2018

A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem

Chatting, M.

Chatting, M. (2018) 'A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem', The Plymouth Student Scientist, 11(2), p. 53-91.
http://hdl.handle.net/10026.1/14184

The Plymouth Student Scientist
University of Plymouth

*The Plymouth Student Scientist, 2018, **11**, (2), 53-91*

A Comparison of Exact and Heuristic Algorithms to Solve the Travelling Salesman Problem

Matthew Chatting

Project Advisor: Dr. Matthew Craven, School of Computing, Electronics and Mathematics, Plymouth University, Drake Circus, Plymouth, PL4 8AA

Abstract

This article provides an overview to the Travelling Salesman Problem (TSP) and the relevant aspects of graph theory and computational complexity theory to understand the TSP. Beyond this, two algorithms capable of solving the TSP are introduced, explained and analysed in terms of their efficiency. The first algorithm is a random search heuristic 'hill climber' and the second is an exact 'branch and bound' algorithm. The benefits and drawbacks of each algorithm are discussed alongside their performance on distinct instances of the TSP. Consideration is also given to current research on the factors contributing to the complexity of instances.

1 Introduction

The Travelling Salesman Problem

In one form or another, the Travelling Salesman Problem (TSP) has been considered throughout most of human history and represents the concept of finding the shortest tour around a set of locations, finishing at the original location and only visiting each location once [6].

More commonly, the TSP has been thought of as a salesman visiting a number of cities. So, for the sake of simplicity throughout this article, the locations shall be referred to as cities. The TSP is relatively simple and can be solved by hand when considering a small set of cities. However, the complexity of the TSP escalates as more cities are considered, making *brute force*, trying every possible solution, inefficient [11]. Hence, it is necessary to utilise logical methodologies to solve the problem.

As the objective of the TSP is to find the shortest tour around a set of cities, a methodology of calculating the *cost* of a tour must be defined. The cost may be the geographical distance between two cities or some other measurement relative to the problem. As such, each *instance* (unique problem) of the TSP has an associated method for calculating the cost between each pair of cities.

A TSP instance is classified as either asymmetric or symmetric. If an instance is asymmetric it indicates that the cost between each city pair is dependent on the direction of travel. Meaning, the distance from city i to j may be different to the distance from city j to i. Conversely, a symmetric instance means that the distance between each pair of cities is the same regardless of the direction of travel. The complexity of a symmetric instance, for n cities, can be quantified by it having $(n-1)!$ feasible tours, as demonstrated in Table 1. Note that all instances included in this article are symmetric, and more detail can be found on each in Section 4.

Table 1: No. of feasible solutions relative to the no. of cities

n	1	2	3	...	10	20	30	40
# Sols	1	1	2	...	$362,880$	1.22×10^{17}	8.84×10^{30}	2.04×10^{46}

Graph Theory

Instances of the TSP can be modelled using the principles of graph theory; the problem is viewed as an undirected weighted graph where cities are vertices, city connections are edges and the objective is to find the optimum valid tour. For a tour to be valid it must visit every city only once, except for the original city which is also the final city. So, the set of feasible solutions is all possible Hamiltonian cycles within the graph and the optimum solution is the shortest Hamiltonian cycle.

Computational Complexity Theory

A brief insight into computational complexity theory shall be provided to allow some of the underlying components of complexity to be explained. This, in turn, shall be used to help understand the complexity of the TSP (as an NP-hard problem [10]) and predict how different approaches to finding the optimum solution to a given instance would work. However it is worth noting that complexity theory is not the main component of this article so only relevant aspects of the topic are included.

Algorithms

An algorithm can be described as a process, or set of rules to be followed explicitly. The idea is that by following an algorithm the process can be completed without any intuitive knowledge, making it ideal for a computer. An algorithm can be written for any number of reasons, perhaps to complete a simple but repetitive task or to complete a very difficult and long task a human would not be able to do within a reasonable amount of time.

This article shall investigate the efficiency of how two distinctly different algorithms operate when applied to various instances of the TSP. The two algorithms considered shall be a *hill climbing* algorithm and a *branch and bound* algorithm. Both algorithms shall be explained, tested and analysed throughout the article.

Hill Climbing

An hill climbing (HC) algorithm is known as a random search heuristic. The use of the word *random* indicates there is an element of randomness in the algorithm; if it is run twice then the same output is not guaranteed, while *search heuristic* means that the algorithm works by considering small iterative changes to improve upon the current solution. When applied to the TSP an HC algorithm works by randomly generating a valid tour and, through a series of permutations, gradually decreases the cost of the tour by iteratively changing the order in which the cities are visited. This algorithm is explained in more detail in Section 5, including the specific permutations applied and small examples.

Branch and Bound

A branch and bound (BB) algorithm is known as an exact algorithm because it is designed to always find the optimal solution to a given problem and does not rely on any form of randomness. So, if a BB algorithm is run multiple times under the same conditions on the same problem, the solution found will remain constant. A BB algorithm is better than a brute force approach because it is able to split the set of all feasible tours into subsets and, by calculating the lower bound of each subset, decide to remove or investigate these subsets in more detail. This sets the BB algorithm apart from a brute force approach because it does not need to fully investigate each branch. This algorithm is explained in more detail in Section 6.

Current Research

The TSP is a heavily researched topic, with a lot of effort being allocated towards proposing new algorithms to solve various instances of the TSP. Much of the current research is looking at the application of evolutionary and genetic algorithms to solve the TSP [1]. Further research is being conducted to better understand the *features* of various instances of the TSP in an effort to predict how effective various algorithms would be in solving the instance [18]. Currently, 'Concorde' [2], is widely regarded as being the best TSP solver available and applies advanced heuristics such as Chained Lin-Kernighan [3].

It should be noted that the objective of this artcile is not to develop upon these current aspects of leading research, but rather to create two very different algorithms designed to solve the TSP and analyse their performance across various instances.

2 Graph Theory

Background of Graph Theory

Graph theory was originally devised by Leonhard Euler, allowing him to solve the famous Königsberg bridge problem, The problem required a circular walk to be found which would allow a person to cross over each of the seven bridges exactly once.

Rather than considering the problem as land masses with bridges spanning rivers; Euler reduced the problem down to a graph, by considering the land and bridges as the vertices and edges of the graph, respectively.

Reducing the problem to a graph allowed Euler to focus on the key aspect of the problem relevant to finding the solution, the connections between land masses. By doing this Euler was able to prove that it was impossible to find a circular walk encompassing each bridge exactly once [15].

A similar logic is taken when considering the TSP. While at first glance the majority of problems may seem too complex to be solved by an algorithm, the principles first used by Euler can still be applied to reduce different instances of the TSP to simple graphs. The following definitions will outline all aspects of graph theory used throughout the artcile.

Definitions

Definition 1. *A **graph**, G, is represented by $G = (V, E)$, where V is the set of n cities and E is the set of edges between each city pair, formally*

$$V = \{1, 2, \ldots, n\} \subseteq \mathbb{N}$$
$$E = \{\, (i, j) : i, j \in V \} \; .$$

Figure 1 is an example of an arbitrary graph. The number of edges adjacent to any

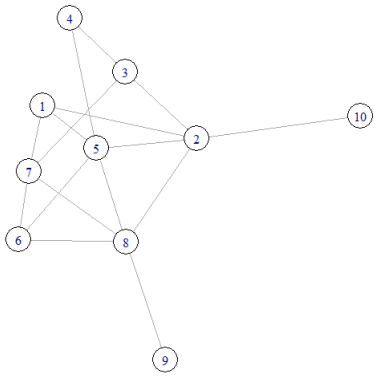

Figure 1: An arbitrary graph with ten vertices

vertex $v \in V$ is known as the *degree* of v, for example, the degree of vertex 6 in Figure 1 is 3.

Definition 2. *If a graph has an edge between a vertex and itself it is known as a **loop**.*

Figure 2: A single vertex with a loop

Figure 2 shows a single vertex with a loop. As we will only be considering simple graphs, there shall be no loops. Note, simple graphs also omit multiple edges connecting the same vertices.

Definition 3. *A graph is **complete** if all pairs of vertices $v \in V$ are connected by some edge in E.*

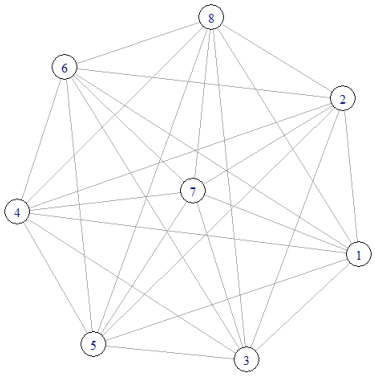

Figure 3: A complete graph with eight vertices

Figure 3 shows a complete, undirected and un-weighted graph with 8 vertices arranged in no particular order, this graph is also known as K_8.

Definition 4. *A **walk** is a sequence of vertices within a graph, G, which are connected via edges. For example,*

$$v_i, v_{i+1}, \ldots, v_{j-1}, v_j$$

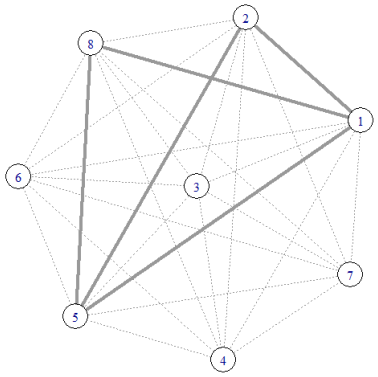

Figure 4: A walk within K_8

is a walk, where v_i and v_j are known as the start and end points of the walk, respectively. Note, in a walk edges and vertices can be used multiple times.

Figure 4 shows an example of a walk (bold edges) in K_8, where the walk follows the sequence $1, 5, 2, 1, 8, 5$. Note this walk visits some of the vertices multiple times but does not use any edge more than once.

Definition 5. *A **path** in a given graph, G, is a walk in which edges and vertices can be used only once.*

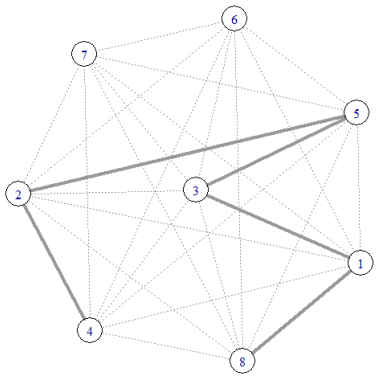

Figure 5: A path within K_8

Figure 5 shows an example of a path (bold edges) in K_8, following the sequence $8, 1, 3, 5, 2, 4$ (or $4, 2, 5, 3, 1, 8$, however in these cases the direction is not relevant). Note, the path does not re-use any edges or vertices to complete this sequence.

Definition 6. *A **cycle** in a given graph, G, is a path which ends at the start point. For example*

$$v_i, v_{i+1}, \ldots, v_j, v_i.$$

Figure 6 shows a cycle (bold edges) following the sequence $1, 8, 4, 2, 5, 3, 1$ on K_8. Note, if a graph cannot contain a cycle, it is known as *acyclic*.

Definition 7. *A **Hamiltonian cycle** is a cycle which visits every vertex, $v \in V$, exactly once.*

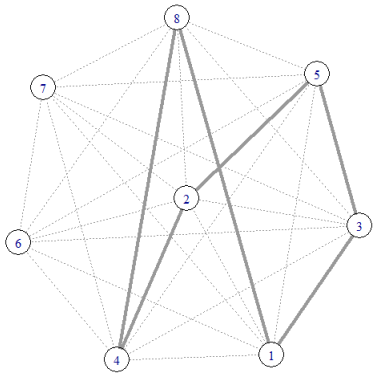

Figure 6: A cycle within K_8

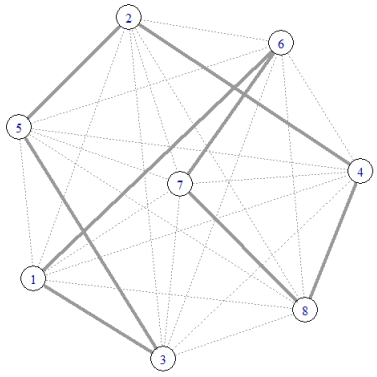

Figure 7: A Hamiltonian cycle within K_8

Figure 7 shows a Hamiltonian cycle within K_8. By the definition of the TSP we are looking for the optimal Hamiltonian cycle in our graph, where the cost of the cycle is dependent upon the weightings of the chosen edges.

