HE DOESN'T KNOW THE TERRITORY

AN INVESTIGATION OF HEURISTICS FOR THE TRAVELING SALESMAN PROBLEM.

TREVOR HARRON

Adviser: Professor Mathew Dickerson

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Arts

May 2014

TABLE OF CONTENTS

1 Introduction **3**

2 The Data and Methodologies **14**

2.1 The Data 14

2.2 Data Preprocessing 16

2.3 Methods 18

3 Algorithms **21**

3.1 Conventions of Pseudo-code 21

3.2 Nearest Neighbor 21

3.3 Greedy 24

3.4 Minimum Spanning Tree 28

3.5 Genetic Algorithms 31

3.6 Christofides 31

4 Possible Improvements **32**

5 Results **33**

6 Conclusions **34**

Bibliography **35**

CHAPTER 1

INTRODUCTION

There are several puzzles that haunt computer scientists in their quest for finding what can and cannot be computed with our current paradigms. Some of these problems have to do with graphs and answering questions about them. Some of these questions turned out to be easy to solve such as the Seven Bridges of Konigsberg where we want to go on each edge of a graph once. Other problems on the other hand are more difficult to solve such as finding Hamiltonian Cycles and arguably more importantly the Traveling Salesman Problem.

The Traveling Salesman Problem (TSP) is simply the question if a salesman is trying to visit each city given a set of cities once without driving on the same road more than once and ending at his starting location again, what route does he take? In essence for a cyclical graph, what is the shortest Hamiltonian Cycle. A Hamiltonian Cycle is the route to visit each city once creating a circuit containing all of the cities. More formally the TSP is defined as given a Graph G is a set of N cities C and a set of edges E such that $e \in E = (c_i, c_j, k)$ where $i \leq N, j \leq N$ and $i \neq j, c \in C$, and with k as the cost to travel on that path, find the path that covers every city with a minimal cost and returns to the starting city c_0 in G. While on the surface this seems like a reasonable problem for a computer to solve it turns out to be quite complex to solve with a reasonable algorithm.

The TSP turns out to be in a class of problems that computer scientists aren't sure if there is a efficient solution to the problem; these problems are called NP-Hard. The algorithmic complexity of the naive solution where we check every possible route turns out to be $\Omega(n!)$. It is important to note that when I say an efficient solution I mean one that can be solved in a Polynomial amount of time (i.e. $O(n^k)$ where k is a constant). Even for smaller instances of the TSP it turns out to be infeasible to use the naive solution for the case of trying to visit every capitol in the United States (starting at Washington

D.C.) it would take an inordinate amount of time and processing capabilities to solve. This also means that for the graphs that are being examined the exact optimal distance cannot be known. Even with improvements to the algorithm such as using Linear Programming (which is outside the scope of this paper) the complexity is $O(n^2 2^n)$, which, while more efficient than the naive solution is still not polynomial making it infeasible to always find the optimal solution. Despite this there is still the need for the TSP from companies such as FedEx and UPS for instance that depend on the quick and efficient delivery of mail and packages to a not fixed set of locations from day to day. In an effort to find other solutions to the TSP computer scientists have for years worked on efficient algorithms that may only find an approximate solution to the TSP. These algorithms are known as heuristics and are the subject of this paper. There are various permutations of the TSP that include the symmetric and the asymmetric TSP, but for the purposes of this paper there is not a distinction drawn between the two major sub-problems. Other forms of the TSP that is not examined at all in this is the Euclidean TSP which is not based on the network distances but instead the Euclidean distances (hence the name). Other instances that are not examined include the Constant TSP and the Small TSP, the Bottleneck TSP, Time-Dependent TSP, and the Stochastic TSP. There are some cases where the TSP has also been solved but those will not be examined either; in short this paper is trying to examine these heuristics in the most General form of the TSP possible with real world data.

For this project the graph that these heuristics will be used on comes from the real set of cities from the continental United States and the edges of the graph are calculated based on the network distance (road distance) in between the cities. Given the impossibly large nature of the dataset, the graphs used are created from the cities and roads of a single state (with islands and other non-reachable cities removed) as opposed to using the continental United States as a whole. The overall problem that this paper

intends to address is the implementations of various heuristic algorithms and showing that these heuristics can be used in real world scenarios and looking at the benefits and detractions of each heuristic. Finally after seeing this, the question then lies of what improvements could be made in terms of algorithmic space and time complexity and with the guaranteed upper bound as well as the complexity and overhead knowledge required for implementing a given heuristic.

Most of the heuristics that are examined in this paper are also called Tour Construction heuristics. These are obviously heuristics that are used in the construction of the route and for the most part can guarantee being 2 times the optimal length. It is important to note that a subset of these heuristics are local search heuristics (such as Nearest Neighbor) and those cannot be guaranteed to be within a multiple of the optimal solution[10]. Among these heuristics are the Nearest Neighbor algorithm, the Greedy algorithm, Minimum Spanning Trees and the Christifolds heuristics. The second heuristic that will be examined is a tour improvement heuristic based on Darwin's Theory of Evolution called Genetic Algorithms. Other Tour Optimization heuristics which focus on taking a given route and such as Simulated Annealing and the $k - opt$ family of tour optimization heuristics. Combinational approaches and Ant Colony approaches are not examined in the scope of this paper. Other approaches that aren't even considered in this paper are Approximation heuristics, Branch and Bound heuristics. In short the heuristics that are covered in this paper are only a small portion of the possible heuristics in the world of TSP heuristic studies. The reason for limiting the research is to allow for a more in depth examination.

