->E);
    list<int>::iterator i;
    for (i = adj[ver].begin(); i != adj[ver].end(); i++)
    {
        int v = *i;

        if (v != -1 && isNext(ver, v))
        {
            eulerianPath->edge[i].from = ver;
            eulerianPath->edge[i].to = v;

            cout << "There is path from " << ver << " to " << v << endl;
        }
    }

    return eulerianPath;
}
```

A6. Code snippet of finding Eulerian path (implemented with reference to Fleury's algorithm from https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/)

```
Graph *findHPath(Graph *graph)
{
    int i = rand() % 4;
    int initial = i;
    int count = 0;
    while (count < graph->E)
    {
        if (!isVisited(graph->edge[i].to))
        {
            int cur = graph->edge[i].from;
            int next = graph->edge[i].to;

            cout << "There is path from " << cur << " to " << next << endl;

            i++;
            i = i % 4;
            count++;
        }

        else if (isVisited(graph->edge[i].to) && (count != graph->E))
        {
            int cur = graph->edge[i].from;
            int next = graph->edge[i].to;

            cout << "Deleted path from " << cur << " to " << next << "," << next << " already visted" << endl;

            i++;
            i = i % 4;
            count++;
        }

        else
        {
            int cur = graph->edge[i].from;
            int next = graph->edge[i].to;
            graph->edge[graph->E + 1].from = i;
            graph->edge[graph->E + 1].to = initial;

            cout << "No more available path, adding path from " << cur << " to " << initial << endl;
        }
    }

    return graph;
}
```

A7. Code snippet of finding Hamiltonian path

```
1   void getWeight(Graph *graph)
2   {
3       int totalWeight = 0;
4       for (int i = 0; i < graph->E; i++)
5       {
6           totalWeight += graph->edge[i].weight;
7       }
8
9       cout << "Total weight of the path is: " << totalWeight << endl;
10  }
```

A8. Code snippet of calculating the total weight of our solution