Definition 8. *Let G be a graph with n vertices. G is a **tree** if the following criteria are met:*

1. *G is connected*
2. *G is acyclic*
3. *G has $n-1$ edges*

Figure 8 depicts an arbitrary tree with 17 vertices and 16 edges.

Definition 9. *Let G be a tree with n vertices. A vertex of G with degree 1 is known as a **leaf**.*

Figure 8 has ten leaves: B, E, G, H, I, J, N, O, P and Q.

Application of Graph Theory to the TSP

An instance of the TSP can be considered as a complete undirected weighted graph where the cities are represented as vertices, with the objective being to find the optimal

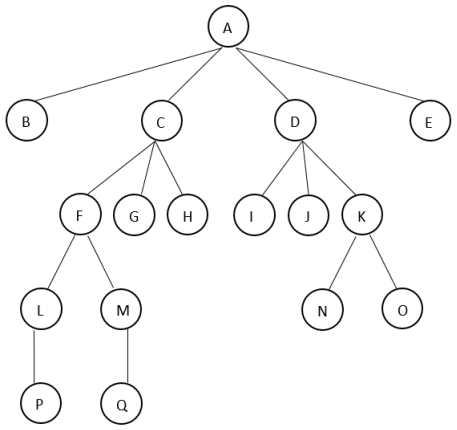

Figure 8: A tree with 17 vertices

tour, T. The set of feasible solutions to the TSP is the set of Hamiltonian cycles and the optimum solution, T, is the Hamiltonian cycle with the least cost. As the graph is defined as being a complete graph it is known that a Hamiltonian cycle exists.

Each edge in a graph representing an instance of the TSP has an associated cost and by summing each of these costs the total cost of a proposed tour can be calculated. So, if we let $X_{i,j}$ define the cost between the city pair $\{i, j\}$ where $i \neq j$ in the graph G, we can say the total cost of the tour, $c(T)$, starting from city i_1 is

$$c(T) = X_{i_1,i_2} + X_{i_2,i_3} + \ldots + X_{i_{n-1},i_n} + X_{i_n,i_1} \tag{1}$$

for distinct $i_j \in \{1, \ldots, n\}$. The methodology for calculating $X_{i,j}$ for each instance is outlined in Section 4, as it is dependent on the edge weight type (metric) of the instance in question. If we let H represent the set of all Hamiltonian cycles present in the graph G we can define the optimum tour as

$$T^* = \min\{\, c(T) : T \in H\}\,.$$

3 Computational Complexity Theory

As complexity theory is a large field, this article shall only provide a brief summary of the key concepts before moving onto the specific applications when considering the TSP. Complexity theory focusses on the complexity of computational problems and allows problems to be categorised depending on their complexity. Complexity theory also allows problems be reduced to show the underlying relationship between them [9]. Reducing a problem down to another (more well known) problem provides a better understanding of how to tackle the problem.

There are many ways to define the complexity of a problem, each with their own merits. The approach taken in this article is to measure the time taken for the algorithm to find the solution. When using time as the measure of complexity, problems often fall within one of two categories, P or NP (although there are other categories). NP is split into a further three subsections: NP, NP-Complete and NP-Hard. Each category shall be discussed in more detail, however some definitions are required first.

Turing Machines

The aforementioned classifications of complexity all depend upon the time taken to find the solution using an algorithm, beyond this, the classifications also rely upon the type of machine running the algorithm. There are two machines which are relevant for this discussion, the *deterministic* Turing machine and the *non-deterministic* Turing machine. Both are named after the world-famous mathematician Alan Turing.

Definition 10. *A **Turing machine** is a conceptual representation of a computer in digital form which follows instructions input via tape. This allows the machine to complete various tasks depending on the input tape, while only storing a limited library of instructions on the machine [5].*

Definition 11. *A **deterministic** Turing machine is representative of a conventional computer, and as such, is only capable of carrying out binary calculations [25].*

Definition 12. *A **non-deterministic** Turing machine is a theoretical machine capable of carrying out more than one action at each step, theoretically allowing it to assess the both TRUE and FALSE outcomes of one variable in one step [25].*

P

P stands for polynomial time and indicates that a given problem can be solved by a deterministic algorithm, on a deterministic Turing machine, within polynomial time. This means that the time taken to reach the solution can be represented by $O(n^k)$, the *order* of the complexity, where k is some constant [14] and n is a problem parameter (for example, the size of the problem). In this case there is a polynomial relationship as n changes because k is kept constant; however, it is worth noting that the magnitude of the complexity will be dependent on k. A sorting problem is a prime example of a problem which falls into complexity class P.

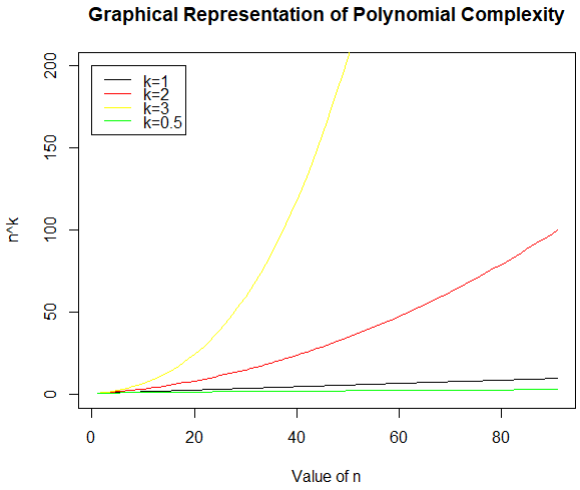

Figure 9: Plot of the relationship between the size of a problem and the time taken to solve for various orders of complexity

Figure 9 shows how much the value of k can affect the time taken to solve a problem within complexity class P. It is clear that while n effects the time taken to solve the problem, k is the dominating term in the definition.

NP

The class NP stands for non-deterministic polynomial time, indicating that a problem can be solved by an algorithm on a non-deterministic Turing machine within polynomial time [16]. As non-deterministic machines do not exist, NP problems cannot be solved in polynomial time. However, a fundamental requirement of an NP problem is that the solution can be verified within polynomial time [14].

NP-Complete

Under the assumption that P$\neq$NP, see [8] for a discussion on this topic, a problem is classified as being NP-complete if it satisfies two criteria. Firstly, the problem must fit the definition of an NP problem. Secondly, all problems within the classification are reducible to the same problem, this means that if a polynomial time algorithm is found to solve one NP-complete problem then the algorithm could be used to solve *all* NP-complete problems [7].

NP-Hard

An NP-hard problem is "at least as hard as any NP-complete problem and quite possibly harder" [13]. Expanding upon this loose definition, an NP-hard problem cannot be solved whilst providing proof of the solution within polynomial time.

As stated in Section 3, an NP problem can have its proposed solution validated within polynomial time, i.e. $O(n^k)$. For the TSP, it is not possible to validate a solution within polynomial time due to the requirement to check that the tour is not just a Hamiltonian cycle but the *optimum* Hamiltonian cycle. This second condition requires that all other feasible tours are known to be longer than the proposed solution, which at its worst would require a brute force check as shown in Table 1 in Section 1. Thus, we can state that the TSP is classified as NP-hard under computational complexity theory [10] and [21].

4 TSP Instances

There are many different TSP instances, with each instance representing a different challenge or application. The majority of these instances have either optimal or nearly optimal solutions already discovered, allowing them to be used as benchmarks when developing a new approach to solving these problems. Table 2 shows the key information for instances used in this article. Note, all instances have been sourced from Heidelberg University [24]. The key difference between instances (other than size) is the the edge weight type, this indicates the manner in which cost of travelling between a pair of cities is calculated. As Table 2 shows, the instances used in this article were of types 'ATT', 'EUC-2D', 'GEO' or 'MATRIX'. Recall that the cost of travelling between a pair of cities, say i and j, is denoted as $X_{i,j}$.

Table 2: Key information of chosen instances

Instance	# Cities	Edge Weight Type	Length of Optimum Solution
ATT48	48	ATT	10,688
BayG29	29	GEO	1,610
BayS29	29	GEO	2,020
Berlin52	52	EUC-2D	7,542
Burma14	14	GEO	3,323
Dantzig42	42	MATRIX	699
Fri26	26	MATRIX	937
GR17	17	MATRIX	2,085
GR21	21	MATRIX	2,707
GR24	24	MATRIX	1,272
Swiss42	42	MATRIX	1,273
Ulysses16	16	GEO	6,859
Ulysses22	22	GEO	7,013

EUC-2D

The edge weight type EUC-2D represents a Euclidean distance in two dimensions. Hence, the cost of travelling between two points is

$$X_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2}$$

This value is rounded to the nearest integer.

ATT

The instance, *ATT48*, represents the 48 capitals of the contiguous mainland U.S. states and ATT indicates the edge weight type used is pseudo-Euclidean [23]. That is, the distance between cities i and j are calculated by finding the difference between the x and y co-ordinates of the cities and summing the square of these differences, dividing by ten and calculating the square root. Put simply

$$X_{i,j} = \sqrt{\frac{(x_i - x_j)^2 + (y_i - y_j)^2}{10}} \tag{3}$$

GEO

The co-ordinates within an instance with the edge weight type GEO are stored as degrees and minutes. So, to calculate the distance between two cities these co-ordinates first need to be converted into decimal degrees before the great circle distance (gcd) can be found. To calculate the decimal degrees one must first split the degree and minute components, e.g. from DDD.MM to DDD and MM. The decimal equivalent of

the co-ordinate is calculated as:

$$\text{decimal degree} = \frac{\pi \times (\text{DDD} + \frac{5}{3} \times \text{MM})}{180} \tag{4}$$

Note that Equation 4 applies for converting both longitude and latitude. After converting the x and y co-ordinates the gcd between the two is calculated as

$$X_{i,j} = 6378.388 \times \arccos\left(\frac{(1 + a_1) \times a_2 - (1 - \mathrm{a}_1) \times a_3 + 1}{2}\right) \tag{5}$$

for $a_1 = \cos(x_i - y_i)$, $a_2 = \cos(x_j - y_j)$ and $a_3 = \cos(x_j + y_j)$ and the final solution is rounded down to the nearest integer. The value 6378.888 in Equation 5 represents a sphere with radius 6378.888, allowing the great circle distance between the two co-ordinates to be calculated.

MATRIX

MATRIX indicates that the instance already has some form of cost matrix associated to it, either a complete cost matrix or only the upper / lower triangles of the matrix. As we know that all cost matrices are symmetric along the main diagonal it is simple to complete the cost matrix where either the upper or lower triangle sections are provided.

Optimum Tour Examples

Figures 10 and 11 show the optimum tours for *Burma14* and *ATT48* respectively. *Burma14* and *ATT48* have been displayed as they are the most referenced problems throughout the article alongside *GR21*, however it is not possible to plot the optimum tour for *GR21* as no (x, y) co-ordinates are provided for this instance.

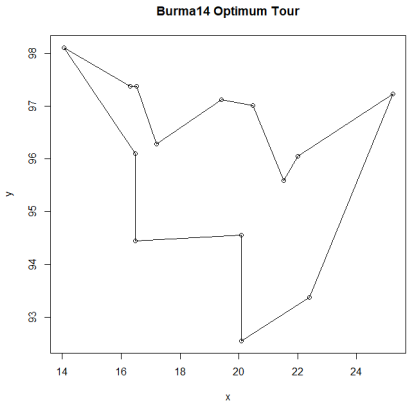

Figure 10: *Burma14* optimum tour

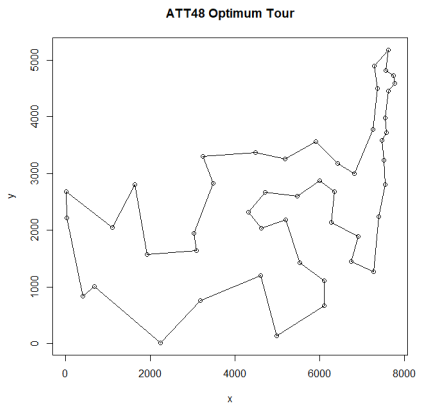

Figure 11: *ATT48* optimum tour

5 Hill Climbing Algorithm

Overview

A Hill Climbing algorithm can be visualised by considering a solution landscape of a maximisation problem, see Figure 12, where the peak is the optimal solution and the dot is the current solution.