To test the heuristics, an Experimenter was made to go through all of the different datasets for each of the solvers multiple times. Of course the number of times that these experiments are is not a small number to eliminate any possibly for chance or circumstance interfering with the results. The results from the various algorithms is not only

the list of cities (in order of visiting them) but also the time in seconds required to solve the graph as well as the Megabytes needed. For each solver, there will be several states used to create the various graphs needed for testing. These graphs, will vary in size of the set of cities as well as the number of routes from one city to the next. Each of these graphs with be used to see the overall effects that the size of the graph has on the performance of the heuristics.

Before moving on to more details of the heuristics being studied, there is one important concept that should be kept in mind, which is the Held-Karp lower bound. The Held-Karp lower bound is a good measurement of comparison of an algorithm's performance. For instance, there will be several references to how close to the Held-Karp Lower bound another algorithm can get to. As for how the Held-Karp lower bound is computed, this is out of the scope of this paper. It is sufficient to say however that the approach is "...the solution to the linear programming relaxation of the integer representation of the TSP" [7]. In short, the Held Karp algorithm used to create the Held-Karp lower bound is a solution using a non-graphical representation of the TSP. This polynomial time algorithm has been found to be within .08% of the optimal length in most cases and at worst within $2/3$ of the optimal tour. As such is a reliable metric to use in evaluating heuristics of the TSP. In short it is important to keep this as a concept since it will be referenced several times as a means of algorithmic evaluation.

As whole, heuristics have been studied as a potential alternative to the naive solution to the TSP. This pursuit then led to the a conclusion stating that if there was a heuristic that guaranteed an optimal solution there then $P = NP$. For the most part, computer scientists agree that $P \neq NP$ and as such there is not an efficient solution that will also guarantee the optimal solution. In lieu of that, to investigate the various heuristics of the TSP, several algorithms have been chosen for their algorithmic complexity and guaranteed maximum distances as an example of most of the conven-

tional approaches that currently exist. These algorithms are Nearest Neighbor, Greedy, Minimum panning Tree, Genetic, and Christofides algorithms. Each algorithm will be implemented and then tested based on their runtime, their CPU usage. After this, it will be explored if their exists a hybrid algorithm that can utilize the best attributes of the traditional heuristics or another approach combining the different kind of construction and path optimization approaches.

The heuristic algorithms implemented will also be measured on several different qualities. These qualities include the algorithmic time and space complexity as well as the actual run times and CPU usage and on the guaranteed upper bound of the various algorithms. In addition to those measurements, the algorithms will also be evaluated on the complexity of implementation. The complexity of implementation means the number additional functions needed for the algorithm, the extra data structures aside from the graph as well and the additional knowledge that the implementer would need. Encompassing some of this complexity is also the additional work needed for the helper functions for the algorithms as well. These issues of complexity, though mostly qualitative in nature, will be examined in the most quantitative ways possible, in looking at the amount of time needed for writing each of these heuristics in addition to any comments about the number of lines of codes, the number of class variables, etc. These will then be put through t-tests to see if there are any significant differences in the results in terms of one algorithm's efficiency in the different qualities. These results in combination with the notes on implementation will then be used to see if there are significant improvements that could be made.

Java is the language that will be used for this project. Java is it is an Object Oriented Programming language that uses a Virtual Machine to run its code. The reason for this is that this provides a layer of abstraction for the developer from the system s/he is working on. This allows for code that is written on one kind of system to be run on

other kinds of systems as long as they all have a compatible version of Java (and thus the Java Virtual Machine or JVM) on them. The choice to use Java is mainly due to its platform independent nature for the implementation for each algorithm. In addition to Java the other languages considered for this project were NetLogo, Python, or C++. NetLogo is a simple multi-agent based language and as such is not suited to the flexible implementation of TSP heuristics or in the gathering of data needed to draw necessary conclusions. Python is a simple interpreted language that would be easy to implement the heuristics with but also can lack the durability of other languages as well as support and use in the workforce for large scale projects where efficiency of the algorithms becomes key. While C++ might be easier to examine the memory usage and is also faster, it is dependent on the system that the algorithm that is running on. C++ also lacks some of the durability of Java making it less than ideal for larger problems and individual efforts such as this to create heuristics. As a result of these different factors, Java, while not the fastest language not only abstracts from the machine's system but also provides the durability of for a individual endeavor and for instances of large datasets at the cost of some difficulty in extracting memory usage. This difficulty is to be expected because of the fact that Java programs run on the JVM instead of the Computer itself. The reason this drawback is a fair trade-off is that it avoids any issues with questions regarding code dependency on one specific operating system or about questions of efficiency based on the system itself. Another reason for using Java is that several companies that depend on these heuristic algorithms also use Java. Part of the purpose of this project is to test the implementation capabilities and as such choosing Java was done with the fact that several organizations whose services require a fast an efficient solution to TSP would also use Java as their main development language. Java will also be used to both parse the data files as well as the visualization of the graph and pathways. Each of the experiments will also be written and run Java naturally. For clarity in each algorithm will then

be displayed in both their pseudo-code format as well as their implementation in Java in addition to the explination. Having said all of this, the purpose of this paper is not to argue the implementation of algorithms in one language over another but instead to focus on the implementation of heuristics for the TSP.

