Liverpool John Moores University
School of Computer Science and Mathematics

6200COMP Project

Final Year Dissertation
submitted by

Dylan Worley

1008307

Computer Science

Title

The Comparison of Exact and Approximate Algorithms for Completing the Travelling Salesman Problem

Supervised by
Pavel Semukhin

Submitted on
11 April 2025

ABSTRACT

This report will find a comparison of exact and approximate algorithms, starting with a brief history and current applications of the Travelling Salesman Problem then a brief history and explanation will be given of the seven different algorithms, Brute Force, Held-Karp, Branch and Bound, Christofides, Nearest Neighbour, Hill Climbing and Simulated Annealing that will be used to create a comparison between the two sets of algorithms.

The algorithms will be tested on a range of problems ranging from sizes of four nodes to seven thousand nodes. Measuring the path taken, cost of route and time taken for the algorithm to complete the problem will mean that a comparison of the performances of each algorithm to highlight strengths and weaknesses to make a comparison.

The results found from the tests will highlight the performance of exact algorithms finding optimal solutions in reasonable times in problems of under twenty nodes and then show the approximation algorithms, while not providing optimal routes doing them in much more time efficient manners to be able to solve problems above five hundred nodes faster than exact algorithms can solve problems of ten nodes. Whilst also comparing the approximation rates of different algorithms solving the TSP problems.

Coming to the overall conclusion to see the most efficient exact algorithm being the Held-Karp algorithm and most efficient approximation algorithm being the Christofides algorithm.

Finally explaining how the project can potentially be expanded in any future work to strengthen the study or provide a better program for users.

ACKNOWLEDGEMENT

I would first like to express my deepest gratitude to my project supervisor Pavel Semukhin for all the meetings, advice, proofreading and feedback that helped to shape and refine my project. I would also like to express my sincere thanks to the faculty at the School of Computer Science and Mathematics at Liverpool John Moores University for all the help and support leading up to this final project. Finally, a special thanks goes to all my friends and family for the constant proofread offers and aid during any stressful moments whilst completing this project and unwavering support, patience and belief in me throughout my entire academic journey.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Background Information

The travelling salesman problem is an NP-Hard algorithm that asks the question, given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city. Due to the complexity of the possible routes, there is no one exact algorithm designed to solve every TSP problem, only different ones sacrificing either optimisation or time.

The TSP is often used in theoretical computer science and operations research as a benchmark for optimisation. The main method used when solving this problem is a brute force search, which tests every single combination of routes to find the most optimal route, following on from this, there have been more recent algorithms developed that use different methods to try and find an optimal route, some focus on optimisation some focus on completion time.

Exact algorithms solve the problem exactly regardless of time and complexity of the problem, the algorithm will find the optimal solution to the problem. Approximate algorithms use heuristics to solve problems which means that the problems will be solved faster but less optimally.

Problem Statement

This report will investigate:

The Comparison of Exact and Approximation Algorithms for Completing the Travelling Salesman Problem

This will be by solving a variety of TSP problems using three different exact algorithms and four approximation algorithms to find out what algorithms complete the most optimal routes and how long it takes for them to do it.

Motivation

The motivation for this product comes from an interest in mapping and route solving algorithms, this has been an interest of mine since learning about the topic during level 4 Computer Science at A – level so being able to use different algorithms to solve different routes and create a comparison between the different algorithms is a project I am interested in and could dive into great detail about.

2. Background Research and Domain Analysis

Background Research

The Travelling Salesman Problem

The idea behind the travelling salesman problem is as follows: A salesman has been given a tour of a specified number of cities. Starting from any one of these cities, he must make a tour visiting each of the other cities on the tour only once, with his destination being his city of departure. This task should be achieved in such a way as to minimise the total distance travelled on the tour. Tours can be symmetrical or non-symmetrical. A symmetrical tour considers that the distance from city ‘a’ to city ‘b’ is the same as that from ‘b’ to ‘a’. A non-symmetrical tour considers that these distances are not the same, such as one-way systems. (Applegate, et al., 2006)

The TSP is an NP-Hard problem. An NP problem is one in which a solution is still being sought after, there is no direct one size fits all way of solving the problem, there are different ways it can be done yet all have strengths and weaknesses. NP problems can be solved in polynomial times whereas NP-Hard problems do not have known polynomial time solutions. There is no one-size-fits-all solution due to multiple different factors, such as the number of destinations and the distance between destinations. (Fortnow, 2013)

The origins of the travelling salesman problem are obscure; it is mentioned in an 1832 manual for travelling salesman, which included example tours of 45 German cities but gave no mathematical consideration. W. R. Hamilton and Thomas Kirkman devised mathematical formulations of the problem in the 1800s. It is believed that the general form was first studied by Karl Menger in Vienna and Harvard in the 1930s. (Yu, 2014)

Hassler Whitney, who was working on his PhD research at Harvard when Menger was a visiting lecturer, is believed to have posed the problem of finding the shortest route between the 48 states of the United States during either his 1931-1932 or 1934 seminar talks. There is also uncertainty surrounding the individual who coined the name “travelling salesman problem” for Whitney’s problem. (Yu, 2014)

The problem became increasingly popular in the 1950s and 1960s. Notably, George Dantzig, Delber R. Fulkerson and Selmer M. Johnson at the RAND Corporation in Santa Monica, California, solved the 48-state problem by formulating it as a linear programming problem. Growing population and code sophistication led to rapid increases in the size of the TSP solved. Martin Grotechel more than doubled this 23 years later, solving 120 instances in 1977. Enoch Crowder and Manfred W. Padberg again more than doubled this in just 3 years with a 318-city solution. This continued growing to 24,978 cities in 2004 until work was done by the University of Waterloo. (Yu, 2014)

Exact Algorithms

Exact algorithms focus on optimisation of the algorithm, better suited to smaller problems. These algorithms will find the most optimal solution to the problem, sacrificing the time taken to complete the algorithm.

Brute–Force Algorithm

This is one of the simplest solutions to sorting the Travelling Salesman Problem, it calculates all available tours and then selects the most optimal once completed, selects the most optimal tour to use. The Brute-Force algorithm has a time complexity of O(n!) (Chase, et al., 2020) The Brute-Force algorithm has a time complexity of O(n!), which means that it is the factorial for the number of routes, so for this example in Figure 1 and as seen in Figure 2, there are 1*2*3*4 different combinations of routes, so 24 in total. This is manageable for smaller routes; however, this increases exponentially with 720 routes for six cities, 40320 routes for eight cities and 3628800 routes for 10 cities, which makes the algorithm much more complex and longer to complete as cities grow; however, it provides the optimal route.

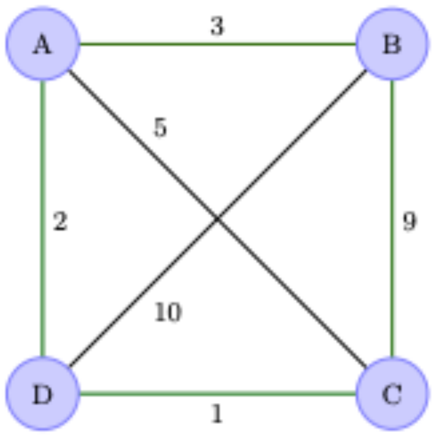


Figure 1: Brute-Force Example

Figure 1 shows a problem with four destinations. (Saiyed, 2012) To solve this problem with a Brute-Force approach, this would look like this:

1. A → B → C → D → A: 3 + 9 + 1 + 2 = 15
2. A → B → D → C → A: 3 + 10 + 1 + 5 = 19
3. A → C → B → D → A: 5 + 9 + 10 + 2 = 26
4. A → C → D → B → A: 5 + 1 + 10 + 3 = 19
5. A → D → B → C → A: 2 + 10 + 9 + 5 = 26
6. A → D → C → B → A: 2 + 1 + 9 + 3 = 15
7. B → A → C → D → B: 3 + 5 + 1 + 10 = 19
8. B → A → D → C → B: 3 + 2 + 1 + 9 = 15
9. B → C → A → D → B: 9 + 5 + 2 + 10 = 26
10. B → C → D → A → B: 9 + 1 + 2 + 3 = 15
11. B → D → A → C → B: 10 + 2 + 5 + 9 = 26
12. B → D → C → A → B: 10 + 1 + 5 + 3 = 19
13. C → A → B → D → C: 5 + 3 + 10 + 1 = 19
14. C → A → D → B → C: 5 + 2 + 10 + 9 = 26
15. C → B → A → D → C: 9 + 3 + 2 + 1 = 15
16. C → B → D → A → C: 9 + 10 + 2 + 5 = 26
17. C → D → A → B → C: 1 + 2 + 3 + 9 = 15

18.	$C \rightarrow D \rightarrow B \rightarrow A \rightarrow C$:	$1 + 10 + 3 + 5 = 19$
19.	$D \rightarrow A \rightarrow B \rightarrow C \rightarrow D$:	$2 + 3 + 9 + 1 = 15$
20.	$D \rightarrow A \rightarrow C \rightarrow B \rightarrow D$:	$2 + 5 + 9 + 10 = 26$
21.	$D \rightarrow B \rightarrow A \rightarrow C \rightarrow D$:	$10 + 3 + 5 + 1 = 19$
22.	$D \rightarrow B \rightarrow C \rightarrow A \rightarrow D$:	$10 + 9 + 5 + 2 = 26$
23.	$D \rightarrow C \rightarrow A \rightarrow B \rightarrow D$:	$1 + 5 + 3 + 10 = 19$
24.	$D \rightarrow C \rightarrow B \rightarrow A \rightarrow D$:	$1 + 9 + 3 + 2 = 15$

Figure 2: Brute-Force Routes Example

Every route is calculated to find the most efficient, and in this example, it is the route of $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$ which costs 15.

Held – Karp Algorithm

The Held–Karp algorithm, also known as the Bellman-Held-Karp algorithm, is a dynamic programming approach that efficiently solves the TSP by systematically evaluating subproblems and building up to the optimal solution. It guarantees an optimal solution by considering all possible routes and leveraging the concept of overlapping subproblems. (Agarwal, et al., 2020)

The Held-Karp algorithm follows a recursive formulation combined with memorisation to avoid redundant computations. The essential steps are:

- Initialisation – set up a dynamic programming table to store the optimal distances between subproblems
- Base Case – Set the initial condition for the smallest subproblems, for example, only two cities.
- Recursive Step – Iterate through each subproblem, calculating the optimal distances by considering all possible intermediate cities.
- Memorisation – Store the computed results in the dynamic programming table to avoid recomputation.
- Backtracking – Construct the optimal tour by tracing back through the dynamic programming table.

The Held-Karp algorithm offers several advantages, guaranteeing the optimal solution and the shortest possible route. Additionally, it has a time complexity of $O(n^2 2^n)$, which is much more optimal than the brute force algorithm; for example, it will take 2304 routes for six cities, 16834 routes for eight cities and 102400 routes for 10 cities. In the example of 10 cities, this is 3500000 fewer routes than for the brute-force algorithm, so it is much more timesaving. (Nguyen, 2020) An example of how this would look would be:

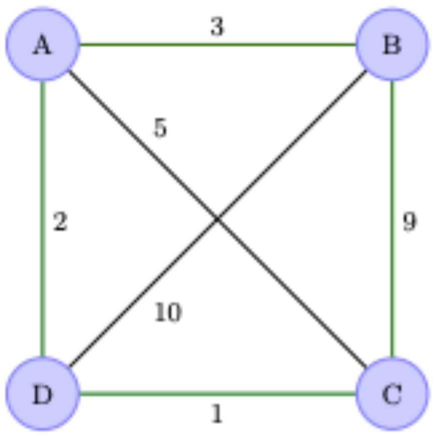


Figure 3: Held-Karp Example

Using the problem in Figure 3, the first step would be to create a distance matrix:

	A	B	C	D
A	0	3	5	2
B	3	0	9	10
C	5	9	0	1
D	2	10	1	0

Figure 4: Held-Karp Distance Matrix

This table shows the distance matrix between all the cities. This is then used to create the dynamic programming table for the subsets.

Visited Cities	**Last City**	**Cost**	**Previous City**
A	A	0	A
A B	B	3	A
A C	C	5	A
A D	D	2	A
A B C	C	3 + 9 = 12	B
A B D	D	3 + 10 = 13	B
A D B	B	2 + 10 = 12	D
A D C	C	2 + 1 = 3	D
A C B	B	5 + 9 = 14	C
A C D	D	5 + 1 = 6	C
A B C D	D	12 + 1 = 13	C
A B C D	C	13 + 1 = 14	D

Figure 5: Dynamic Processing Table

This example starts at A, and then finds the second route between B, C and D to calculate the cost, storing the cost of the routes, then proceeds to calculate the cost of travelling from the second city to the third and then the third to the fourth calculating the costs before backtracking to find the optimal route.

Branch and Bound Algorithm

The branch-and-bound algorithm starts by creating an initial route, typically from the starting point to the first node in a set of cities. Then, it systematically explores different permutations to extend the route beyond the first pair of cities, one node at a time. Each time a new node is added, the algorithm calculates the current path’s length and compares it to the optimal route found so far. If the current path is already longer than the optimal route, it bounds or prunes that branch of the exploration, as it would not lead to a more optimal solution. (Smith, et al., 2015)

This pruning is the key to making the algorithm efficient. By discarding less efficient paths, the search continues to get narrowed down, and the algorithm can focus on exploring only the most promising paths. This process is continued until all possible routes are explored, and the shortest one is identified as the optimal solution to the problem. (Smith, et al., 2015)

This algorithm also can have a time complexity of O(n!) in the worst case and O(2^n) with good pruning; this means that there is no definitive complexity and can change with the problem. How the branch and bound would work on a real example would look like:

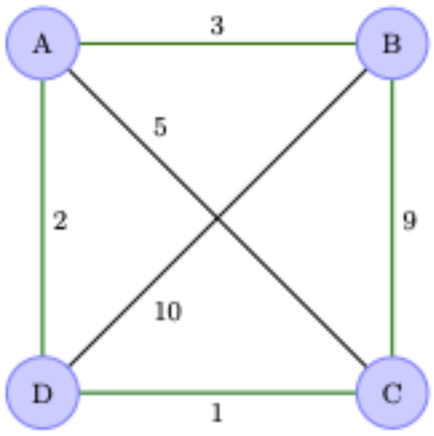


Figure 6: Branch and Bound Example

Just as with the Held-Karp algorithm, a distance matrix would also be created looking like:

	A	B	C	D
A	0	3	5	2
B	3	0	9	10
C	5	9	0	1
D	2	10	1	0

Figure 7: Branch and Bound Distance Matrix

Once this distance matrix is created, the lower bounds can be found in the rows and the columns.

	Smallest Edges	Sum
A	2, 3	5
B	3, 9	12
C	1, 5	6
D	1, 2	3

Figure 8: Branch and Bound Lower Bound Calculation

A lower bound is calculated by adding the minimum two edges for each vertex and dividing by 2.

5 + 12 + 6 + 3 = 26
26/2 = 13

The lower bound is 13.

Figure 9: Branch and Bound Lower Bound Calculation

Branching can then be made, and the lower bounds for the branches can be calculated.

A → B = Cost = 3
A → C = Cost = 5
A → D = Cost = 2

Figure 10: Level 1 Branching

The most promising branch is A to D initially.

The lower bound at each branch can then be calculated by adding edge costs and recomputing bounds for remaining nodes.

A → B = Cost = 3
Remaining nodes: C, D, A.
- *Node A: already used a to b. smallest unused A → D = 2*
- *Node B: already used b to a. smallest unused B → C = 9*
- *Node C: Two smallest edges 1,5*
- *Node D: Two smallest edges = 1,2*

Sum = 2 + 9 + 6 + 3 = 20
Lower Bound = 3 + 20/2 = 13

A → C = Cost = 5
Remaining nodes: B, D, A.
- *Node A: already used a to c. smallest unused A → D = 2*
- *Node B: Two smallest edges = 3, 9*
- *Node C: already used c to a. smallest unused C → D = 1*
- *Node D: Two smallest edges = 1,2*

Sum = 2 + 12 + 1 + 3 = 18
Lower Bound = 5 + 18/2 = 14

A → D = Cost = 2
Remaining nodes: B, C, A.
- *Node A: already used a to d. smallest unused A → B = 3*
- *Node B: Two smallest edges = 3, 9*
- *Node C: Two smallest edges = 3,9*
- *Node D: already used d to a. smallest unused D → C = 1*

Sum = 3 + 12 + 6 + 1 = 22
Lower Bound = 2 + 22/2 = 13

Figure 11: Calculate Bounds for Each Branch at Level 1

Then the next node can be branched to, A to D due to having the lowest bound. So, the algorithm will continue branching from A to D.

Current Path A → D = 2
Nodes remaining B, C
Possible next branches:
- *D → B = 10*
- *D → C = 1*
- *A → D → C = 3*
- *A → D → C → B → A = 15*

This is the most optimal tour

Figure 12: Continued Branching

The optimal tour with a cost of 15 is found. Other routes are then evaluated.

- *A → B = 13 still has potential so test on full path*
- *A → B (3) B → C (9) = 12*
- *C → D (1) D→ A (2) = 15*
- *A → B → C → D→ A = 15*
- *Same cost so route is also acceptable no improvement though.*
- *A → C lower bound = 14 so will not be more efficient.*

Figure 13: Branch and Bound Evaluate Other Promising Branches

The most optimal route was then found which had a total cost of 15.

Approximation Algorithms

Approximation algorithms compromise on optimisation and instead focus on completion time, finding a faster result instead of an optimal one. This can result in algorithms with much smaller time complexities however can provide routes that travel more than double the distance of an optimal route.

Nearest Neighbour Algorithm

The nearest neighbour algorithm was one of the first algorithms used to solve the Travelling Salesman Problem. This is a greedy algorithm that begins at any vertex and follows the edge of least weight from that vertex. At every subsequent vertex, it follows the edge of least weight that leads to a city not yet visited until it returns to the starting point. (Koether, 2016)
Because it is a greedy algorithm, the solutions may not be optimal. However, it is easier to implement and quick to run with a worst-case time complexity of O(n^2), which means that for a problem with ten cities, it will have 100 routes; this is 3628700 less than a brute force search, so it is exponentially faster. (Chase, et al., 2020)

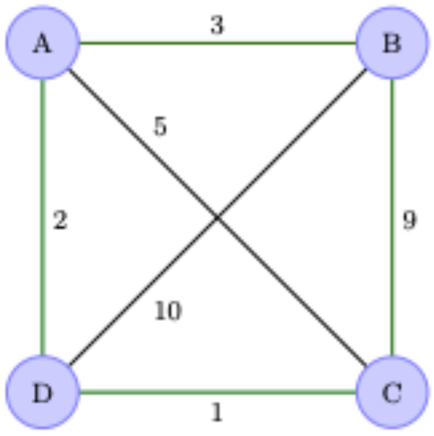


Figure 14: Nearest Neighbour Example

In the example problem, a Nearest Neighbour problem would start at a random city to complete the problem. In this example, it will be city A.

1. A → B = 3
2. A → C = 5
3. A → D = 2

Figure 15: Nearest Neighbour Step 1

Starting at A, all the routes are checked to see which is the closest; A to D is the shortest route, so the salesman will move to D.

1. D → B = 10
2. D → C = 1

Figure 16: Nearest Neighbour Step 2

Once the salesman has moved to city D, the algorithm will search for the nearest unvisited city, which is C.

1. C → B = 9

Figure 17: Nearest Neighbour Step 3

The only other unvisited city is B, so the algorithm will move to city B. Every city has been visited now so the algorithm will return to the starting city.

1. B → A = 3

Figure 18: Nearest Neighbour Step 4

This will then show the completed route of:

1. A → D → C → B → A: 2 + 1 + 9 + 3 = 15

Figure 19: Nearest Neighbour Step 5

This has completed the route much easier than brute-force, and although this example has produced an optimal route, it will not always produce the optimal route.

Hill – Climbing Algorithm

Hill–climbing is a local search algorithm that iteratively moves towards the direction of increasing improvement, aiming to find a better solution in the search space. Starting from a random initial solution in the search space. Starting from a random initial solution, it generates neighbouring solutions and selects the one with the lowest distance. The process repeats until no better neighbours are found.
The algorithm aims to generate an initial solution, which is now the best solution. Then, a neighbouring solution is selected from the best solution. If the neighbouring solution is better than the best solution, set the best solution to be the neighbouring solution. Repeat until the neighbour solution is not better than the best solution. (Howell, 2023)
Using the previous example, this is how the Hill Climbing algorithm looks like in a practical example:

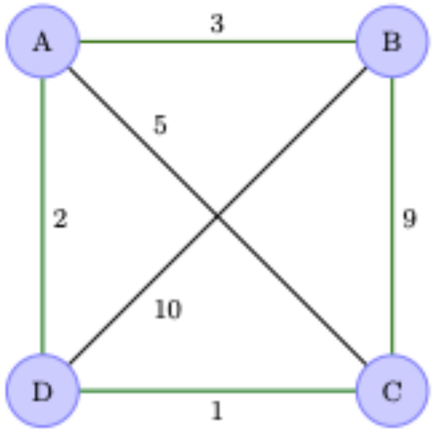


Figure 20: Hill Climbing Algorithm

Firstly, the algorithm will start by selecting a random initial tour:

1. A → B → C → D → A: 3 + 9 + 1 + 2 = 15

Figure 21: Hill Climbing Step 1

The initial tour cost is calculated at 15. Then, the neighbours are swapped and compared:

1. Swap B and C = A → C → B → D → A = 26
2. Swap B and D = A → D→ C → B → A = 15

3. Swap C and D = A → B → D → C → A = 19

Figure 22: Hill Climbing Step 2

Then, once this has been completed, the new routes need to be compared to see if the neighbouring routes are more optimal.

