The College of Wooster

CS200 Final Project

# The Traveling Salesman Problem

**- An Approach Using Genetic Algorithms and Heuristics -**

*Nicholas Hagopian-Zirkel*

advised by
Dr. Denise Byrnes

April 10, 2017

# 1 Abstract

The Traveling Salesman Problem is an optimization problem that asks to find the shortest distance one can take through a set of points. Runtime increases exponentially for any direct solutions so other methods must be used. Heuristic models and genetic algorithms both provide decent approximate results with polynomial runtime. This paper strives to examine possible solutions to the traveling salesman problem and how they can be made more effective.

# 2 Summary of the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a thoroughly discussed problem in the fields of applied mathematics and theoretical computer science. The interest in the problem stems from its applicability and from it as one of the earliest examples of an NP-hard problem. The problem can be easily generalized, however in its original form it is presented from the point of view of a traveling salesman. The salesman has a list of cities that have to be visited but there is no specific order in which to visit them. The only things that is known is the location of each city, that each city has to be visited exactly once, and that when he has visited every city, he must return home. Given that he wants to reduce the amount of time he's working, it is in the salesman's interest to find the shortest route to hit every city. More generally, given any set of points in a 2d space, what is the order that points should be visited in order to produce the shortest route. For clarification, all points must be connected to the route and the route starts and ends at the same location.

The original TSP is attributed to mathematicians Sir William Rowan Hamiliton and Thomas Penyngton Kirkman in the 1800s. The general form however was not studied until the 1920s by Karl Menger. The largest tour proven to have no shorter length is a tour of 24,978 cities in Sweden approximating out to be 72,500 kilometers. [cite ARS] Before continuing too deep into the problem it is important to define some terms that will be used often in this paper. Firstly all points that we are trying to connect may also referred to as cities or vertices. The way in which we

connect these points are through routes which also may be referred to as tours. Two points are connected by an edge. Lastly much later when we talk about genetic algorithms, gene recombination is another way of saying gene crossover.

Earlier it was mentioned that the TSP is considered NP-hard, this stands for "non-deterministic polynomial-time hard." The standard version of TSP, as presented in this paper, is also not considered NP-complete. Being that it is NP-hard means that large versions of this problem cannot be computed in reasonable time and not being NP-complete means that any presented solutions to this problem cannot be verified in a reasonable time either since verifying it requires solving the problem in the first place. There are many approximations of an optimal tour but the only way to get the definite shortest route is to check all permutations of connected cities. The runtime for this method is $O(n!)$ which can get quite out of hand rather quickly for large values of n. This is why approximations must be used which can be generated through methods such as heuristics and genetic algorithms.

# 3 Variations of the Traveling Salesman Problem

The traveling salesman problem may seem niche, but it is actually a basis for many other problems. The TSP has simple rules. Its simplicity is likely because it is already so hard to solve in the case of large problems. Given how generalized it is, it is not incredibly applicable to real world situations. Perhaps the most heinous assumption is measuring distance simply as lines between points. In real life, roads have twists and turns and sometimes there are mountains or rivers between points. Other factors get more complicated, including the varying profitability of visiting each city and the assumption that it is in the salesman's best interest to visit every city. Mathematically these problems can be solved by giving edges and vertices varying weights and maximizing or minimizing these weights. Some variations on the TSP are widely used in robotics, particularly in the design of circuits on chips. Some variations include the bottleneck traveling salesman problem, minimize a TSP with edge weight; traveling politician problem, visit each state at least once; and the traveling purchaser problem, which deals with purchasing a range of products

from various cities where cities sell different products for different prices. Although some modification is needed, heuristic models and genetic algorithms can both be used to solve more applicable problems.

# 4 Held-Karp lower bound

Measuring the success of any TSP approximations proves to be rather difficult. Given that it is not considered NP-complete in its standard form there should be no way to verify whether an approximation turns out to be the correct solution. In order to find out how close to the correct answer an approximation is then the correct answer must be calculated directly. This results in a circular problem however and thus other options must be analyzed. Instead of trying every approximation on every problem and choosing the best result we use the Held-Karp lower bound.