Having said that everything is in Java, it is important to give a brief description of the basic lasses/interfaces used for such a project. The interface most concerned with the graph's construction (apart from the Graph interface) is the Reader interface who's job it is to parse information in either a Comma Separated Value (csv) document or a key markup language (kml) document. As a note: to be able to parse and use kml documents required to construct the graph an external library designed by the makers of the KML format (citation here) was used to read the of all of the cities was used. Readers can also be implemented to take any other number of formats in theory but for the purposes of this implementation kml files and csv files were focused on. The Reader is able to take the documents preprocessed from the shapefile format as described in Chapter 2 into either csv (if used from the GIS OD Cost Matrix) or the kml format preprocessed from Census city data. The different readers that are implemented from the Reader Interface are then used to parse these different formats: csv for roads, csv for locations, and kml for city locations.

After the Reader interface is more importantly the Graph interface. This interface allows for different representations of the graph to be used abstractly through some key methods. There are some basics that each graph has such as Nodes and Edges to represent the cities and roads. The two graph implementations used for this were the Matrix model and the list model. The matrix model is where the edges of a graph a represented in an nxn matrix, where n is the number of cities, and the edge c_i, c_j, k goes from city c_i to city c_j at a cost k and is at position (i, j) in the matrix such that $i \neq j$. Those edges that are on the diagonal have been declared $null$ for the purposes of simplifying the im-

plementation. The list model has a slightly different structure where the Nodes contains a list of pairs of Nodes and distances from Node to Node. The reason for such a difference is that certain algorithms work on one graph implementation than another. These two different kinds of graphs also represent the two major ways that directed graphs are represented mathematically. It is also important to note that due to some of the restrictions of programming languages (and for the sake of simplicity in parsing the data) The Edges of the graph are directed though the for each city pair there is an edge going from one city to the other and visa-versa (this happens to be the case in the instances presented in this paper). In essence, this makes abstracts away from the complicated potential of changing the algorithms for directed roads and non-directed and allows for us to better examine the heuristics themselves.

For the Edge and Node classes they are relatively straight forward in implementation. For the Edge class (there is only one) the edge holds the object identifiers for both the city the road is going from as well as the city the road is going to and the distance from the two points. As stated earlier, this distance is not Euclidean but is instead based off of the network distance traveled by road to get from one place to the other. The final point to make in regards to the Edge class is that it implements the Comparable interface (from the Java standard library) allowing it to be sorted in the case where the sorting of Edges is needed. The Node super-class is very simple in that basically it holds the basic information about the city/point of interest such as the coordinate x and y, the name of the city, the unique identifier of the location (generally a number) and the position it was read in. The last variable is to allow for the Node to know where it lies in say a matrix based graph. There are also two sub-classes of the Node class which are Matrix Node and List Node. This implementation is similar except that the List Node keeps track of the roads that it is attached too. This is to allow for easier implementation with various tree based algorithms or algorithms that focus on the Edges more than the nodes.

The final and arguably most important interface to mention is the Solver. The basic idea of the solver interface is that the classes that implement the solver interface are the ones that implement the heuristics tested in the paper. Each solver deals with the implementation of the heuristics. These are all created by a factory to help better abstract away from the direct implementations of the solvers. This is used by both the Experimenter class (used for the running of experiments) and for the TSPApplet which is the GUI for looking at how the solvers are solving the TSP for the various heuristics. for each different heuristic, there is a solver that goes with it. Also for purposes of clarity in the Java code, the helper functions of the algorithm are not displayed as that would clutter the examination of the algorithms (except in certain cases where they are the defining factor of the algorithm's time and or space). Each solver also contains the necessary container classes to solve the problem and methods that go along with that; for example a Minimum Spanning Tree Solver (or MSTSolver) would contain a Spanning Tree object and the method for transversing the tree. These more specific cases will be gone into more detail in the Algorithms section of this paper. For purposes of clarity, the result of the solver is an ArrayList (re sizable array) of the cycle of cities, the distance of that cycle as well and the time, and space used to solve the graph. The For each solver, testing has been done to ensure the accuracy of the algorithm's implementation, the accuracy of the solution on a smaller (less than 10 cities) dataset as well as the durability of the code in place. This dataset has no bearing on an place in reality and as such is not considered in the analysis of the heuristic algorithms.

The final set of classes that should be discussed are the ones pertaining to the visualization and the experimentation of the project. These classes are pretty self explanatory and focus on using the Readers to create the graph and the Solvers to solve it. In the terms of the differences between the Experimenter and the TSPApplet, the Experimenter finds the solution and repeats the experiment a certain number of times adding

the results to a csv file to be analyzed with t-tests or to use for graphic representations of the results. The TSPApplet is a Java applet whose purpose it is to view the results of a single iteration of the the solver for a given graph. The graph and the solver can both be chosen by the user to allow for easy viewing of the heuristics of the TSP. Overall the need for these classes while not strictly what is being examined are paramount to the study, analysis and debugging of the various heuristics implemented in the course of this paper.

