Extended Essay
Mathematics

Research Question:

How can we solve and analyse the Travelling Salesman Problem?

Word Count: 3998

Contents

Introduction

Optimizing time and energy is crucial in this industrious and technological era. Consequently, mathematics reigns supreme -- from mass production of energy, to downsizing phone batteries, complex mathematics eliminates waste.

An example of optimality is Travelling Salesman Problem (TSP). Originally formulated by Euler through The Knights Tour in Chess,[1] and further examined by Hamilton.[2] Studies by Karl Menger and George Dantzig recently using integer programming provided us with indispensable mathematics.[3]

TSP is about salesman needing to travel n cities and return to the initial city in the shortest distance or cost possible. This can be mathematically defined as Hamiltonian Cycle, which entails that in this undirected graph, every vertex is visited. I'll solve and analyse Symmetric TSP, where distance or cost, C_{ij}, is same from city $i - j$ and city $j - i$, otherwise the problem is Asymmetric.

TSP is valuable as it's very applicable. The general ideas and theory are useful in optimization, however there are also direct applications which strongly influence their fields of research.[4] These can range from easily understandable applications such as networking to highly specific and dissociated research areas like genome sequencing. The mathematics involved in TSP lends itself to problems in optimisation such as the assignment problem.

[1] Hülya Demez "Combinatorial Optimization: Solution Methods of Traveling Salesman Problem" Page 14 January 2013https://pdfs.semanticscholar.org/af88/88753dd016dce9b5277003a41ecbd6cf3e40.pdf
[2] Martin Grötschel "The Travelling Salesman Problem and its Applications" Page 17 September 2009 http://co-at-work.zib.de/berlin2009/downloads/2009-09-21/2009-09-21-1600-MG-TSP-and-Applications.pdf
[3] E.L. Lawler ... [et (1985). *The Traveling salesman problem : a guided tour of combinatorial optimization* (Repr. with corrections. ed.). Chichester [West Sussex]: Wiley. ISBN 0471904139.
[4] "Applications of the TSP" Accessed 4/6/2017 http://www.math.uwaterloo.ca/tsp/apps/index.html

This essay focuses on Combinatorial Optimisation, a branch of mathematics and computer science with close relation to operations research,[5] which reduces long, possibly complex problems to optimal solutions. An example of combinatorial optimization problem is Minimum Spanning Tree (MST) from Graph Theory, which I use in my research. Combinatorial Optimization utilizes Integer Programming, a necessary part in formulation of TSP, approximation algorithms (Heuristics) and other Branch-and-Cut algorithms. This essay shows an analysis of TSP.

[5] Schrijver, Alexander (February 1, 2006). A Course in Combinatorial Optimization https://homepages.cwi.nl/~lex/files/dict.pdf

Understanding Computational Complexity

An algorithmic approach to TSP is critical. Hence, understanding ideas behind Computational Complexity is necessary. We must optimize TSP as running a brute force algorithm (exhaustive search) would be impossible at a large number of cities, due to highly exponential growth in possible routes. For TSP to be applicable and useful, we need to devise ways in which we can obtain a suitable answer in a relatively appropriate time, so that in optimizing our routes or tasks, our process to obtain said optimized tasks is also optimized. To have a stronger understanding of this, we look at Time Complexity and Complexity Classes. However, since this is a background understanding for my EE, I've included explanations in Appendix.

Raw TSP has a time complexity of $O(n!)$, which is highly non-polynomial and therefore cannot be used to compute TSP solutions. That shows why TSP is classified as $NP - Hard$,[6] explaining the need for heuristics and lower time complexity algorithms.[7]

[6]Puget, J. *No, The TSP Isn't NP Complete (IT Best Kept Secret Is Optimization)*. [Accessed 4/6/2017]. https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en
[7] Stack Overflow "What are the differences between NP, NP-Complete and NP-Hard?" Accessed 4/6/2017 https://stackoverflow.com/questions/1857244/what-are-the-differences-between-np-np-complete-and-np-hard

Preliminary Work

Checking that the matrix satisfies triangle inequality:

$$C_{ij} \leq C_{ik} + C_{kj}$$

This makes solving TSP easier and more practical. My matrix which satisfies the triangle inequality is:

$$\begin{bmatrix} - & A & B & C & D & E \\ A & - & 11 & 6 & 9 & 8 \\ B & 11 & - & 8 & 10 & 9 \\ C & 6 & 8 & - & 12 & 7 \\ D & 9 & 10 & 12 & - & 10 \\ E & 8 & 9 & 7 & 10 & - \end{bmatrix}$$

This is a 5x5 matrix, therefore I'll be solving the TSP with an instance of 5 cities, A, B, C, D, E, or $1, 2, 3, 4, 5$. Testing to see that triangle inequality is satisfied:

$$C_{AB} \leq C_{AC} + C_{CB}$$

$$11 \leq 6 + 8$$

$$11 \leq 14$$

I'll formulate the problem using Integer Programming, getting an exact solution using a precise variant of a branch-and-cut algorithm, the Reduced Matrix Method. I shall look at heuristic approaches such as Nearest Neighbour Search and Twice Around the Tree method, finally, looking at a Brute Force Algorithm I programmed using Lexicographic Ordering.

Integer Programming Formulation

Integer programming is used when it's favourable to limit certain variables to being integers.[8] This proves as a comprehensive and useful mathematical model and TSP was formulated the same way in 1954 by Dantzig.[9] To formulate TSP, I'll be using zero-one integer-linear-programming, which means that variables are binary, which is:

$$x_{ij} = \begin{cases} 1 \; if \; edge(i,j) \; is \; included \; tour \\ 0 \qquad if \; not \; included \; in \; tour \end{cases}$$

Some necessary preliminary constraints are:

$$x_{ij} = x_{ji}$$

This makes sure that the graph is undirected. Another necessary constraint is:

$$x_{ii} = 0 \approx C_{ii} = \infty$$

This eliminates the possibility of going from one city to the same city, as it's possible for a subtour to emerge and this eliminates that possibility by setting cost to infinity. The main optimizing function for TSP is:

$$\sum_{(i,j)\in T} \sum C_{ij} x_{ij}$$

Here T is the variable for Tour, C is the cost variable, and x is the binary variable.

Some necessary arc constraints are added, so that the requirements for TSP are met:

$$\sum_{j=1}^{n} X_{ij} = 1 \; \forall \, i$$

[8] Beasely, J. *Integer programming*. http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html [Accessed 7/6/2017].
[9] Dupont, E. and Maes, C *The Travelling Salesman Problem with Integer Programming and Gurobi* http://examples.gurobi.com/traveling-salesman-problem [Accessed 7/6/2017].

This constraint makes sure each city is visited just once.

$$\sum_{i=1}^{n} X_{ij} = 1 \ \forall \, j$$

This constraint allows departure from each city once only. Consequently, each node is visited once, and one node is connected to another by only one arc. Subtours are possible, [10] as an optimised solution may be: $1 - 4 - 5$ and $2 - 3$. This leads to a disjoint tour. The requirements of TSP aren't met. A length 1 subtour consists of one node, a Length 2 subtour consists of 2 nodes and so on. Subtours can go from lengths 1 to $(n - 1)$, so maximum is Length 4. Hence, we have some subtour constraints. The L1 (Length 1) constraints I addressed, where cost matrix C_{ii} was set to infinity.

L2 Constraint: $x_{ij} + x_{ji} \leq 1$

This eliminates chance of a subtour between two nodes, as the x value between two nodes is 1 or 0, meaning there can be only one arc in the two nodes. Since there are 5 nodes, there are $\binom{5}{2}$ or 10 possibilities for Length 2 subtours.

L3 Constraint: $x_{ij} + x_{jk} + x_{ki} \leq 2$

This doesn't allow a secluded 3 node subtour, meaning a maximum of two arcs. There are $\binom{5}{3}$ or 10 possibilities for Length 3 subtours.