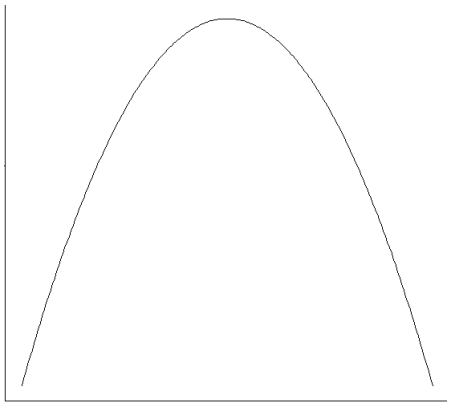

Figure 12: Convex landscape

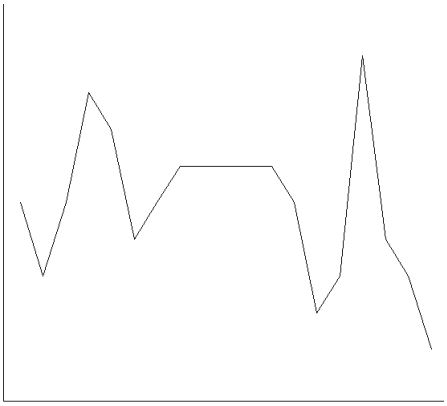

Figure 13: Rugged landscape

HC algorithms generate arbitrary solutions and make iterative improvements. As the algorithm converges on the optimal solution the dot 'climbs' up to the peak. A typical HC algorithm only adopts 'uphill' changes that improve the solution, and thus the HC algorithm is classified as a local neighbourhood search heuristic. If applied to a problem with multiple local maxima, Figure 13 for example, the HC algorithm would stop at the first local maximum reached. The algorithm would not identify any changes which would improve the solution and wrongly conclude the optimal solution had been reached. Therefore, the effectiveness depends greatly upon the landscape of the instance being solved. It is worth noting the ruggedness of the non-convex landscape in Figure 13. As the complexity of the problem increases, often the landscape will become more rugged. This indicates a greater risk of the HC algorithm returning a local maximum.

Although Figures 12 and 13 represent a continuous solution landscape this is not strictly true for most problems, and any movement on the landscape would be discrete not continuous. However these have been included as a visual aid to assist in understanding the concept of a solution landscape.

As the TSP is a minimisation problem, the HC algorithm developed will only accept 'downhill' steps. Further, the solution landscape of a TSP will often have multiple local minima, so there is a risk of getting stuck in a 'valley'. By this definition, the HC algorithm applied to the TSP can be viewed as trying to maximise the negation of a function. So, when applying an HC algorithm to find the optimal solution to an instance of the TSP, the aim is to minimise the objective function $f(x)$. This is equivalent to maximising the function $-f(x)$. Here, the objective function $f(x)$ is represented by the cost of the tour, $c(T)$. At each iteration the algorithm will adjust the tour in some way.

This creates the new tours, each with their own costs

$$c(T'),\, c(T''),\, \ldots,\, c(T^m)$$

where $m \in \mathbb{N}$ and represents the number of permutations applied. Each new tour will be compared to $c(T)$, and, if any

$$\{c(T'),\, c(T''),\, \ldots,\, c(T^m)\} < c(T),$$

then the tour with the smallest cost is accepted as the new tour and becomes $c(T)$. This process is repeated until a tour is found which satisfies the condition

$$\{c(T'),\, c(T''),\, \ldots,\, c(T^m)\} \geqslant c(T),$$

and then, the tour represented by $c(T)$ is classified as the optimum solution found as no possible swap can improve upon the current tour.

The first step for the HC algorithm is to use an initialisation function to create an initial tour, considering the example in Figure 14, a completely random valid tour might be

$$1, 4, 3, 5, 2, 1,$$

which would have an associated cost of 35.

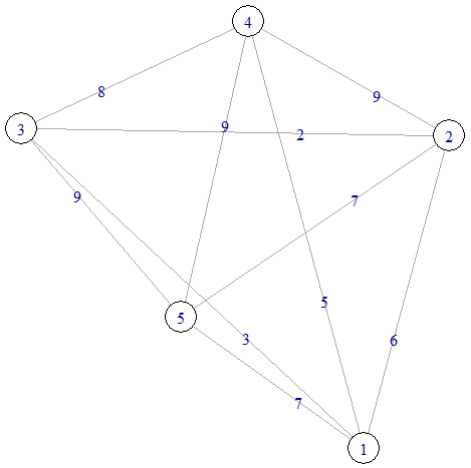

Figure 14: Five city example

Next the potentially shorter tours, known as 'children' are generated and compared against the 'parent'. Children could be generated in various ways by applying different permutations to the current tour. One potential permutation is a simple city swap heuristic, where children are generated by swapping two cities in the tour [19].

Being a Hamiltonian cycle the starting city does not affect the overall efficiency of a tour so there is no need to swap the first city. So following on with our example, the first city considered for swapping is city 4. The impact of swapping this city with all other cities (i.e. 3, 5 and 2) would be calculated in search of a shorter tour, generating the following children:

$$1, 3, 4, 5, 2, 1;$$
$$1, 5, 3, 4, 2, 1;$$
$$1, 2, 3, 5, 4, 1.$$

The cost of the children is compared against the initial tour and the one with the biggest benefit is adopted as the new tour.

This is repeated for the next city in the tour, i.e. 4 or 3, depending on which tour is chosen. The algorithm would iteratively apply the swapping process to every city in the tour and restart at the second city after the penultimate one has been swapped, until no better tour can be found. At this point either a local, or global, minimum has been found and the search is complete.

The HC algorithm is an effective way of finding optimum solutions to convex problems quickly and at least a locally optimum solution for non-convex problems. Additionally, it will return a valid solution even if it is interrupted mid-way. For all these benefits, the biggest drawback of the HC algorithm is the risk of it returning a local optimum, with no way of the user knowing if their result was not the best possible outcome. Several methods exist to minimise this risk (one of these methods is 'random restart').

Random restart iteratively repeats the HC algorithm a chosen number of times, and each run of the algorithm has a new randomly generated starting tour produced by the initialisation function. Before using a random restart the optimum cost discovered is stored and after all restarts have been completed the shortest tour is kept as the optimum result. This iterative method increases the chance of finding the global optimum, although it increases the computational time required because the HC algorithm is run multiple times. When the algorithm is applied to a problem with a known optimum (see Table 2), random restarts can be used until the global optimum tour is found. However, for previously unsolved problems this would not be possible so it is important to remember the solution found by a hill climbing algorithm may not necessarily be the optimum solution in these cases.

Developed Hill Climbing Algorithm

As stated previously, the HC algorithm is split into 4 distinct steps:

1. Generate initial tour;
2. Apply heuristics given on pages 68-71 to the current tour;
3. Repeat step 2 until no better solution can be found;
4. Apply random restarts if desired.

Each step shall be explained in detail, giving visual aids where appropriate.

Initial Tour Generator

Two potential methods of generating the initial tour were considered, a completely random valid tour and a (logical) *modified nearest neighbour* tour. The modified nearest neighbour approach requires selecting a city, say i, at random and the next city, $i+1$, is

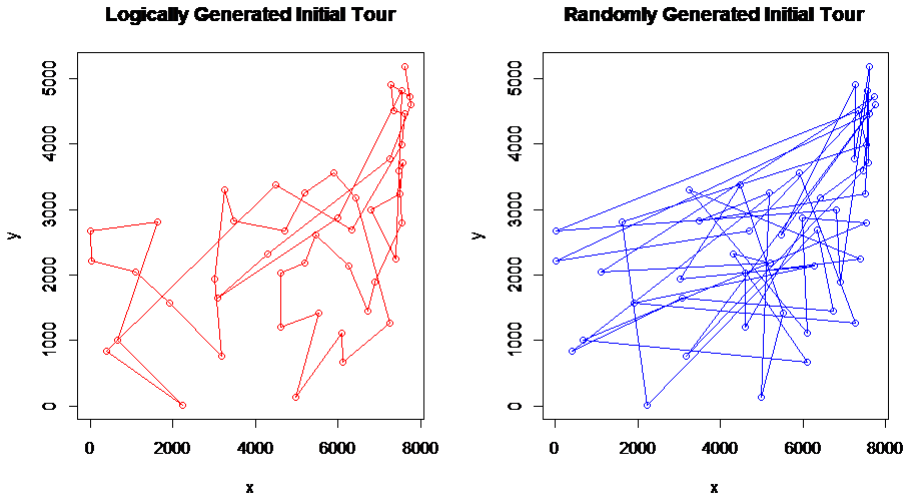

Figure 15: Comparison of initial tour generators on *ATT48*

randomly chosen from the set of the three closest cities to i not already visited. Each potential city has an equal probability of being chosen, $\frac{1}{3}$, so the initial tour still involves randomness. It is proposed that taking this logical approach reduces the time required for the algorithm to converge on the optimum solution - this is investigated in detail in Section 7. An example of the difference between the two methods can be seen in Figure 15. See Figure 11 for the optimum tour of *ATT48*.

Swap Heuristic

The first permutation applied is a straightforward city swap heuristic as outlined in the example in Section 5. A for loop is used to consider all possible swaps and identify the most beneficial city swap available. Additionally, a while loop is used to apply the swap heuristic repeatedly until it can provide no further improvement.

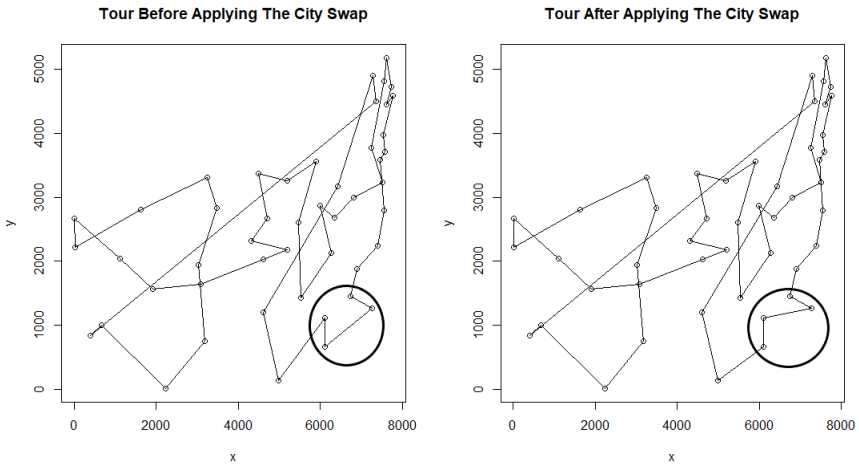

Figure 16: Example of the swap heuristic

Figure 16 shows a change (circled) implemented by the swap heuristic. As shown, a

more favourable route has been found by swapping the order in which the cities are visited. The change in cost, Δ, due to a swap can be calculated easily. Recall we have defined $\mathrm{X}_{i,j}$ as the cost of travelling from city i to city j. If $j = i + 1$, this Δ is

$$\mathrm{C}_1 = \mathrm{X}_{i-1,i} + \mathrm{X}_{j,j+1}, \tag{6}$$
$$\Delta = \mathrm{C}_1 - (\mathrm{X}_{i-1,j} + \mathrm{X}_{i,j+1}). \tag{7}$$

However, if $j \neq i + 1$ then the Δ in cost is

$$\mathrm{C}_2 = \mathrm{X}_{i-1,i} + \mathrm{X}_{i,i+1} + \mathrm{X}_{j-1,j} + \mathrm{X}_{j,j+1}, \tag{8}$$
$$\Delta = \mathrm{C}_2 - (\mathrm{X}_{i-1,j} + \mathrm{X}_{j,i+1} + \mathrm{X}_{j-1,i} + \mathrm{X}_{i,j+1}). \tag{9}$$

These calculations can only be applied to a symmetric TSP as, in Equation 7, the cost of $X_{i,j} = X_{j,i}$ so this edge does not need to be taken into account. This would not apply for an asymmetric TSP. In either case, if Δ is positive then the proposed modification would be beneficial. In this way, Δ may be seen as the benefit or dis-benefit arising from the swap. This tour is stored until all other city pairs have been swapped and the change resulting in the greatest benefit is adopted as the new tour. The process is repeated until the swap heuristic can produce no shorter tours can be produced.