As a final note pertaining the the code used within this paper, it was all also written with good coding practices in mind. As a result of this, the code developed was tested incrementally on small data sets as mentioned before and tried to use elements of good Object Oriented design. As for the exact specifications of good Object Oriented Design, those are outside the scope of this paper. However, in all stages of development, code was written to be flexible and reusable and attempted to minimize the amount of rewriting that can go on in a project such as this. As such all of the classes are in packages appropriate to their function, names either are intuitive or were named to mimic the pseudo-code (for readability in side by side comparisons), inheritance was used as a means of preventing rewriting and Interfaces were used in order to abstract away from implementation dependencies. In short, the code developed in the course of this paper was made to try and match industry standards given that there is a strong undertone usability of various algorithms in a space where a trade off must be made in between efficiency and accuracy in the real world.

Going forward there are a few things key concepts and terms that will be the core of this paper. The first of these is that the TSP is a difficult problem to solve and as such the heuristic algorithms used to approximate the solution are the main focus of study. The various algorithms are implemented in Java as well as the testing, visual, and other necessary parts to creating the graph. The data (which will be explained in more detail)

is based off of real world data and as such the coding practices used focus on real world implementing of these algorithms. In lieu of this the focus of this paper lies not only in the time and space of the algorithm but also in the complexity of the implementation as well.

CHAPTER 2

THE DATA AND METHODOLOGIES

2.1 The Data

The data used for this project was a combination of road census data for the edges of the graphs and their costs. The road and city data are from 2006 and as such it should be noted that the main use of this data is to show some of the practical applications of the algorithms that are implemented in the course of of this paper. The scope of this data will be examined as well as the contents and implications that this may have on the paper as a whole. While this is not critical to the algorithms or the overall results of the paper it is important for the data to be examined in detail to better to understand the scope of the paper as the whole.

The cities (or vertexes) are also from US census data containing the major towns and cities in each of the 48 continental states. Each of these files were originally in a format called a shapefile. Shapefiles are used by GIS software and contains the geometric data for the objects within the shapefile. These various features include an unique identification number, the geometric shape of the objects, relevant information regarding the geometry, the coordinates of the object, its name, and other pertinent features. While shapefiles are useful for GIS studies the format is not parsable for the purposes of this paper. Before going into the preprocessing of the data first the details of the data should be examined for both the cities and the roads that will be used for this project.

For the instance of the cities, the geometry was simply a point with no additional data attached to it apart from the coordinates for the point, the cities also have a name and a state that they are in, the basic information including the city's id and also the additional information that might be useful for GIS investigation including population size, range, and the county that the city is a part of. However, lacking from the data is

the road or roads or any mention of the roads that, the city is closest to. Most importantly, it should be noted that the cities are by no means a comprehensive list of all of the populated places within the United States though admittedly is it a larger set than the average layman would think of.

The roads are made up of several different segments and the junctions that they connect to. These junctions and lengths are also held are small vectors that make up an arc. It is important to note that these are not exact and in some cases required some additional preprocessing that will be described later. However, this property of the road while approximate in nature is still reasonably as accurate as one could hope. Also, the additional soon to be further explained preprocessing changes did not change in any way the underlying distance data of the lines. For the distance data that each segment has, for the purposes of easy comparison and accuracy the use of meters was chosen. This is not a choice based of the practicality of the use of meters in the United States but instead the practicality of the metric system.

The basic notion of the data is that the there are two different kinds of raw data that are used for this paper. The two kinds of data are based of the cities and the roads. Both of these are originally in a shapefile to be read for a set of GIS programs which this paper does snot implement. Despite this the benefit of this format is the additional data that is involved. Though while most of this meta-data is not useful for the purposes of this paper there still are some pieces that are vital to the paper including the geometric data in particular. It is important to finally note that the data that has been up until now as a whole is truly two separate sets of data. The two sets are based on the two different kinds of objects required for the paper: cities and roads. The combining and binding of these two different sets of data happens in the data preprocessing stage.

2.2 Data Preprocessing

To gather the data, US Census data from 2006 was used to for the cities and roads in the US and pre processed using ArcGIS. The set of cities was saved from its original shapefile format to a readable format called kml (Key Markup Languages) which maintained all of the spatial and geometric properties of the original format. These properties include the state that the city resides in, the coordinates of the city, and additional properties that a user might useful. These features are the ones that are in the shapefile previously mentioned. To transfer the shapefiles into the more readable kml and csv file formats a tool was used called ogr2ogr (pronounced ''Oger to Oger""). Ogr2ogr is a open source tool whose sole purpose is to ''translate" shapefiles to other formats. These formats range from other GIS formats like raster, to a variety of json called geojson, kml, csv and more. Among these formats there was research done as to what the best format to use for both the cities and the roads before chosing the final formats to store and parse the data in.