L4 Constraint: $x_{ij} + x_{jk} + x_{kl} + x_{li} \leq 3$

This doesn't allow a secluded 4 node subtour. There are $\binom{5}{4}$ or 5 possibilities for Length 4 subtours.

[10] Nptelhrd *Lec – 24 Traveling Salesman Problem (TSP)* Accessed 7/6/2017 https://www.youtube.com/watch?v=-cLsEHP0qt0&list=PL004010FEA702502F&index=24&t=817s

Noticeably, for Length k constraint: $x_{ij} + x_{jk} \dots\ k$. The number of possibilities for this subtour are $\binom{n}{k}$. This method of subtour elimination is inefficient, but it visually lays out constraints and shows which subtour is being eliminated at what level. This can be simplified.[11] If we successfully eliminate L1 subtours, then possibility of L4 subtours disappears. This is because we are dealing with 5 nodes and for a L4 subtour to exist, there must be leftover L1 subtour. Hence we see that if we successfully eliminate L2 subtours, we will eliminate L3 subtours subsequently.

Summing up:

$$x_{ij} = \begin{cases} 0 \\ 1 \end{cases}$$

$$x_{ij} = x_{ji}$$

$$\sum_{(i,j)\in T} \sum C_{ij} x_{ij}$$

Arc Constraints:

$$\sum_{j=1}^{n} X_{ij} = 1\ \forall\, i$$

$$\sum_{i=1}^{n} X_{ij} = 1\ \forall\, j$$

Subtour Elimination Constraints:

$$\text{L1: } x_{ii} = 0 \approx C_{ii} = \infty$$

$$\text{L2: } x_{ij} + x_{ji} \leq 1$$

[11] Nptelhrd *Lec – 24 Traveling Salesman Problem (TSP)* Accessed 7/6/2017 https://www.youtube.com/watch?v=-cLsEHP0qt0&list=PL004010FEA702502F&index=24&t=817s

Hungarian Assignment Reduced Matrix Method

This method is used to solve TSP in polynomial time and is a variant of branch-and-cut algorithms. It finds precise and accurate solutions, using penalties and minimizations.

The Hungarian Method solves another important problem in Combinatorial Optimisation known as the Assignment Problem, formed in the 19th century by Hungarian mathematicians. It's strongly polynomial and has a time complexity of $O(n^4)$.[12]

Methodically solving TSP using Hungarian Matrix Method.[13]

Initial matrix:

$$\begin{bmatrix} - & A & B & C & D & E \\ A & - & 11 & 6 & 9 & 8 \\ B & 11 & - & 8 & 10 & 9 \\ C & 6 & 8 & - & 12 & 7 \\ D & 9 & 10 & 12 & - & 10 \\ E & 8 & 9 & 7 & 10 & - \end{bmatrix}$$

The first step is row minimization. You subtract minimum element of each row by remaining elements. The first row the minimum element is 6, so we subtract 11, 9, and 8 by 6. This process is repeated for every row:

$$\begin{bmatrix} - & A & B & C & D & E \\ A & - & 5 & 0 & 3 & 2 \\ B & 3 & - & 0 & 2 & 1 \\ C & 0 & 2 & - & 6 & 1 \\ D & 0 & 1 & 3 & - & 1 \\ E & 1 & 2 & 0 & 3 & - \end{bmatrix}$$

Next, we do a column minimization, meaning that we subtract the minimum element of each column, by remaining elements, similarly to before:

[12] Topcoder.com *Assignment Problem and Hungarian Algorithm – topcoder*. https://www.topcoder.com/community/data-science/data-science-tutorials/assignment-problem-and-hungarian-algorithm [Accessed 7/6/2017].

[13] Happy Learning *Travelling Salesman Problem Minimizing Distance* Accessed 7/6/2017 https://www.youtube.com/watch?v=vNqE_LDTsa0

$$\begin{bmatrix} - & A & B & C & D & E \\ A & - & 4 & 0 & 1 & 1 \\ B & 3 & - & 0 & 0 & 0 \\ C & 0 & 1 & - & 4 & 0 \\ D & 0 & 0 & 3 & - & 0 \\ E & 1 & 1 & 0 & 1 & - \end{bmatrix}$$

The next step is to find the penalties for any zero in our matrix. Meaning we find the minimum element in its column and row and add them together, and assign it as a penalty to the associated zero. If we take the cost for $A - C$, which is a zero, we can see that the minimum element in its row is 1 and the minimum element in its column is 0. Hence penalty associated with the 0 at C_{AC} is 1. This process is repeated for every zero:

$$\begin{bmatrix} - & A & B & C & D & E \\ A & - & 4 & 0^1 & 1 & 1 \\ B & 3 & - & 0^0 & 0^1 & 0^0 \\ C & 0^0 & 1 & - & 4 & 0^0 \\ D & 0^0 & 0^1 & 3 & - & 0^0 \\ E & 1 & 1 & 0^1 & 1 & - \end{bmatrix}$$

Now, we must find highest penalty and strike it off. If there is more than one highest penalty, we can choose any one of them arbitrarily. Here, the highest penalty is at $A - C$, hence we can strike off row A and column C to give reduced matrix:

$$\begin{bmatrix} - & A & B & D & E \\ B & 3 & - & 0 & 0 \\ C & - & 1 & 4 & 0 \\ D & 0 & 0 & - & 0 \\ E & 1 & 1 & 1 & - \end{bmatrix}$$

Row A and the column C are no longer in the matrix. We can assume that one of the routes taken is $A - C$, and hence we put a dash through $C - A$ as the salesman cannot traverse that edge, as this would create a subtour. Row 4 does not have any zeroes, so repeat the row and column minimizations:

$$\begin{bmatrix} - & A & B & D & E \\ B & 3 & - & 0 & 0 \\ C & - & 1 & 4 & 0 \\ D & 0 & 0 & - & 0 \\ E & 0 & 0 & 0 & - \end{bmatrix}$$

Now calculate all penalties for every zero in the matrix.

$$\begin{bmatrix} - & A & B & D & E \\ B & 3 & - & 0^0 & 0^0 \\ C & - & 1 & 4 & 0^1 \\ D & 0^0 & 0^0 & - & 0^0 \\ E & 0^0 & 0^0 & 0^0 & - \end{bmatrix}$$

The highest penalty is 1 at $C - E$. We can strike off row C and column E. We can ascertain that one of the routes taken is $C - E$. The newly reduced matrix is as follows, and does not need to be minimized. Therefore, penalties have been calculated:

$$\begin{bmatrix} - & A & B & D \\ B & 3 & - & 0^3 \\ D & 0^0 & 0^0 & - \\ E & 0^0 & 0^0 & 0^0 \end{bmatrix}$$

Now we can see that highest penalty is 3 at $B - D$, this means we can strike off row B and column D, and know that one of our routes taken is $B - D$. To eliminate the chance of a subtour forming we put a dash through $D - B$.

$$\begin{bmatrix} - & A & B \\ D & 0^0 & - \\ E & 0^0 & 0^0 \end{bmatrix}$$

At this stage, it is possible to make any assignment from the 3 available zeroes. However, if we do $E - A$, we will be left with $D - B$, and since this is not possible as it would create a subtour, our two remaining tours are given to us as: $E - B$ and $D - A$, completing our assignments and solving the TSP.