The Held-Karp lower bound won't give the minimum tour with 100

# 5 Direct Solutions

## 5.1 Brute Force Solution

A brute force solution to the TSP is simple yet impractical. Given below is a standard brute force solution given in steps.
step 1 - Generate a list of all tours
step 2 - Calculate the length of each tour
step 3 - Find the minimum tour length
It is possible to reduce space complexity by instead of generating a list of all tours, sequentially progress through all possible tours and only store the tour data if it is the minimum. This, of course, requires a good way to sequentially generate all permutations and it also ignores the underlying problem of time complexity. The C++ code for a brute force solution is presented below. [to be added with final draft]

### 5.2 Dynamic Programming

At first this problem may seem like a candidate for dynamic programming. The problem has a hallmark features of dynamic programming: overlapping subproblems. Intuitively it is known that if permutations of every possible tour are being calculated then many of the same subtours will be calculated multiple times. There are at most $O(n2^n)$ subproblems and since they take linear time to solve the time complexity is $O(n^2 2^n)$. This is significantly faster than $O(n!)$ but is still exponential. In addition the space complexity is now also exponential which creates new issues. The reason dynamic programming doesn't significantly help time complexity is that it does not have an optimal substructure. Dynamic programming is useful so as to not repeatedly compute the same subtrees, however it still has to check all the possible combinations since a tour is only known to be the minimum length once all points are in the set. Thus a locally optimal tour is not necessarily globally optimal. Although this is true, many approximations involve greedy algorithms.

## 6 Heuristic Solutions

There are many heuristic solutions but the following are some of the most basic. More complicated solutions may use all of the following heuristic methods.

### 6.1 Nearest Neighbor

This is perhaps the simplest method of approximation. As the name suggests nearest neighbor attempts to create an optimal tour by connecting points together through their nearest neighbor. Runtime for this is $O(n^2)$ which is acceptable given the vast difference between it and the direct $O(n!)$. Nearest neighbor takes the form of the following steps. [cite ARS]

step 1 - Select a random city

step 2 - find the nearest unvisited city and connect it

step 3 - If there are still unvisited cities repeat step 2

step 4 - Return to the first city

It stays withing 25

## 6.2 Shortest Edge

This method is a greedy algorithm similar to Kruskal's algorithm which is used to generate a minimum-spanning-tree. This method creates a tour using the shortest edges. Each edge is checked to make sure it doesn't break the rules of the TSP. The more precise steps are given below. [cite Nillson]

step 1 - Sort all edges into a set and connect the shortest edge

step 2 - If the vertices of the next edge are disjoint the connect this edge

step 3 - If the set of edges is not empty then repeat step 2

step 4 - Connect the two ends of the route

The problems with this algorithm is that it has to create a set of all the edges, which can take some time, and the algorithm must be careful not to connect points that are already connected by non-direct edges. It stays within 15-20

## 6.3 Insertion

There are many variants of insertion heuristics. The general form is to find some subtour of all cities and then insert cities individually into the tour by finding the cheapest insertion. The following examples of insertion heuristics are taken directly from Nilsson. [cite Nilsson] Nearest Insertion, $O(n^2)$

step 1 - Select the shortest edge and make a subtour of it

step 2 - Select a city not in the subtour, having the shortest distance to any one of the cities in the subtour

step 3 - Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will be minimal

step 4 - Repeat step 2 until no more cities remain

Convex Hull, $O(n^2 log_2(n))$

step 1 - Find the convex hull of our set of cities, and make it our initial subtour

step 2 - For each city not in the subtour, find its cheapest insertion (as in step 3 of Nearest Insertion). Then choose the city with the least cost/increase ration, and

insert it.

step 3 - Repeat step 2 until no more cities remain

### 6.4 Heuristic Improvement

Once a tour is constructed using any of these techniques it is possible to improve it so the solution is even closer to the correct result. The most common ways to improve a tour is by using 2-optimal (2-opt) and 3-optimal (3-opt) local searches. An n-opt algorithm removes n edges from the tour and then checks for the shortest way to reconnect these n edges. Since runtime of each n-opt check increases exponentially with the size of n, 2-opt and 3-opt are the most common options. Using 2-opt heuristic will often result in a tour less than 5