Before even considering translating the formats into a parsable format the data needs to be processed from its raw form into the graphical form that is used. Then to calculate the network distances of each city, ArcGIS was used. To gather the distances first in ArcGIS is that an OD Cost Matrix was created from the cities of a given state and all of the roads from the census data. Naturally, to ensure that there are not any errors that occur either from either the preprocessing that ArcGIS does in laying down roadways on the map there was some manual labor involved in ensuring the roads that are supposed to connect to each other do. Such errors can occur due to slight miscalculations in the data itself or in how the shapefiles are processed in ArcGIS. The corrections that occurred include the connection of roads that are continuous, the linking of cities to the closest road, and even telling the OD Cost matrix that some roads end at the coastline away from cities and should not be considered. In the creation of the OD Cost

Matrix the cities are filtered by the state that is being used in the creation of the graph for the vertexes. All of these edges are placed as the origins and the destinations of the matrix to try and find all of the edges for the graph that are possible. To ensure the completeness of the graph and that there are not any disconnected cities, the cities that are on unreachable locations such as islands have also been removed. Such features would also make several of the algorithms infeasible.

The Reader class is primarily responsible for parsing the various formats and converting it for the uses within the Graph. In terms of the implementations, the classes for reading the kml and the csv formats are distinctive to what the data is used for. For instance the KMLReader reads the kml file of cities and uses the meta-data to create the cities for the graph to hold. With the CSVReader, the initial implementation meant that the roads were the only part to be read but this could easily be changed to accept arbitrary locations of interest to visit as well as the edges. To ensure that the algorithms and parsing work correctly, there is also a sample dataset that have been created consisting of five vertexes which are interconnected by edges with costs. This ample data uses the same formats as the actual data.

For the distances and edges, a variant of Dijkstra's algorithm is used[9]. These distances were then stored in line objects that consists of an id, a pair of cities in the form of ''to - from" and ending in the distance of the route. All of these line objects were then stored in a shapefile tied to the OD Cost Matrix. To read these lines the shapefile takes from export it and then translate it into a csv file. In the case of a csv file there is not the geometry of the lines at all. This lack of geometry is not of concern however since the lines are tied to the cities meaning that the geometry of the lines is not needed at all. In fact the only line geometry that is regarded is the lines that are stored in the roads which has been mentioned before. For the lines that are then represented in the graphics are represented as a straight line instead of the network distances. This is purely for the

ease of displaying the graph as a whole.

2.3 Methods

The methods used for testing are very simple in terms of concepts needed. The information that is being gathered is simply the amount of time needed to run the algorithm, the space, and the distance of the route that the algorithm produces. To gather all of this extra data, it is simplest to gather it when obtaining the result, which is simply an array of the order of cities are visited in to get the route. Of course also to gather this information there is extra code that has been added to the algorithms which have no effect on the result itself nor its time or space usage in any significant way and of course have no impact on the algorithmic evaluations.

To gather the time that it took a given heuristic to run, Java has some built in features to help keep track of the time an opteration was started which is the basis for the time keeping that is used. Admittedly, the fully time keeper is gathering the starting time which Java keeps as a long then immediately after the program is complete finding the current time (also a long), find the difference between the two, and then converting it into a readable format. The amount of time is recorded in the final solution in terms of seconds. For the smaller datasets concern is understandable for the purposes of measuring time some might find the use of tenths of seconds more useful but for the larger cities examined, it would soon become clear that seconds is not only valid but easily puts the results in context without meaningless conversion. The amount of time that it takes the program to run is the first of measurements to be added to the result.

As stated in the introduction, since Java is used as the language of implementation, gathering memory usage is not a straight-forward matter. However, to be able to look

at the memory usage, all that is simply done is use a function that calls Java's garbage collector and then look at the total memory and free memory to see how much was used. The function called is ''System.gc()' for 'Garbage Collection'. This technique is frequently used in programs to check for memory leaks (i.e. data that is not cleaned up properly. The calls to get the used amount of memory and the total memory are System.totalMemory() and System.freeMemory(). By finding the difference in the two we can find the amount of used memory there was. However, also as mentioned earlier, the memory is being collected in terms of KB so the difference is divided by 1024 (since 1024 bytes go into a KB). To avoid any issues this might have on the time, the space is taken after the time is taken. This number of KB that is used is also then added to the result that is returned.

Distance is the final quantitative measure that the heuristics are being evaluated on. The distances themselves come from the road distances that were calculated as stated earlier in Data Preprocessing. The gathering of distance occurs at different times depending on the heuristic in question but are either gathered during the construction of the route or after the route has been constructed. For the gathering of the distances during the tour's construction, the distance is only aggregated after it has been confirmed that a route from one city to another has already been added to the final tour produced. Also, for each of the tours it is needless to say that the final distance back to the original starting city has been included in the final distance. Also all the distances are kept in their original form of meters for easy base 10 conversions for later calculations of the results. The distance is the final measurement added to the results.