1. A → D→ C → B → A = 15
2. A → B → C → D → A = 15

Figure 23: Hill Climbing Step 3

Two optimal routes are found and can be taken.

Simulated Annealing Algorithm

Simulated annealing is a computational method that borrows inspiration from the field of physics. It simulates the physical process of solid annealing. In this process, a material such as metal or glass is raised to a high temperature and then left to cool, allowing the local regions of order to grow outward, thereby reducing stress and increasing the ductile strength of the material. It is a method generally known to escape local optima. (Ming Tan, 2008)

This is a probabilistic technique that explores the search space by allowing less optimal solutions at the beginning, gradually reducing this acceptance as the temperature decreases. This helps avoid local minima and increases the likelihood of finding the global minimum. (GeeksForGeeks, 2024)

This algorithm follows the same swapping style as the hill climbing algorithm. However, it factors in two variables: temperature and cooling rate. Temperature controls how likely the algorithm is to accept worse solutions; at high temperatures, the algorithm accepts worse solutions more frequently. And as temperature decreases, the algorithm becomes more selective, eventually turning into the hill climbing algorithm.
Then, there is the cooling rate, which determines how quickly the temperature decreases; a slower cooling rate allows for more exploration, increasing the chance of finding a global optimum. A quicker cooling rate reduces execution time but may cause the algorithm to settle in a local optimum.

Christofides Algorithm

The Christofides algorithm is a heuristic algorithm based on the observation that the TSP is equivalent to finding a minimum-weight perfect matching in a complete graph, where the weight of an edge is the distance between two cities. The Christofides algorithm consists of the following steps:

1. Start by finding a minimum spanning tree of the graph.
2. Find the set of nodes that have an odd degree in the minimum spanning tree and form a subgraph with these nodes.
3. Find a minimum-weight perfect matching in the subgraph formed in Step 2.
4. Combine the minimum spanning tree and the perfect matching to form a Eulerian graph.
5. Find the Eulerian tour of the Eulerian Graph.
6. Convert the Eulerian tour back into a Hamiltonian cycle by removing repeated edges.

The Christofides algorithm guarantees a solution that is at most 3/2 times the optimal solution. (Christofides & Whitlock, 1977)

Domain Analysis

This section covers real world applications of the TSP.

Applications of the TSP

There are many different applications of the travelling salesman problem shown in (Gutin & Punnen, 2002).

- Machine Scheduling Problems – the most well studied application area of the TSP. A scheduling algorithm where there are no jobs and the most efficient way to schedule and complete them are solved.
- Cellular Manufacturing – in a cellular manufacturing families of products that require similar processing are grouped together and processed together in a specialised cell to achieve efficiency and cost reductions.
- Arc Routing Problems – a general arc routing problem, where a graph is given and certain edges need to be serviced such as ploughing snow, collecting garbage or street sweeping.
- Frequency Assignment Problem – In a communication network with a collection of transmitters the frequency assignment problem is to assign a frequency to each transmitter from a given set of available frequencies satisfying some interferences constraints. Those constraints can be represented by a graph where each nodes represents a transmitter.

Other Real-World Applications

The first is logistics and transport, where companies need to optimise their delivery routes to reduce costs and improve efficiency. For example, a delivery company may need to find the shortest route that visits all its customers in each area to minimise fuel consumption and delivery time.

Second is manufacturing and production planning, where companies must optimise their production processes to reduce costs and improve efficiency. For example, a manufacturing plant may need to find the shortest route to visit all the machines that need maintenance to minimise downtime and maximise productivity.

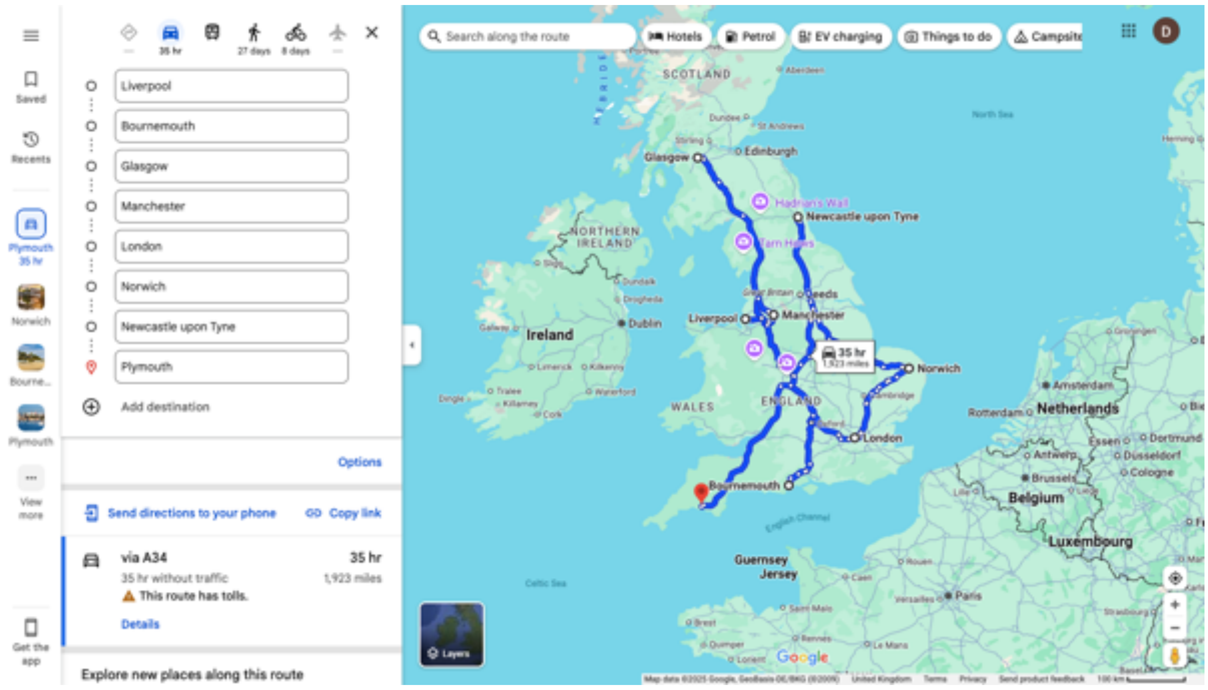
Third is in network design and optimisation, where companies must optimise the routing of information, goods, or services through a network of nodes. For example, a telecommunication company may need to find the shortest route to connect its network nodes to minimise signal loss and improve network performance.

Fourth is DNA sequencing, where researchers must determine how to sequence the DNA fragments to reconstruct the original genome. For example, the TSP can be used to determine the order in which to sequence the overlapping fragments of a genome to reconstruct the original sequence.

And fifth is in robotics and automation, where companies need to optimise the movements of robots or machines to reduce costs and improve efficiency. For example, a robot may need to find the shortest path to visit all the locations in a factory to perform tasks such as picking and placing objects or inspecting machines. (Eyelit Technologies, 2023)

Route Planner – Routora

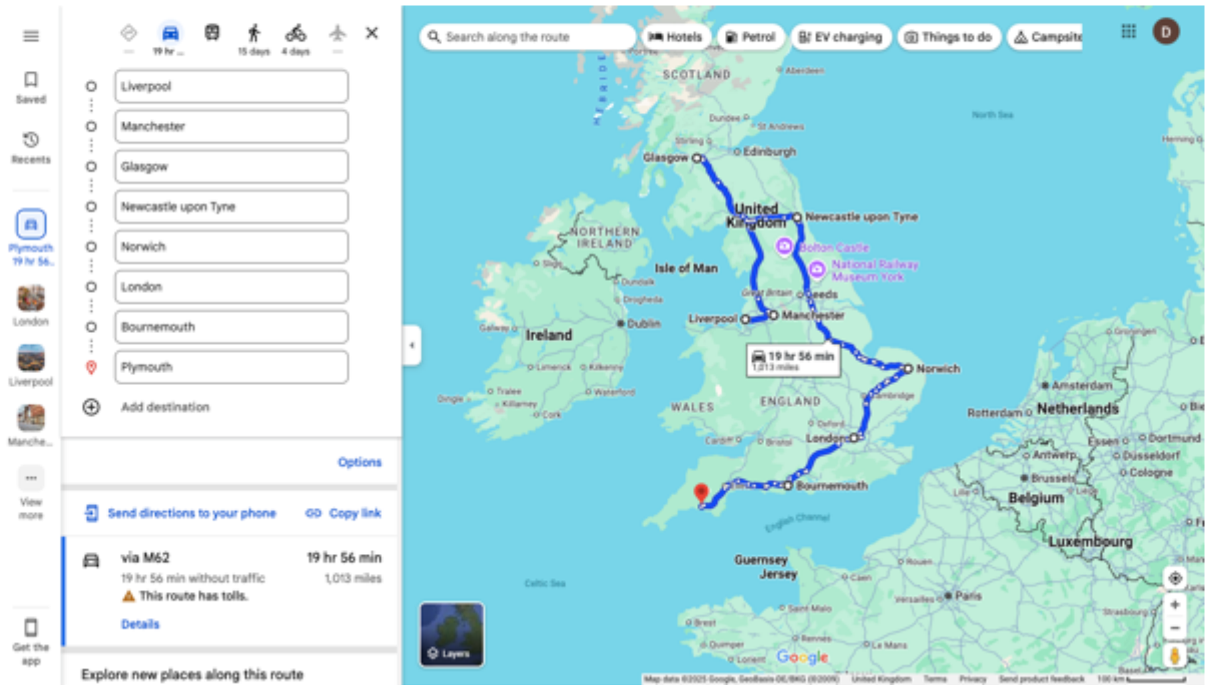
A real-world TSP solver is called Routora, which is both an app and a browser extension. This allows users to enter stops that need visiting, and then the route is optimised and presented to the user. An example of this is visiting different cities. (Routoura, n.d.)


Liverpool
Bournemouth
Glasgow
Manchester
London
Norwich
Newcastle upon Tyne
Plymouth
Add destination

via A34 35 hr
35 hr without traffic 1,923 miles
This route has tolls.

Figure 24: Routera Uncalculated Route

This is the route that needs to be taken; it visits eight different cities in the UK and has a total travel distance of 1923 miles, totalling 35 hours of travel time. Once it has run through routera, the new route now looks like.


Liverpool
Manchester
Glasgow
Newcastle upon Tyne
Norwich
London
Bournemouth
Plymouth
Add destination

via M62 19 hr 56 min
19 hr 56 min without traffic 1,013 miles
This route has tolls.

Figure 25: Routera Calculated Route

This new route is now only 1013 miles with 20 hours of travel time, which shaves 15 hours off the previous route and is much more optimal.
This application is useful for a smaller number of cities; however, there can be uses where there are hundreds and thousands of cities in the problem where the routes need to be computed.

Case Study – University of Waterloo

The Washington Post said in 2018 that it would take an average laptop 1000 years to compute the most efficient route between 22 cities. After seeing this, the University of Waterloo set out to find the optimal route between 49,687 pubs in the UK. This set the record as the longest route ever calculated, taking 250 years of combined computer time, taking 63,739,687 metres or about a sixth of the distance to the moon. The full solution looks like this:

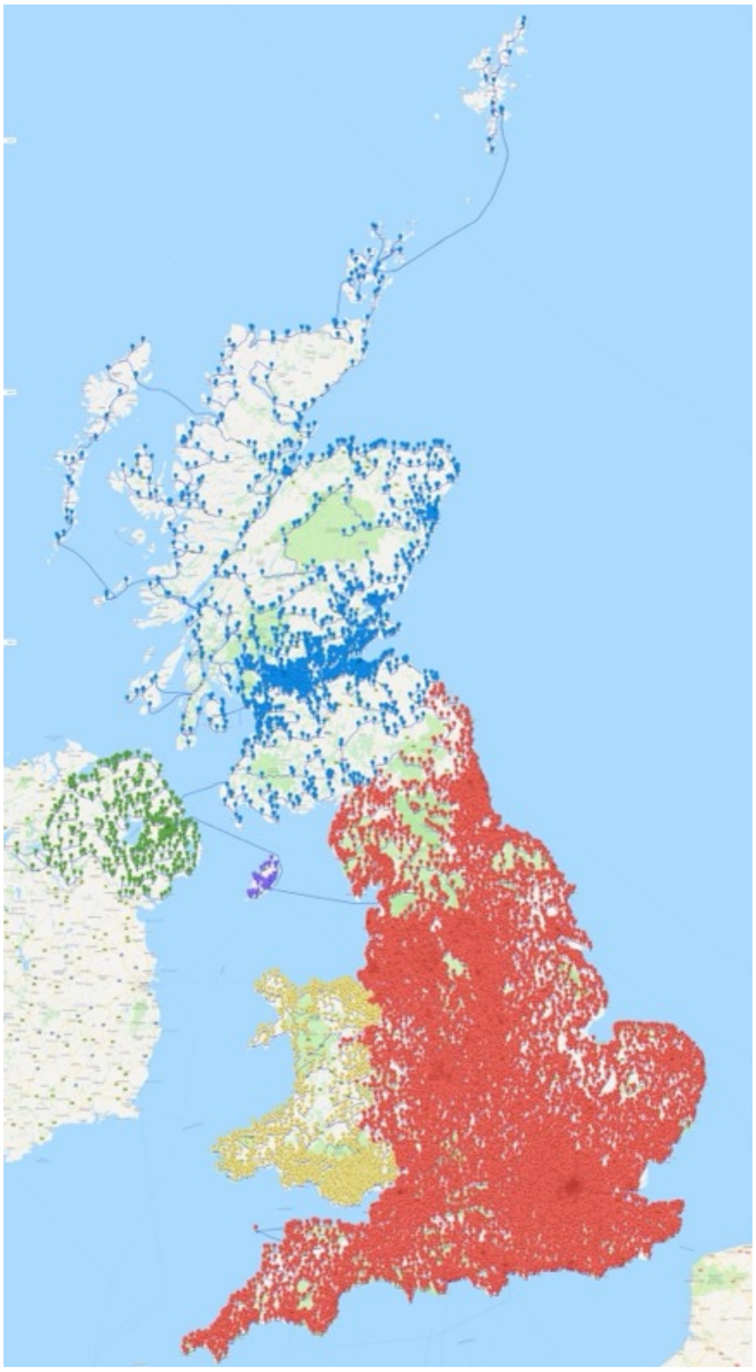


Figure 26: Solved Pub Problem

This set the record for the biggest problem solved, this was not solved via a brute force method because, for a brute force method to solve this problem, it would require 3 followed by 211,761 zeroes, which would be impossible to compute. This was solved by assigning fractions to each road and then using linear programming and dynamic processing to find what fractions added up to an integer and using that to influence the route. (Cook, 2018)

3. Requirement Analysis and Methodology

Requirement Analysis

This requirement analysis will cover all the requirements needed to complete the project.

Problem Definition

The research problem is comparing exact and approximation algorithms when solving the travelling salesman problem. This is to test the capabilities of each algorithm and discover what they are best used for. This is important because the TSP is an NP-Hard problem, and there is no one-size-fits-all solution to the problem, so a clear analysis of the different equations is useful to see how they work, where they can be applied, and which algorithm should be applied to solve the program efficiently.

Stakeholders & User Needs

There are a multitude of different stakeholders this can be applied to, although it may not be helpful to researchers as supercomputers can solve large TSP problems optimally. This technology is not yet available to the public, so the type of stakeholders that can use this information are logistics companies, as they can choose the correct algorithm to suit their needs for whatever route needs to be taken. Furthermore, another stakeholder this can assist is robotics and AI developers to assist with route scheduling and taking the correct path. Furthermore, telecommunications companies can also use this when structuring new network infrastructure, for example, for laying out network and telephone masts.

With these stakeholders, a range of algorithms is required for a complete comparison. The information required in the report for each algorithm should include the route taken as well as the time taken to complete it, the total cost, and the distance travelled. This will show the main features of the algorithms and will allow for the comparison between them.

Functional Requirements

The requirements of this project must be to:

- Use at least three different exact and three different approximate algorithms.
- Calculate the path taken by the algorithm, the cost of the route, the time taken to complete the algorithm and create a visualisation of the route taken.
- Present the data collected in tables and use graphs and visualisations as an easier way of presenting the data.
- Present a comparison between exact and approximate algorithms.

The program must use a range of different algorithms, at least three exact algorithms and three approximate algorithms; more algorithms means that there will be more information to compare, making for a better comparison.

The exact algorithms used are the brute-force algorithm, branch and bound and the Held-Karp algorithm. These have been selected because they provide exact optimal solutions to the problem and can be used as the baseline for the correct route, and choosing different algorithms will also show a time difference in the algorithms. Additionally, nearest neighbour, hill climbing, and simulated annealing will also show different routes taken, as well as the time category, to see if there are any real-time differences between the approximate algorithms.
To gain a complete comparison of the two sets of algorithms, a range of different TSP problems will need to be solved, ranging in both size and complexity. This will fully test all the algorithms to gain knowledge of what algorithms work best with what problems.

The metrics measured will be the route taken; this will show the path taken by the algorithm, which can be used to compare between the optimal route and the route taken to see any differences. To go along with this, visualisations of the route can also be made, which will provide a more abstract view of the route taken. This means that for longer routes, the differences in paths are easily comparable instead of lists of numbers. Then, the total distance travelled will also need to be captured. By doing this, it is possible to see which algorithm is providing the most efficient route and which one is the most inefficient. Additionally, the time taken to complete the algorithm will also need to be recorded; this is to compare the speeds of algorithms and how the speed can be compared to the efficiency of the route. A final metric that can be recorded is the number of cities visited; this will show how many algorithms end up visiting the same place and show the efficiency of the algorithm.

The values collected will be presented in tables to store the data. Not all the data, such as the total route and visualisation, can be presented as this is too much to fit in a table, so this will have to be presented another way. Additionally, the data recorded can be presented in different graphs and visualisations to also have an abstract view of the data recorded, aiding the comparison between the algorithms.

Non-functional Requirements

The system does not deal with sensitive data, such as user information, passwords or any confidential information so no security is needed for the program, if the program did deal with user information such as addresses then a username and password system would be needed for the program as well as a hashing function would be required to hash the stored user information to stop it being accessed incorrectly.

Another factor that will need to be considered will be the usability and presentation of the algorithm; for example, changing the TSP problem should be easy, and changing the algorithm should be easy. The code should be set out into subroutines and commented on to make it easy to understand and read.

To improve usability all code needs to be commented to outline what each subroutine does or what a crucial line of code does additionally it needs to have the same way across all algorithms to change tsp problem this will be the same function across every algorithm in the same place, for example, the function will be the last subroutine of the program so users know to change the algorithm the code goes at the bottom.
Furthermore, when comparing algorithms, one crucial thing is keeping the computer the same for comparing all the algorithms; the algorithms will be run on a standard university computer

at one go to maintain consistency of results. Running the algorithms at different times can cause changes in the performance of the machine, thus affecting results.

Constraints & Assumptions

One large problem will be due to the completion times of exact algorithms on large problems. This is because the time taken to complete TSP problems with a brute force method increases exponentially, so this may cause an issue that will need to be factored in when choosing problems and then running the algorithms. So, an assumption will be made that the TSPLIB solutions will be correct to allow for a correct analysis.
Another assumption is that the visualisations are correct, and that the data collected is accurate so a correct comparison can be made.

Methodology

The methodology that will be used to complete the project will be:

Research Approach

This project will focus on quantitative research, which means it will take exact values from the algorithms and use this data to make the comparison, and this will be used to make a direct comparative analysis of exact vs approximate algorithms.

Data Collection

For the TSP problems used, a library of TSP problems will be used called TSPLIB 95. This is a library of sample instances for the TSP from various sources and various types collated by the University of Heidelberg. (Reinelt, 2013)
The TSPLIB library contains over 100 problems with sizes ranging from 14 cities to 85,900 cities, which will offer a wide range of problems to use to compare the algorithms. The use of the TSPLIB library is that these are solved problems, so in cases where the problem is too large to run, a brute-force search on an optimal value can still be found and used to compare the other algorithms.
Data can be collected and stored in a table; this data can potentially be visualised to help easier understand and present results.

Tools & Technologies

This program will be developed using the IDE Visual Studio, and the code is developed in Python t because of the large number of libraries available to assist the development process.
The libraries that will be used are:

- TSPLIB – this will allow easy integration of the TSPLIB files into the program, allowing for different problems to be solved. (Grant, 2020)
- Itertools – this will provide the function for generating permutations and combinations, which will allow for the different combinations of routes. (Python, 2023)
- NumPy – provides support for multi-dimensional arrays and mathematical operations such as distance matrices. (Oliphant, 2025)
- Matplotlib – Used for visualising TSP solutions such as plotting routes taken. (John D. Hunter, 2025)
- Basic libraries such as time and math to measure execution times and make basic calculations.