## 7 Genetic Algorithm Overview

Genetic algorithms are another clever option for approximating solutions to problems in the field of combinatorial optimization. The traveling salesman problem and variations of it are precisely the type of problems where the application of genetic algorithms should be considered. The idea behind how genetic algorithms work came from evolution and the idea of breeding solutions to problems. In evolution and genetic algorithms alike when a generation is presented with a problem, due to genetic randomness, some of the members of that generation will have genes that allow them to cope with a problem more successfully than other members. If the problem is life threatening then those who are the most suitable to handle that problem are more likely to survive to pass on their superior traits to the next generation. The steps for a genetic algorithm are as follows.

step 1 - Randomly generate initial population

step 2 - Evaluate fittest members of the population

step 3 - Kill off unfit members of a generation

step 4 - Breed fit members and apply crossover and mutation to generate a full population size

step 5 - If the solution is not yet acceptable go to step 2

step 6 - Else return shortest route

One of the most important factors of genetic algorithms is randomness. This is why step 1 and the mutations in step 4 introduce randomness. Even step 3 has some randomness involved for the members in the middle of a generation. A central philosophy of genetic algorithms is that with enough randomness eventually a member of a generation will get lucky. That's not where it ends however, once the algorithm gets lucky it has to be able to evaluate when it got lucky with a fitness function and it has to build off of this luck using breeding between generations.

## 7.1 Initial generation

The members of the first generation can either be randomly varied, or they can be mutated from a heuristic solution. The initial generation is likely not going to produce the correct answer. It may however have some good ideas—as in genes from some of its members may have optimal subtours.

## 7.2 Fitness Function

The fitness function is a necessary component of all genetic algorithms. This is the part of the algorithm that answers the question of whether a member of a generation should survive to pass on its traits. The term "fitness" refers to the evolutionary idea of "survival of the fittest" in which those who are more "fit" are more likely to survive long enough to reproduce. In the case of the TSP the fitness function will rate all the members of a generation by shortest tour divided by average tour length of a generation. The more successful members will then be awarded a higher chance of survival. The purpose of not simply eliminating a bottom percent is so that generations don't become homogeneous. Under a similar chain of logic a higher chance of survival should certainly be awarded to the top members of a generation but it should also be awarded to any members who are relatively successful while also have drastically different tours then the most successful members. This variance can lead to tours that are better than either of the parent tours. Killing too many members should only be a worry if a large number of successful members with

varying tours are killed. Killing too few members is also a problem, however it is only minimal. The only negative effect comes in terms of a slower runtime. Compared to the TSP direct solution however, the runtime will never be all that bad. Both killing too many and killing too few members of a generation will eventually lead to optimal solutions.

## 7.3 Crossover and Mutation

It is helpful to think of each new generation as a two-step process. The first step is where the unfit members of a generation are killed in order to generate an intermediate population. Then recombination and mutation are applied to the intermediate population to create the next generation. In all genetic algorithms, member of a generation have their own "genes" (typically represented by a string of bits) that can be used in crossover or mutation. In the case of the TSP these genes refer to the order in which cities are visited. It may seem confusing why crossover and mutation are both used in genetic algorithms. The purpose for crossover is to pass on the successful genes so as to get the best out of every generation. The goal of mutation however is to introduce new opportunities to succeed. Without mutation the only variance would be whatever was created in the initial generation. It is also important not to overdue crossover, one tour that was successful may need all of the components in it to maintain this form. Likewise too much mutation leads to a generation that did not get the success of the parent generation passed on to it. Crossover is incredibly important and tricky. If just single points are crossed over then the success of a generation is not shared well. This is because shortest length is relevant to a collection of points, not just the placement of a single point.

**The following is a vague guideline of the work to be completed for the final draft of the paper. These sections require finished software. In addition, some of the above sections are incomplete without software examples.**

# 8 Methodology

Here the work done for this project is discussed and any algorithms generated are represented. Segments of code will also be shown here.

## 8.1 Software

This section discuss all the software used and why that software was chosen for this topic. It also will discuss why C++ was chosen over other languages.

## 8.2 Work Completed

This section header is subject to change. It will be analyzing the algorithms created and their functionality.

# 9 Conclusion

Analysis of the usefulness of results is presented. Comparison to a heuristic solution to the problem may be presented as well to evaluate usefulness of a genetic algorithm.