2-opt Heuristic

The 2-opt heuristic (2-opt) is the first permutation which applies a 'k-opt' transformation. A k-opt transformation is defined as removing k edges from the current tour and replacing them in all possible combinations [12]. Thus, 2-opt involves removing two edges and replacing them in all possible ways. When removing two edges there are two new ways of reconnecting them, but only one which results in a valid tour.

Visually, 2-opt can shorten the tour by removing segments where the proposed cycle overlaps. This is completed within the algorithm by selecting a segment of the current tour, i.e. the i^{th} city to the j^{th} city, reversing it and inserting it back into the tour in the original location. In an instance with n cities, the algorithm uses nested for loops with $2 \leq i \leq n - 1$ and $i + 1 \leq j \leq n$ to evaluate all possible 2-opt swaps.

If we consider the random initial tour from the example in Section 5, which was $1, 4, 3, 5, 2, 1$, and calculate all possible 2-opt permutations with $i = 2$ (the second city visited, in this case city 4) we would produce:

$$1, 3, 4, 5, 2, 1;$$
$$1, 5, 3, 4, 2, 1;$$
$$1, 2, 5, 3, 4, 1,$$

with $j = 3$, $j = 4$ and $j = 5$ respectively. After calculating the cost associated to these tours we would consider $i = 3$ and $i = 4$, at this point we have $i = n-1$ so the algorithm

would identify the most beneficial swap, set that as the new tour and repeat until no better tours can be found with this permutation.

The impact of this permutation is calculated by considering the change in cost where adjacent vertices have changed, similarly to the swap heuristic, and the change resulting in the greatest benefit becomes the new tour.

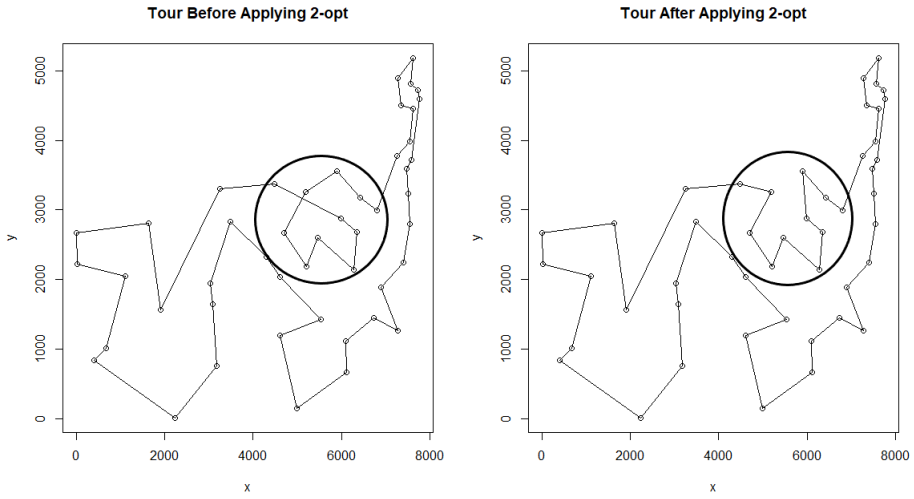

Figure 17: Example of the 2-opt permutation applied

Figure 17 shows 2-opt applied to *ATT48*, where a shorter tour has been created by modifying a segment of the tour which overlaps (circled). As with the swap heuristic, a for loop is used to asses the impact of all potential 2-opt permutations on the current tour.

3-opt Heuristic

The 3-opt heuristic (3-opt) applies a k-opt transformation where $k = 3$. As expected, 3-opt involves removing three edges from the current tour and reconnecting them in all possible combinations resulting in a valid tour. This is achieved by removing two segments from the current tour and rejoining them in all possible combinations, however the HC algorithm omits combinations which are equivalent to a 2-opt as the tour has already been optimised under this heuristic. The total cost of each new tour is calculated similarly to 2-opt. While 3-opt allows for more flexibility in optimising the tour it requires more calculations, taking longer to compute each step. Figure 18 shows 3-opt applied to *ATT48*.

In an instance with n cities, the algorithm uses three nested for loops with $2 \leq i \leq n-4$, $i+1 \leq j \leq n-2$ and $j+2 \leq k \leq n$ to evaluate all possible 3-opt swaps. An example of the 3-opt transformation can be provided by taking a generic sequence of 8 cities, $1, 2, 3, 4, 5, 6, 7, 8, 9, 1$ and letting $i = 3$, $j = 5$ and $k = 8$. This would mean rearranging the blue and red portions from $1, 2, 3, 4, 5, 6, 7, 8, 9, 1$ in all possible combinations not equivalent to the 2-opt transformations. Note, the red portion is always from $j+1$ to k. This example would create the children:

$$1, 2, 6, 7, 8, 3, 4, 5, 9, 1;$$
$$1, 2, 8, 7, 6, 5, 4, 3, 9, 1;$$
$$1, 2, 5, 4, 3, 8, 7, 6, 9, 1;$$
$$1, 2, 8, 7, 6, 3, 4, 5, 9, 1;$$
$$1, 2, 6, 7, 8, 5, 4, 3, 9, 1.$$

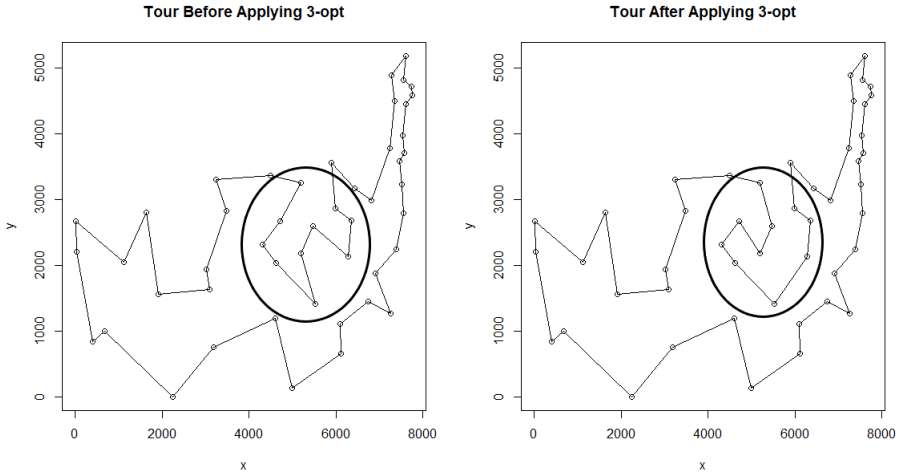

Figure 18: Example of the 3-opt permutation applied

Enhanced Swap

The enhanced swap is a modified version of the swap heuristic designed specifically to increase the robustness of the algorithm after observing the performance of all other permutations on the instance *ATT48*. The enhanced swap is applied after all other permutations and, quite simply, moves the position of a city, say i, within the tour.

Completed Algorithm

The completed HC algorithm is an amalgamation of the swap heuristic, 2-opt, 3-opt and enhanced swap applied consecutively. It fully optimises under each permutation and progresses onto the next when no shorter tours can be found by applying the current heuristic. The efficiency of this algorithm when applied to the instances listed in Table 2 is analysed in Section 7. Figure 11 shows the optimum tour for *ATT48* which, additionally, is found by the hill climbing algorithm.

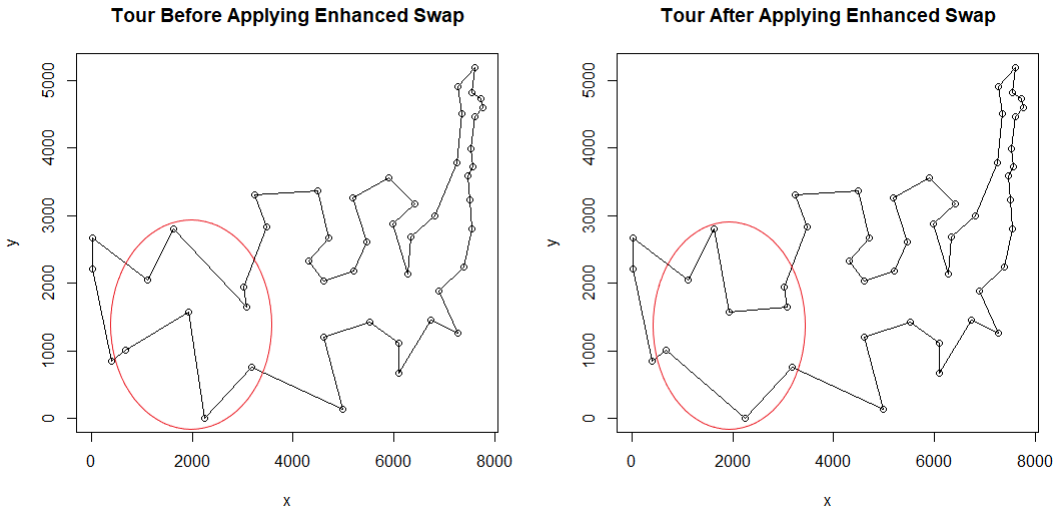

Figure 19: Example of the enhanced swap

6 Branch and Bound Algorithm

Overview

The Branch and Bound algorithm relies on splitting the set of all feasible tours into subsets and calculating the lower bound (LB) of the subset, where the LB is defined as the cheapest possible solution within the subset. Note that some of the constraints of the problem are relaxed to allow this to be calculated easily. Branches with LBs greater than a defined bounding criterion are pruned until only one tour remains, the optimal tour [17]. The pruning (also known as bounding) stage of BB separates it from a brute force approach as the LB of a branch is calculated without every path being individually computed. Because of this, the efficiency of the BB algorithm is dependent upon the choice of bounding criterion, this is discussed in more detail in Section 7.

The overall LB of the problem is calculated by summing the minimum two elements of every row in the cost matrix (CM) and dividing the total by two [22]. Dividing by two to avoids double counting edges, as each edge is incident with two vertices and all vertices are in the final tour, also the initial LB is always rounded up and the method is known as the *two neighbour bound*. As the BB algorithm builds up potential tours the LB for each branch would change depending on the cities already visited.

Developed Branch and Bound Algorithm

The BB algorithm is split into six distinct steps:

1. Assign a bounding criterion and calculate an overall lower bound;
2. Set an initial city, e.g., city 1;
3. Evaluate valid neighbours adjacent to the current city;

4. Prune any branches which now exceed the bounding criterion;
5. Repeat steps 3 and 4 until all branches reach a 'leaf';
6. Identify optimum solution from those remaining.

Bounding Criterion and Lower Bound

Step one of the algorithm is one of the most important aspects of a BB algorithm as the closer the bounding criterion is to the optimum solution, the faster the algorithm will identify the optimum solution. There are several ways of calculating the bounding criterion, one could generate a *nearest neighbour* tour (where the next city visited is the closest city) and use the cost of this tour as the bounding criterion. Alternatively the user could use the output of a HC algorithm, for example to ensure they have found the optimum tour. Figure 20 shows a nearest neighbour tour on *ATT48*.

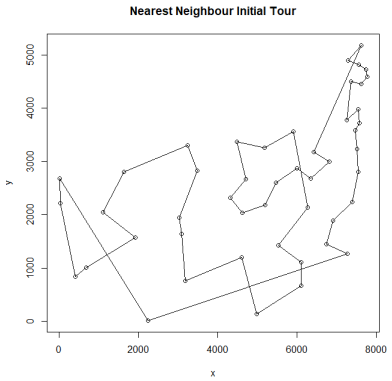

Figure 20: Nearest neighbour initial tour on *ATT48*

From the example in Figure 14, a nearest neighbour tour from city 1 is

$$1, 3, 2, 5, 4, 1,$$

which provides a bounding criterion of 26. The next step is to calculate the overall lower bound, the cost matrix for Figure 14 is

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} \infty & 6 & 3 & 5 & 7 \\ 6 & \infty & 2 & 9 & 7 \\ 3 & 2 & \infty & 8 & 9 \\ 5 & 9 & 8 & \infty & 9 \\ 7 & 7 & 9 & 9 & \infty \end{bmatrix} \end{array}$$

and the two neighbour bound produces an overall lower bound of

$$\mathrm{LB} = \frac{(3+5)+(2+6)+(2+3)+(5+8)+(7+7)}{2} = 24. \tag{10}$$

It is worth noting that this does not necessarily represent a valid tour.