Development Methods

There are two development types that could be used in this project:

- Agile Development – software is developed in iterations that contain mini-increments of the new functionalities added. This minimises risks such as bugs, cost overruns and changing requirements.
- Waterfall – the waterfall method is a rigid linear model that consists of sequential phases these are requirements, design, implementation, verification and maintenance focusing on distinct goals. Each phase must be 100% complete before the next phase can start. There is usually no process for going back to modify the project or direction.

Agile would be used to develop the code for example doing a function at a time, for example developing the selected algorithm, then importing a tsp file then adding route visualisations. Whereas waterfall could be used for developing each algorithm to completion for example fully developing and testing the brute force algorithm and then once it is fully complete moving on to the next algorithm only moving to another algorithm once the current one has been fully completed. This project will use a more agile style of development opting to focus on the algorithms before adding features such as route visualisation and tsp file importation.

Implementation Plan

The implementation plan that will be followed when completing this project will be:

- Gathering TSP problems for testing - means that a fair test can be completed amongst all the different values to get a proper comparison.
- Algorithm Selection – choose the correct algorithms to get a proper comparison between exact and approximate algorithms.
- Run different algorithms on the same data sets and change data sets based on size and complexity.
- Compare performances of algorithms.
- Interpret results to see the similarities and differences.

Testing & Validation

The algorithms will be tested to view the performance of the algorithms this will be by running them on different sizes and complexities of problems. This can be used to measure execution time, solution cost and algorithm efficiency.

Testing can be set into functional testing and non-functional testing.

- Functional testing focuses on unit, integration, system and acceptance.
- Nonfunctional focuses on security, performance, usability and compatibility.

This code is not focused on the nonfunctional focuses, there is no need for security testing as there is no log in features or user data stored in the code so no security testing is not needed, there will not be mass use of the program as this is used for comparing different algorithms, so no performance, usability or compatibility testing is needed either.

The focus on this testing is for functional testing so this will be tests to ensure that each aspect of the program is working correctly and handles errors correctly when presented.

4. Design of Artefact

Design Overview

The main purpose of the artefact is to use different exact and approximation algorithms to measure their strengths and weaknesses on a range of different Travelling Salesman Problems to get a good comparison of the two. The algorithms will be presented with a TSP problem and will then solve it.

This artefact aims to allow the user to select a TSP Problem and then allow the user to run the problem against different algorithms, once this has been completed, results will be outputted, such as completion time, total cost, route taken as well as a visualisation of the route completed for an abstract comparison.

Due to the size and complexities of the different algorithms, they will be kept in different files in the same directory and named with the algorithm name, so the user must select the name of the algorithm and run the file. This is also because some algorithms take the cities as they are, and others will need to convert the TSPLIB file into a distance matrix, so keeping them separate is the tidiest and most efficient way of presenting the algorithms. Users will have to select the TSPLIB file and enter it before running the program as the files all have unique complex names that are harder to remember, such as "brd14051", so users can carefully select and input the file name before having to run the program, so they are not trying to remember the file name and potentially inputting incorrectly.

Considerations need to be made to ensure units and scale are the same across all algorithms, for example, the visualisation needs to be to the correct scale, and time taken needs to be in the same units, for example, in seconds, to make the comparison as easy as possible. Furthermore, the code will need to be modular and structured into neat subroutines, with properly named variables as well as all code commented this is to ensure that it can easily understood or adapted to any other user's needs.

System Architecture

A system architecture diagram shows the functionality of the artefact and what each part of the program does. The system architecture diagram for this artefact will show each algorithm and the process carried out by all of them.

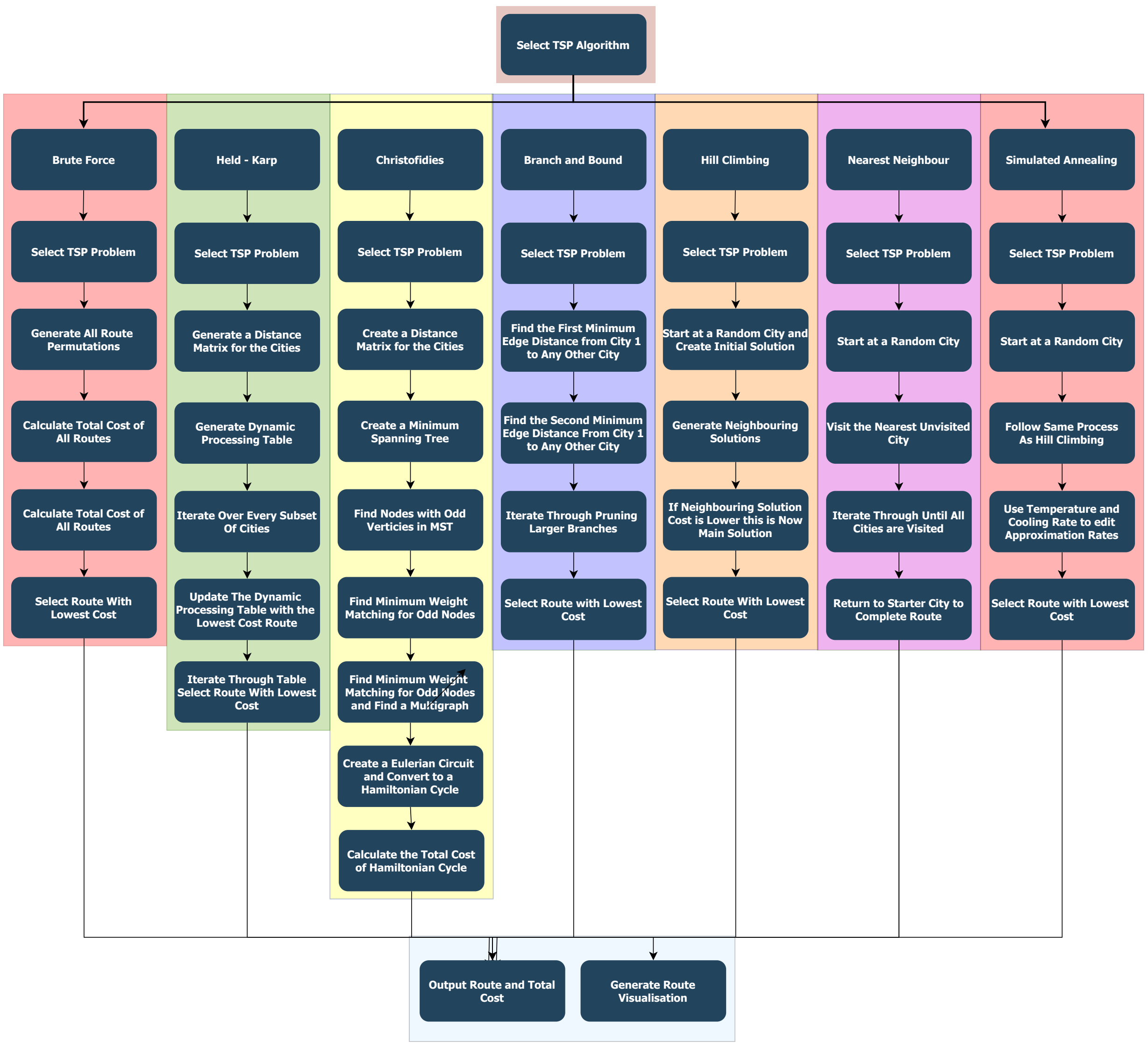

Select TSP Algorithm
Brute Force
Select TSP Problem
Generate All Route Permutations
Calculate Total Cost of All Routes
Calculate Total Cost of All Routes
Select Route With Lowest Cost
Held - Karp
Select TSP Problem
Generate a Distance Matrix for the Cities
Generate Dynamic Processing Table
Iterate Over Every Subset Of Cities
Update The Dynamic Processing Table with the Lowest Cost Route
Iterate Through Table Select Route With Lowest Cost
Christofidies
Select TSP Problem
Create a Distance Matrix for the Cities
Create a Minimum Spanning Tree
Find Nodes with Odd Verticies in MST
Find Minimum Weight Matching for Odd Nodes
Find Minimum Weight Matching for Odd Nodes and Find a Multigraph
Create a Eulerian Circuit and Convert to a Hamiltonian Cycle
Calculate the Total Cost of Hamiltonian Cycle
Branch and Bound
Select TSP Problem
Find the First Minimum Edge Distance from City 1 to Any Other City
Find the Second Minimum Edge Distance From City 1 to Any Other City
Iterate Through Pruning Larger Branches
Select Route with Lowest Cost
Hill Climbing
Select TSP Problem
Start at a Random City and Create Initial Solution
Generate Neighbouring Solutions
If Neighbouring Solution Cost is Lower this is Now Main Solution
Select Route With Lowest Cost
Nearest Neighbour
Select TSP Problem
Start at a Random City
Visit the Nearest Unvisited City
Iterate Through Until All Cities are Visited
Return to Starter City to Complete Route
Simulated Annealing
Select TSP Problem
Start at a Random City
Follow Same Process As Hill Climbing
Use Temperature and Cooling Rate to edit Approximation Rates
Select Route with Lowest Cost
Output Route and Total Cost
Generate Route Visualisation

Figure 27: System Architecture Design

This diagram has all seven algorithms as well as the main process that the algorithm does to complete the TSP problem. These main processes will be slightly different for every algorithm and result in all algorithms outputting the optimal route and total cost as well as generating the route visualisation.

Algorithm Design

This section will showcase the design of the different algorithms needed in the artefact, in both an ordered list as well as pseudocode following the pseudocode standards set by Cambridge International. (Cambridge Assessment International Education, 2023)

Brute Force

The Brute Force Algorithm is one of the simplest in the program, it works by taking the TSPLIB File, generating a list of cities from this and then generating all possible permutations of routes taken to visit these cities to find the shortest route. This algorithm will need to run the following steps:

1. Select TSP Problem

2. Convert problem nodes into a list of cities
3. Create all possible permutations of routes that can be taken between cities
4. Select a route with the shortest possible cost
5. Output shortest route, total cost, time taken and generate route visualisation

Pseudocode

This pseudocode shows the functions needed specifically by the brute force algorithm, creating the total cost of the route as well as the brute force algorithm. The total cost is calculated in a separate function where the distance from the current city is added to the next city, and that is added to a variable called total_cost. then the main algorithm is run, calling in that calculate_cost function and running every permutation, comparing the cost of each permutation until the smallest value is found.

```
//function to calculate total cost of journey
FUNCTION calculate_cost:
   SET total cost TO 0
   FOR EACH city IN the route
      ADD the distance from the current city to the next city TO
total_cost
   RETURN total cost
END FUNCTION

//function to run brute force algorithm
FUNCTION brute_force:
   SET optimal route TO none
   SET min cost TO infinity
   GENERATE all permutations of cities

   FOR EACH permutation in permutations
      COMPUTE total_cost using FUNCTION calculate_cost
      IF total_cost < min_cost THEN:
         UPDATE optimal_route to current permutation

   Return optimal_route and min_cost
END FUNCTION
```

Figure 28: Brute-Force Pseudocode

Held – Karp

The Held-Karp algorithm is different to the brute force algorithm because it requires the list of cities to be stored in a distance matrix, this is a table where the distances between every city are recorded, as shown in the example in a previous chapter. Once the distance matrix is created, a dynamic processing table also needs to be generated; this stores the most optimal routes between each city as it iterates through all the possible routes. Once this has been completed, the algorithm will recursively trace back through the dynamic processing table to find the most efficient route. The steps will look like:

1. Select TSP Problem.
2. Generate a distance matrix for the cities.
3. Generate a dynamic processing table.
4. Iterate over every subset of cities.
5. Update the dynamic processing table with the route with the least cost.
6. Iterate through the table and select the route with the lowest cost.

Pseudocode

```
FUNCTION tsp_dynamic_programming
   LOAD the TSP problem
   CREATE a list of cities
   CREATE a distance matrix to store city to city distance

   CREATE a dynamic processing table
   SET DP table to 0 for the starting city

   FOR EACH subset of cities:
      FOR EACH last visited city in subset:

      SKIP if city not in subset

      FOR EACH next city not in subset:
         CALCULATE new mask by adding next cities distance
         CALCULATE new distance

         IF new distance < current mask in DP:
            UPDATE DP with new distance
            MOVE to next city

   BACKTRACK through DP from last city to find optimal route

END FUNCTION
```

Figure 29: Held-Karp Pseudocode

This pseudocode starts by extracting the weights between the cities and storing them in the distance matrix, and then running loops to go through cities, checking if they are visited and then updating the dynamic processing table, which was initialised at 0. When a more optimal route is found, a new mask is added to the dynamic processing table, and upon completion, the algorithm backtracks through the table to complete the most optimal route.

Branch and Bound

The branch and bound algorithm start at a random city and then finds the first and second closest links to the city and then if the path looks too expensive that branch is pruned and moves on to another branch. The steps this algorithm takes are:

- Select and load the TSP problem
- Create a distance matrix of problems
- Find the first minimum edge distance from City 1
- Find the second minimum edge distance from City 1
- Recursively go through the distance matrix pruning branches that look too costly on the first and second minimum edge.
- Construct branches to find the optimal path.

Pseudocode

The branch and bound pseudocode will look like:

```
FUNCTION branch_and_bound_tsp:

   LOAD the TSP problem
   CREATE a list of cities
   CREATE a distance matrix with all city-to-city distances

   FIND the first minimum edge distance from City 1
   FIND the second minimum edge distance from City 1

   SET best_cost TO a very large number
   SET best_route TO empty

   CREATE an empty path starting at City 1
   MARK City 1 as visited

   CALL recursive_search(current_city = City 1, path,
current_cost = 0)

   RETURN best_route and best_cost
END FUNCTION
```

```
FUNCTION recursive_search(current_city, path, current_cost):

   IF all cities have been visited THEN
      ADD distance from current_city back to starting city
      SET total_cost = current_cost + return_distance

      IF total_cost < best_cost THEN
         UPDATE best_cost with total_cost
         SAVE current path as best_route
      END IF
      RETURN
   END IF

   FOR each city not yet visited DO
      ADD  distance  from  current_city  to  next_city  to
current_cost

      IF current_cost is less than best_cost THEN
         ADD next_city to path
         MARK next_city as visited

         CALL recursive_search(next_city, path, current_cost)

         REMOVE next_city from path
         MARK next_city as unvisited
      END IF
   END FOR
END FUNCTION
```

Figure 30: Branch and Bound Pseudocode

Christofides

The Christofides algorithm finds cities that have an odd number of vertices, this is because they are limited in which way they can be visited and will sometimes require the salesman to backtrack to the previous city, it will find the minimum weight perfect matching of the odd nodes and then connect that with the even nodes to complete the route. The steps the algorithm will need to take are:

1. Select TSP Problem
2. Create a distance matrix from the problems
3. Find the minimum spanning tree from the distance matrix
4. Find nodes with an odd degree in the MST
5. Find the minimum weight perfect matching for the odd-degree nodes
6. Combine the MST and the matching to form a multigraph
7. Find a Eulerian Circuit in the multigraph
8. Convert the Eulerian Circuit to a Hamiltonian Cycle by skipping repeated nodes
9. Calculate total cost of the Hamiltonian cycle.

Pseudocode

The pseudocode for this will look like:

```
FUNCTION christofides_tsp:

   CREATE a distance matrix using the coordinates of all cities

   CREATE a minimum spanning tree (MST) from the distance matrix

   FIND all nodes in the MST that have an odd degree

   FIND the minimum weight perfect matching between all odd-degree nodes

   COMBINE the MST and the matching edges to create a multigraph

   FIND a Eulerian circuit in the multigraph

   CONVERT the Eulerian circuit to a Hamiltonian cycle by skipping repeated nodes

   CALCULATE the total cost of the Hamiltonian cycle

   RETURN the route and its total cost
END FUNCTION
```

Figure 31: Christofides Pseudocode

Nearest Neighbour

The nearest neighbour algorithm is another simpler algorithm, the algorithm gets all the cities, starts at a random city and then visits the closest unvisited city until there are no more unvisited cities. The steps the algorithm will need to take are:

1. Select TSP Problem
2. Create a list of cities
3. Start at city 1
4. Move to the nearest city
5. Move to the nearest unvisited city
6. Iterate through the list until all cities have been visited
7. Return to start city
8. Calculate the total cost of the trip

Pseudocode

```
// Function to run the nearest neighbour algorithm
FUNCTION nearest_neighbour(graph, cities):
    SET num_cities TO LENGTH(cities)
    SET start_city TO cities[0]
    SET unvisited_cities TO COPY OF cities
    REMOVE start_city FROM unvisited_cities

    SET current_city TO start_city
    SET path TO [start_city]
    SET total_cost TO 0

    WHILE unvisited_cities IS NOT EMPTY:
        SET nearest_city TO city IN unvisited_cities WITH
MINIMUM graph[current_city][city]
        ADD graph[current_city][nearest_city] TO total_cost
        SET nearest_city TO current_city
        APPEND current_city TO path
        REMOVE nearest_city FROM unvisited_cities
        SET current_city TO nearest_city

    // Return to the starting city
    ADD graph[current_city][start_city] TO total_cost
    APPEND start_city TO path

    RETURN path, total_cost
```

Figure 32: Nearest Neighbour Pseudocode

This algorithm put the list of cities in an array called unvisited cities, it then finds the closest city from the start city, then setting the second city to a variable called current city, then using the graph function finds the closest unvisited city, moving through the cities, when a city has been visited it is removed from the unvisited cities list to prevent the algorithm moving to the same city, once all the cities have been visited then the algorithm returns to the start. The values found from the graph function also are used to increment the cost of every trip so that the correct cost of the journey can be found.

Hill Climbing

The hill climbing algorithm starts at a random city, creates an initial route, and then swaps out certain routes with others to find more efficient routes, swapping the places of certain cities as shown in the initial example in Figure 14, swapping cities b and d until the most optimal route is found. The steps the algorithm would need to take are:

1. Select TSP problem.
2. Start at a random city and create an initial route.
3. Swap out each node with a different node and compare routes.
4. Swap out nodes until the most efficient is found and repeat for every node in the route.
5. Calculate total cost.

Pseudocode

```
FUNCTION hill_climbing_tsp:

   START with a random route that visits all cities once
   CALCULATE the total distance of the current route

   REPEAT
      GENERATE all neighbouring routes by swapping two cities
      FIND the neighbour with the lowest total distance

      IF the best neighbour is not better than the current
route THEN
         STOP the loop
      ELSE
         REPLACE current route with best neighbour
   UNTIL no better neighbour is found

   RETURN the best route found and its total distance
END FUNCTION
```

Figure 33: Hill Climbing Pseudocode

This pseudocode starts by creating a random starting point and then creates a random route, then swapping out each neighbour with a different node comparing the total cost to find the shortest and most efficient route. Returning the best route in the end.

Simulated Annealing

The simulated annealing algorithm works very similarly to the hill climbing algorithm, however factoring in cooling rate and temperature to alter the efficiency of the program. The steps this algorithm would need to take are:

- Select TSP problem
- Set cooling rate and temperature
- Start at a random city and select initial route
- Swap out each node with a different node and compare routes
- Swap nodes until the shortest and most efficient route is found and calculate cost.
- Higher temperature means less optimal route is found but route is found faster
- Higher cooling rate means algorithm will search for route for longer.

Pseudocode

```
FUNCTION simulated_annealing_tsp:

   START with a random route that visits all cities once
   CALCULATE the total distance of the current route
   SET best_route to current route
   SET best_distance to current distance

   SET initial temperature
   SET cooling rate
   SET number of maximum iterations

   FOR each iteration from 1 to max_iterations DO

      GENERATE all neighbouring routes by swapping two cities
      SELECT one random neighbour from the list

      CALCULATE the distance of the neighbour route

      IF neighbour distance is better than current distance
THEN
         ACCEPT the neighbour as the new current route
      ELSE IF a random probability is less than the acceptance
probability THEN
         ACCEPT the worse route with some chance

      IF current route is better than best_route THEN
         UPDATE best_route and best_distance

      REDUCE the temperature by multiplying with cooling rate

   END FOR

   RETURN the best route found and its total distance
END FUNCTION
```

Figure 34: Simulated Annealing Pseudocode

This pseudocode is very similar to hill climbing starting with a random route and swapping out nodes only allowing users to set cooling rates and temperatures to alter acceptance and probability when solving the problem.

Data Structures and Representation

The data structures in the report consist of: TSPLIB files to store the TSP problems as well as the presentation of the results from the algorithms and the visualisations created from the results. The presentation of these algorithms must be planned to maintain consistent results to allow for a correct comparison.