To evaluate the code and the heuristics overall there are two main forms of evaluation, quantitative and qualitative. It is simple enough to understand the quantitative measures, these are based on the use of time and space and are measured in terms of big-O notation. In addition to the algorithmic time and space the actual time and space

and distance result of the algorithm are also measured. Slightly in between these two approaches is the evaluation of the difficulty of the implementation itself. The quantitative approach for evaluation is the number of extra helper functions and the algorithmic complexity and implementation of these functions. The qualitative approach to the heuristics comes from looking at the knowledge required to implement these algorithms. This is a qualitative measure given that this information while measurable in terms of number of concepts, does not lend itself to more definite measures. More of the qualitative measures also include the intuitive difficulty of the heuristics in both understanding the approach itself as well as the intuition of the implementation. The quantitative measures of the algorithms are then confirmed as different ranging from heuristics to heuristic with t-tests to show if there are significant differences between the results of the algorithms. In conclusion, the methods for evaluation are both quantitative and qualitative and for the msot part, there are many attempts of quantify all of the data.

CHAPTER 3

ALGORITHMS

3.1 Conventions of Pseudo-code

The pseudo-code used in this paper does not conform to a single form of pseudo-code but instead attempts to be as readable as possible. Some points to keep note of include the use of plain English and mathematical symbols to try and concisely give the algorithm in a readable format. To augment this endeavor some plainly named, black box functions have also been used. The pseudo-code also appears in a more Object Oriented Programing approach to better link the understanding between the pseudo-code and the Java code.

3.2 Nearest Neighbor

One of the simplest heuristic algorithms is Nearest Neighbor (or NN for short). This algorithm's is to first start at a random city (or a specified city) look locally at the cities nearby and then adds the closest unvisited city to its to the solution. Then second keep adding cities in this method until there are no cities left and then finally find and add the edge back to the starting city. This is obviously a local and greedy approach to solving the TSP. It is to be noted that while in some cases the cost of the algorithm is taken after the route has been found but for this implementation of nearest neighbor the distance of the route is taken when the city is added to the route. The basic pseudo-code for the algorithm briefly described above is shown below:

NNHEURISTIC(G):

1 $U \leftarrow getStart(G)$ //random start city

2 $V \leftarrow getCitiesExceptStart(G)$

3 **while** V not empty:

4 $u \leftarrow$ most recently added vertex to U

5 Find vertex $v \in V$ with smallest cost u

6 $U.add(v)$

7 $V.remove(v)$

8 $findRoute(U, getStart(G))$

9 **return** U

And below the Java equvilent:

```
PUBLIC ARRAYLIST<STRING> SOLVE() throws NoSolutionException{

    //Initializing variables
    long startTime = System.nanoTime();
    long seed = System.nanoTime();
    Collections.shuffle(V,new Random(seed));
    try {           U.add(V.get(0));//random start
        V.remove(0); double distance = 0.0;
        while (!V.isEmpty()){//The algorithm
            String u = U.get(U.size()-1);
            Edge e = findMinVertex(graph.getCity(u),graph);
            graph.setRoadVisited(true,e);
```

```
                String nextCity = e.getTo();
                distance += e.getDistance();
                U.add(nextCity); V.remove(nextCity);
            }
            Edge road = findRouteToStart(U);
            U.add(road.getTo());



            distance += road.getDistance();
            U.add(""+((System.nanoTime()-startTime)*1.0e-9));
            System.gc();
            double usedMB = (Runtime.getRuntime().totalMemory() -
                    Runtime.getRuntime().freeMemory()) / 1024.0;
            U.add(""+(usedMB)); U.add(""+distance);
            return U;
        } catch(Exception e){
            throw new NoSolutionException(e.getMessage());
        }
    }
```

As the runtime of this algorithm is $O(n^2)$[7] with a space complexity of $O(n)$ and in most cases though it will be within 25% of the Held-Karp lower bound[7]. The obvious issue is that due to the greedy local property of this heuristic, the final edge that could be used could be arbitrarily long. Due to this there is not a guarentee for there to be a upper bound that is a multiple of a constant k of the optimal route. This is not optimal in terms of the distances the algorithm produces.

As you can see the implementation of Nearest Neighbor is relatively simple and

only requires a few additional helper functions including findMinVertex() and findRouteToStart(). For the purposes of focusing on the algorithm these functions are not shown above. The thing to also notice is that the only additional knowledge required for this implementation (aside from Java) is the use of a loop and some minor intuition. With this in mind it is nice to have a simple implementation of such an algorithm that can find some approximate solution very quickly. An additional part to keep in mind is that depending on the starting city for the tour (which is random) for the simplicity and testing purposes of the implementation, it is possible for several different routes to be produced. With a few simple tweaks the implementation could be altered to accommodate a specified starting city.

It should be noted in the data (to be provided) that the actual route and distances vary in both terms of the cities to be visited and in terms of the distance that it takes to travel. This while not optimal in terms of having a reliable answer is not necessarily an impairment to the approach as a whole. The reason for this logic is that the algorithm could be run multiple times (as is the case of testing) and keep track of the most efficient route found thus far, however in terms of the algorithm itself its benefits are that it is the simplest algorithm in terms of the concepts needed for the implementation as well as the amount of code for the implementation itself.