Branching and Bounding

The next step in the algorithm is to generate, assess and prune branches representing all possible tours. The algorithm will select a city, say i, as the first city (without loss of generality as all cities are visited regardless). The impact the choice of initial city has on the time taken to solve the problem is investigated in Section 7. As this is the initial city, no other cities have been visited yet and therefore any of the remaining cities can be visited. So, if the problem contains n cities there are $n-1$ potential cities to visit. For each the associated cost of visiting them must be evaluated.

To do this a data-frame with $n-1$ rows is created, this can be visualised by imagining a decision tree (see Figure 21), where each row represents an edge in the decision tree for this problem. As each row represents a potential tour, the minimum LB for the tour is calculated and stored in the data-frame and any row with an LB greater than the bounding criterion is removed from the data-frame. This simulates the bounding aspect of the algorithm. Figure 21 shows this first step of branching and bounding applied to the example from Figure 14 with the LB for each tour shown alongside each branch.

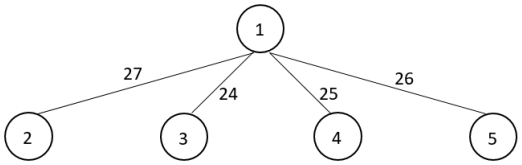

Figure 21: Initial step for branch and bound algorithm

The LB varies for each branch because of the different decisions made. On each branch, the LB of the tour being constructed is the overall LB plus the cost of the tour so far, $c(T)$, minus the minimum edges adjacent to the two cities in the tour. The calculations representing city 1 to city 5 are

$$c(T) = 7 \tag{11}$$

$$\text{LB} = 24 - \frac{3+7}{2} + c(T) = 26. \tag{12}$$

The fraction $\frac{3+7}{2}$ represents the minimum costs of the edges adjacent to city 1 and city 5. By subtracting this value from the LB we are subtracting as little as possible from the LB before adding the cost of the edge chosen. This keeps the LB as large as possible, which helps when it comes to pruning branches. It is also worth noting that the value being subtracted ($\frac{3+7}{2}$) is always rounded up to the nearest integer using the `ceiling` function in `R`.

As this LB is not greater than the bounding criterion (26), this branch is not be pruned. However, the LB for city 1 to city 2 is greater than 26 so all subsequent nodes in this branch would not be considered.

Next, all possible tours from the rows remaining in the data-frame are evaluated. This

is accomplished by duplicating each row $n-2$ times; $n-2$ is used as all tours under consideration are currently of length two and therefore there are $n-2$ cities that can be visited given the definition of the problem. For each iteration within the algorithm this value will change to represent the number of cities which can be visited. Specifically, this can be written as

$$\text{Number of duplicates} = n - \text{length of current tour.} \tag{13}$$

Again, each of these rows can be represented by branches on a decision tree and for each the LB is calculated and the relevant branches are pruned. This process is repeated until all potential tours are complete. At this point there will be one (or more) tours remaining, and the shortest tour is the optimum tour. As the BB algorithm is an exact algorithm, this is known to be the global optimum solution to the problem.

For these stages, the LB calculations for each branch are slightly different. Instead of subtracting the cost of the minimum edge adjacent to each city, the second cheapest adjacent edge value is used for the city which the salesman is departing from and the minimum adjacent edge cost is used for the city the salesman is travelling to. This is because the minimum edge adjacent to the city the salesman is departing from has already been subtracted from the LB in the previous iteration of the algorithm, so subtracting the second cheapest adjacent edge keeps the LB as accurate as possible. For example the calculations for the new LB from city 3 to city 2 would be

$$c(T) = 2 \tag{14}$$

$$\text{LB} = 24 - \frac{3+2}{2} + c(T) = 23 \tag{15}$$

Note, the new LB (23) is smaller than the LB calculated at the previous stage for this branch (24). Recall that this is because we always round up, using `ceiling`, when subtracting the cheapest adjacent edges to a city to avoid erroneously pruning a branch.

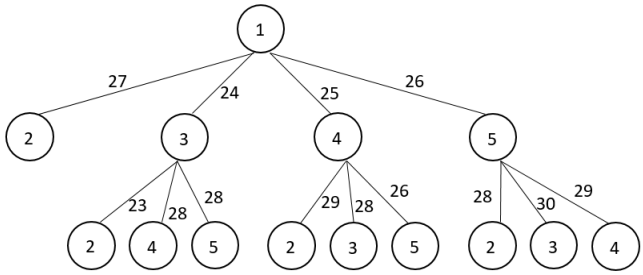

Figure 22: Second step for branch and bound algorithm

Figure 22 shows all branches and respective LBs considered in the second iteration of the algorithm. Figure 22 indicates the tours $1,3,2$ and $1,4,5$ are the only branches not

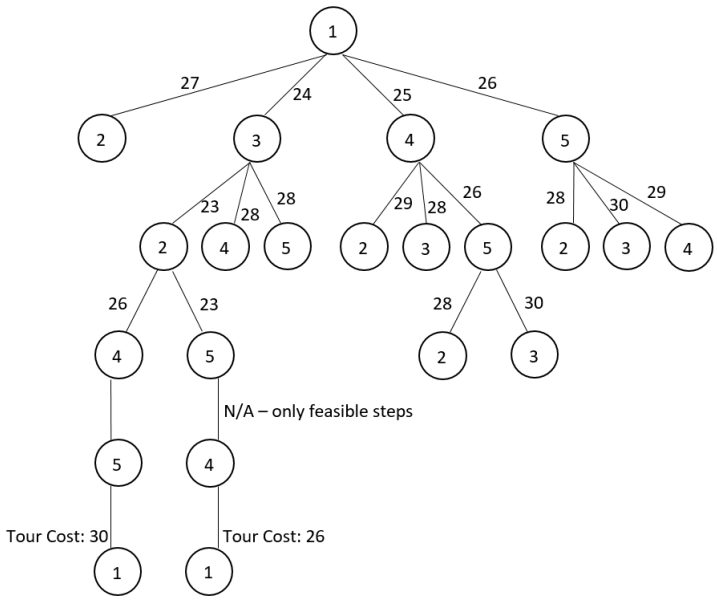

Figure 23: Completed branch and bound algorithm

to be pruned at this stage, as the lower bounds of all other branches exceeds the cost of the original nearest neighbour tour.

Figure 23 shows the completed tree for the example problem. In this case the cost of the optimum tour is 26, and the Hamiltonian cycle is $1, 3, 2, 5, 4, 1$. While this is the same as the initial nearest neighbour tour produced this can be considered a coincidence.

7 Analysis

Linear Regression

Linear regression can be described as an analytical approach to predict an outcome using the linear relationship between an *outcome variable* and one, or more, *predictor variables* [20]. For linear regression to be applicable the following conditions must be true:

- the predicted variable is continuous;
- the predictor variable (or variables) are continuous;
- the predicted and predictor variables have a linear relationship;
- the predictor variable is distributed normally.

If there is only one predictor variable, which will be the case in our analysis, it is known as *simple* linear regression. The equation for a simple linear regression model is

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \tag{16}$$

which, with the exception of the error term, can be equated to the standard line equation

$$y = mx + c,$$

and, therefore, represents a straight line. In Equation 16, β_1 represents the gradient of the line, β_0 represents the y-axis intercept, Y is the predicted value and X is the predictor value, while ϵ denotes the random error terms which accounts for the randomness in the predicted variable.

As our analysis shall investigate how the number of cities in an instance effects the time taken to solve the problem, our predictor variable is discrete. Despite having a discrete predictor variable, we can still use a linear regression model because our predictor is not a categorical variable and there is a continuous scale as the instance size changes. This means that the difference between an instance having 20 or 21 cities is equivalent to the difference between an instance having 40 or 41 cities. So, despite the discrete nature of the data the scale is continuous. Whereas if categorical data had been converted into a numerical representation, for example the results of a survey, the difference between 1 and 2 may not be equivalent to the difference between 3 and 4.

Analysis of the Hill Climbing Algorithm

Initial Tour Analysis

In Section 5 it was claimed that the time taken to find the optimum solution could be reduced by using a modified nearest neighbour (logical) initial tour, instead of a completely random valid tour. To assess the validity of this claim the instances: *Burma14*, *BayG29*, *BayS29*, *GR17*, *GR21* and *GR24* were each run sixty times on the hill climbing algorithm, thirty times with a random initial tour and thirty times with a modified nearest neighbour initial tour. Each time the algorithm was run, the time taken to find the optimal solution and the cost of the initial tour was collected.

Figure 24 shows the time, in seconds, to find the solution relative to the number of cities in the instance. Note that each instance had a discrete number of cities but a jitter plot has been used as a visual aid only. Looking at Figure 24, in the smaller instances there is a very small difference between the logical and random initial tours. However, Figure 25 shows the clear difference in mean times per initial tour for each instance size and would appear to show that there is an exponential growth in time as the number of cities increases. So, unless stated otherwise, all future analysis shall be conducted with the modified nearest neighbour initial tour.

Application to Various Instances

Each instance listed in Section 4 was run through the HC algorithm 100 times using the logical initial tour generator. As the optimum for each instance is known this was used to stop the algorithm once the optimum tour was found. Each time the algorithm was run the following metrics were collected:

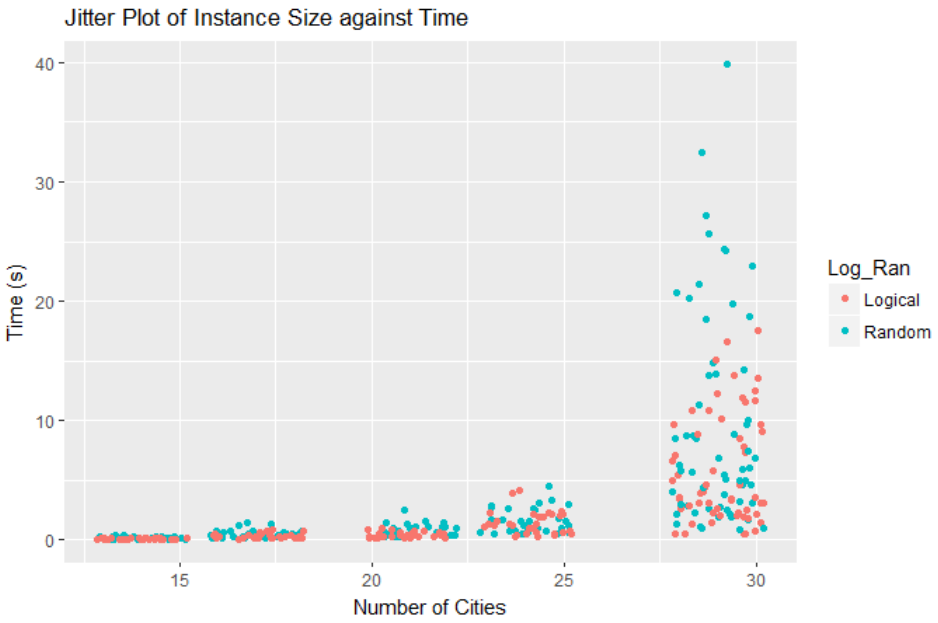

Figure 24: Jitter plot of instance size against time taken to solve

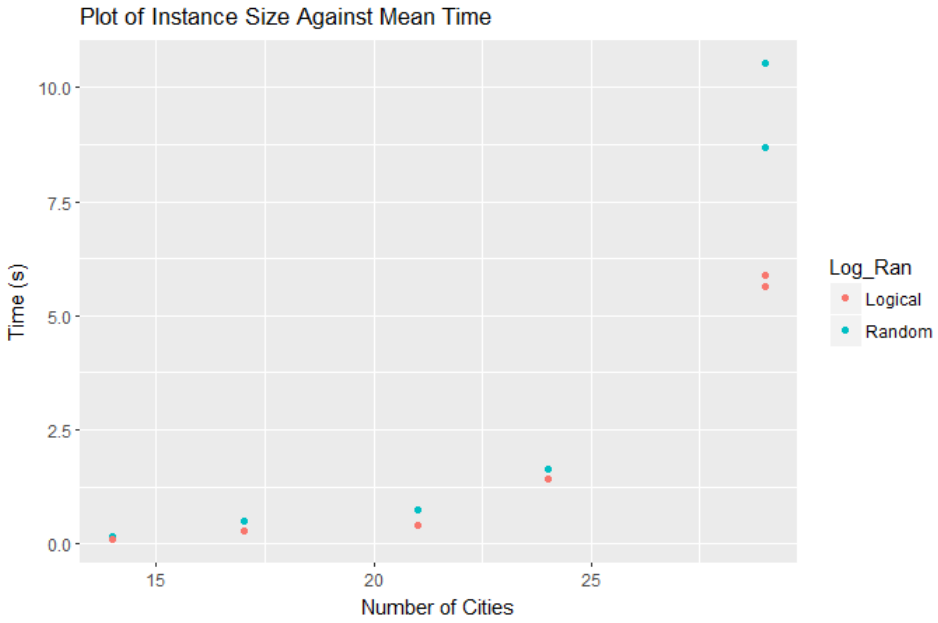

Figure 25: Size of instance against mean time taken to solve

- Time - time taken, in seconds, to find the optimum solution;
- Restarts - number of random restarts to find the optimum solution;
- Steps - number of permutations applied in the successful restart.