TSPLIB Design

TSPLIB files allow users to read and write TSP problems easily this is essential for this artefact as different TSP problems are needed, so not only is being able to test the same problem across multiple algorithms important, but also being able to test different problems across an algorithm is important to complete a proper test of the algorithms. For example, if the distances were to be hardcoded into the problems, it would become much more difficult to swap problems and duplicate the problems onto different algorithms as mistakes typing in the coordinates could ruin the results and then the comparison. A standard TSPLIB file will look like this:

```
NAME: ulysses16.tsp
TYPE: TSP
COMMENT: Odyssey of Ulysses (Groetschel/Padberg)
DIMENSION: 16
EDGE_WEIGHT_TYPE: GEO
DISPLAY_DATA_TYPE: COORD_DISPLAY
NODE_COORD_SECTION
```

```
1 38.24 20.42
2 39.57 26.15
3 40.56 25.32
4 36.26 23.12
5 33.48 10.54
6 37.56 12.19
7 38.42 13.11
8 37.52 20.44
9 41.23 9.10
10 41.17 13.05
11 36.08 -5.21
12 38.47 15.13
13 38.15 15.35
14 37.51 15.17
15 35.49 14.32
16 39.36 19.56
EOF
```

Figure 35: TSP File Example

- The file will start with the name of the problem, the type of file, any comments and the size of the problem. In the example in the figure above the problem is called Ulysses16.tsp and is a TSP file modelled off the Odyssey of Ulysses and has a dimension of 16 so contains 16 co-ordinates.
- The next bullet point is for 'EDGE_WEIGHT_TYPE' in this problem, it is 'GEO', which means that the distances between the nodes are based on geographical longitude and latitude coordinates. (Grant, 2022)
- Thirdly, the next bullet point is 'DISPLAY_DATA_TYPE', in this example, it is 'COORD_DISPLAY', which means the points can be used as co-ordinates on a 2D plane, and that is the direct co-ordinate provided. (Grant, 2022)
- After this, the nodes are listed, for example, in the figure above, "1 38.24 20.42" is listed as the first node, the first number stands for the index number of the node so this would be the first node, and the second numbers are the x and y coordinates of the node. (Grant, 2022)
- Finally, after every node has been listed, the file must end with "EOF", standing for End of File and signifies the end of the file and the termination of the problem, signalling that the problem has been completed. So, the program would never be terminated upon completion.

TSP Pseudocode

TSP files still need to be called into the program to be used, so to do this a function will be created and located at the top of every algorithm to maintain consistency and best practice amongst all the algorithms.

```
IMPORT TSPLIB95 //TSPLIB library needs importing

FUNCTION load_tsp_file(filename)
   Problem = LOAD TSP problem FROM filename
   Cities = LIST of all nodes IN problem
   Graph = FOR each city in cities:
      GET weight BETWEEN city i AND city j FROM problem
   END FOR
   G = GET graph FROM problem

   Return cities, graph, problem
```

Figure 36: Import TSPLIB Pseudocode

- This pseudocode starts by importing the TSPLIB library, as this is crucial for extracting the TSP problem.
- Secondly, a function is defined to load the TSP file and call the variable filename, as this will store the file path of the TSP problem.
- The problem variable will load the tsp problem from the path stored in the filename variable.
- The cities variable will then store a list of all the cities in the TSP problem.
- Then, the graph and G variable will find the distance between the two cities to be used when creating a visualisation of the route taken.
- The cities, graph and problem variables are returned for later use.

Route Visualisation

Route visualisations can be used to visualise the route taken by the algorithm, this allows for an abstract interpretation of the results and allows for a quick comparison of the route, as it is easier to visualise a graph than a list of 100 cities to see the differences in the algorithms. The visualisations will be plotted using the library matplotlib; this is a visualisation library designed for Python, allowing users to plot graphs with data created from the code. (matplotlib, 2024)
The graph will need to be a plot, plotting every node location with a line connecting each node to where they visit next. An axis on the side will also be useful to show the distances between the cities.
An example of this from the matplotlib website is like this:

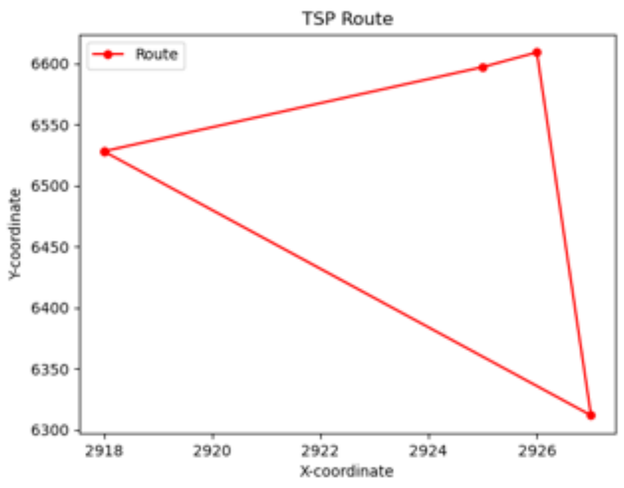


Figure 37: Example Simple Plot

This uses the matplotlib.pyplot library to create a visualisation of red dots against a red line, which is what the artefact visualisation will be modelled after. (matplotlib, 2024)

Pseudocode

```
FUNCTION plot_route
   City_coords GET co-ordinates from PROBLEM
   Route_coords = city_coords for city in route
   Route_x = x for x, y in route_coords
   Route_y = y for x, y in route_coords

   APPEND route_x[0] TO route_x
   APPEND route_y[0] TO route_y

   PLOT route_x, route_y AS a red line with markers
   SET title TO [algorithm name]
   DISABLE grid
   DISPLAY plot
```

Figure 38: Plot Visualisation Pseudocode

- Firstly, getting the coordinates from the problem and assigning them to the city_coords variable.
- Secondly, get the coordinates of each city in the route and assign it to a variable called route_coords. This is so it can be run for each city in the problem.
- Then extract all x coordinates and place them in the route_x variable. And do the same for the y coordinates placing them in the route_y variable.
- Once all the data is collected, plot the graph as a red line with marks without a grid and set the name to the algorithm name.

User Interface Design

There is a very limited need for a user interface in this artefact as there is no user input whilst the program is running; the only user interface requirements are outputting data from the algorithm once it has been run. This requires the output of the route, algorithm cost, time taken and visualisation in a clear format that can be easily understood and read. This should look like:

```
[ALGORITHM NAME]
BEST PATH: //path taken
LENGTH OF PATH: //length of path
MINIMUM COST: //total distance travelled in the route
TIME TAKEN: //time taken to run in minutes and seconds
```

Figure 39: Output Design

It is important to keep the units the same in the time taken and minimum cost sections, as changing units will ruin the comparison between the algorithms.
This user interface is a command line interface; however, it could be further developed into a graphical user interface. This will be discussed in a later chapter titled, evaluation of artefact.

Artefact Testing and Evaluation Criteria

Once the artefact has been developed, it will need to be tested to ensure that the program is working as intended, one example is through robustness and error testing will be used by trying to force errors to make sure the code is running correctly.
Once the code has been tested, the results can start to be recorded for the different tests. Multiple tests will be running with different size TSP problems to measure a range of strengths and weaknesses on different algorithms.

Why TSPLIB Files

TSPLIB files are being used for this test because they are TSP problems used for regular problem solving. They are a collection of benchmark problems that can be used to fairly test algorithms against a different range of products. This will provide a better test of results than custom generating a set of problems which will be time consuming in cases of large 1000+ node problems.

Functionality Tests

Functionality tests will be run on different aspects of the algorithms to ensure no bugs or errors in the code before it is run on the official testing to compare the algorithms.
An example of a functionality test is on the functionality of the algorithms, a testing table will be used for example:

Test	TSP Problem	Purpose	Outcome
Brute Force Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Held Karp Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail

Branch and Bound Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Christofides Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Nearest Neighbour Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Hill Climbing Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Simulated Annealing Functionality	Att8.TSP	Ensure algorithm runs without error	Pass/Fail
Visualisations are different for different algorithms.	Ulysses16.TSP Using Nearest Neighbour and Christofides algorithms	Ensure different visualisations are being plotted not just the same route regardless of algorithm	Pass/Fail
Passing an incorrect TSP file name does not return a result	Att9.TSP	Ensures that the loading tsp file function loads the file correctly	Pass/Fail
Simulated Annealing cooling rate and temperature changes return different results	Ulysses16.TSP	Ensures that the temperature and cooling rate affect results, so algorithm works correctly	Pass/Fail

Table 1: Functionality Test Example

This will test each algorithm to ensure they run correctly without any issues as well as additional functions such as loading in a tsp file and creating route visualisations also work correctly.

Result Criteria

The algorithms will be tested against sixteen different TSP problems these are:

- D4 – 4 node problem modelled off 4 locations in Germany.
- Att8 – 8 node problem based off 8 capital cities in mainland USA.
- Burma10 – 10 node problem based off 10 states in Myanmar.
- Burma14 – 14 node problem based off states in Myanmar.
- Ulysses16 – 16 node problem based off the Odysses of Ulysses.
- Ulysses22 – 22 node problem based on the Odyssey of Ulysses.
- Att48 – 48 node problem based on the 48 capitals of mainland USA.
- Eil51 – 51 node artificially generated route.
- Berlin52 – 52 node problem based off 52 locations in Berlin, Germany
- Brazil58 – 58 node matrix based of 58 locations in Brazil
- Pr76 76 node artificially generated problem.
- Ch150 – 150 node artificially generated problem.
- Gr202 – 202 node problem based on American cities.

- Ali535 – 535 node problem based on airports around the globe.
- Nrw1379 – 1379 node, artificially generated problem
- Pla7939 – 7939 node problem that has been artificially generated to test upper bounds of approximation algorithms.

The sixteen different tests range in size from 4 nodes to 7939 nodes this is to properly test the algorithms. The upper size limit is 7939 nodes due to the time taken to complete the algorithm being limited to 10 minutes in execution time as this is considered a reasonable time to complete the problem. Due to the complexity of some algorithms, such as the brute force algorithm, testing this against larger problems will take too much time, so the time taken will have to be worked out mathematically to get a full set of results for a proper comparison. As the TSPLIB library comes with solution information, there is still a minimum cost, which can be substituted for the brute force minimum cost. it is important to use different size tests because some algorithms are stronger on smaller algorithms whilst not efficient on larger problems, and some are stronger on larger algorithms and not efficient on smaller problems.

Four measures will be output to the user: the best path taken to complete the route, the length of the path taken to see if there is any backtracking in the algorithm and to ensure all cities are being visited when the algorithm runs, the minimum cost to show the distance travelled when compared with other algorithms will show which algorithms are travelling further and total time taken to run the algorithm.

Results Table

The results will be recorded in separate results tables for each TSP problem an example of how this will look is:

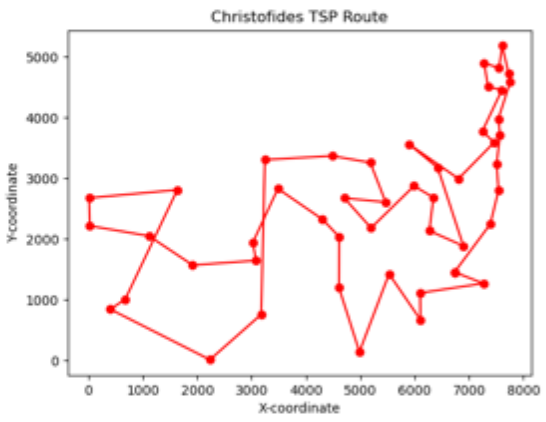
Name of Algorithm	Minimum Cost	Time Taken
Brute-Force		
Held-Karp		
Branch and Bound		
Nearest Neighbour		
Hill Climbing		
Christofides		
Simulated Annealing		

Figure 40: Example of Results Table

This table will store the data for each algorithm per algorithm, so there will be sixteen different tables for all the problems. This means that each algorithm can be compared per problem. This is different to having one table per algorithm storing all six problems because then it turns into a comparison of the problems and not a comparison of the algorithms.

Results Visualisations

Two types of visualisations will be used in the results sections the first is a visualisation of the route taken; this will be used because it will be so much easier to view differences in routes graphically which is something that is done every day than viewing a list of a 100-node route seeing where the differences are. For example, this is the comparison between the two:

1, 8, 22, 16, 3, 41, 34, 14, 25, 39, 48, 5, 42, 10, 35, 45, 24, 26, 4, 2, 29, 32, 21, 47, 11, 13, 23, 12, 15, 40, 9, 33, 20, 46, 18, 36, 6, 30, 43, 27, 17, 19, 37, 28, 7, 44, 31, 38, 1 *Att48 Christofides Route*	 *Att48 Christofides Route Visualisation*

This shows that the route visualisation is much clearer than printing the route.

Data visualisations can be used to present the results discovered from the program running; these can be used to show the comparison graphically between exact and approximate algorithms. Two examples of the visualisations that can be plotted are:

- Stacked bar chart – stacking the time taken for each algorithm for different problems.
- Heat map – showing the approximations of the approximate algorithms highlighting good and bad performances.

This information can be used alongside the route visualisations to gain a graphical comparison between the algorithms.

5. Development of Artefact

This section will cover the development of the Artefact to run and test the algorithms. A full view of the code written will be available in the appendix section at the end of the report.

Implementation and Environment Tools

This artefact is developed in the programming language Python, which is a high-level language that is widely used in web applications, software development, data science and machine learning, Python is popular because of its versatility and efficiency. Python is also an interpreted language which means the code is translated and run line by line. This is useful because it means that errors can easily be found quicker during development. (Amazon Web Services, 2025)
The artefact will be developed in Visual Studio Code, which is an integrated development environment designed by Microsoft. Available for free on Windows, macOS and Linux, supporting development languages such as Java, Python, C, C++, C#, PHP, Go, .NET and many more. Containing debugging, translating and syntax support to assist with the development process. (Microsoft, 2025)

GitHub Repository

The code for this project as well as being available in the appendix is in a GitHub repository, the reason for this is because it stores the code of the project on a GitHub server not on a local device which means it can be accessed remotely by multiple devices without having to share the code files. Additionally, an advantage to this is it will show version control so any changes made can be viewed. (Worley, 2025)

Code Organisation and Structure

This section will now cover the methods needed to structure the code.

SOLID Principles

The code will be developed in a modular structure following best practices when it comes to variable names, comments and indentation. Additionally, following SOLID principles these are:

- Single responsibility principle – every class or module is responsible for one part of the
- Open-Closed Principle – classes should be able to be extended without having to be modified or changed.
- Liskov Substitution Principle – every derived class should be substitutable for its parent class. This is to ensure that derived classes extend the base class without changing behaviour.
- Interface Segregation Principle – make fine-grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use. It is better to have many smaller interfaces with essential information than a few bigger ones with features that aren't used.
- Dependency Inversion Principle – high-level modules should not depend upon low-level modules. Both should depend on abstractions; this is to make the code more flexible, agile and reusable.

Following these principles will mean that the code is easier to understand, maintain and extend designs, avoiding issues and building adaptive, effective and agile software using these SOLID principles. (Watts, 2024)

Each algorithm will be in a separate file to prevent any crossovers in the code and to keep each algorithm independent. Each file is named after the correct algorithm and will be in the same directory as the TSP problems.

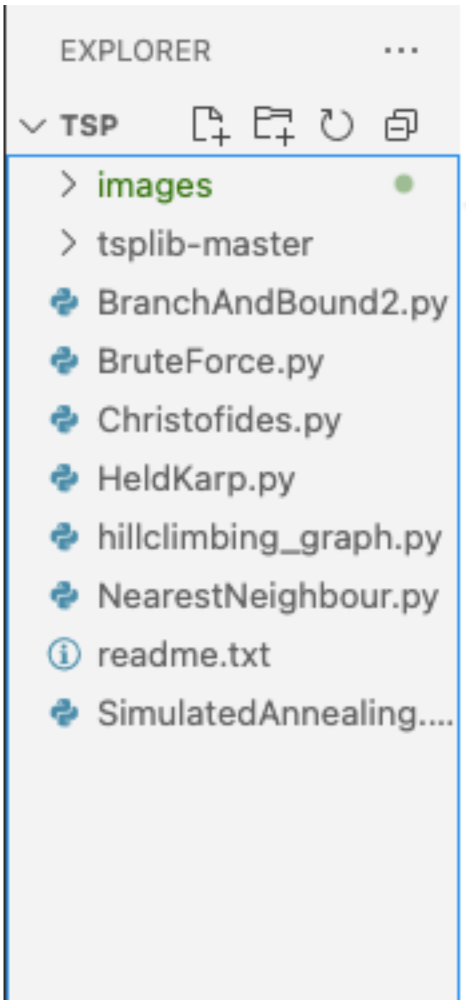


Figure 41: Artefact Directory

The figure above shows the artefact directory; each file is named correctly, and TSPLIB problems are stored in the tsplib-master folder. Any images that can be used in visualisations are presented in the images folder, and a read-me file has also been added to assist users in running any algorithms.

Algorithm Implementation

This section will cover how the exact and approximate algorithms were implemented in Python to solve the Travelling Salesman. The goal of the algorithms is to accept a TSPLIB problem and then solve that problem using the algorithm and return the cost, time, number of cities visited and the route that was taken.

Brute-Force

The initial code implementation is based on an example seen in (Liang, 2024). However, it was adapted and rewritten to fit the needs of this project.

Load TSP File Function

This first function is used to load the TSP problem from the TSPLIB file to run the problem.

```
# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem
```

Figure 42: Brute - Force Load File

1. This code starts by using the 'tsplib95.load' function to load the TSP problem from the filename and assign it to the variable problem.
2. The next line extracts all the nodes in the problem and assigns them to the variable cities.
3. After this, the next two lines are used to gather the distances between the cities and then use those distances to create a graph between all the nodes.
4. Returns the cities, graph and problem variable for later use.

Calculate Cost Function

The second function is used to calculate cost of the route that has been taken:

```
# Function to calculate the cost of the route
def calculate_cost(route, graph):
    total_cost = 0
    num_cities = len(route)
    for i in range(num_cities):
        current_city = route[i]
        next_city = route[(i + 1) % num_cities]  # Wrap around to the start of the
route
        total_cost += graph[current_city][next_city]
    return total_cost
```

Figure 43: Brute-Force Calculate Cost

1. This function then calls the route and graph variable from the previous function.
2. Initialising the 'total_cost' variable and setting it to 0 as there is no cost at the start of the route because nowhere has been travelled yet.
3. The variable labelled num_cities is then assigned to calculate the number of cities in the route.
4. A for loop is then used to iterate for the number of cities in the problem.
5. Inside the loop, the current_city variable is set to the number city in the loop.
6. The next_city variable is then found by incrementing the current_city value by one until the value of i is greater than the value in the num_cities variable and then wraps around to the start of the route.
7. The Total cost is then calculated by adding the distance between the current_city and next_city variable to the total_cost total.
8. The total_cost variable is then returned for later use.

Brute-Force Function

The third function is the brute force algorithm to solve the problem.

```python
def brute_force(cities, graph):
    begin_time = time.time() #start the timer

    # Generate all permutations of the cities and initialise variables
    all_permutations = itertools.permutations(cities)
    min_cost = float('inf')
    optimal_route = None

    # Iterate over all permutations and calculate costs
    for perm in all_permutations:
        cost = calculate_cost(perm, graph)
        if cost < min_cost:
            min_cost = cost
            optimal_route = perm
    end_time = time.time() #end the timer
    return optimal_route, min_cost, begin_time, end_time
```

Figure 44: Brute-Force Function

1. The function starts by calling the cities and graph variables into the brute force function.
2. The second line of code is used to start the time function, this will be used to calculate how long algorithm execution takes.
3. Third, all the permutations of the cities are generated, a permutation is a route that can be taken. This is found using the itertools.permutations function to generate all possible routes. This is then stored in the variable all_permutations.
4. Then the min_cost variable is initialised and set to infinity this will be used to find the minimum cost.
5. And the optimal_route is initialised and set to empty.
6. A for loop is initialised for every route in the total number of routes.
7. The cost variable is then used to compare the cost of that route to the minimum cost, if the cost is lower than that value becomes the new minimum cost, and that number permutation becomes the optimal route. And stored in the optimal_route variable.
8. This for loop is then iterated by comparing the cost of each route until all routes have been checked.
9. The timer is then stopped to find the end time.
10. The optimal route, minimum cost, start time and end time are then returned.

Main Code

The main code to run the algorithm will look like:

```python
# Main code
if __name__ == "__main__":
    filename = "./TSP/tsplib-master/att8.tsp" #TSP file path
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance, begin_time, end_time = brute_force(cities, graph)
    print("Best Route: " , best_route)
    print("Number of Cities: ", len(best_route))
    print("Total Cost: ", best_distance)
    print("Execution Time: ", (round(end_time - begin_time)), "seconds")
    # Plot the best route
    plot_route(cities, best_route, problem)
```

Figure 45: Brute-Force Main Code

1. The main code starts by loading the TSPLIB file into the filename value, as the TSP problems are stored in the same directory it does not need a long file path.
2. Then the variables, cities, graph and problem are unpacked from the load_tsp_file function while filename is declared with the function so the correct file can be processed.
3. The best route, distance, and times are unpacked from the brute force function and cities and graph are also declared with the function.
4. The best route, number of cities, total cost and execution time are all printed, and the route visualisation is created. The route visualisation code will be accessible in the Data Visualisation sub chapter and can be seen in place in the full code in the appendix.

This completes the main function of the brute force code.

Held – Karp

The code implementation for the Held-Karp algorithm is based on an example also seen in (Liang, 2024), however, rewritten and adapted to fit the needs of this project.