From our calculations, we saw that the salesman has to go from:

A – C	C – E	B – D	E – B	D – A

Hence the route the salesman needs to follow to be most efficient is: A – C – E – B – D – A

Therefore, the total minimized distance can be calculated, using the values in the initial matrix:

6 + 7 + 9 + 10 + 9 = 41 units

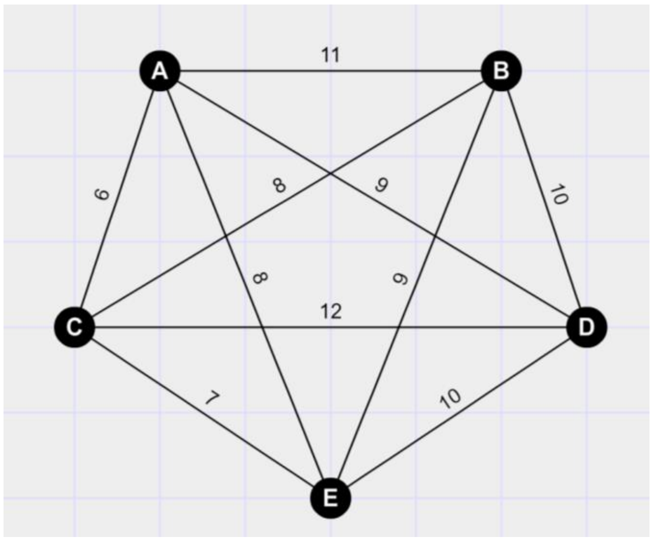
This is the complete reduced matrix method solution for the Travelling Salesman Problem. The algorithm is relatively slow and has many iterations, however it still runs in polynomial time and provides a confirmed accurate and precise solution.

Heuristics for TSP

A heuristic is defined as an algorithmic technique to find a near optimal solution. Although the solution may not be as precise as branch-and-cut method, it will still be reliable and near optimal.[14] A heuristic is quicker than most other precise algorithms and solvable in weakly polynomial time.

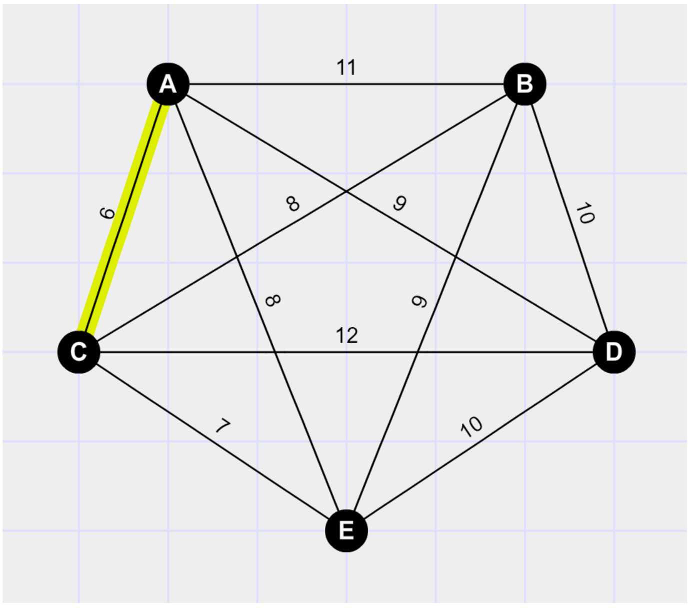
I'll solve TSP using two types of heuristic algorithms. One is a rudimentary heuristic, Nearest Neighbour Search (NNS) and the other's a more technical and precise technique, Twice Around the Tree (TATT).

NNS provides a feasible solution to the problem which acts as an Upper Bound (UB) to the optimal solution.[15] NNS runs relatively quickly, specifically in an $O(n^2)$ time complexity. To begin NNS algorithm, a visual representation of our cities is useful, and is as follows:

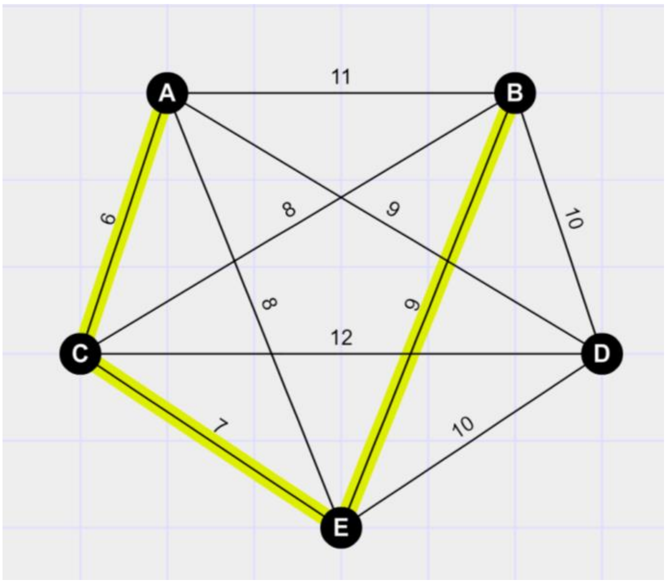


[14] Techterms.com *Heuristic Definition* https://techterms.com/definition/heuristic [Accessed 7 Sep. 2017].
[15] YouTube *Lec-26 Heuristics for TSP* https://www.youtube.com/watch?v=LjvdXKsvUpU [Accessed 7/6/2017].

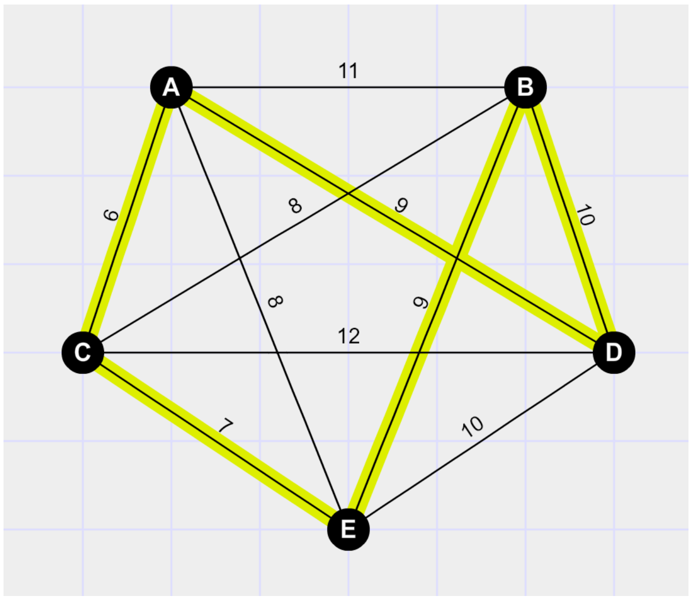
The nodes and edges are from our initial matrix. To begin the algorithm, arbitrarily choose any starting city, or node, and then traverse across the edge of the least cost. Choosing A as our starting node, the least cost arc is from $A - C$, hence we would traverse that arc.



From node C, next shortest edge is E with a cost of 7. From E, shortest edge is A, however this creates a subtour, therefore, next edge we traverse would be B, as this has the least cost of 9.



From node B, traverse to remaining node D, as any other step creates a subtour. From D we go to A, as the NNS algorithm and the TSP require that you go back to the initial node. Our final diagram is as follows:



Now that we have found a feasible route to take, we can add up the costs for every edge. The path we took was:

$A - C - E - B - D - A$

Therefore, the associated values are:

6 + 7 + 9 + 10 + 9 = 41 units

This solution is exactly the same as the solution we got with our more precise Hungarian Assignment Method. However, this is not always the case with Heuristic algorithms and is purely coincidental. I only came upon this solution as I arbitrarily chose A as my starting node. Had I chosen B as the starting node, the solution would've been:

$B - C - A - E - D - B$

This gives a cost sum of 42 units. Choosing C gives the same solution with a cost sum of 42 units and starting with nodes D and E give other solutions. Hence, we need to mathematically represent the relationship between the heuristic solution L_H, which acts as a UB, and the actual optimal solution L_O.

$$L_O \leq L_H$$

The optimal solution can never be more than heuristic solution. To get a more specific understanding of the heuristic, we examine its goodness, which arithmetically put is $\frac{L_H}{L_O}$. The goodness of this specific heuristic is given by:

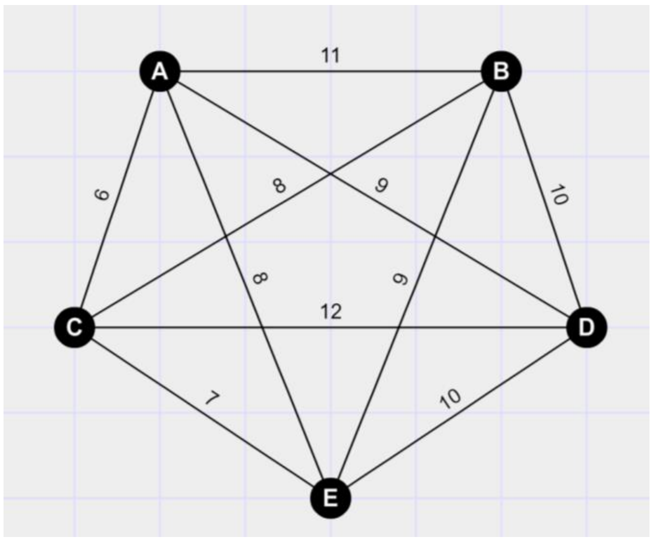
$$\frac{L_H}{L_O} \leq \frac{1 + \log_2 n}{2}$$

Here, n is the number of cities or nodes. The measure of goodness in this example is simply how much bigger the heuristic solution can be compared to the optimal solution, and the equation above gives us that very ratio. So, if we were dealing with an 8 city TSP, so $n = 8$, the ratio would be $\frac{L_H}{L_O} \leq 2$, meaning that the heuristic solution L_H can be up to 2 times as big as the optimal solution. Hence, in our NNS algorithm, where $n = 5$, the goodness ratio is $\frac{L_H}{L_O} \leq 1.66$. This ratio is a type of performance measure for the heuristic.

As the number of cities increases, the goodness of the algorithms decreases, and when dealing with higher city instances, this can lead to highly varied solutions. Hence, we must look at a heuristic algorithm which has a fixed goodness ratio of 2. This algorithm is called Twice Around the Tree (TATT).

This algorithm is more complex and combines other components from Combinatorial Optimisation and Graph Theory. Firstly, a Eulerian Circuit is one that requires you to traverse every edge and return to the initial vertex. A Hamiltonian Circuit is one that visits all the vertices and returns to the initial. The TSP is a Hamiltonian Circuit; however, the TATT method uses a Eulerian Circuit with a Minimum Spanning Tree (MST).

A spanning tree is a subgraph of an undirected single connected component graph G. It has every vertex from G, just not every edge. Therefore, a MST is a spanning tree with minimum weight. Meaning that the edges in the spanning tree have the least possible sum of any spanning tree from the graph G.[16] In this case, G is the overall set of nodes and vertices:



The first step in the TATT algorithm is to get the MST for the above graph G. I shall be using Prims Algorithm to get MST.[17] Prims algorithm uses a greedy approach to find the MST, meaning that at each iteration, it adds the least cost vertex to the spanning tree. The algorithm is quite efficient as it runs in logarithmic time complexity.

Firstly, find the minimum weight edge. This is $A - C$, as it has a weight of 6 units. We make two sets to compare values from. The first set, is that of elements already added to our spanning tree, which at this point is only A and C. The other set is of the other 3 remaining elements.

$$\{A\ C\}$$

[16] HackerEarth. *Minimum Spanning Tree Tutorials & Notes | Algorithms | HackerEarth*. https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial [Accessed 7/6 2017].
[17] YouTube *Lec-19 Network Models*. https://www.youtube.com/watch?v=Dmw3OsyT5yw [Accessed 7/6/2017].

$$\{B\ D\ E\}$$

We will now use these two sets to get the other edges in our spanning tree. If we take vertex A from set 1 and vertex B from set 2, notice the edge joining them is of weight 11. Next, we look at the edge $A - D, A - E$, keeping the minimum edge weight in mind. Then we move onto the next element in set 1, and repeat, looking at edges $C - B, C - D, C - E$. Finally, this tells us that the minimum weight edge sorted in such manner is $C - E$, which is of weight 7. Hence, add E to set 1:

$$\{A\ C\ E\}$$

$$\{B\ D\}$$

Now we repeat the process. This means we look at the following edges: $A - B, A - D, C - B, C - D, E - B, E - D$. From this, we learn that $C - B$, has the lowest weight, of 8. Now we add B to our first set, which leaves only D in the second set. Making our set comparisons, the next lowest edge is $A - D$. This means that we have found our MST. It is as follows:

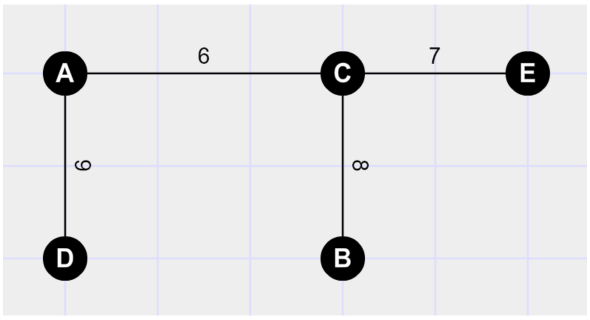
$A - C$

$C - E$

$C - B$

$A - D$

This can be graphically represented as:



This MST satisfies the theoretical requirements, meaning that all vertices, n, are included and $n - 1$, in this case 4, edges are in the MST. It is a connected subgraph from the original graph G. The length of our MST, $L_{MST} = 30$. To mathematically show that the L_{MST} is a lower bound (LB) for the optimal solution L_O, we imagine that the optimal solution, which has n vertices and edges, has one edge removed, making it a spanning tree, we will get a new length, L_{ST}.

Now given:

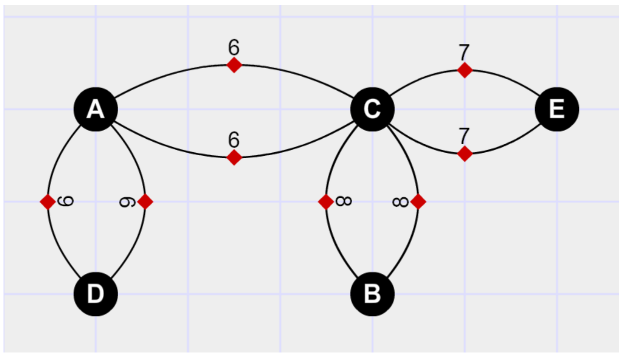
$$C_{ij} \geq 0$$

$$L_{ST} \leq L_O$$

$$L_{MST} \leq L_{ST} \leq L_O$$

$$L_{MST} = 30 \leq L_O$$

Therefore, the MST is a LB for the optimal solution of the TSP. However, to get more useful solutions using the TATT heuristic, we can still expand further. Next, take the MST and create an Eulerian Graph from it. To make this happen, all the vertices need to have an even degree, meaning each vertex has an even number of edges coming off. Duplicating the number of edges creates an Eulerian Circuit (or graph).



The length of the Eulerian Circuit, L_E, is simply double the L_{MST}, making it 60. Therefore, one possible Eulerian Circuits is: $A - C - E - C - B - C - A - D - A$. By extracting a Hamiltonian circuit from our Eulerian circuit we can find useful heuristic solutions of length L_H. We need to extract a single component connected graph, which visits all the vertices once and comes back to the initial vertex. To get this more refined circuit we follow our Eulerian circuit, abstaining from creating any subtours. This means we cannot go to the same node more than once. Demonstratively, we take our original circuit $A - C - E - C - B - C - A - D - A$. For the heuristic, we add the $A - C - E$ edges without going back to C afterwards, as we cannot create subtours and hence cannot have repeated nodes. Therefore, our Hamiltonian path becomes $A - C - E - B - D - A$. Notice that the third repeated C, the second A nodes were not included and the circuit ends on the initial node A. $A - C - E - B - D - A$ is a heuristic solution to the TSP, and has a length, $L_H = 41$. Having already found the solution using the precise Hungarian Assignment Method, we know this to be the exact solution to the TSP, proving the usefulness of the heuristic algorithm. The heuristic solution we just found, is one of many available ones from our Eulerian circuit. Another one is $A - E - B - C - D - A$, where $L_H = 46$. There is an array of heuristic solutions to pick from the Eulerian circuit, moreover, we can tell that there is a variety of Eulerian circuits we can make from the graph above. Regardless, we arrive at a feasible heuristic solution, L_H, efficiently, as the heuristic does run in polynomial time.