While most statistical inference generally relies on at least 30 data points, exceeding this value only increases the accuracy of the regression models created. The algorithm was run 100 times to allow for plenty of data to be collected within a reasonable amount of time, also allowing for the easier identification and removal of outliers without having

a significant impact on the number of data points remaining. For example, if only 30 runs were completed, removing an outlier would bring the total number of data points below 30, potentially reducing the accuracy of the models.

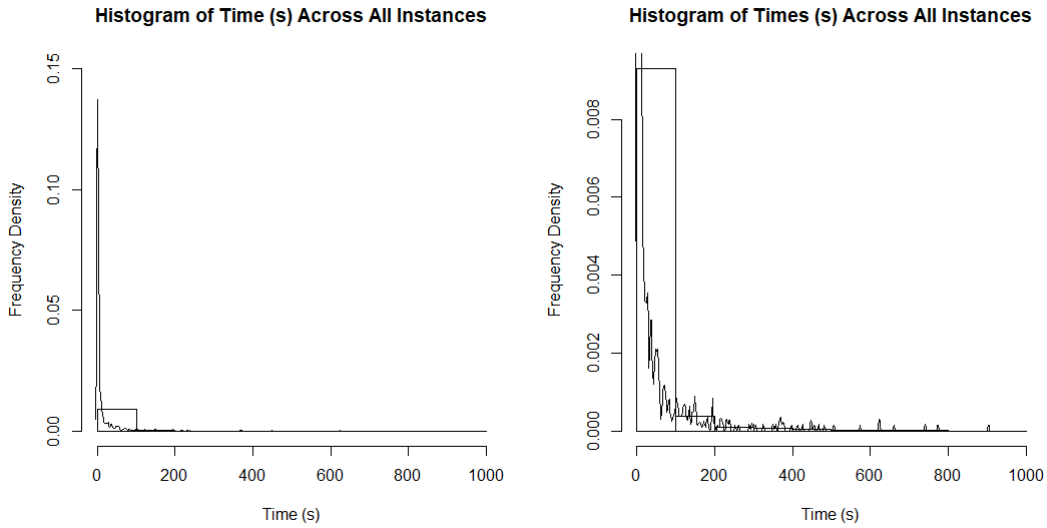

Figure 26: Histogram of time in seconds with added density line

Figure 26 shows the histogram of the time taken to find the optimum solution across all instances with different scales on the vertical axes to emphasise the peak in the density line. Figure 26 implies that a Poisson regression model may be a good fit due to the distribution of the data, however a Poisson model is only suitable when the predicted variable is an integer which does not hold in our case without coercing the data into an integer.

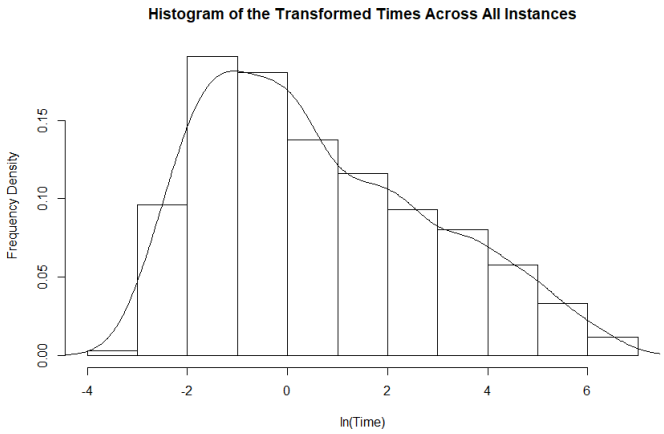

Figure 27: Histogram of the transformed time, with added density line

Figure 27 is a histogram of the time taken to find the optimum solution across all instances with a logarithm transformation applied to the time. After the transformation has been applied the data is normally distributed, with a slight skew, and a linear regression model may now be suitable.

Figures 28 and 29 show the QQ plot and residuals plot, respectively, of a Poisson model that has been created using instance size to predict the integer time taken to

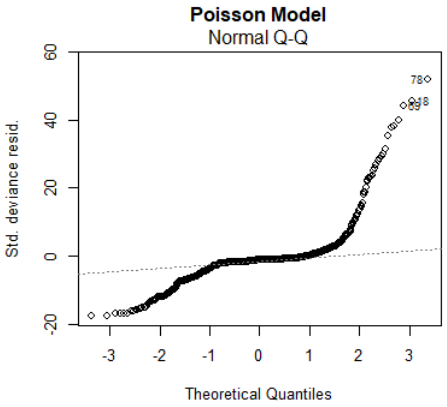

Figure 28: Poisson model QQ plot

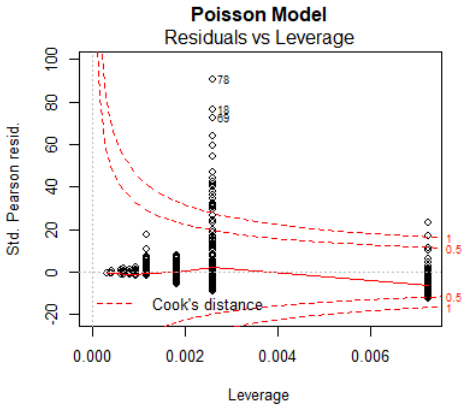

Figure 29: Poisson model residuals

find the optimum solution. Figure 28 shows the model is unfit because the plotted values do not follow the dotted line in a linear fashion as expected and Figure 29 reinforces this interpretation as there are many points outside of *Cook's distance*. Cook's distance can be described as a measure by which the influence of a data-point can be measured, where the greater the residual value (e.g., the point labelled 18 in Figure 29) the more influential the point is on the model [27] (see [4] for a complete definition of Cook's distance). By considering Figures 28 and 29 we can see that the outliers identified in the QQ plot are also identified as being highly influential on the overall model, which would imply that a Poisson regression model is not a good fit for the data. Hence a simple linear regression model of the log adjusted times was considered, where the instance size is used to predict the logarithm of the time required to find the optimum solution.

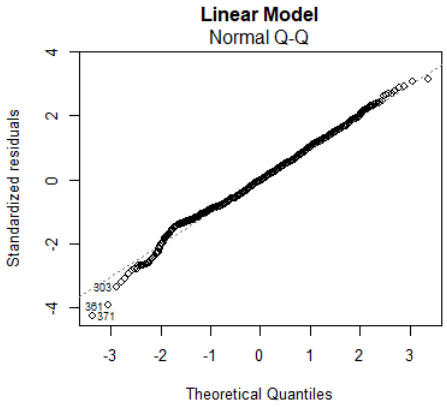

Figure 30: Linear model QQ plot

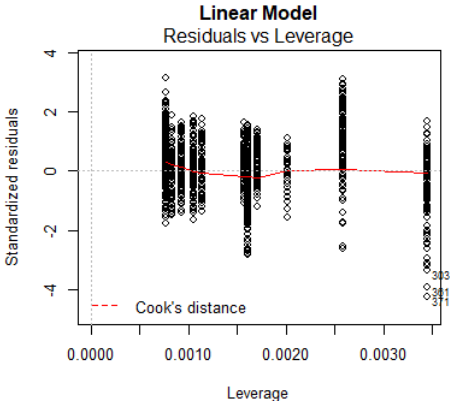

Figure 31: Linear model residuals

Figures 30 and 31 show the QQ and residuals plots, respectively, of the linear model. Both plots clearly show this is a better model in comparison to the Poisson model. Note that Figure 30 shows nearly all data points fit closely to the dotted line and Figure 31 shows no highly influential data-points (as the absolute value of the standardised residuals is less than four). However, both plots do still indicate that the outliers in the data are somewhat influential - these can be seen clearly at the bottom tail of Figure 30.

These potential outliers can be investigated in more detail by producing a box-plot of the data, Figure 32, for each instance size run through the hill climbing algorithm. The horizontal line within each box represents the 50^{th} percentile, while the top and bottom of each box represent the 75^{th} and 25^{th} percentiles respectively. The ends of the whiskers represent extensions of the box-plot designed to include the majority of the data and identify any outliers, which can be seen in *ATT48* and *Berlin52*.

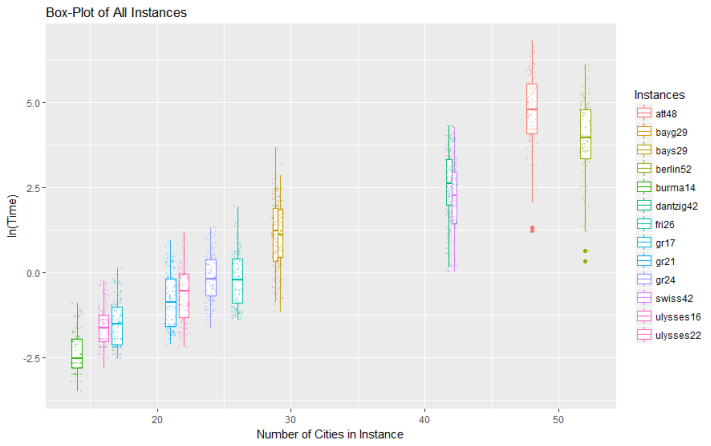

Figure 32: Box plot of transformed time per instance

After removing these observed outliers and creating a new simple linear regression model, again using instance size to predict time taken to solve, we can see in Figure 33 that this new linear regression model (blue line with 95% confidence interval) closely reflects the trend visible in the data.

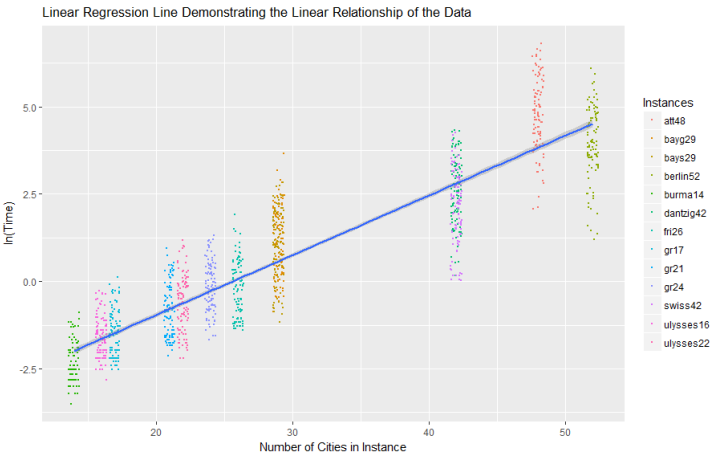

Figure 33: Plot of ln(time) with a linear regression line

The regression line in Figure 33 follows the equation

$$y = 0.17 \times x - 4.38. \tag{17}$$

Relating this back to Equation 16, $\beta_0 = -4.38$ and $\beta_1 = 0.17$. Recall that β_0 represents the y-axis intercept and β_1 is the gradient of the line. However, as this line represents the relationship between the number of cities in the instance and the natural logarithm of the time we cannot draw any conclusions about the actual time taken to solve a given instance yet.