Load TSP File

```
# Function to load a TSPLIB file, extracting both distance matrix and problem data
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    num_cities = len(cities)
    distance_matrix  =  [[problem.get_weight(cities[i],  cities[j])  for  j  in
range(num_cities)] for i in range(num_cities)]
    return problem, cities, distance_matrix
```

Figure 46: Held-Karp Load TSP File

This function is like the brute force function that loads the file:

1. Loading the TSP file into the problem variable, generating a list of nodes in the city's variable collecting the number of cities and assigning it to the num_cities variable.
2. However, the Held-Karp algorithm requires the cities to be placed in a distance matrix:
3. The weight is found between the start city, which is city I and the ending city which is city J.
4. This is repeated in two for loops one for the start city one for the end city which both iterate for the number of cities in the problem.
5. The variables problem, cities and distance_matrix are then returned.

Dynamic Processing Function

```
# Function to solve TSP using dynamic programming
def tsp_dynamic_programming(filename):
    begin_time = time.time() #start the timer
    problem, cities, distances = load_tsp_file(filename)
```

```python
#Step 1: Initialise the DP table
n = len(distances)
dp = [[math.inf] * n for _ in range(1 << n)]
parent = [[None] * n for _ in range(1 << n)]

dp[1][0] = 0

#Step 2: Fill the DP table
for mask in range(1 << n):
    for last in range(n):
        if not (mask & (1 << last)):
            continue
        for next in range(n):
            if mask & (1 << next):
                continue
            new_mask = mask | (1 << next)
            new_dist = dp[mask][last] + distances[last][next]
            if new_dist < dp[new_mask][next]:
                dp[new_mask][next] = new_dist
                parent[new_mask][next] = last

#Step 3: Find the minimum cost and reconstruct the path
min_cost = math.inf
end_city = None
full_mask = (1 << n) - 1
for last in range(1, n):
    cost = dp[full_mask][last] + distances[last][0]
    if cost < min_cost:
        min_cost = cost
        end_city = last

#Step 4: Reconstruct the optimal path
tour = []
mask = full_mask
last = end_city
while last is not None:
    tour.append(cities[last])
    new_last = parent[mask][last]
    mask ^= (1 << last)
    last = new_last

# Step 5: Reverse the path to get the correct order
tour = tour[::-1]
tour.append(cities[0])

# Step 6: Return the result and end timer
end_time = time.time() #end the timer
return problem, cities, tour, min_cost, begin_time, end_time
```

Figure 47: Held-Karp Dynamic Processing

Due to this being a larger function, it is split up into 6 different steps.

- Step 1: Initialise the dynamic processing table.
    - The number of cities is calculated and stored in the variable n
    - A 2D array is then created called dp; it is initialised as infinity as the cost was in the brute force example, this stores the shortest distance visited to each city as they are computed.
    - Another 2D array is created called parent, this keeps track of the route to help with reconstructing the optimal route.
    - The dp table is then initiated to set the initial cost from the first city to 0.
- Step 2: Fill the dynamic processing table.
    - Three loops are created and nested in each other.
    - The first is created, to loop through each possible combination of cities, with each potential route being represented in a bitmask, this is where each value is stored as bits and binary numbers instead of lists or strings.
    - The second loop will iterate over each last visited city possible for the route.
    - The third loop will check for the next city to visit and will check that it has not been visited already.
    - If it hasn't been visited the new route will be added to the bitmask.
    - This is then checked to see if the route is faster than the previous route.
    - If it is not then continued, if it is then the new route is recorded, and the parent array is updated as this is the new fastest route.
- Step 3: Find the minimum cost and reconstruct the path.
    - Initialise the minimum cost variable and set it to infinity and set the end city variable to empty.
    - The route bitmask is loaded.
    - A loop is created to iterate through all the last visited cities in the problem.
    - The cost of the route is calculated by adding together the distances of the cities as the algorithm moves through the route.
    - If the cost is lower than the previous route's cost, then set this new route's cost to the current cost as this is now the fastest route.
- Step 4: Reconstruct the Optimal Path
    - A new array is initialised called tour to store the full route.
    - A while loop is used to add each last visited city to the array repeating until there are no more cities yet to be added to the array and it is back at the starting city.
- Step 5: Reverse the path to get the correct order
    - Reverse order of path because the algorithm backtracks so route needs to be reversed to be correct.
    - Add the starting city to the start of the array to get the full route.
- Step 6: Return the result and end the timer
    - End the timer to get execution time.
    - Return the problem, cities, tour, cost and time variables.

This completes the steps needed to execute the problem using the Held – Karp method. Now the main part of the code needs to call the functions.

Main Code

```python
# Main execution
if __name__ == "__main__":
    filename = "./TSP/tsplib-master/att8.tsp"
    problem,    cities,    best_path,    min_cost,    begin_time,    end_time    =
tsp_dynamic_programming(filename)
    print("Best path:", best_path)
    print("Number of cities:", len(cities))
    print("Total cost:", min_cost)
    print("Execution time:", (end_time - begin_time), "seconds")
    plot_route(cities, best_path, problem)
```

Figure 48: Held - Karp Main Code

1. Just like with the earlier brute force example, the TSPLIB file is loaded from the directory and stored in the filename variable.
2. The variables used in the dynamic processing function are all called, and the function is executed.
3. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
4. The full code can be seen in the appendix chapter later in the document.

Branch and Bound

The code for the branch and bound algorithm is interpreted from an online example found in (GeeksForGeeks, 2024) however being interpreted to meet the requirements needed for the project.

Load a TSP File

```
# Load TSPLIB file using tsplib95
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    adj_matrix = np.array([[problem.get_weight(i, j) for j in cities] for i in
cities])
    return adj_matrix, cities, problem
```

Figure 49: Branch and Bound Load TSP File Function

1. This code starts by using the ‘tsplib95.load’ function to load the TSP problem from the filename and assign it to the variable problem, as done in the two earlier examples.
2. The next line extracts all the nodes in the problem and assigns them to the variable cities.
3. After this, the next two lines are used to gather the distances between the cities and then use those distances to create a graph between all the nodes.
4. An adjacency matrix is created to store the distances between the cities; this is different to a distance matrix as an adjacency matrix checks to see if there is a connection between nodes, not just measure the distances.
5. Returns the cities, graph, adjacency matrix and problem for later use.

First Minimum Function

```
def first_min(adj, i): #finds the first minimum edge distance from i to any other
node
```

```
    min_val = np.inf
    for k in range(len(adj)):
        if adj[i][k] < min_val and i != k:
            min_val = adj[i][k]
    return min_val
```

Figure 50: Branch and Bound Find First Minimum Function

1. Initialise the minimum value and set it to infinity.
2. Uses a loop to iterate through all the nodes in the problem.
3. If the distance between city one and city two is less than the minimum value, then set that distance to be the new minimum value.
4. Return the minimum value.

Second Minimum Value

```
def second_min(adj, i): #finds the second minimum edge distance from i to any other
node
    first, second = np.inf, np.inf
    for j in range(len(adj)):
        if i == j:
            continue
        if adj[i][j] <= first:
            second = first
            first = adj[i][j]
        elif adj[i][j] <= second:
            second = adj[i][j]
    return second
```

Figure 51: Branch and Bound Find Second Minimum Function

1. Initialise the first and second variables and set them to infinity.
2. Create a loop and loop through each node in the problem.
3. If I (city 1) am equal to j (city 2) then skip. If not, then if the distance between city 1 and city 2 is less than the first then assign the current first value to the second. And assign the new value to first.
4. Otherwise, if the weight is only less than or equal to the second, update the second.
5. Return the second variable.

Recursive Pruning Function

```
def  tsp_rec(adj,  current_bound,  current_weight,  level,  current_path,  visited,
final_res, final_path): #recursive function to solve by pruning branches
    N = len(adj)

    if level == N:
        if adj[current_path[level - 1]][current_path[0]] != 0:
            current_res    =    current_weight    +    adj[current_path[level    -
1]][current_path[0]]
            if current_res < final_res[0]:
                final_path[:N + 1] = current_path[:]
                final_path[N] = current_path[0]
                final_res[0] = current_res
        return
    for i in range(N):
        if adj[current_path[level-1]][i] != 0 and not visited[i]:
            temp = current_bound
            current_weight += adj[current_path[level - 1]][i]

            if level == 1:
                current_bound  -=  (first_min(adj,  current_path[level  -  1])  +
first_min(adj, i)) / 2
            else:
                current_bound  -=  (second_min(adj,  current_path[level  -  1])  +
first_min(adj, i)) / 2

            if current_bound + current_weight < final_res[0]:
                current_path[level] = i
                visited[i] = True

                tsp_rec(adj,   current_bound,   current_weight,   level   +   1,
current_path, visited, final_res, final_path)

            current_weight -= adj[current_path[level - 1]][i]
            current_bound = temp

            visited = [False] * N
            for j in range(level):
                if current_path[j] != -1:
                    visited[current_path[j]] = True
```

Figure 52: Branch and Bound Recursive Pruning Function

1. The number of cities in the problem are calculated and stored in the variable N.
2. Then check if all nodes have been visited by comparing the current recursion level with the total number of nodes.
3. If all nodes have been visited, it verifies there is an edge from the last node back to the starting node to complete the tour.
4. It then compares this cost with the most efficient route so far, if it is lower, then this becomes the new optimal route.
5. If all nodes are not visited the function will iterate through all the nodes looking for the next move to be made, for each potential move the adjacency matrix is checked to ensure the city has not been visited yet.
6. The current lower bound is stored in a temporary variable called the current bound to allow for backtracking later in the function.
7. The cost for the move is stored in the current weight variable, and this is used to create a cumulative tour cost.
8. The lower bound is adjusted based on the level of recursion, at level one half the sum of the first minimum distance is subtracted from the current city and potential city. At other levels, half the sum of the second minimum is subtracted is from the current city and potential city.
9. The function then checks if the sum of the new lower bound and current cumulative cost is less than the best-known tour cost, pruning non-promising branches.
10. If the branch is promising, the candidate city is marked as visited, added to the current path and the function repeats for the next level.

11. After exploring the function backtracks by reverting the cumulative cost and lower bound to their previous values and rebuilding the visited list based on the current path.

Branch and Bound Solver

```
# Main TSP solver using Branch and Bound
def solve_tsp_branch_bound(adj):
    begin_time = time.time() #start the timer
    N = len(adj)
    current_bound = 0
    current_path = [-1] * (N + 1)
    visited = [False] * N

    final_res = [np.inf]
    final_path = [-1] * (N + 1)

    for i in range(N):
        current_bound += (first_min(adj, i) + second_min(adj, i))

    current_bound = np.ceil(current_bound / 2)

    visited[0] = True
    current_path[0] = 0

    tsp_rec(adj,  current_bound,  0,  1,  current_path,  visited,  final_res,
final_path)
    end_time = time.time() #end the timer
    return final_res[0], final_path, begin_time, end_time
```

Figure 53: Branch and Bound Solver Function

1. The function starts by setting the length of the problem to the variable N.
2. Initialising the current bound variable as well and setting it to 0.
3. Creating the current path list and setting it to a size of N+1 will store the sequence of cities in the route.
4. Sets up another variable called visited which is a Boolean that will display true or false to determine if a city has been visited or not.
5. The final res variable is also initialised and set to infinity this will store the tour cost and be used to compare tour costs.
6. A loop is then used to calculate the lower bound by iterating over each city adding the sum of the first and second minimum distances for each city.
7. The current bound is then halved and ran through the ceiling function which rounds the bound up to the nearest integer.
8. The starting city is then changed to visited and is set to the start of the current path.
9. The tsp recursive function is called and initialised.
10. The most efficient tour cost and route is then returned.

Main Code

```
# Example usage
if __name__ == "__main__":
    filename = "./TSP/tsplib-master/att8.tsp"  # Replace with your TSP file path
    adj_matrix, cities, problem = load_tsp_file(filename)
    best_cost,          best_route,          begin_time,          end_time          =
solve_tsp_branch_bound(adj_matrix)
    print("Total Cost:", best_cost)
    print("Number of Cities:", len(best_route))
    print("Optimal Path:", [cities[i] for i in best_route])
    print("Execution Time:", end_time - begin_time)
    plot_route(problem, best_route)
```

Figure 54: Branch and Bound Main Code

1. Just like with the earlier examples, the TSPLIB file is loaded from the directory and stored in the filename variable.
2. The variables used in the dynamic processing function are all called, and the function is executed.
3. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
4. The full code can be seen in the appendix chapter later in the document.

Christofides

The code for the Christofides algorithm is taken from this online source; (Ghosh, 2025) however, it has been amended to fit the needs of this project.

Load TSP File

```
# Load TSPLIB file
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem, G
```

Figure 55: Christofides Load TSP File

1. Following the same pattern as the other algorithms this starts with importing the TSPLIB file and assigning variables for the file.

Calculate Cost

```
# Cost calculation
def calculate_cost(route, graph):
    total_cost = 0
    for i in range(len(route)):
        current = route[i]
        next_city = route[(i + 1) % len(route)]  # wrap around
        total_cost += graph[current][next_city]
    return total_cost
```

Figure 56: Christofides Cost Calculation Function

1. The total cost is initialised and set to 0.
2. A loop is created that iterates through every node in the loop.

3. The current node is assigned to the variable current and the next city is assigned to the next city variable.
4. The total cost is then added by using the graph function to gain the distance between the current city and the next city, this is then incremented for every city to accumulate a total cost.
5. The total cost is returned.

Christofides Function

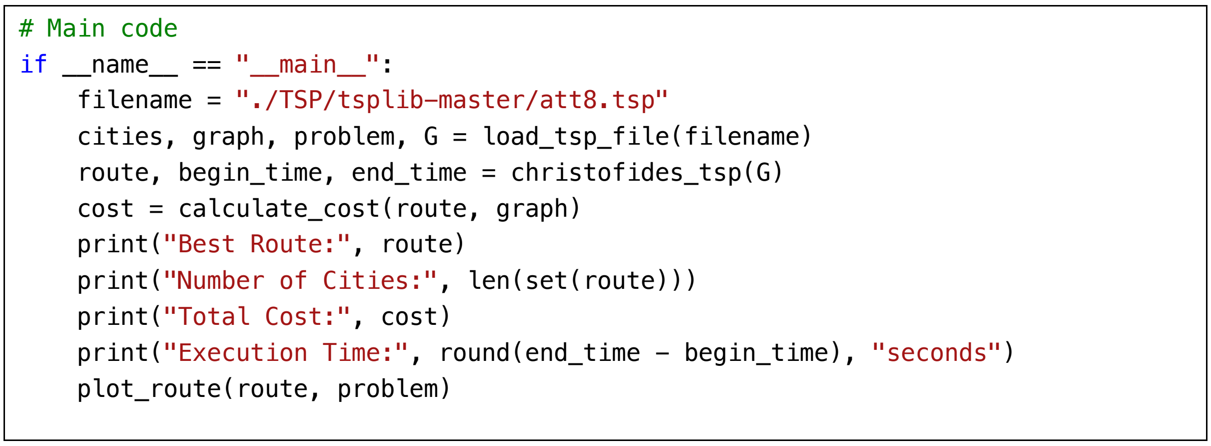
```python
# Christofides Algorithm
def christofides_tsp(G):
    begin_time = time.time()
    # Step 1: Minimum Spanning Tree
    mst = nx.minimum_spanning_tree(G)

    # Step 2: Find odd degree nodes
    odd_nodes = [v for v, d in mst.degree() if d % 2 == 1]

    # Step 3: Minimum Weight Perfect Matching among odd degree nodes
    subgraph = G.subgraph(odd_nodes)
    matching        =        nx.algorithms.matching.min_weight_matching(subgraph,
maxcardinality=True)

    # Step 4: Combine MST and Matching
    eulerian_graph = nx.MultiGraph(mst)
    eulerian_graph.add_edges_from(matching)

    # Step 5: Find Eulerian Circuit
    euler_circuit = list(nx.eulerian_circuit(eulerian_graph))

    # Step 6: Shortcutting to TSP Tour
    tour = []
    visited = set()
    for u, v in euler_circuit:
        if u not in visited:
            tour.append(u)
            visited.add(u)
    tour.append(tour[0])
    end_time = time.time()
    return tour, begin_time, end_time
```

Figure 57: Christofides Function

- Time is started to measure the execution time of the algorithm.
- Step 1: Minimum Spanning Tree
  - A minimum spanning tree is assigned to the variable mst.
  - It is created by using the NetworkX function which connects all the nodes with the shortest possible edge weight in a weighted connected graph. (NetworkX, 2024)
- Step 2: Find odd-degree nodes
  - Create an array called odd_nodes and then use a loop to iterate through all vertices, degrees are vertices that are connected to nodes.

    - Check if the node degrees are odd or even. If they are odd add the vertices to the odd_nodes to the array.
- Step 3: Minimum Weight Perfect Matching among odd degree nodes
    - Create a subgraph using the odd_nodes array.
    - Use the NetworkX minimum weight matching function to match vertices to get perfect weight matching. (NetworkX, 2024)
- Step 4: Combine minimum spanning tree and matching graph
    - Using the NetworkX multigraph function the mst variable is added to the eulerian_graph variable.
    - A Eulerian graph is a graph where every node has an even degree so that each node is visited exactly once. (Weisstein, 2025)
    - Then, edges are added to the eurlerian_graph by using the add edges function to add the matching tree as well.
- Step 5: Find Eulerian Circuit.
    - A Eulerian circuit is created by creating a list of all the nodes in the Eulerian graph.
- Step 6: Shortcutting to TSP Tour
    - The tour array is initialised and empty.
    - The visited list is initialised as well.
    - A loop is created to check whether a node has been visited yet.
    - Iterating through each node in the circuit, if it has not been visited then the tour is appended, and the node is added and that node is added to visited.
    - The tour is then appended to add the start city.
- Time is then stopped, and variables are returned.

Main Code



```python
# Main code
if __name__ == "__main__":
    filename = "./TSP/tsplib-master/att8.tsp"
    cities, graph, problem, G = load_tsp_file(filename)
    route, begin_time, end_time = christofides_tsp(G)
    cost = calculate_cost(route, graph)
    print("Best Route:", route)
    print("Number of Cities:", len(set(route)))
    print("Total Cost:", cost)
    print("Execution Time:", round(end_time - begin_time), "seconds")
    plot_route(route, problem)
```

Figure 58: Christofides Main Code

1. Just like with the earlier examples, the TSPLIB file is loaded from the directory and stored in the filename variable.
2. The variables used in the christofides function are all called, and the function is executed.
3. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
4. The full code can be seen in the appendix chapter later in the document.

Nearest Neighbour

The code for the nearest neighbour algorithm is interpreted from this online source, (Liang, 2024), however the code needs adapting to fit the needs of the project such as importing TSP files and creating visualisations.

Load TSP File

```
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem #returns variables for cities and graph
```

Figure 59: Nearest Neighbour Load TSP Problem

1. The tsp problem is loaded into a variable called problem, and the list of nodes are added to a list called cities.
2. Creates a graph for all the cities in the problem so the distances can properly be measured.
3. Returns the variables for later use.

Nearest Neighbour Function

```
def tsp_nearest_neighbour(graph, cities): #defines function for nearest neighbour
algorithm
    begin_time = time.time() #start time
    num_cities = len(cities) #gets the number of cities from the tsp file and then
stores it in variable
    start_city = cities[0]
    unvisited_cities = set(cities)
    unvisited_cities.remove(start_city)

    current_city = start_city #starts at start city
    path = [start_city]
    total_cost = 0

    while unvisited_cities: #while there are unvisited cities
        nearest_city      =      min(unvisited_cities,      key=lambda      city:
graph[current_city][city])
 #creates a temporary variable called lambda and sets it to the nearest city
        total_cost += graph[current_city][nearest_city]
        path.append(nearest_city)
        unvisited_cities.remove(nearest_city)
        current_city = nearest_city

    # Return to the starting city
    total_cost += graph[current_city][start_city]
    path.append(start_city)
    end_time = time.time() #end time
    return path, total_cost, begin_time, end_time
```

Figure 60: Nearest Neighbour Function

1. The number of cities in the problem is assigned to a variable called num_cities.
2. Then the start_city is set to the start of the cities list.
3. Another list is created called unvisited cities, and all the items in the cities list are added to it.
4. The start city is removed from this list as this city has been visited.
5. A loop is used to iterate through the unvisited cities, while there are unvisited cities, the algorithm will continue to iterate.
6. A temporary variable is created and set to the nearest city to the current city.
7. The total cost is incremented with the distance from the current city to the nearest city.
8. The nearest city is added to the route and removed from the unvisited cities list. And the nearest city is now set to the current city.
9. Once there are no more unvisited cities the algorithm returns to the start city and the distance between the current city and the start city is added to the total cost.
10. The time and variables are returned at the end of the function.

Main Code

```
if __name__ == "__main__": #main function
    filename = "./tsplib-master/d2103.tsp"
    cities, graph, problem = load_tsp_file(filename)

    best_path,  min_cost,  end_time,  begin_time  =  tsp_nearest_neighbour(graph,
cities)
    print("Best path: ", best_path)
    print("Minimum cost: ", min_cost)
    print("Execution Time: ", ((end_time - begin_time)))
    # Plot the best route
    plot_route(cities, best_path, problem)
```

Figure 61: Nearest Neighbour Main Code

1. Just like with the earlier examples, the TSPLIB file is loaded from the directory and stored in the filename variable.
2. The variables used in the nearest neighbour function are all called, and the function is executed.
3. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
4. The full code can be seen in the appendix chapter later in the document.