3.3 Greedy

The second tour construction heuristic algorithm that is being examined is also a greedy approach similar to the Nearest Neighbor algorithm but is called the Greedy heuristic. As the name suggests the Greedy heuristic uses a Greedy approach to constructing the tour of the traveling salesman. The idea is simple and intuitive, simply sort all of the edges and use the shortest edge to construct the tour as long as you don't

raise the degree of the vertexes more than two and that the edge doesn't make a cycle of less than N where N is the number of vertexes in the graph. The pseudo-code for the algorithm is displayed below.

GREEDYHEURISTIC(G):

```
1  E ← G.getRoads()
2  sort(E)
3  R ← []
4  N ← number of Vertices in G
5  while R.size <= N:
6          r ← E[0]
7          if r does not makeCycle(G, r) and does not raiseDegreeMoreThan2(G, r)
8                  R.add(r)
9          remove r from E
10 return findRoute(R)
```

The Java implementation:

PUBLIC ARRAYLIST$< String >$ SOLVE() throws NoSolutionException{

```
//Initializing variables
long startTime = System.nanoTime();
ArrayList¡String¿ result = new ArrayList¡String¿();
//sorting and ensuring there are no duplicates
Collections.sort(roads);
maxEdges = graph.getCities().keySet().size();
route = new ArrayList¡Edge¿();
//finding the route
```

```
        try{
            int numEdges = 0;
            while (numEdges < maxEdges){
                Edge road = roads.get(0);
                if (!moreThanTwoDegrees(road,route)
                    && !makesCycleLessThanN(road, route)){
                    pathGraph.addEdge(road);
                    route.add(road);
                    numEdges++;
                }
                roads.remove(0);
            }
            //obtaining the result
            double distance = findRoute(result, route);
            //final preparing of the data
            result.add(""+((System.nanoTime()-startTime)*1.0e-9));
            System.gc();
            double usedKB = (Runtime.getRuntime().totalMemory() -
                Runtime.getRuntime().freeMemory()) / 1024.0;
            result.add(""+usedKB);
            result.add(""+distance);
            return result;
            } catch(Exception e){
                throw new NoSolutionException(e.getMessage());
            }
    }
```

As can be seen above this is also a very intuitive construction heuristic based of the theory that using the shortest edges will help create the shortest route. However, like Nearest Neighbor, this algorithm has no guarantee of a upper bound on the distance returned though the Greedy algorithm averages within 15-20% of the Held-Karp lower bound[7].

Before moving forward with the analysis of this solution, it should be noted that there are two different graphs here. The first (named graph) is similar to the Nearest-Neighbor approach and is simply is the graph that the solver is working with. The second graph (named pathGraph) is just the vertexes of graph and is used to help simplify the examination of the edges added to the graph in the form of the functions makesCycle-LessThanN() and moreThanTwoDegrees(). These functions each take $O(n)$ time in the worst case scenario. The check for making a cycle is simply an iterative linear-search since the graph used (the pathGraph) only has the set of roads in the route making checking for a cycle simply going from one node to the next and seeing if we have seen it before. The process of checking if an edge would increase the degree to be greater than two also relies on the path graph and count the instances that a edge is being used to go to a node and from a node and if the sum of any of counts for a city are more than 2 then returning true. This means that the overall runtime of the algorithm is at most $O(EN)$ where E is the number of Edges that the graph has. Now I say this in the instance that there is no guarantee that in the number of cities N a solution will be found but in iterating over the Edges we find the solution doing $O(N)$ work from the previously mentioned functions for each Edge. It should be noted however, that the sorting of the edges is done int $O(nlogn)$ time and thus in this case is $O(ElogE)$ time and $O(E)$ space used. Therefore the entire algorithm runs in $O(ElogE+NE)$ time and $O(E+N)$ space.

From this approach a few things are especially reverent, the first is that since we

are looking at the same set of sorted roads each time the solver is run, while the start of the route may change, the amount of time, distance and space stays relatively static. This provides for a consistency guarantee that might be desirable in some instances. The amount of overhead knowledge that is required by this algorithm is also very small, all that is required for the helper functions is some knowledge of graph theory with some simple algorithms related to finding cycles. If you are also constructing the actual graph (as was done above) then this task become even simpler to accomplish. Overall, this is a simple algorithm to also implement but it should be no surprise that the developer would need to not only have the Edges implement a comparable interface but also ensure that the destructive behaviors of the algorithm do not affect the graph as a whole.

3.4 Minimum Spanning Tree

The final construction heuristic that this paper examines is the Minimum Spanning Tree heuristic. The basic approach is to using the graph make a minimum spanning tree and then create the tour from a preorder transversal of the tree. For some explanation, a minimum spanning tree is an acyclic graph connecting all of the vertexes with each of the vertexes having a degree no more than two. This is also an unintuitive algorithm since for most know do not study computer science, they would not necessarily think of creating a tree to make a cycle. The deceptively simple pseudo-code for the approach is below:

MSTHEURISTIC(G):

1 T $\leftarrow$ make a MST of G

2 **return** the inorder transversal of T

This is a little more complicated in practice as seen in the Java Implementation:

PUBLIC ARRAYLIST$< String >$ SOLVE() throws NoSolutionException {

```
long startTime = System.nanoTime();
ArrayList<String> V = new ArrayList¡String¿();
for (String v: graph.getCities().keySet())
    V.add(v);
try {           long seed = System.nanoTime();
    Collections.shuffle(V,new Random(seed));//random start
    //make the Tree
    Tree mst = makeMST(V.get(0));

    //get the result
    ArrayList<String> result = mst.treeWalk();

    //get the metrics
    result.add(result.get(0));
    double distance = getRouteDistance(result);
    result.add(""+((System.nanoTime()-startTime)*1.0e-9));
    System.gc();
    double usedKB = (Runtime.getRuntime().totalMemory() -
        Runtime.getRuntime().freeMemory()) / 1024.0;
    result.add(""+usedKB);
    result.add(""+distance);
return result;
```

```
        } catch(Exception e){
            throw new NoSolutionException(e.getMessage());
        }
    }
```

As stated before, this is not an intuitive heuristic; this approach involves a couple of more complex algorithms than in previous approaches. First of all this involves the makeMST() function. For the purposes of this paper, the algorithm used was based off of Primm's MST algorithm. Primm's MST algorithm is a greedy algorithm, and with the implementation of this algorithm, there is a slight difference with the original algorithm. The original algorithm uses a data structure called a heap used as well as an adjacency list for the graph so it runs in $O(ElogN)$ time, but the implementation that is used collects the edges on the frontier and sorts them. While this takes $O(NElogE)$ time, it is a slightly simpler implementation then the use of a heap and exacting the values no longer applicable from it when the frontier changes. The time $O(NElogE)$ also is the upper bound time for this algorithm since the transversal of the tree takes $O(n)$ time since an iterative preorder to save space on the virtual stack uses an iterative traversal.

The benefits to this heuristic is that there is a guarantee of a upper bound on the distance that the route produces of 2 times the optimal route. This is a nice change from the other heuristics that might have better run times because the other algorithms explained thus far do not have a guarantee making them potentially impractical for real time use. The second benefit is that if there are data structures and functions already in place that can be used such as a Tree, the function to make the MST, and the tree transversal function, then the actual code necessary is unbelievably straight forward (as the pseudo-code demonstrated). However as was said before there were a fair number of additional classes that were created so that this function was possible which make the implementation more complex than the previous heuristics.

3.5 Genetic Algorithms

The only tour optimization heuristic that this paper examines is the popular family of algorithms called genetic algorithms. These algorithms mimic the evolutionary process by having a population, a fitness characteristic, a breeding mechanism, and a chance at mutation within individuals. The basic concept is to implement each of the characteristics and at the end of a number of iterations pick the fittest example as the result. This while not the most intuitive algorithm also makes a measure of sense if one thinks about it briefly. The psuedo-code for the algorithm helps show this approach below:

GENETICHUERISTIC(G):

```
1  V ← G.vertices()
2  population ← several random orderings of V
3  for k times:
4          find the best canidates from population
5          breed those candates
6          randomly mutate some of the offspring
7  return the most fit member of the population
```

And the Java implementation below:

3.6 Christofides

CHAPTER 4

POSSIBLE IMPROVEMENTS

CHAPTER 5

RESULTS

CHAPTER 6

CONCLUSIONS

BIBLIOGRAPHY

[1] Codenotti B., Margara L. Heuristics for the traveling salesman problem *Internal note*. IEI-B4-14, 1991.

[2] M.L. Fredman, D.S. Johnson, L.A. McGeoch, G. Ostheimer. Data structures for traveling salesman. *Journal of Algorithms*, 18, 1995, pp. 432-479.

[3] C.-P. Hwang, B. Alidaee and J. D. Johnson. A Tour Construction Heuristic for the Traveling Salesman Problem. *The Journal of the Operational Research Society*, Vol. 50, No. 8 (Aug., 1999), pp. 797-809.

[4] Levin, A. and Yovel, U. 2-Opt heuristics for the traveling salesman problem. *Nonoblivious*, Networks 62: 201219, (2013), doi: 10.1002/net.21512.

[5] D. Gamboa, C. Rego, F. Glover. Implementation analysis of efficient heuristic algorithms for the traveling salesman problem. *Computers and Operations Research*, 33 (2006), pp. 11541172.

[6] N. Christofides, S. Eilon. Algorithms for large-scale traveling salesman problems. *Operations Research Quarterly*, 23 (1972), pp. 511518.

[7] Nilsson, Christian. Heuristics for the Traveling Salesman Problem. *Linkoping University*. http://web.tuke.sk/fei-cit/butka/hop/htsp.pdf.

[8] Ayudhya, Wichitsawat Suksawat Na and Scott E. Grasman. A new heuristic algorithm for the traveling salesman problem. *IIE Annual Conference.Proceedings: 1-6*, 2005.

[9] Algorithms used by Network Analyst. *ArcGIS Resource Center*, 2012. Web. http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html

[10] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. The Traveling Salesman Problem. 1985, John Wiley and Sons Ltd.

[11] Valenzuela, Christine L. and Jones, Antonia J. Evolutionary Divide and Conquer (I): a novel genetic approach to the TSP. *Evolutionary Computation*, 313-333, 1994. Web. http://citeseerx.ist.psu.edu