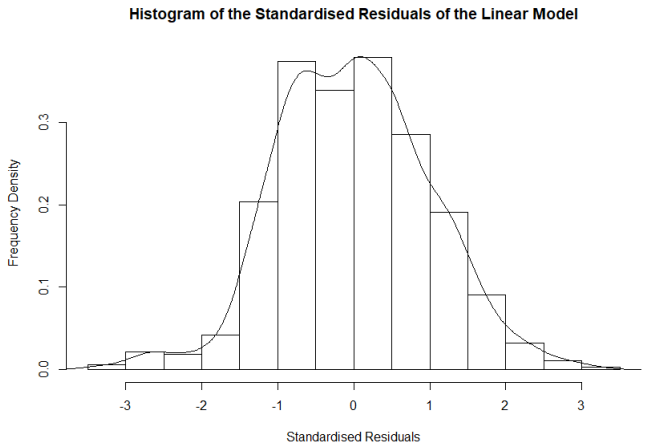

Figure 34: Histogram of standardised residuals of the linear regression model

Figure 34 is a histogram of the standardised residuals of the linear regression model, showing they are (roughly) normally distributed. This reinforces our belief that the linear regression model generated is a good fit for our data and is equivalent to creating a scatter plot of the standardised residuals displaying no pattern or trend.

To confirm everything we have inferred from the plots we can look at the summary of the liner regression model within `R` using the `summary` function from the package `pander`. This shows us that $p << 0.05$ which means the model accurately reflects the data and an R^2 value of 0.82 indicates the model is representing a significant portion of the variation present. Because of this, we can be confident that exponentiating predictions from our linear regression model will provide a good estimation of the actual time taken to solve an instance depending on the instance size.

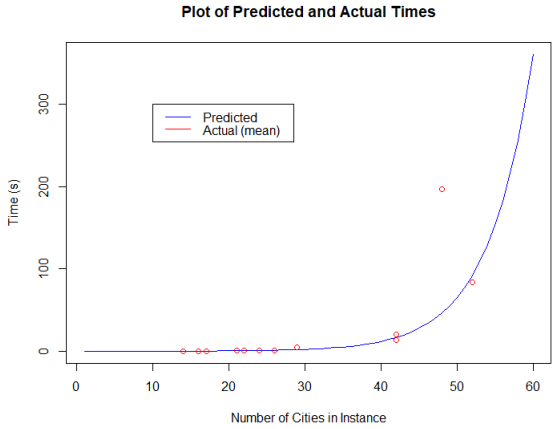

Figure 35: Plot of predicted time to find optimum solution with actual time taken

Figure 35 shows the predicted time, in seconds, to find the optimum solution depending on instance size plotted with the mean actual time taken to solve each instance. Note that the prediction has been displayed as a continuous line for visual aid only and predictions would only be made for discrete values. The equation governing these

predictions is

$$y = \exp^{(0.17 \times x - 4.38)} \tag{18}$$

where y is the time in seconds.

Figure 35 shows that our linear regression model is a good predictor for all instances except *ATT48*. This could indicate that there are features beyond the size of the algorithm that impact the time taken for the HC algorithm to find the optimum solution. In the next subsection, we analyse the BB algorithm outputs.

Branch and Bound Analysis

It is worth noting that, unfortunately, due to computational limitations the BB algorithm was not able to be run successfully across all instances listed in Section 4. The algorithm was successfully run on *Burma14* and *GR21*. From these successful runs, analysis shall be conducted to attempt to determine why these instances could be solved within a 'reasonable' amount of time while other instances could not be solved by looking at the instances sizes, initial cities chosen and the effect of the bounding criterion used.

Solved Instances

Figure 36 shows a violin plot of the times taken to solve the two instances *Burma14* and *GR21* after the BB algorithm was run 15 times for each instance. Note, that city 1 was used as the starting city for all runs. A violin plot displays the time taken in the form of a density plot, where the width of the violin indicates the frequency of the variable.

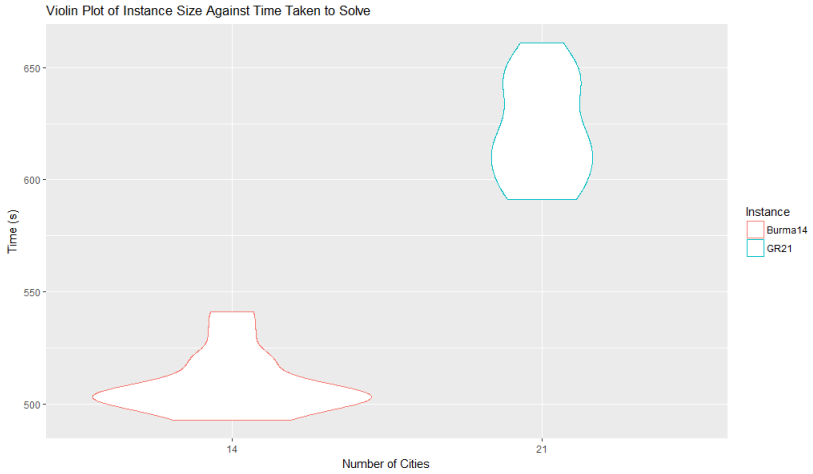

Figure 36: Violin plot of instance against time (s)

The violin for *Burma14* indicates that the instance was generally solved in just over 500 seconds, but on some occasions took up to around 540 seconds. While the violin for *GR21* shows a more uniform distribution of time from 590 to 660 seconds. The key

point to take away from Figure 36 is that even the quickest runtime for *GR21* is greater than the longest runtime for *Burma14*, which would imply that the larger instance is more computationally demanding than *Burma14*.

Bounding Criterion Analysis

As seen in Section 6, the BB algorithm aims to solve the problem by quickly identifying (and removing) tours which are clearly not optimal depending on some defined bounding criterion. The two neighbour method was explained; however, given the computational restrictions it quickly became apparent that this would not be a suitable approach for defining the bounding criterion.

Figure 37 and Table 3 show how modifying the bounding criterion effects the time taken for the BB algorithm to find the optimum solution on the instance *Burma14*. Note that for this subsection the BB algorithm used city 13 as the starting city as it was identified to reduce the time taken for the optimum solution to be found, this is explained in detail in Section 7.

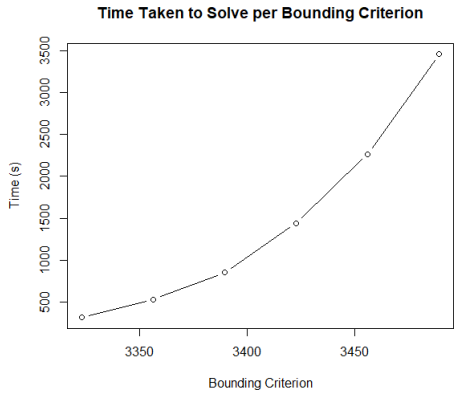

Figure 37: The effect of increasing the bounding criterion on *Burma14*

Table 3: Time taken to find optimum solution per bounding criterion on *Burma14*

Bounding Criterion	Time (s)	Time (min)
$3,323.00$	315.90	5.3
$3,356.23$	534.43	8.9
$3,389.46$	854.02	14.2
$3,422.69$	$1,442.36$	24
$3,455.92$	$2,266.93$	37.8
$3,489.15$	$3,455.66$	57.6

Using the nearest neighbour approach to generate an initial tour provides a bounding criterion of $3,638$, which is roughly 9.5% greater than the initial $3,323$. From Table 3 it is clear that using the bounding criterion generated from the nearest neighbour tour would take the algorithm well over 1 hour (probably more) to find the optimum solution. Therefore, unless stated otherwise all future applications of the BB algorithm will use the currently known optimum as the bounding criterion to allow the algorithm to the run

within a reasonable space of time. This can be considered equivalent to testing a HC algorithm output to ensure it is the optimum solution.

Size of Instance

Sub-instances were generated by selecting a set number of cities, say p, where $p \in \{5, 6, 7, 8, 9, 10\}$ from the instances *Burma14* and *GR21*. This provided some insight into how the the number of cities in the instance effects the time taken for the BB algorithm to find the optimum solution. Note that in this subsection city 1 was used as the starting city.

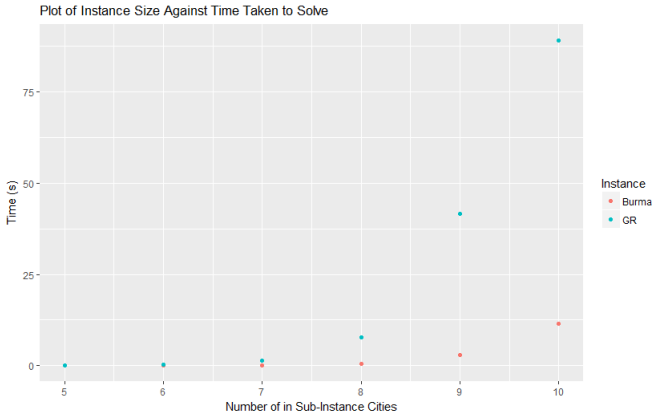

Figure 38: Relationship between the number of cities in the (sub-)instance and time taken to solve for two instances

Figure 38 shows the time taken to find the optimum increases as the size of the sub-instance increases in both cases. However, there is clearly more to the complexity of the instance than the number of cities because the algorithm is capable of solving *GR21* in less than 250 seconds, see Figure 39, but was not able to solve *GR17* after running for 24 hours.

Because of this, it was decided to attempt to identify what allowed some instances to be solved relatively quickly while others took longer. To first try and gain some insight, the choice of starting city which the BB algorithm uses was considered.

Starting City

The choice of starting city can be made without loss of generality, as every city is visited regardless, however it is believed that the choice of initial city does impact the time taken to find the solution using the BB algorithm. This was investigated by running the BB algorithm on *Burma14* and *GR21* with every possible starting city. Note that for all of these runs, the value of the known optimum solution to the instance was used as the bounding criterion, the reason for this was explained in Section 7.

Figure 39 shows the time taken for the BB algorithm to find the optimum solution plotted against the variance in edge costs adjacent to the starting city. We cannot say with any certainty if there is a direct relationship between the two from Figure 39 alone,

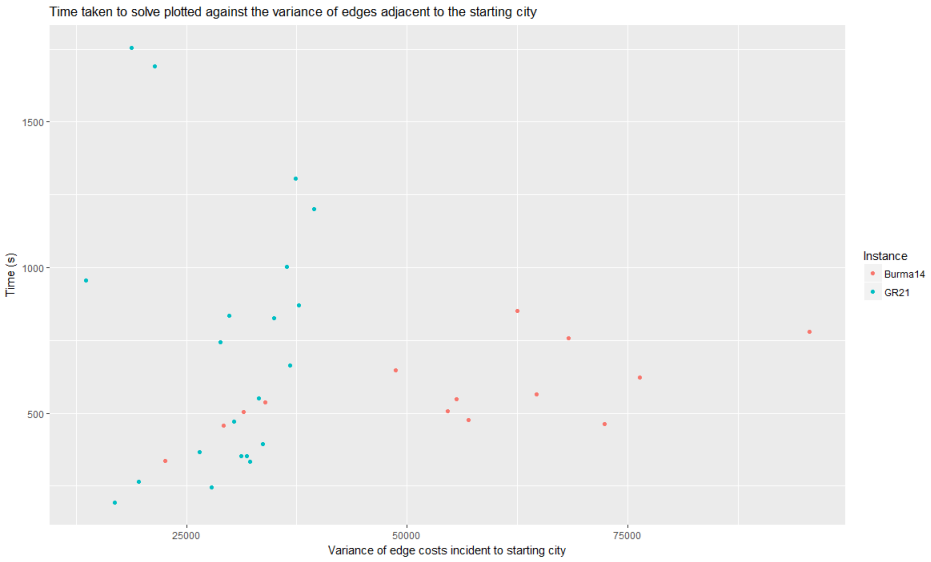

Figure 39: Effect of the starting city on time taken to solve

although it does support the belief that there is more to the instance complexity than the number of cities. *Burma14* seems to show a stronger relationship between the two metrics than *GR21*, as several starting cities in the latter had low variance but large solving time. Overall, this provides a small insight into a very large area of ongoing research which is expanded upon in Section 8.

Table 4: Costs of edges adjacent to city 1 in *Burma14*

City	2	3	4	5	6	7	8	9	10	11	12	13	14
Cost	153	510	706	966	581	455	70	160	372	157	567	342	398

Table 4 shows the cost of all edges adjacent to city 1, using these costs the variance was calculated in `R` using the `var` function, this was repeated for all cities in each instance. Note that city 1 is omitted from Table 4, this is because it is not possible for the salesman to travel from city 1 to city 1 so the associated cost within the cost matrix is ∞.