Hill Climbing

The hill climbing algorithm is interpreted from this online source, (GeeksForGeeks, 2024) However, adapted for the project to include features such as TSPLIB file importation and route visualisation.

Load TSP File

```
# Set seed for reproducibility
np.random.seed(42)
random.seed(42)
```

```
# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem
```

Figure 62: Hill Climbing TSP Load Function

1. The first line uses a random number generator to set the random start city for the problem.
2. The TSPLIB file is loaded into a variable named problem, and the nodes are added to a list called cities.
3. Creates a graph for all the cities in the problem so the distances can properly be measured.
4. Returns the variables for later use in the algorithm.

Calculate Cost Function

```
def calculate_cost(route, graph):
    total_cost = 0
    n = len(route)
    for i in range(n):
        current_city = route[i]
        next_city = route[(i + 1) % n]  # Wrap around to the start of the route
        total_cost += graph[current_city][next_city]
    return total_cost
```

Figure 63: Hill Climbing Calculate Cost Function

1. The total cost is initialised and set to 0.
2. The number of cities is added to a variable called n.
3. A loop is created and will iterate through every node in the problem, and the next city's distance and total cost are returned.
4. The total cost is returned for later use.

Create a Random Route Function

```
# Function to create a random initial route
def create_initial_route(cities):
    return random.sample(list(cities), len(cities))
```

Figure 64: Hill Climbing Create Random Route Function

- The random library is used to create a random order of the list of cities to create a random route.

Create Neighbouring Solutions Function

```
# Function to create neighbouring solutions
```

```
def get_neighbours(route):
    neighbours = []
    for i in range(len(route)):
        for j in range(i + 1, len(route)):
            neighbour = route.copy()
            neighbour[i], neighbour[j] = neighbour[j], neighbour[i]
            neighbors. Append(neighbour)
    return neighbors
```

Figure 65: Create Neighbouring Solutions Function

1. An array is created called neighbours and is set to empty.
2. A loop is used to iterate through every node in the route. Inside that loop is a nested loop that iterates through the different neighbouring cities.
3. The neighbours are reversed to prevent any duplication.
4. The pair of neighbours are added to the neighbour array.
5. The neighbour array is returned for later use.

Hill Climbing Function

```
#hill climbing function
def hill_climbing(cities, graph):
    begin_time = time.time()
    current_route = create_initial_route(cities)
    current_distance = calculate_cost(current_route, graph)

    while True:
        neighbors = get_neighbors(current_route)
        next_route = min(neighbors, key= lambda x: calculate_cost(x,graph))
        next_distance = calculate_cost(next_route, graph)

        if next_distance >= current_distance:
            break

        current_route, current_distance = next_route, next_distance
    end_time = time.time()
    return current_route, current_distance, begin_time, end_time
```

Figure 66: Hill Climbing Function

1. A random route is created and assigned to the current_route list.
2. And the current distance is calculated using the calculate cost function using the current route list and the graph variable.
3. A loop is used to compare sets of neighbours to fund the shortest route.
4. A temporary variable to find the next city is created and that is added to the next distance variable.
5. The loop will continue to iterate until the next distance is greater than the current distance variable.
6. The shortest route and shortest distance are set as the current route and current distance variable.
7. The current route and current distance and time values are returned.

Main Code

```
# Main Code
if __name__ == "__main__":
    filename = "./TSP/tsplib-master/rat99.tsp"
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance, begin_time, end_time = hill_climbing(cities, graph)
    print("Best Route: ", best_route)
    print("Total Cost: ", best_distance)
    print("Execution Time: ", ((end_time - begin_time)))
    plot_route(cities, best_route, problem)
```

Figure 67: Hill Climbing Main Code

1. Just like with the earlier examples, the TSPLIB file is loaded from the directory and stored in the filename variable.
2. The variables used in the hill climbing function are all called, and the function is executed.
3. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
4. The full code can be seen in the appendix chapter later in the document.

Simulated Annealing

Due to the similarities of the algorithm, the code is inspired by the same source as the hill climbing algorithm, (GeeksForGeeks, 2024) being adapted as well for the needs of this project. This code follows the base structure of the hill climbing code with the simulated annealing function added.

Hill Climbing Functions

```
# Set seed for reproducibility
np.random.seed(42)
random.seed(42)


# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem

def calculate_cost(route, graph):
    total_cost = 0
    n = len(route)
    for i in range(n):
        current_city = route[i]
        next_city = route[(i + 1) % n]  # Wrap around to the start of the route
        total_cost += graph[current_city][next_city]
    return total_cost
```

```
# Function to create a random initial route
def create_initial_route(cities):
    return random.sample(list(cities), len(cities))

# Function to create neighbouring solutions
def get_neighbors(route):
    neighbors = []
    for i in range(len(route)):
        for j in range(i + 1, len(route)):
            neighbour = route.copy()
            neighbour[i], neighbour[j] = neighbour[j], neighbour[i]
            neighbours. append(neighbour)
    return neighbors
```

Figure 68: Hill Climbing Function

1. This follows the same structure as the hill climbing algorithms.
2. A random route is created using the random library to generate a random order of routes.
3. The TSPLIB files are loaded into the code and the cost function is defined.
4. Additionally, the neighbouring solutions function is also defined as in the hill climbing software.

Simulated Annealing Function

```
def    simulated_annealing(cities,    graph,    initial_temp,    cooling_rate,
max_iterations):
    current_route = create_initial_route(cities)
    current_distance = calculate_cost(current_route, graph)
    best_route = current_route.copy()
    best_distance = current_distance
    temperature = initial_temp

    for _ in range(max_iterations):
        neighbors = get_neighbors(current_route)
        next_route = random.choice(neighbours)
        next_distance = calculate_cost(next_route, graph)

        if    next_distance    <    current_distance    or    random.random()    <
np.exp((current_distance - next_distance) / temperature):
            current_route, current_distance = next_route, next_distance

            if current_distance < best_distance:
                best_route, best_distance = current_route, current_distance

        temperature *= cooling_rate
    end_time = time.time()
    return best_route, best_distance, begin_time, end_time
```

Figure 69: Simulated Annealing Function

1. The current route and current distance is created and is immediately set to the best route and best distance.
2. Temperature is initialised.
3. A loop is initialised that loops for the number of iterations that is defined by the user.
4. The neighbouring pairs are compared by comparing distances between nodes until the closest neighbours are found just as in the hill climbing algorithm.
5. The only difference is a worse solution can be accepted from the probability of the temperature. So, a higher temperature means a higher acceptance of worse solutions.
6. The temperature decreases from the cooling rate which can also be set by the user.
7. Values are returned at the end of the function for use later.

Main Code

```
# Parameters for Simulated Annealing
initial_temp = 200
cooling_rate = 0.99
max_iterations = 1000

# Set name of file
if __name__ == "__main__":
    filename = "/./TSP/tsplib-master/att8.tsp"  # Replace with your TSP file path
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance = simulated_annealing(cities, graph, initial_temp,
cooling_rate, max_iterations)
    print("Best Route: " , best_route)
    print("Total Cost: ", best_distance)
    print("Execution Time: ", ((end_time - begin_time)))
    # Plot the best route
    plot_route(cities, best_route, problem)
```

Figure 70: Simulated Annealing Main Code

1. The initial temp can be set between 1 and 100.
2. Cooling rate can be set between 0 and 1.
3. And max iterations can be set to any value.
4. Just like with the earlier examples, the TSPLIB file is loaded from the directory and stored in the filename variable.
5. The variables used in the simulated annealing function are all called, and the function is executed.
6. Best path, number of cities, total cost and execution time are all printed, and the route visualisation is shown.
7. The full code can be seen in the appendix chapter later in the document.
8. For the experiments later in the report the algorithm will be set to find the fastest solution possible regardless of optimality.

User Interface

The user interface is limited as there is no graphical user interface only a command line output after the problem has been solved. This can be seen in the code for the algorithms in the main code sections.

```
    print("Best Route: " , best_route)
    print("Total Cost: ", best_distance)
    print("Execution Time: ", ((end_time - begin_time)))
    plot_route(cities, best_route, problem)
```

Figure 71: Output Information Code

This will print all the data collected by the code into clear lines like this:

```
Best Route: [1, 7, 6, 4, 2, 5, 3, 8, 1]
Number of Cities: 8
Total Cost: 6276
Execution Time: 0.013 seconds
```

Figure 72: Output Example

This will make up the main part of the algorithm output, and there will be a graphic visualisation to go with it.

Data Visualisation

The function to plot the graph is the same for every algorithm so can be seen in place in the code in the appendix chapter. The code to plot the route on a graph will look like:

```
# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords

    # Plot the route
    route_coords = [city_coords[city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])

    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("TSP Route")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()
```

Figure 73: Data Visualisation Code

1. The coordinates from the problem are found and stored in the variable city_coords.
2. The route is then plotted using loops to iterate through each coordinate and plot it on the graph.
3. The coordinates are added for the starting city to complete the route.
4. The visualisation is using red markers and a red line and is labelled as route in the key for the graph.
5. The title and label axis are added as well.
6. The grid is set to false to remove the grid to make the route easier to view.
7. The route is then plotted and displayed.

Functionality Testing

The first test that will be ran is to test the functionality of the algorithms to ensure they run correctly and output the correct data:

Test	TSP Problem	Purpose	Outcome
Brute Force Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Held Karp Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Branch and Bound Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Christofides Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Nearest Neighbour Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Hill Climbing Functionality	Att8.TSP	Ensure algorithm runs without error	Pass except time error
Simulated Annealing Functionality	Att8.TSP	Ensure algorithm runs without error	Pass
Visualisations are different for different algorithms.	Ulysses16.TSP Using Nearest Neighbour and Christofides algorithms	Ensure different visualisations are being plotted not just the same route regardless of algorithm	Pass
Passing an incorrect TSP file name does not return a result	Att9.TSP	Ensures that the loading tsp file function loads the file correctly	Pass
Simulated Annealing cooling rate and temperature changes return different results	Ulysses16.TSP	Ensures that the temperature and cooling rate affect results so algorithm works correctly	Pass

Table 2: Algorithm Functionality Test

The integration tests were mainly successful with only a couple of bugs appearing that can be fixed. All algorithms passed a basic functionality test on the small dataset however were not able to be tested on large data sets because of algorithm complexity on the exact algorithms. To test the brute force algorithms on a large data set it could take days which is not sustainable to fix bugs when they appear so the assumption is made that if it can run on a smaller data set with 8 nodes it will work on larger data sets.

Integration Testing Fixes

The fixes for the integration testing may only be small but ensure the code is working properly and as intended.

Time Fix

The time value in the nearest neighbour algorithm was returning a negative value despite the formula still being correct for calculating time. The original code looked like this:

```
    best_path,  min_cost,  end_time,  begin_time  =  tsp_nearest_neighbour(graph,
cities)
    print("Best path: ", best_path)
    print("Minimum cost: ", min_cost)
    print("Execution Time: ", ((end_time - begin_time)))
```

Figure 74: Nearest Neighbour Time Error

And was returning this:

```
Best path:  [1, 8, 3, 5, 2, 4, 7, 6, 1]
Cities Visited:  9
Minimum cost:  6396
Execution Time:  -8.7738037109375e-05
```

Figure 75: Nearest Neighbour Incorrect Output

This was a simple fix because it is to do with how the time variables are returned and unpacked, the variables are being returned by the function in a different order they are being unpacked so to fix this the variable order needs to be swapped in the main code. This now should look like:

```
    best_path,  min_cost,  begin_time,  end_time   =  tsp_nearest_neighbour(graph,
cities)
    print("Best path: ", best_path)
    print("Minimum cost: ", min_cost)
    print("Execution Time: ", ((end_time - begin_time)))
```

Figure 76: Nearest Neighbour Time Fix

So now the amended output should look like:

```
Best path:  [1, 8, 3, 5, 2, 4, 7, 6, 1]
Cities Visited:  9
Minimum cost:  6396
Execution Time:  8.702278137207031e-05
```

Figure 77: Nearest Neighbour Correct Output

Although not a major issue, this can still provide problems if it is not fixed and working as intended.

SOLID Principles

The adhesion to the SOLID principles can be seen here:

Single Responsibility Principle

The single responsibility principle has been adhered to by keeping each algorithm in their own respective file keep as well as using distinctive function to load tsp files and produce visualisations.

Open/Closed Principle

The modular set up means that new algorithms and TSP problems can be added without affecting any other algorithms because they are all kept in their own files separate from all the algorithms.

Liskov Substitution Principle

Python doesn’t force strict class inheritance the structured outputs that are used across all the algorithms, help support the LSP. This consistency allows for comparison across the algorithms as will be seen in the results chapter.

Interface Segregation Principle

Code is separated for visualisation, problem loading, path generation, execution timing, and algorithm logic which means that these parts can be removed when irrelevant and nit bundled into the one algorithm.

Dependency Inversion Principle

The main code blocks that run the algorithms, depend on utility functions to load data and import tsp files not on hard coded details.

Challenges and Solutions

A challenge faced during this project is with making visualisations of the routes not all TSPLIB problems have coordinates as some problems are explicit this means that visualisations cannot be produced for these problems so will be harder to compare and test these problems.

Another challenge faced is testing exact algorithms on larger tsp problems, for example the brute force algorithm searches every route possible so it is not possible to test this algorithm on large scale tsp problems as it cannot be completed within a reasonable time. To try and prevent this the algorithms have been tested on various smaller tsp problems to ensure a consistent performance across all algorithms.

Another problem faced is the performance of the machine that this experiment is being performed on; an example is in the nearest neighbour algorithm for large scale problems the graph creates a data structure of the squared value of that number of problems for the problem pla7397, which has 7397 nodes a data structure of 54 million routes before the nearest neighbour algorithm runs. This means that although the algorithm should take seconds to run this can turn to over twenty minutes to build which is classified as an unreasonable time to execute which will spoil the maximum range of the algorithm. This can be solved by measuring total execution time of the program and not just the execution time of the algorithm itself. However, means that more mid-sized algorithms will be needed to test.

6. Testing and Results from Artefact

To test the strengths and weaknesses of the algorithms, they will have to be run through several different TSP problems to test scaling, efficiency, performance and functionality with different size problems.

Test Datasets

The TSP data sets that will be used for the testing range in size and symmetry these different problems are:

D4

This is a 4-node problem modelled off 4 locations in Germany. This is a very small symmetric using an edge weight type of EUC_2D which is a two-dimensional Euclidean distance between the nodes. This has been selected for verifying the correctness of the algorithms as well as being favoured towards the brute force and held karp algorithms due to their smaller size.

Att8

The Att8 data set is a smaller and reduced set of data from the att48 data set, using an edge weight type of ATT this problem represents eight of the mainland states in the USA. A test of this size is great at testing the middle bound for exact algorithms and the lower bounds of approximation algorithms.

Burma10

The Burmw10 algorithm is a shorter version of the Burma14 algorithm, this uses geographical coordinates to plot the nodes and is also a great size problem for a direct comparison of exact and approximation algorithms.

Burma14

Burma14 is a 14-node problem small sized problem, that is based off 14 states in Myanmar. This is a small real-world inspired problem that uses an edge weight type of GEO, which uses geographical coordinates to separate the nodes. This sized problem will be good for testing the accuracy and the approximation rates of the approximate algorithms as they can be compared to the exact algorithms on a problem this size.

Ulysses16

Ulysses22 is a 16-node small to mid-sized problem that is inspired off the path of Ulysses from the Odyssey. This also has an edge weight type of GEO using geographical coordinates, this sized problem is also great for comparing exact and approximate algorithms as it is just about short enough to run exact algorithms but large enough to see performance differences in the approximate algorithms.

Ulysses22

Ulysses22 is a 22-node mid-sized problem that is inspired off the path of Ulysses from the Odyssey. This also has an edge weight type of GEO using geographical coordinates, this sized problem is also great for comparing exact and approximate algorithms as it is just about short enough to run exact algorithms but large enough to see performance differences in the approximate algorithms.

Att48

Att48 is another mid-size problem modelled off the 48 capital cities of mainland America, this has an edge weight type of ATT, which is pseudo-Euclidean and is a good size to compare approximation algorithm differences as well as differences in route visualisation to allow for better comparisons.

Eil51

Eil51 is a mid-sized problem that has an edge weight type of EUC_2D featuring 51 artificially generated coordinates which will be good for testing both exact and approximation algorithms.

Berlin52

Berlin52 is a 52-node problem that is modelled off 52 locations in Berlin, Germany and is also a good mid-sized problem that can be used on both sets of algorithms within a reasonable time. Having an edge weight type of EUC_2D meaning visualisations can be made.

Brazil58

This is an explicit matrix data type that is modelled off 58 cities in Brazil, because this does not have explicit coordinates no visualisation will be produced but is still important to test to see how different algorithms handle different TSP problems.

Pr76

A randomly generated 76-node problem that is used for testing algorithms, another EUC_2D problem that will be useful to use on approximation algorithms and to get a route visualisation.

Ch150

Another artificially generated problem that has 150 nodes so will be great to see the different approximation rates of the algorithms, which may be too large to run exact algorithms within a reasonable time.

Gr202

Gr202 is a mid to large-size symmetric problem that is modelled off 202 different American cities with an edge weight type of EUC_2D which will be excellent at testing the differences in approximation algorithms as well as great for comparing the routes taken and shown in the visualisation.

Ali535

Ali535 is a large-scale geographical data set that is based off 535 airports around the world and will be great for testing performance of approximation algorithms and showing how exact algorithms cannot compete when it comes to problems of this size.

Nrw1379

This is a much larger sized problem used to test the power of the approximation algorithms to view the differences on a much larger scale will also provide better visualisations because of the different paths that will be taken. Using EUC_2D on a 1379 node problem.

U2152

A 2152 node problem that is used for testing the larger bounds of approximation algorithms to see what the largest problems that can be solved are.

Rl5934

A 5934-node problem to test the upper bounds of the approximation algorithms to gain insight into how different the algorithms are.

Test Metrics and Evaluation Criteria

The values that will be tested are the total cost of the journey, which is how far the salesman would have to travel to complete the route, the number of cities visited to ensure that the route is running correctly or to see if there have been any duplicate cities, the path taken to see what different paths the algorithms are taking and finally a visualisation of the route taken to graphically view the differences in routes.

The testing of the algorithms will be conducted in one sitting using the same hardware and development environment, this is to ensure all tests are ran using the same software with the same memory allocation processing speed and same efficiency. This will ensure that the results are fair across every TSP problem and produce consistent results. Each algorithm will be tested at once meaning brute force will be tested against multiple problems then held karp moving on until every algorithm has been tested against every problem.

Each test will be conducted once this is because of the size of the TSP problems and the time it will take to test multiple times meaning it will be hard to do in one sitting and then could potentially change the consistency of the results if done over multiple days.

Results Tables and Visualisations

This section will cover the results recorded from the algorithm tests on the TSP problems. The best performing algorithm for cost will be highlighted in green and the worst performing highlighted in red.

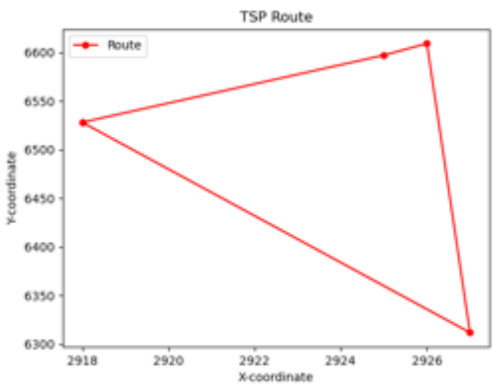
D4 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	594	1
Held-Karp	594	1
Branch and Bound	594	1
Nearest Neighbour	594	1
Hill Climbing	594	1
Christofides	594	1
Simulated Annealing	594	1

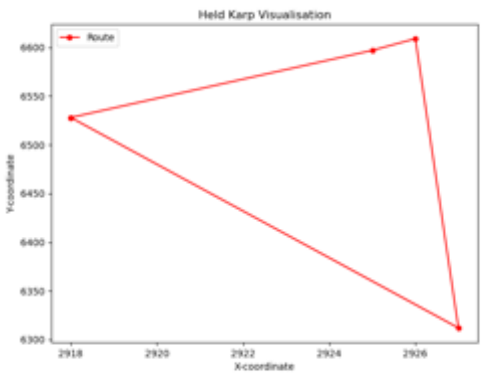
Table 3: D4 Results Table

The results recorded are all very even with only slight variations in path taken the cost and time taken remains the same across all algorithms. All algorithms performed well so highlighting not necessary.