The final step is to analyse the goodness of this heuristic, and to arrive at a ratio for $\frac{L_H}{L_O}$. A preliminary requirement is that our initial cost matrix satisfies triangle inequality so that in all cases $L_H \leq L_E$. Then we can obtain a goodness ratio as follows:

$$L_{MST} \leq L_O$$

$$L_E = 2L_{MST}$$

$$L_H \leq L_E$$

$$L_H \leq 2L_{MST}$$

$$L_H \leq 2L_O$$

$$\frac{L_H}{L_O} \leq 2$$

Hence, we can see that the TATT heuristic algorithm has a constant goodness ratio, or performance measure, of 2, meaning that the heuristic will be within 2 times the optimal solution to the TSP, regardless of size. The TATT heuristic is a construction heuristic, as we had to construct the MST and work off of it.

This concludes my look at the heuristic approaches for the TSP. The heuristics were considerably easier to solve than the longer Hungarian method. The NNS algorithm could even be solved visually and even gave the correct solution to the TSP, as did the TATT heuristic, though this was coincidental. We can tell that the heuristic algorithms perform very satisfactorily and the trade-off for computing time in exchange for performance uncertainty seems very worth it.

Brute Force Algorithm Using Lexicographic Ordering

I thought running a brute force algorithm would be useful to see how the raw checking-every-possibility pathway leads us into problems with time complexity. Since there is no pre-processing phase, there is a constant need for extra space as the values in the problem increase and an exponentially increasing run-time. This type of exhaustive approach is highly inefficient.[18]

The TSP is an NP – Hard problem, meaning that an exhaustive algorithm will definitely not run in polynomial time. The number of possible routes from an n city problem is $n!$. Hence the time complexity for an exhaustive algorithm is also $O(n!)$, which is extremely high and cannot be run by computers, certainly for higher city instances.[19]

I wrote the algorithm myself, however to generate every single permutation, I needed to look into more advanced algorithmic approaches. Therefore, I used Lexicographic Ordering which allowed me to successfully generate all the possible permutations, meaning routes in this case.[20] After calculating the permutations, I used my own methods to find the shortest route. As for the Lexicographic Ordering algorithm, the method I found was highly complex. Our initial permutation is $1 - 2 - 3 - 4 - 5$, to go to the next, we need to follow these steps:

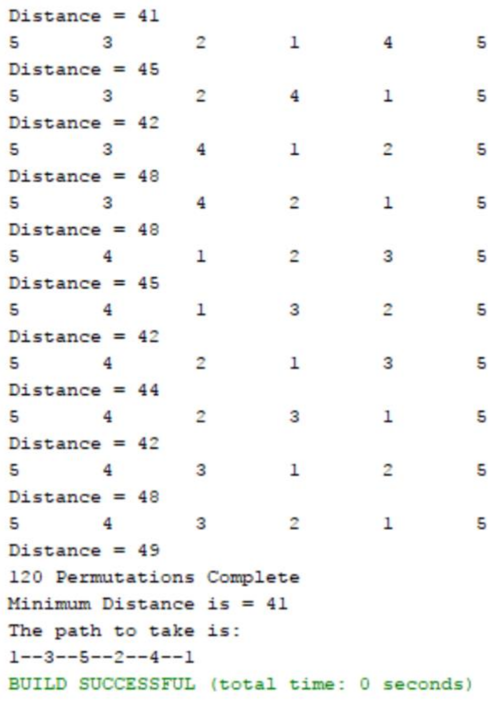
1. Find the largest index i, such that the $C[i] < C[i + 1]$. This means that find the largest index such that the value of the index after is bigger.
2. Find the largest index j, such that $C[i] < C[j]$. This step asks us to find the biggest index that represents a value bigger than the value in index i.

[18] *What exactly is the brute force algorithm* https://stackoverflow.com/questions/8103050/what-exactly-is-the-brute-force-algorithm [Accessed 9/6/2017].
[19] *Brute force algorithm*. http://www-igm.univ-mlv.fr/~lecroq/string/node3.html [Accessed 9/6/2017].
[20]*How would you explain an algorithm that generates permutations using lexicographic ordering?* https://www.quora.com/How-would-you-explain-an-algorithm-that-generates-permutations-using-lexicographic-ordering Accessed 9/6/2017

3. Now we have to swap the values of indexes i and j.
4. The final step is to take the entire permutation, from index i until the end, and reverse all the values in specifically that portion

Clearly the process to generate permutations is not easy, however I was able to successfully program this. The programming techniques incorporated into this include but are not limited to: one-dimensional and two-dimensional arrays, recursive programming, multiple looping, and more. The code is in the Appendix.

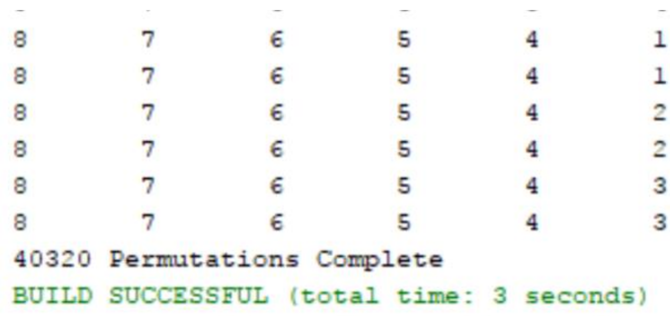
Upon execution, I got the following results:



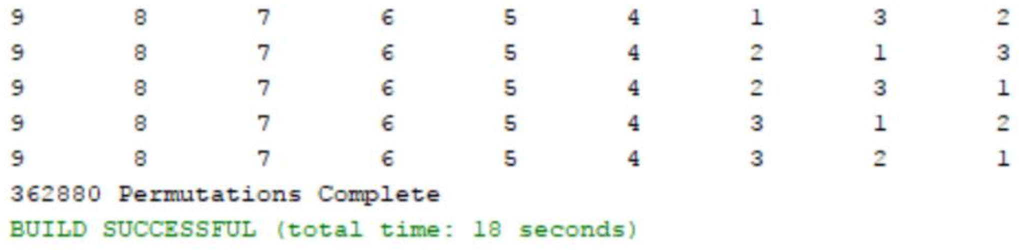
```
Distance = 41
5       3       2       1       4       5
Distance = 45
5       3       2       4       1       5
Distance = 42
5       3       4       1       2       5
Distance = 48
5       3       4       2       1       5
Distance = 48
5       4       1       2       3       5
Distance = 45
5       4       1       3       2       5
Distance = 42
5       4       2       1       3       5
Distance = 44
5       4       2       3       1       5
Distance = 42
5       4       3       1       2       5
Distance = 48
5       4       3       2       1       5
Distance = 49
120 Permutations Complete
Minimum Distance is = 41
The path to take is:
1--3--5--2--4--1
BUILD SUCCESSFUL (total time: 0 seconds)
```

The brute force algorithm gives us $1 - 3 - 5 - 2 - 4 - 1$, which is the same as $A - C - E - B - D - A$. This was one of 120 possible solutions as that is 5!. We can see that in this small sample of solutions, many are the same distance. This is because, even though there are $n!$ number of possible solutions, there are $(n-1)!$ possible unique solutions.