Comparing Hill Climbing and Branch and Bound

Due to the limited number of instances the BB algorithm was able to be successfully applied to, it is difficult to draw a strong comparison between the HC and BB algorithms. In order to enhance the data available, the HC algorithm was also run on the sub-instances defined in Section 7.

The HC algorithm was run 100 times on each sub-instance, however, as these sub-instances are not official TSP instances there are no published known optimum solutions. Because of this, the solution provided from the BB algorithm was used within the HC algorithm to identify when the optimum solution was identified to allow for comparable results to be generated. If this was not done there would be no way of fairly identifying how long it took the HC algorithm to find the optimum solution between the sub-instances and the complete instances.

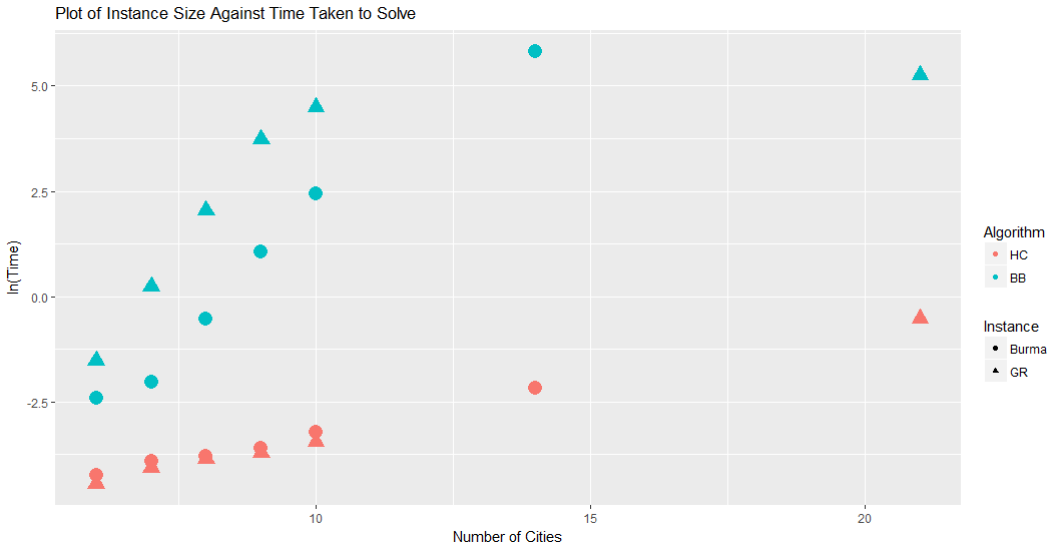

Figure 40: Comparison between HC and BB algorithms

Figure 40 shows the (log adjusted) times taken to find the optimum solution for both the HC and BB algorithms on the *Burma14* and *GR21* instances in addition to the sub-instances generated from each instance. These sub-instances were of size

$$\{6, 7, 8, 9, 10\}.$$

A logarithm transformation was applied to the times to allow difference in time to be seen more clearly. It is clear from Figure 40 that, in all cases, the HC algorithm is faster than the BB algorithm. It can also be seen that, as the size of the instance grows, the difference between the time taken for each algorithm to find the solution grows too. It is also worth noting that, in all sub-instances the starting city for the BB algorithm was city 1, while for the complete instances the initial city chosen was the optimum city found for that instance in Section 7.

This approach was taken to enable a fair comparison between the times taken to solve the complete instances *ATT48* and *Burma14* on both algorithms, as the optimum approach was taken in generating each plotted value. However, as Figure 39 shows, we cannot assume that starting from the city with the least variances in adjacent edge costs will result in the quickest time to find the solution. So, for simplicity the decision was made to start form city 1 in all sub-instances.

8 Discussion and Conclusions

Discussion

Page 77 provided some insight into the performance of the HC algorithm across a range of instances, and indicated that there was a relationship between the number of cities in the instance and the time taken for the HC algorithm to identify the shortest tour. However, by observing Figure 32 and Figure 33 and comparing the times taken to solve *ATT48* and *Berlin52* we can infer that the time taken to solve is not entirely

dependant on the number of cities in the instance. Indeed, on average, *Berlin52* was solved faster than *ATT48*. This could imply that there are other features of instances which impact the efficiency of the algorithm employed. This concept was investigated in more detail in Section 7, looking at the BB algorithm.

After noticing that the BB algorithm was able to solve *Burma14* and *GR21* quickly, but unable to solve *GR17* in over 24 hours, it was clear that the features impacting the performance of the BB algorithm were having a far greater impact than the features affecting the HC algorithm. In order to try and identify why only these two instances could be solved within a reasonable space of time, they were run through the algorithm again using every possible starting city. Rather than answering any questions however, this process only raised more. Figure 39 shows how, for *Burma14*, there appears to be a relationship between the time taken to solve and the variance in edge costs adjacent to the starting city. The same investigation into *GR21* displayed no clear pattern.

From this, we can surmise that there may be features present in different instances which have a substantial impact on the time taken for a BB algorithm to solve them. This inference is supported by the existence of ongoing research into the relationships between features of different instances of the TSP and how different algorithms are affected by these features [26].

Conclusion

Throughout this article we have introduced the Travelling Salesman Problem and the relevant components of graph theory and computational complexity theory to allow us to understand the TSP and why it is such a difficult problem. Two approaches to solving the TSP were proposed, each with their own merits and drawbacks.

The Hill Climbing algorithm, an amalgamation of random search heuristics, was initially investigated. Before being applied to the TSP it was known that the HC algorithm was an approximation algorithm designed to give an estimate of the optimum solution within a short space of time, but with no way of guaranteeing the optimum solution had been found. The written algorithm was explained in detail, specifying the four heuristics applied: the city swap, 2-opt, 3-opt and the enhanced city swap. It was also proposed that the efficiency of the algorithm was dependant on the initial tour generated.

The Branch and Bound algorithm was also proposed as an approach to solving the TSP. The BB algorithm is known as an exact algorithm because it is guaranteed to find the optimum solution; however, this guarantee comes at the expense of time. The process of branching and bounding in search of the optimum solution was explained and it was proposed that the choice of bounding criterion would have a substantial impact on the time taken for the algorithm to run.

The HC algorithm was applied to a variety of instances, ranging in size and metric (edge weight type) where the known optimum solutions were used to stop the algorithm after finding the optimal solution. This allowed analysis to be conducted on the time taken to solve each instance, from initial analysis it was clear that there is, in general, a relationship between time taken to solve and instance size. However, beyond this

relationship it was also visible that other features of the instances impacted efficiency of the HC algorithm. By applying a linear regression model we were able to show the strength of this relationship. Using this model, we would also able to predict how long the algorithm would probably need to be run for to find the solution to larger, potentially unsolved problems.

After applying the HC algorithm the BB algorithm was applied to the same instances, although with limited success. It was quickly noted that the BB algorithm was not able to solve the majority of instances within a reasonable space of time. With only two instances successfully solved by the BB algorithm analysis was conducted to try and identify what affected the algorithms success. Initially the effect of changing the bounding criterion was considered, and, as expected this was shown to have a dramatic impact on the time taken to find the optimal solution. Beyond this, limited analysis was conducted on the relationship between the BB algorithm and the size of the instance being solved. However, as only two instances had been solved this could only be conducted by creating sub-instances and solving these.

This investigation revealed the expected results, that the size of the instance did appear to impact the time taken to solve, and that as the number of cities increased so too did the time. In an effort to better understand the algorithm and the instances which were successfully solved, each possible city for both instances was considered as the starting city. *Burma14* revealed a relationship between the variance in the edge costs adjacent to the starting city and the time taken to solve. However, this relationship was not present in *GR21*, and so there is clearly more to be considered when it comes to the BB algorithm.

Both algorithms revealed significant performance changes when the initial conditions were modified. For the HC algorithm, the modified initial condition was the initial tour generated. Using a logical initial tour decreased the time taken to solve all instances tested. The BB algorithm displayed a relationship between the bounding criterion and the time taken to solve.

Overall, we were able to show that the HC algorithm is consistently faster than the BB algorithm at finding the optimum solution. However without knowing the optimum solution beforehand it would be impossible to know, for certain, when the HC algorithm had found the global optimum solution to a given problem. On the other hand the BB algorithm took longer but guaranteed the optimum solution.

We can conclude that, operationally, the HC algorithm would be the more reliable algorithm as it is faster and we can predict its performance prior to applying the algorithm based upon the size of the instance. However if used in conjunction with the BB algorithm, it may be possible to validate the output of the HC algorithm given enough time.

Sample code may be downloaded from `http://math-sciences.org/datasets`.

Acknowledgements

I would like to thank my supervisor, Matthew Craven, for all of his help and support throughout this project. He went above and beyond the requirement of a project supervisor and was always happy to help wherever possible, from discussing complexity to correcting my (often poor) grammar. Alongside Matthew I would also like to thank my partner, Kieran, for all of his support. Especially in the long weeks, days and hours up to the submission of my work.

References

[1] Albayrak, M. and Allahverdi, N. (2011). Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms. *Expert Systems with Applications*, 38(3):1313–1320.

[2] Applegate, D., Bixby, R., Chvatal, V., and Cook, W. (2006). Concorde tsp solver. *http://www.tsp.gatech.edu/concorde*.

[3] Applegate, D., Cook, W., and Rohe, A. (2003). Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92.

[4] Cook, R. D. and Weisberg, S. (1982). *Residuals and influence in regression*. New York: Chapman and Hall.

[5] Copeland, B. J. (2004). *The essential turing*. Clarendon Press.

[6] Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

[7] Farhi, E., Goldstone, J., Gutmann, S., Lapan, J., Lundgren, A., and Preda, D. (2001). A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem. *Science*, 292(5516):472–475.

[8] Fortnow, L. (2009). The status of the p versus np problem. *Communications of the ACM*, 52(9):78–86.

[9] Goldreich, O. (2008). Computational complexity: a conceptual perspective. *ACM Sigact News*, 39(3):35–39.

[10] Grefenstette, J., Gopal, R., Rosmaita, B., and Van Gucht, D. (1985). Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168.

[11] Held, M., Hoffman, A. J., Johnson, E. L., and Wolfe, P. (1984). Aspects of the traveling salesman problem. *IBM journal of Research and Development*, 28(4):476–486.

[12] Helsgaun, K. (2006). *An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic*. PhD thesis, Roskilde University. Department of Computer Science.

[13] Hillar, C. J. and Lim, L.-H. (2013). Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):45.

[14] Hogan, S. (2011). A gentle introduction to computational complexity theory, and a little bit more. *available at* *https://www.math.uchicago.edu/~may/VIGRE/VIGRE2011/REUPapers/Hogan.pdf*.

[15] Jungnickel, D. (2005). *Graphs, networks and algorithms*. Springer.

[16] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

[17] Little, J. D., Murty, K. G., Sweeney, D. W., and Karel, C. (1963). An algorithm for the traveling salesman problem. *Operations research*, 11(6):972–989.

[18] Mersmann, O., Bischl, B., Bossek, J., Trautmann, H., Wagner, M., and Neumann, F. (2012). Local search and the traveling salesman problem: A feature-based characterization of problem hardness. In *Learning and Intelligent Optimization*, pages 115–129. Springer.

[19] Montgomery, J. (2007). Tackling the travelling salesman problem: introduction. *available at* *http://www.psychicorigami.com/2007/04/17/tackling-the-travelling-salesman-problem-part-one/*.

[20] Neter, J., Kutner, M. H., Nachtsheim, C. J., and Wasserman, W. (1996). *Applied linear statistical models*, volume 4. Irwin Chicago.

[21] Puget, J. (2013). No, the tsp isn't np complete. *available at* *https://www.ibm.com/developerworks/community/blogs/jfp/entry/no_the_tsp_isn_t_np_complete?lang=en*.

[22] Reinelt, G. (1994). *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag.

[23] Reinelt, G. (1995). Tsplib95. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*.

[24] Reinelt, G. (2008). Index of tsp library. *available at* *http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/*.

[25] Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192.

[26] Smith-Miles, K., van Hemert, J., and Lim, X. Y. (2010). Understanding tsp difficulty by learning from evolved instances. In *International Conference on Learning and Intelligent Optimization*, pages 266–280. Springer.

[27] Williams, D. (1987). Generalized linear model diagnostics using the deviance and single case deletions. *Applied Statistics*, pages 181–191.