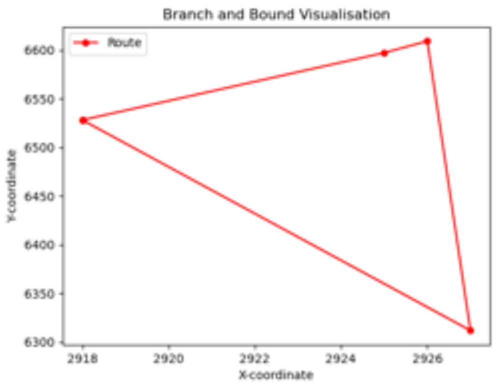
The visualisations for these routes look like:



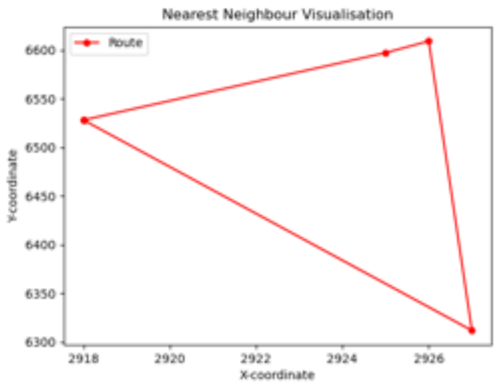
Brute Force Visualisation



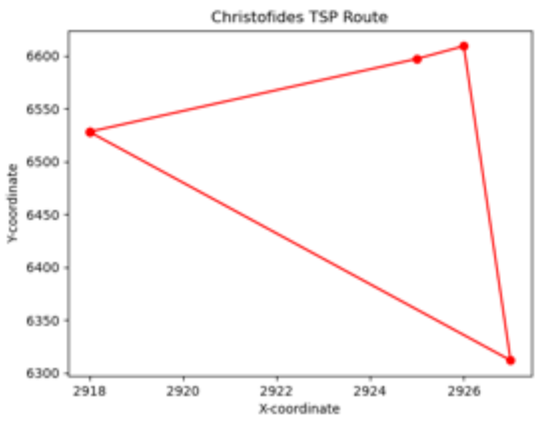
Held Karp Visualisation



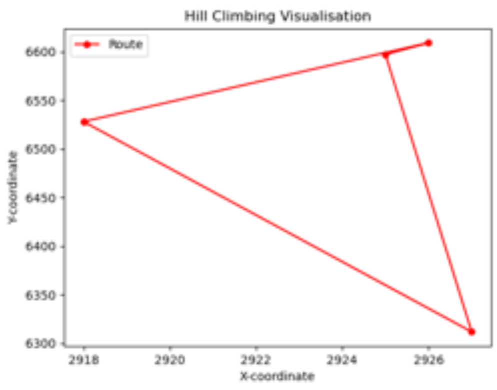
Branch and Bound Visualisation



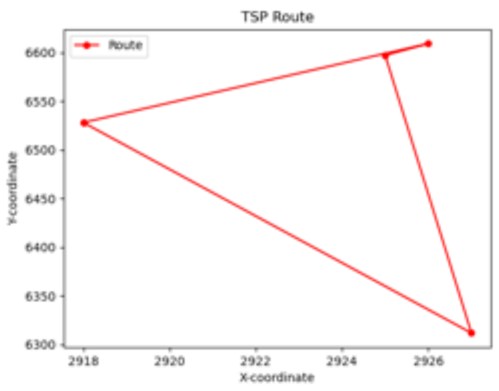
Nearest Neighbour Visualisation



Christofides Visualisation



Hill Climbing Visualisation



Simulated Annealing Visualisation

Table 4: D4 Visualisation Table

The visualisations show that every algorithm follows the same path except the hill climbing and simulated annealing algorithms which have a slight deviation in path which is to be expected as they are the same style of algorithm.

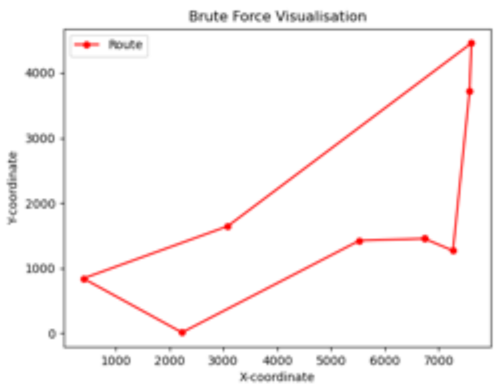
Att8 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	5919	1
Held-Karp	5919	1
Branch and Bound	5919	1
Nearest Neighbour	6396	1
Hill Climbing	6153	1
Christofides	6207	1
Simulated Annealing	5919	1

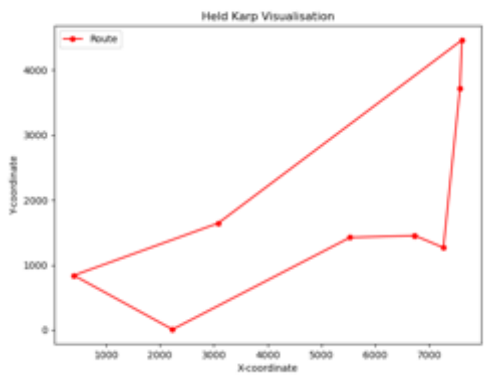
Table 5: Att8 Results Table

This results table shows that the simulated annealing algorithm found the most optimal route in the shortest amount of time with the nearest neighbour algorithm being the least efficient algorithm with the highest distance cost.

The visualisations for these routes look like:



Brute Force Visualisation



Held Karp Visualisation

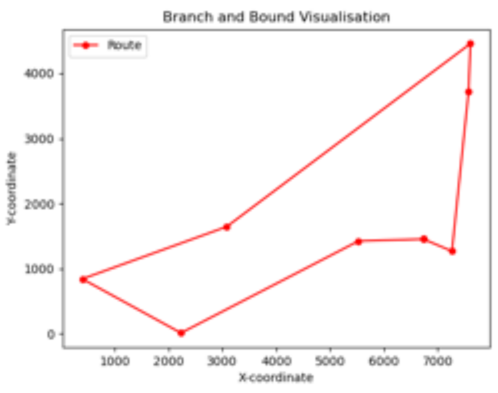
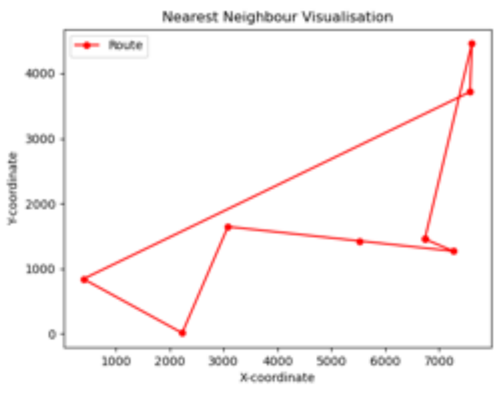
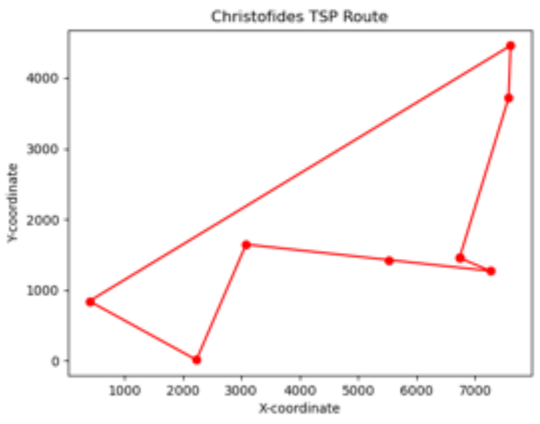
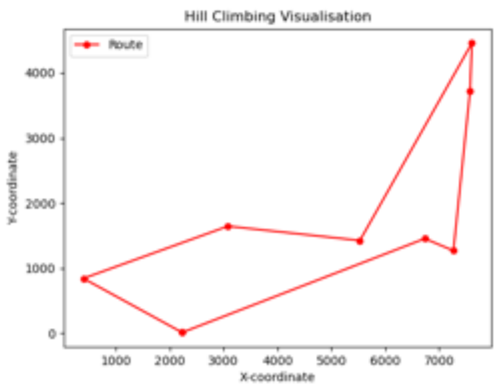
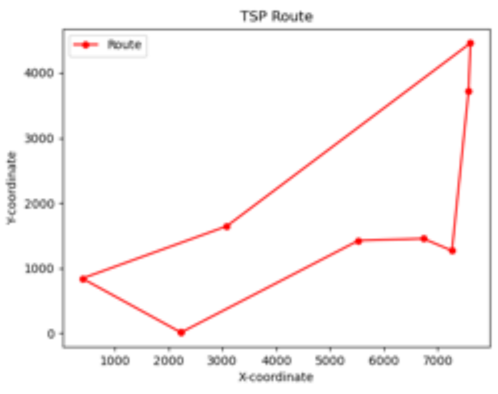
 *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 6: Att8 Visualisations

This table shows visualisations of the different routes taken by the algorithms, all three exact algorithms taking the same route which is the most optimal route as well as simulated annealing which can be seen as being the most efficient route in the results table. The other three visualisations for the approximate algorithms all take different routes with the nearest neighbour having the most inefficient route which can be seen in the visualisation because it has the longest lines which means it's travelling the longest distance overall.

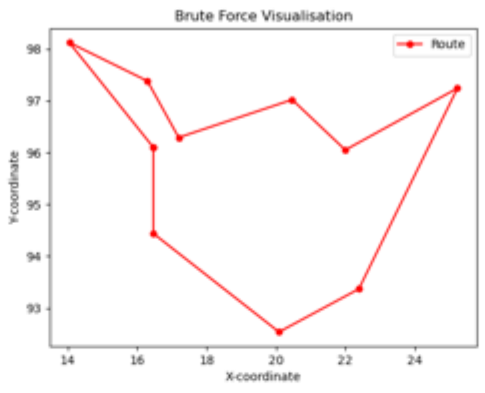
Burma10 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	3114	34
Held-Karp	3114	1
Branch and Bound	3114	1
Nearest Neighbour	3603	1
Hill Climbing	3114	1
Christofides	3275	1
Simulated Annealing	3154	1

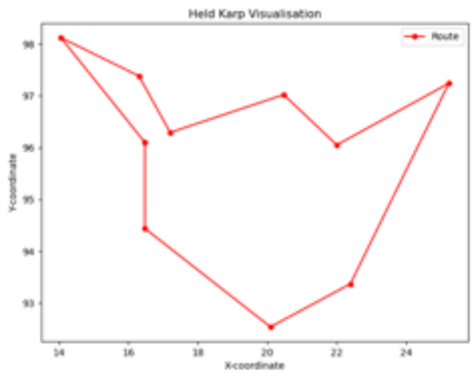
Table 7: Burma10 Results Table

The minimum cost for this route is 3114, as shown by exact algorithms only the held-karp and branch and bound algorithm completing this much faster than the brute force algorithm. The hill climbing found the most efficient route with the simulated annealing algorithm finding a near optimal route. All approximation algorithms completing this problem in under 1 second.

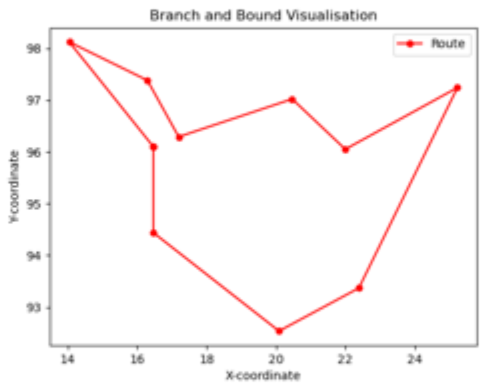
The visualisations for these routes look like:



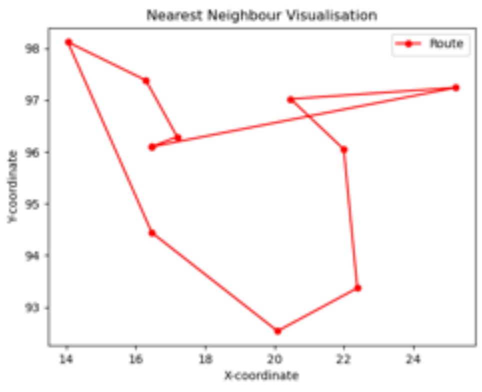
Brute Force Visualisation



Held Karp Visualisation



Branch and Bound Visualisation



Nearest Neighbour Visualisation

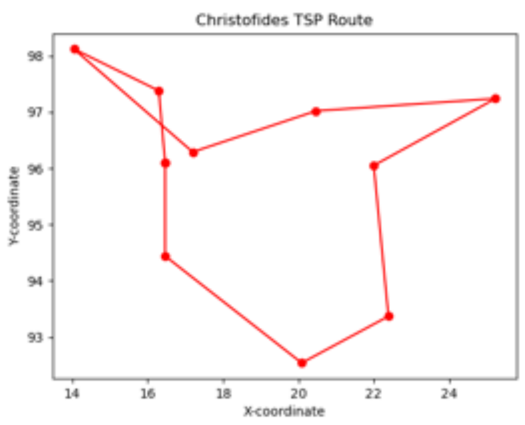
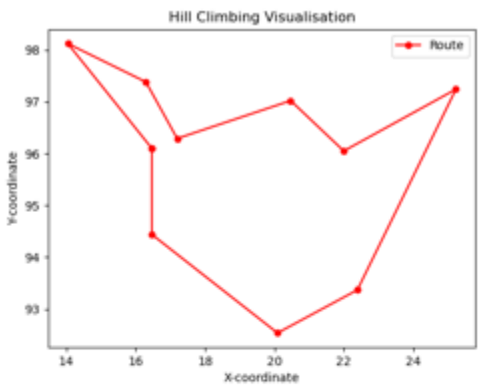
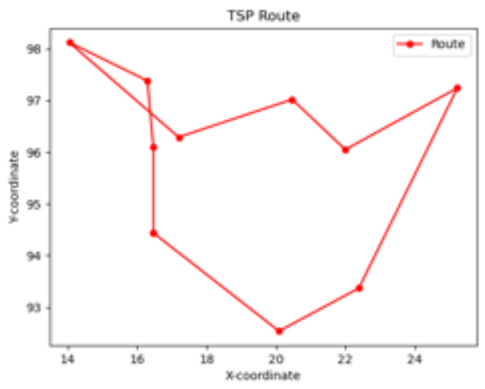
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 8: Burma10 Visualisations

This table of visualisations show the most efficient path taken in the exact algorithms and the hill climbing algorithm and shows the slight variance in the hill climbing and simulated annealing that causes the slight difference in cost. The nearest neighbour algorithm producing the least efficient route which can be seen by having the largest lines meaning there's more distance travelled.

Burma14 Test

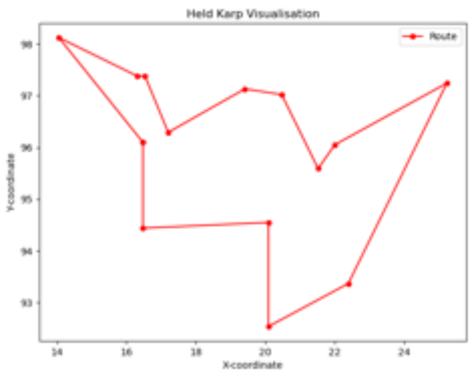
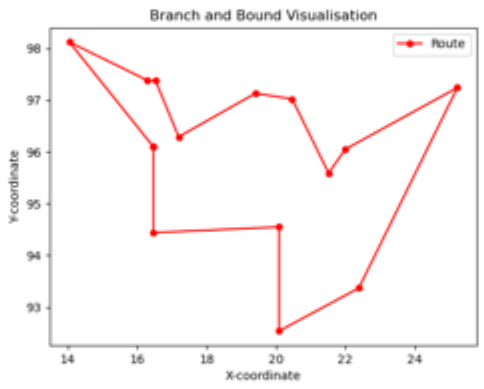
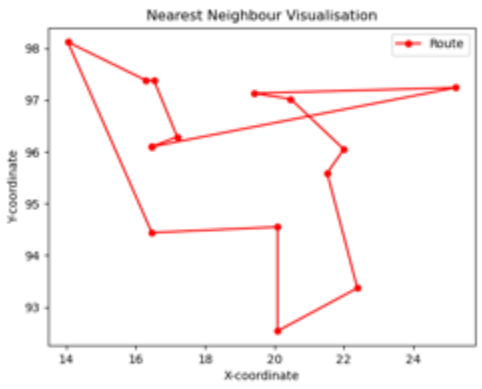
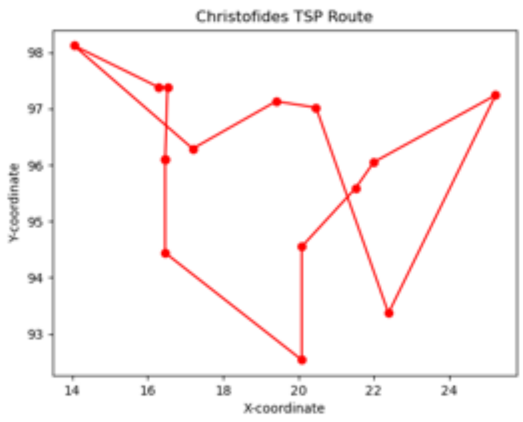
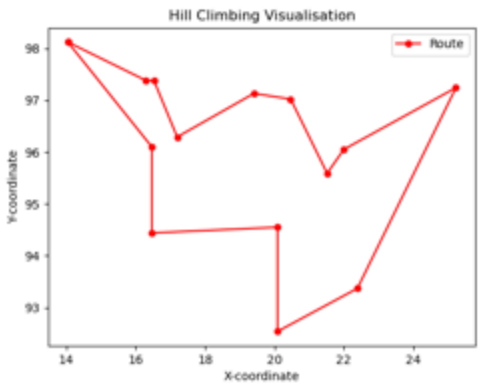
Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	3323	2
Branch and Bound	3323	53
Nearest Neighbour	4048	1
Hill Climbing	3323	1
Christofides	3607	1
Simulated Annealing	4204	11

Table 9: Burma14 Results Table

The size of this problem meant that the brute force algorithm could not complete the problem in the reasonable time specified which was ten minutes. The other two exact algorithms completed the problem with a minimum cost of 3323, however there was a 51 second difference between the two with the held karp algorithm completing the problem in only two seconds.

The hill climbing algorithm also completed the algorithm to optimality as well and completed it in the fastest time. The simulated annealing algorithm was the most inefficient having the largest cost out of any of the algorithms. However, all approximation algorithms completed the problem in under a second.

The visualisations for these routes look like:

No visualisation as problem is too large for algorithm *Brute Force Visualisation*	 *Held Karp Visualisation*
 *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*

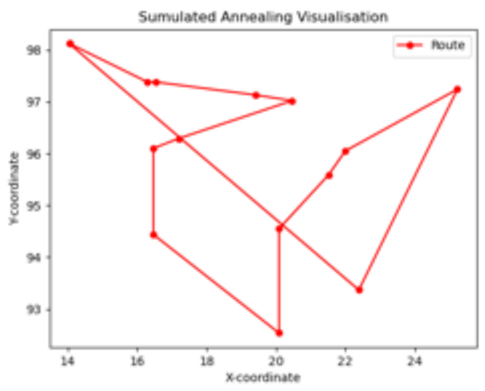
 *Simulated Annealing Visualisation*	

Table 10: Burma14 Visualisations

This table shows the visualisations of the routes taken by the different algorithms; the brute force algorithm could not complete the route so there was no route to visualise. The hill climbing algorithm took the same route as the exact algorithms to make the most efficient route. With the Christofides algorithm having the second most efficient route the slight difference in route can be seen when comparing that to the exact algorithms. The simulated annealing algorithm had the largest route which can be seen on the visualisation with the largest lines meaning the most distance is travelled.

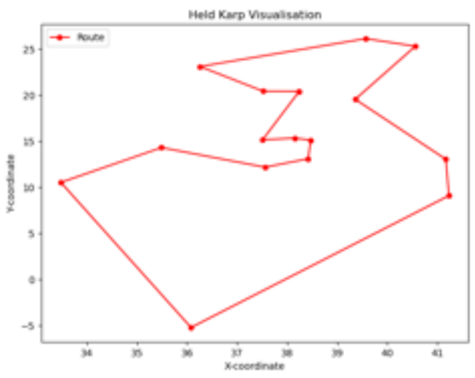
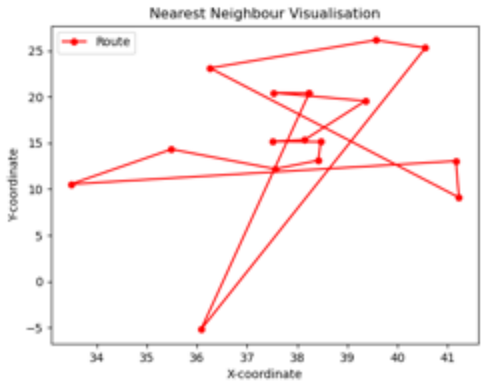
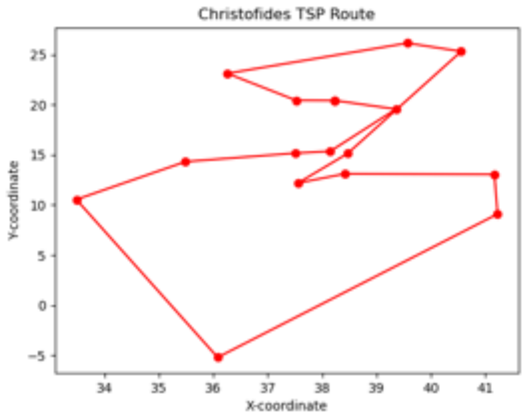
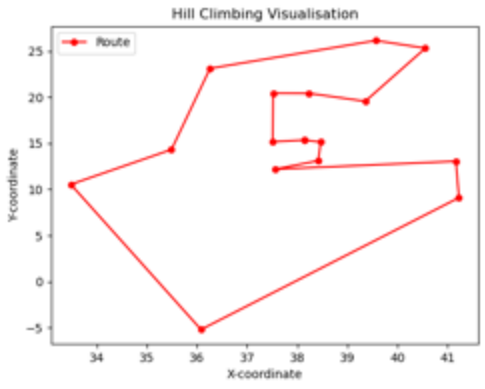
Ulysses16 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	6859	11
Branch and Bound	N/A	N/A
Nearest Neighbour	9988	1
Hill Climbing	6982	1
Christofides	6984	1
Simulated Annealing	8665	1

Table 11: Ulysses16 Results Table

The brute force and branch and bound both could not complete the algorithms in reasonable times with worst case time predictions of 12000 seconds, so the only exact algorithm that could complete the route is the held-karp algorithm with a minimum cost of 6859. No approximation algorithm found the most efficient route the two closest were the hill climbing and Christofides algorithms which were within 200 of the minimum cost. The nearest neighbour algorithm once again had the least efficient route, and all approximation algorithms completed the problem in less than a second.