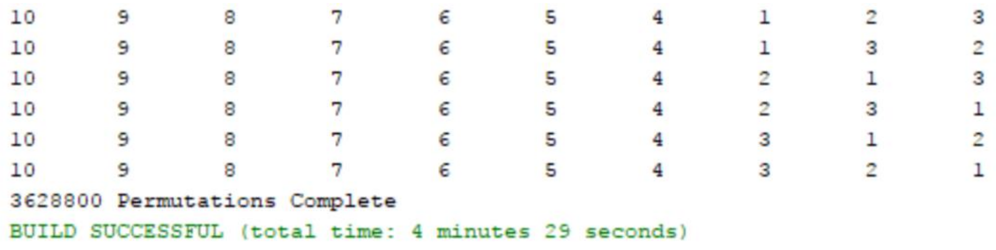
A final notable deduction is that the computational time taken to get this answer was 0 seconds. This seems as though the brute force algorithm is highly efficient and very practical. However, to prove it's very much the opposite and becomes impossible to compute, I decided to modify my program only to compute all the permutations rather than deal with the distances and other calculations. My results from this test can be seen below:



```
8        7        6        5        4        1        2        3
8        7        6        5        4        1        3        2
8        7        6        5        4        2        1        3
8        7        6        5        4        2        3        1
8        7        6        5        4        3        1        2
8        7        6        5        4        3        2        1
40320 Permutations Complete
BUILD SUCCESSFUL (total time: 3 seconds)
```



```
9        8        7        6        5        4        1        3        2
9        8        7        6        5        4        2        1        3
9        8        7        6        5        4        2        3        1
9        8        7        6        5        4        3        1        2
9        8        7        6        5        4        3        2        1
362880 Permutations Complete
BUILD SUCCESSFUL (total time: 18 seconds)
```



```
10       9        8        7        6        5        4        1        2        3
10       9        8        7        6        5        4        1        3        2
10       9        8        7        6        5        4        2        1        3
10       9        8        7        6        5        4        2        3        1
10       9        8        7        6        5        4        3        1        2
10       9        8        7        6        5        4        3        2        1
3628800 Permutations Complete
BUILD SUCCESSFUL (total time: 4 minutes 29 seconds)
```

I decided to jump to an 8-city instance and noted the first increase in time to 3 seconds, which is still quite insignificant. At a 9-city instance there was an exponential jump to 18 seconds, however this was still only a small inconvenience. It was at the 10-city instance where there was a massive jump in time taken. Upon attempting an 11-city instance I gave up as it took too long. This simple demonstration of the highly non-polynomial nature of the raw time complexity of TSP shows that brute force algorithm is an impossible approach for today's computers.

Conclusion

Having successfully examined and analysed various possible methods of solution for TSP, I am astonished by how such a seemingly trivial problem can be filled with such a deep and broad field of study.

This problem is not solely about optimising route, but about optimizing method of obtaining optimized route. TSP shows how to efficiently solve problems and portrays the usefulness of computer complexity. This essay analysed trade-off between solution and formulation time and accuracy and precision of answer. Finding the sweet spot between these two categories can lead to efficient and optimised solution. After fully formulating TSP using integer programming, I initially looked at a long linear programming solution called Hungarian Assignment Method. Although this gave a definite and confirmed answer, its time complexity was strongly polynomial making it hard for quick and easy use. Looking at the Heuristics, highly efficient methods which ran in light polynomial time and still provided feasible solutions. The NNS, although it lost its reliability as the n-city instance increased, the fact that an extremely quick almost visual method was available for such a complex problem was astonishing. The TATT heuristic was more complicated but the performance measure was fixed and the refined method provided a consistent reasonable and accurate solution. To conclude, I programmed a brute force method to look at TSP in raw state and understand how a simplistic exhaustive method can be useless and wasteful at higher city instances.

Overall, TSP is a crucial element of optimisation which is further generalised amongst many more advanced problems and ideas in applied mathematics. This problem has shown me a new depth to mathematical optimisation and in its practicality.

Bibliography

"Applications of the TSP" Accessed 4/6/2017
http://www.math.uwaterloo.ca/tsp/apps/index.html

Beasely, J. *Integer programming*. http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html [Accessed 7/6/2017].

Brute force algorithm. http://www-igm.univ-mlv.fr/~lecroq/string/node3.html [Accessed 9/6/2017].

Dupont, E. and Maes, C *The Travelling Salesman Problem with Integer Programming and Gurobi* http://examples.gurobi.com/traveling-salesman-problem [Accessed 7/6/2017].

E.L. Lawler ... [et (1985). *The Traveling salesman problem : a guided tour of combinatorial optimization* (Repr. with corrections. ed.). Chichester [West Sussex]: Wiley. ISBN 0471904139.

HackerEarth. *Minimum Spanning Tree Tutorials & Notes | Algorithms | HackerEarth*. https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial [Accessed 7/6 2017].

Happy Learning *Travelling Salesman Problem Minimizing Distance* Accessed 7/6/2017 https://www.youtube.com/watch?v=vNqE_LDTsa0

How would you explain an algorithm that generates permutations using lexicographic ordering? https://www.quora.com/How-would-you-explain-an-algorithm-that-generates-permutations-using-lexicographic-ordering Accessed 9/6/2017

Hülya Demez "Combinatorial Optimization: Solution Methods of Traveling Salesman Problem" Page 14 January 2013
https://pdfs.semanticscholar.org/af88/88753dd016dce9b5277003a41ecbd6cf3e40.pdf

Martin Grötschel "The Travelling Salesman Problem and its Applications" Page 17 September 2009 http://co-at-work.zib.de/berlin2009/downloads/2009-09-21/2009-09-21-1600-MG-TSP-and-Applications.pdf

Puget, J. *No, The TSP Isn't NP Complete (IT Best Kept Secret Is Optimization)*. [Accessed 4/6/2017]. https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en

Schrijver, Alexander (February 1, 2006). A Course in Combinatorial Optimization https://homepages.cwi.nl/~lex/files/dict.pdf

Stack Overflow "What are the differences between NP, NP-Complete and NP-Hard?" Accessed 4/6/2017 https://stackoverflow.com/questions/1857244/what-are-the-differences-between-np-np-complete-and-np-hard

Techterms.com *Heuristic Definition* https://techterms.com/definition/heuristic [Accessed 7 Sep. 2017].

Topcoder.com *Assignment Problem and Hungarian Algorithm – topcoder*. https://www.topcoder.com/community/data-science/data-science-tutorials/assignment-problem-and-hungarian-algorithm [Accessed 7/6/2017].

What exactly is the brute force algorithm https://stackoverflow.com/questions/8103050/what-exactly-is-the-brute-force-algorithm [Accessed 9/6/2017].

YouTube Nptelhrd *Lec-19 Network Models*. https://www.youtube.com/watch?v=Dmw3OsyT5yw [Accessed 7/6/2017].

YouTube Nptelhrd *Lec – 24 Traveling Salesman Problem (TSP)* Accessed 7/6/2017 https://www.youtube.com/watch?v=-cLsEHP0qt0&list=PL004010FEA702502F&index=24&t=817s

YouTube Nptelhrd *Lec-26 Heuristics for TSP* https://www.youtube.com/watch?v=LjvdXKsvUpU [Accessed 7/6/2017].

Appendix

Introduction:

Integer Programming – A program in which all or some variables are restricted to integers. Typically for optimisation or feasibility purposes.

Undirected Graph – A graph where all edges can be traversed without any restrictions in direction.

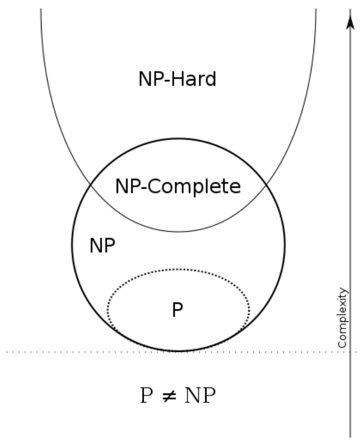
Genome Sequencing – The process of DNA ordering and determining of an organism's genome.

Assignment Problem – A fundamental problem in Combinatorial Optimisation, which consists of finding an optimised weight matching in a weighted bipartite graph.

Graph Theory – A branch of mathematics which consists of studying graphs that represent a relationship in the form of nodes or vertices connected by edges or arcs.

Understanding Computational Complexity

Note that some preliminary understandings are that: In this topic, decision problems are specifically defined as problems which have a Boolean answer, meaning that it can be solely true or false. Another important understanding is that of the concept of Reduction. This is the process by which we can convert an unsolvable problem A, into a solvable problem B, in polynomial time, which gives an algorithmic time complexity $O(n^k)$ where $k > 0$. These are important as algorithms which run in polynomial time can be run by computers. Algorithms of a higher time complexity such as $O(n!)$, the time complexity of a brute force method for TSP, cannot be run in polynomial time, and hence cannot be feasibly run by current computers. For algorithms to run in polynomial time is important, hence any decision problems which are *solved* in polynomial time are put in complexity class P. Next, we have the class NP, this is the set of *verifiable* solutions to decision problems, in polynomial time. Whether or not the problems in this class can be solved in polynomial time is unknown. I'll assume that $P \neq NP$, meaning that problems in NP, cannot be solved in polynomial time. The next complexity class to look at is $NP - Complete$, the set of all problems which all problems in NP can be reduced to. The final class is $NP - Hard$, the set of all problems to which any $NP - Complete$ can be reduced to, in polynomial time. These problems needn't be decision problems and are considered to be at least as hard as the problems in NP.



Preliminary Work:

Lexicographic Ordering – The generalization of a set of numbers or letters to create every possible variant.

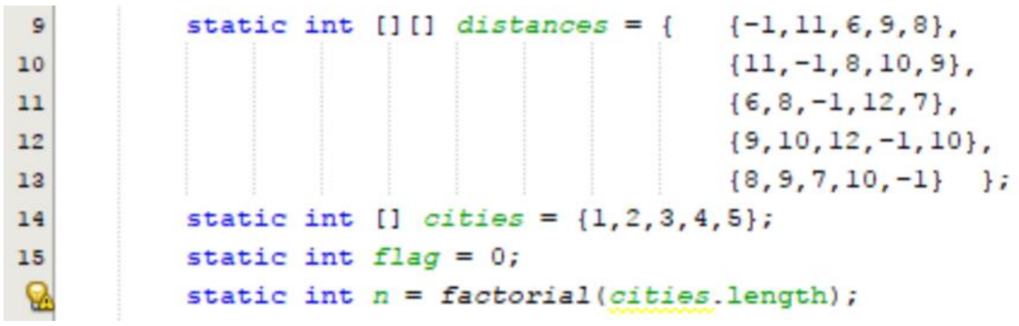
Triangle Inequality – States that the length of two sides of a triangle should always add up to be greater than the third side. Can be incorporated into nodes in a graph.

Hungarian Assignment Reduced Matrix Method:

Penalties – These are temporary values assigned to any number in the matrix which help in the reduction process.

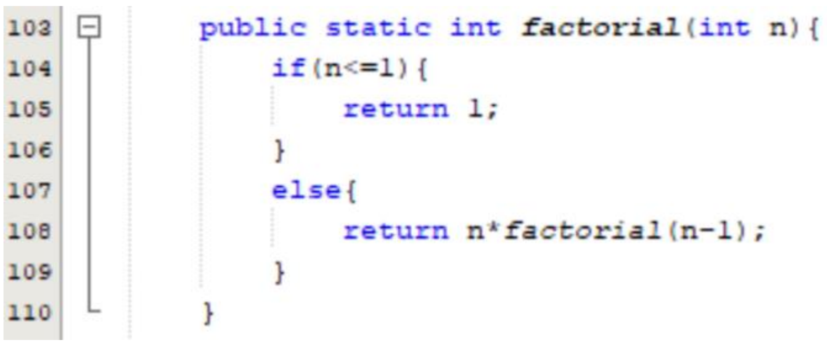
Brute Force method with Lexicographic Ordering

Index - An index is a sort of temporary value used to mark the spot of each element in the permutation.



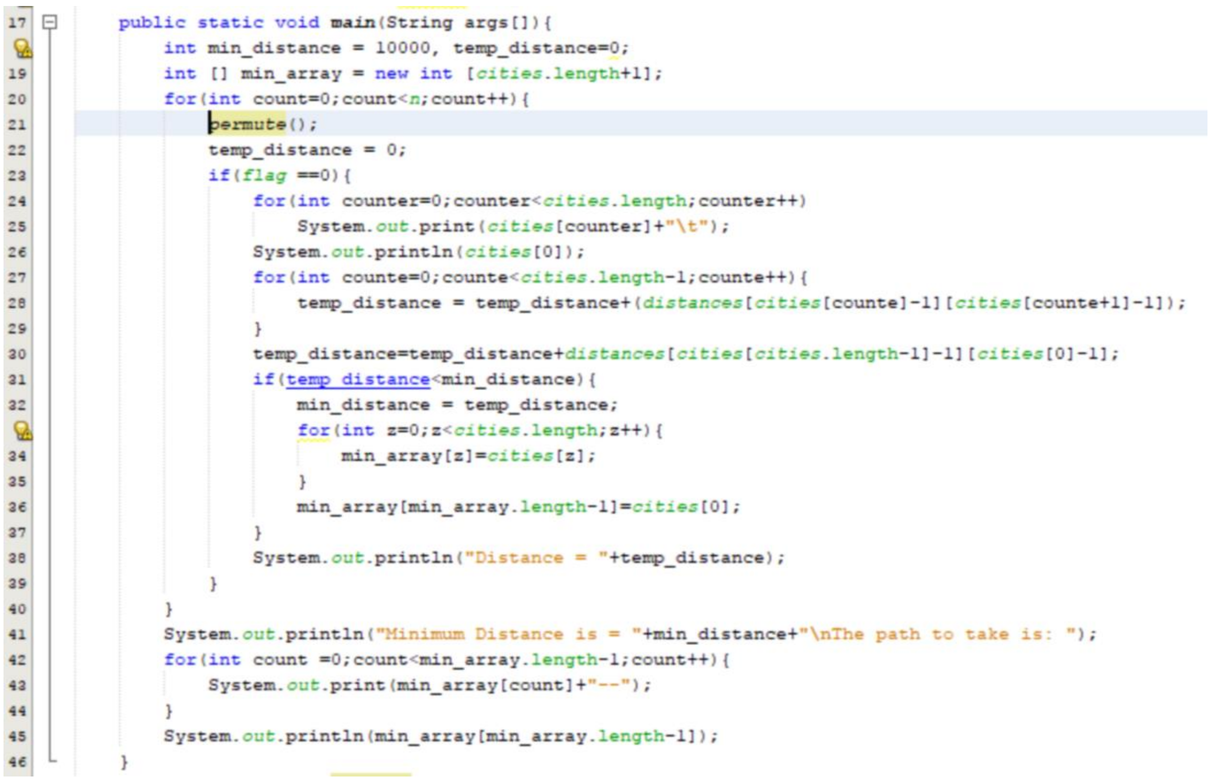
```
static int [][] distances = {   {-1,11,6,9,8},
                                {11,-1,8,10,9},
                                {6,8,-1,12,7},
                                {9,10,12,-1,10},
                                {8,9,7,10,-1}   };
static int [] cities = {1,2,3,4,5};
static int flag = 0;
static int n = factorial(cities.length);
```

This is the beginning of the program, here I initialised my distance or cost matrix, which is the same as the one used throughout this paper. I also initialised an array to hold the number of cities in the running instance and a flag variable. The variable n is used to calculate how many permutations will be run, I use a self-written recursive factorial function which is as follows:



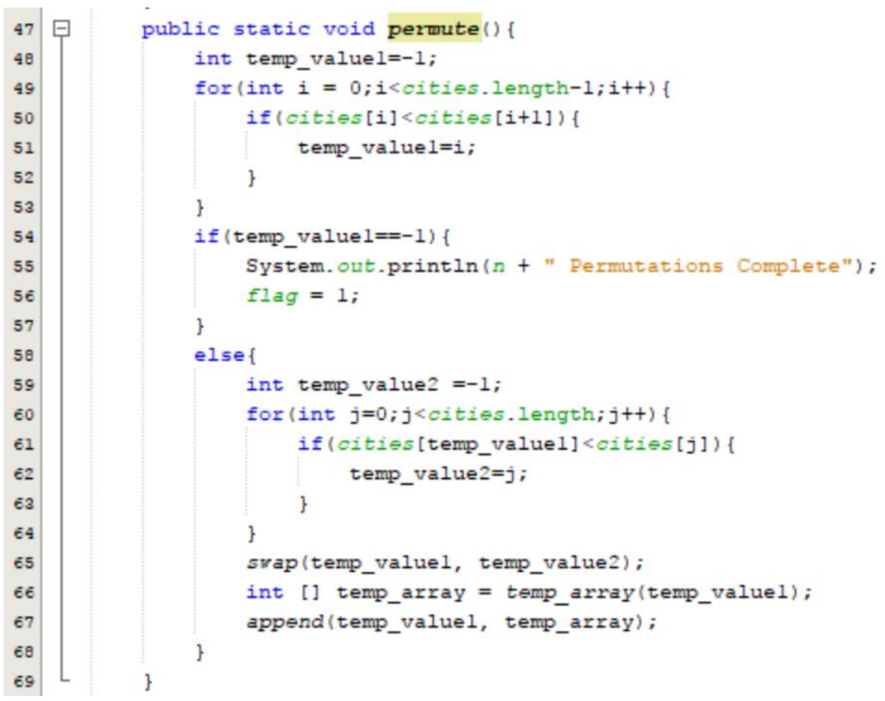
```
public static int factorial(int n){
    if(n<=1){
        return 1;
    }
    else{
        return n*factorial(n-1);
    }
}
```

Next, we have the main method for this programme. In this section, the routes (permutations) are all output to the main console one by one. The distance for each one of those routes is also outputted and there are checks in place to see if it the minimal distance so far. At the end, the minimal distance and the corresponding route is outputted. The highlighted function $permute()$ calculates every next permutation using lexicographic ordering,



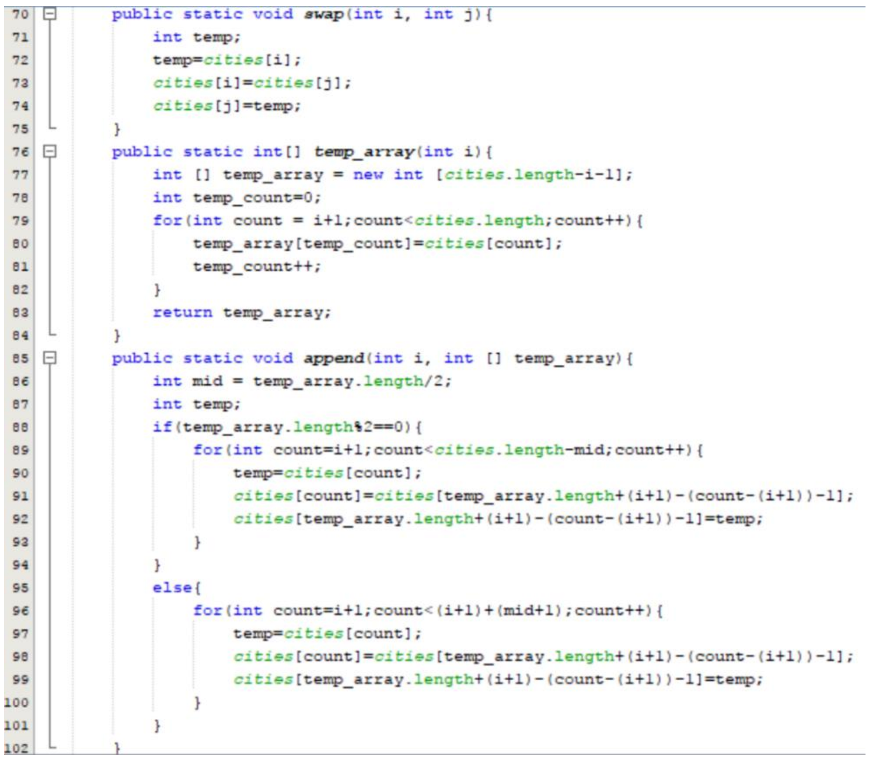
```java
public static void main(String args[]){
    int min_distance = 10000, temp_distance=0;
    int [] min_array = new int [cities.length+1];
    for(int count=0;count<n;count++){
        permute();
        temp_distance = 0;
        if(flag ==0){
            for(int counter=0;counter<cities.length;counter++)
                System.out.print(cities[counter]+"\t");
            System.out.println(cities[0]);
            for(int counte=0;counte<cities.length-1;counte++){
                temp_distance = temp_distance+(distances[cities[counte]-1][cities[counte+1]-1]);
            }
            temp_distance=temp_distance+distances[cities[cities.length-1]-1][cities[0]-1];
            if(temp_distance<min_distance){
                min_distance = temp_distance;
                for(int z=0;z<cities.length;z++){
                    min_array[z]=cities[z];
                }
                min_array[min_array.length-1]=cities[0];
            }
            System.out.println("Distance = "+temp_distance);
        }
    }
    System.out.println("Minimum Distance is = "+min_distance+"\nThe path to take is: ");
    for(int count =0;count<min_array.length-1;count++){
        System.out.print(min_array[count]+"--");
    }
    System.out.println(min_array[min_array.length-1]);
}
```

The highlighted function $permute()$ calculates every next permutation using lexicographic ordering, and it can be seen as follows:



```java
public static void permute(){
    int temp_value1=-1;
    for(int i = 0;i<cities.length-1;i++){
        if(cities[i]<cities[i+1]){
            temp_value1=i;
        }
    }
    if(temp_value1==-1){
        System.out.println(n + " Permutations Complete");
        flag = 1;
    }
    else{
        int temp_value2 =-1;
        for(int j=0;j<cities.length;j++){
            if(cities[temp_value1]<cities[j]){
                temp_value2=j;
            }
        }
        swap(temp_value1, temp_value2);
        int [] temp_array = temp_array(temp_value1);
        append(temp_value1, temp_array);
    }
}
```

Here I find the previously stated values i and j, and terminate beforehand if all the permutations have already been found. At the end, I do the 3rd and 4th step in the lexicographic ordering , the swap and reversal, also using self-written programs, which can be seen as follows:



```
    public static void swap(int i, int j){
        int temp;
        temp=cities[i];
        cities[i]=cities[j];
        cities[j]=temp;
    }
    public static int[] temp_array(int i){
        int [] temp_array = new int [cities.length-i-1];
        int temp_count=0;
        for(int count = i+1;count<cities.length;count++){
            temp_array[temp_count]=cities[count];
            temp_count++;
        }
        return temp_array;
    }
    public static void append(int i, int [] temp_array){
        int mid = temp_array.length/2;
        int temp;
        if(temp_array.length%2==0){
            for(int count=i+1;count<cities.length-mid;count++){
                temp=cities[count];
                cities[count]=cities[temp_array.length+(i+1)-(count-(i+1))-1];
                cities[temp_array.length+(i+1)-(count-(i+1))-1]=temp;
            }
        }
        else{
            for(int count=i+1;count<(i+1)+(mid+1);count++){
                temp=cities[count];
                cities[count]=cities[temp_array.length+(i+1)-(count-(i+1))-1];
                cities[temp_array.length+(i+1)-(count-(i+1))-1]=temp;
            }
        }
    }
```