The visualisations for these routes look like:

No route visualisation as the route could not be completed. *Brute Force Visualisation*	 *Held Karp Visualisation*
No route visualisation as the route could not be completed. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*

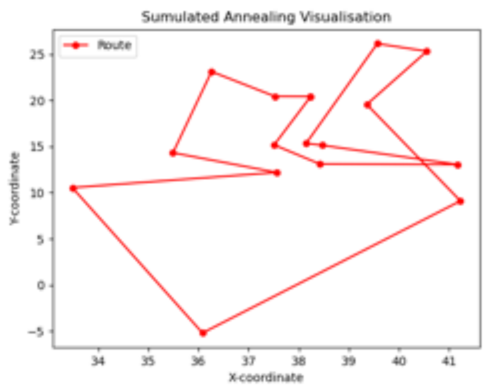
 *Simulated Annealing Visualisation*	

Table 12: Ulysses16 Results Table

The results table shows that hill climbing has the closest path taken to the exact algorithm and why it is the most efficient of the approximation algorithms which Christofides following a similar path but not the same and being slightly less efficient. Nearest neighbour and simulated annealing algorithms not following any similar path and having long connecting lines show why the cost is so large for these algorithms.

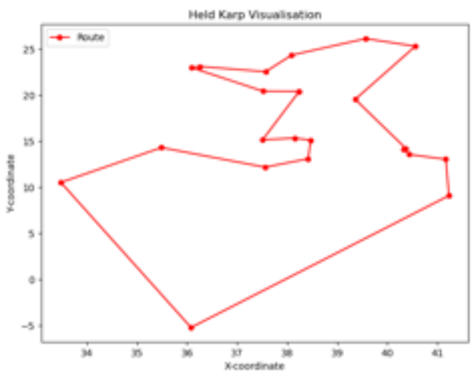
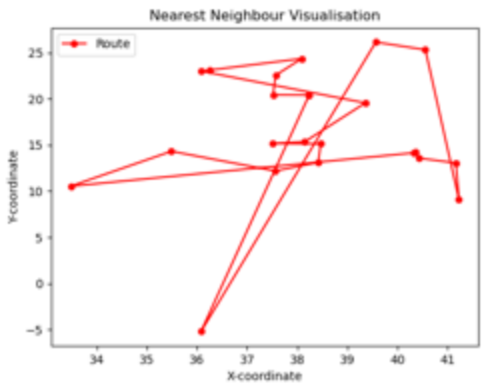
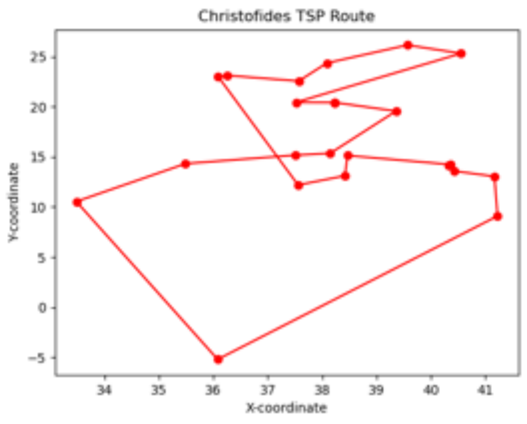
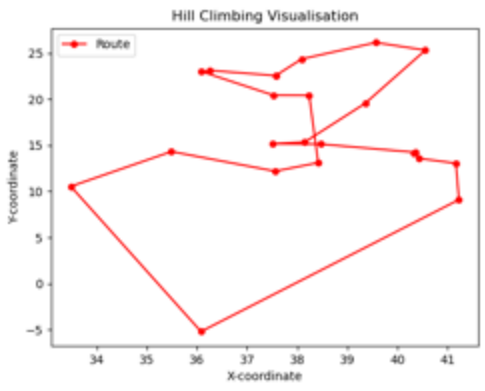
Ulysses22 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	7013	596
Branch and Bound	N/A	N/A
Nearest Neighbour	10586	1
Hill Climbing	7187	1
Christofides	7412	1
Simulated Annealing	9033	1

Table 13: Ulysses22 Results Table

The brute force and branch and bound algorithms are still unable to complete the problem however this is the maximum the held karp algorithm can execute as the algorithm takes ten minutes to solve so any problem larger would be greater than a reasonable time, the optimal solution had a cost of 7013. Every approximation algorithm completes the problem in under 1 second however the minimum costs have a range of 3400 with hill climbing being the most efficient and the nearest neighbour being the least efficient.

The visualisations for these routes look like:

No route visualisation as the route could not be completed. *Brute Force Visualisation*	 *Held Karp Visualisation*
No route visualisation as the route could not be completed. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*

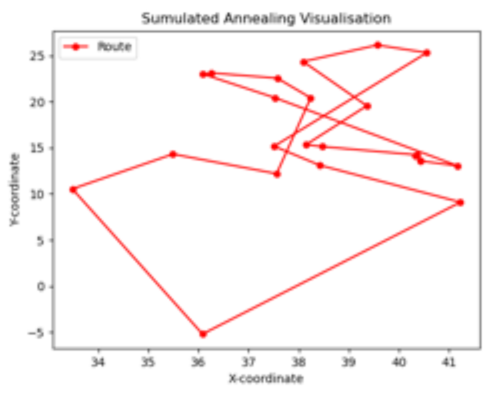
 Simulated Annealing Visualisation	

Table 14: Ulysses22 Route Visualisations

This table shows the different paths taken by the algorithms, with the hill climbing algorithm having the most similar pattern to the held - karp algorithm showing how it is nearly optimal. The nearest neighbour algorithm is clearly the least efficient route as there are long path line between the nodes meaning that the salesman would be travelling the longest distances.

Att48 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	12861	1
Hill Climbing	17159	3
Christofides	12613	1
Simulated Annealing	28402	1

Table 15: Att48 Results Table

The att48 problem is too large for exact algorithms to be ran on, so to find the most optimal solution, the solutions will need to be gained from the TSPLIB mater file, the optimal cost is 10628. The most efficient route found by the approximation algorithms, is from the christofides and nearest neighbour algorithms, both completing these problems in under 1 second. The hill climbing algorithm took the longest time out of the approximation algorithms taking 3 seconds to complete the problem however having a much higher cost of 17159. The simulated annealing algorithm had the least efficient route with nearly three times the cost of the most optimal route however completing the problem in under a second.

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. Brute Force Visualisation	No route visualisation as the route could not be completed in a reasonable time Held Karp Visualisation

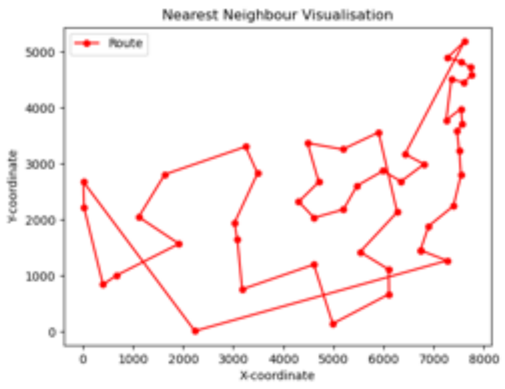
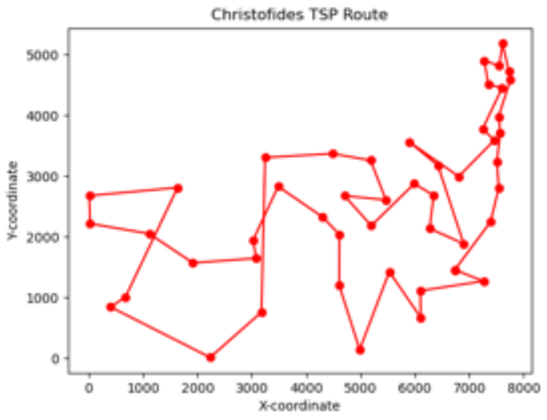
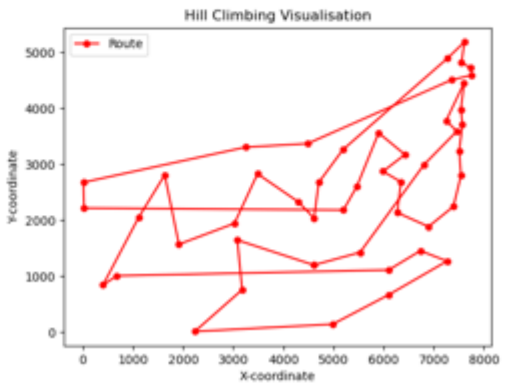
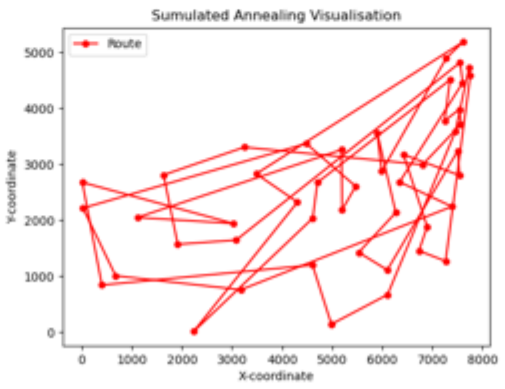
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 16: Att48 Results Table

This results table shows that nearest neighbour and christofides follow a path that outlines the states in the USA which is what the problem is modelled after. The hill climbing and simulated annealing algorithms do not follow a set path or show a pattern which can show why the costs are so high because of the long distances travelled between the nodes.

Eil51 Test

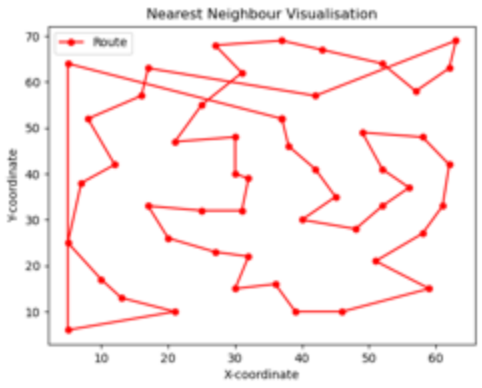
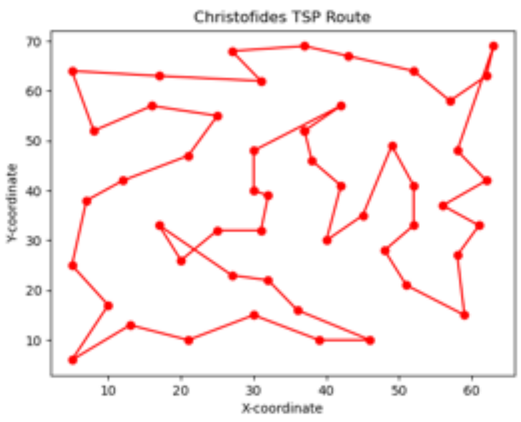
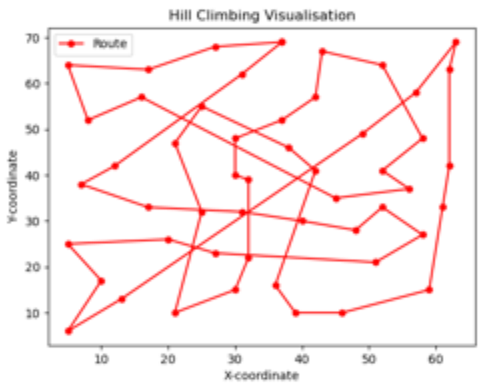
Name of Algorithm	Minimum Cost	Time Taken (seconds)

Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	511	1
Hill Climbing	651	3
Christofides	462	1
Simulated Annealing	1027	1

Table 17: Eil51 Results Table

The approximation algorithms completed the problems in similar times, to the previous example, the most optimal solution that has been found for the solution has a cost of 426, however the most optimal route found by any of these algorithms was from the Christofides algorithm with a cost of 462 which is very close to optimal. The simulated annealing algorithm requiring more than double the cost at 1027.

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*

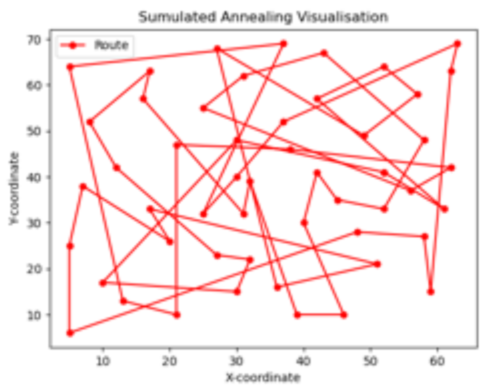
 *Simulated Annealing Visualisation*	

Table 18: Eil51 Visualisation Table

This results table highlight the differences in route and cost, it is clear to see the difference between the Christofides algorithm which is the most efficient and the simulated annealing algorithm which is the least efficient.

Berlin52 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	8980	1
Hill Climbing	10680	4
Christofides	8591	1
Simulated Annealing	18511	1

Table 19: Berlin52 Results Table

The most optimal solution of the Berlin52 problem is a cost of 7542. The most efficient route created was from the Christofides algorithm with a cost of 8591. All approximation algorithms completed the problem in under 1 second except for the hill climbing algorithm which required 4 seconds to complete the problem. The least efficient algorithm was the simulated annealing algorithm with a cost of 18511 more than double the most efficient route.

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*

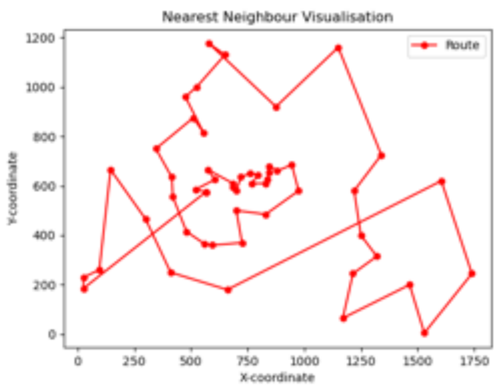
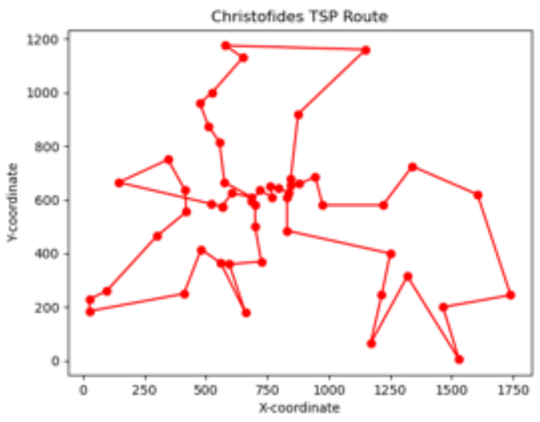
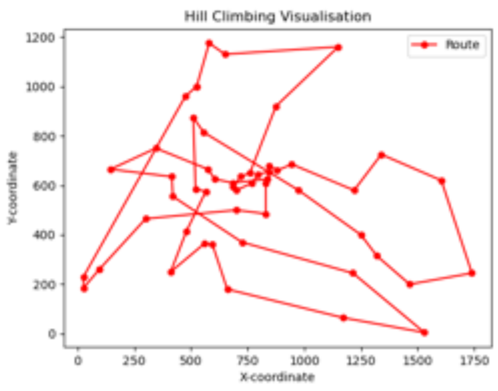
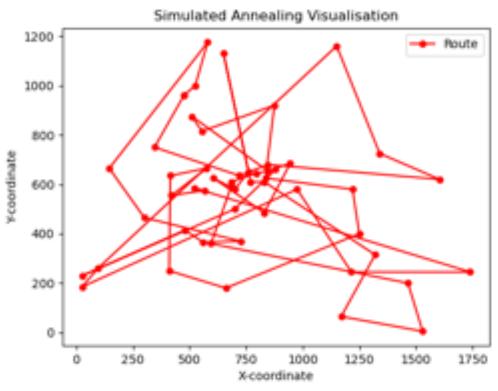
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 20: Berlin52 Route Visualisations

The table above shows the visualisations of the four different routes taken by the algorithms, the difference between the most efficient in the christofides algorithm and the least efficient simulated annealing algorithm can be seen around the middle of the visualisation where the route connects all of the middle visualisations whereas for the simulated annealing algorithm there are more connections to the outer nodes into the middle meaning more distance is travelled thus a higher cost.

Brazil 58 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	30774	1
Hill Climbing	34787	4
Christofides	27205	1
Simulated Annealing	74722	1

Table 21: Brazil 58 Results Table

The Brazil58 problem is different to the other TSP problems as this set of data is an explicit matrix and not a list of coordinates like the other problems. As the problem is too large to run with an exact algorithm the solution for the problem has a cost of 25395. The only algorithm coming close to this optimal path is the Christofides algorithm with an optimal cost of 27205. Other algorithms such as nearest neighbour and hill climbing managed to complete the problem with route costs in the 30,000s however the simulated annealing algorithm required a cost of 74722. Even though the route is almost double the simulated annealing algorithm completed the problem at the same time as the nearest neighbour and Christofides algorithm all in under a second. Hill climbing taking the longest time again with an execution time of 4 seconds.

No visualisation of these routes is produced.

Pr76 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	153462	1
Hill Climbing	143474	20
Christofides	116684	1
Simulated Annealing	349122	1

Table 22: Pr76 Results Table

The pr76 is the first large problem that the approximate algorithms will solve, the optimal cost of the problem is 108151, the christofides algorithm produced the most optimal route out of all the other algorithms with a cost of 116684, with the hill climbing and nearest neighbour algorithms producing slightly longer routes, simulated annealing again producing a route over triple the cost of the most optimal route. The hill climbing algorithm saw a jump in time from the four second execution time previously seen in the Brail58 problem to now 20 seconds which means that the algorithm is nearing an upper bound. The other three algorithms completed the problem in one second or under.

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*

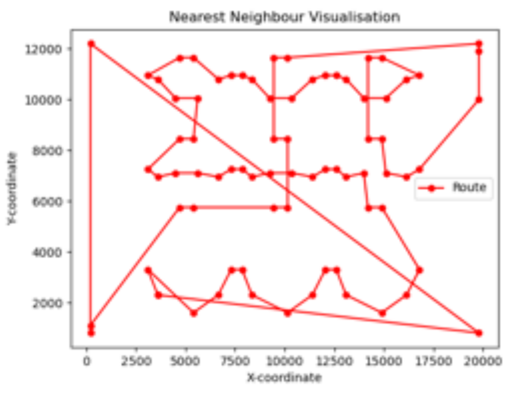
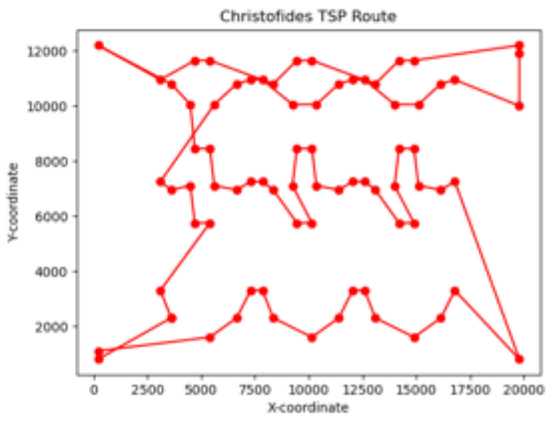
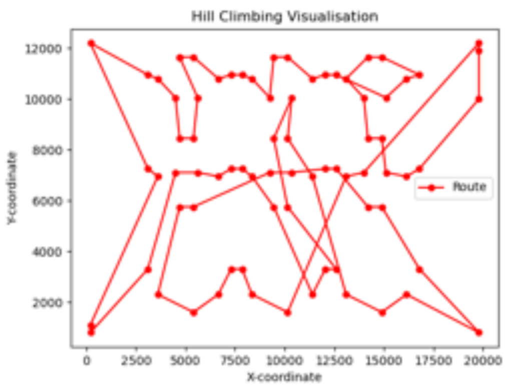
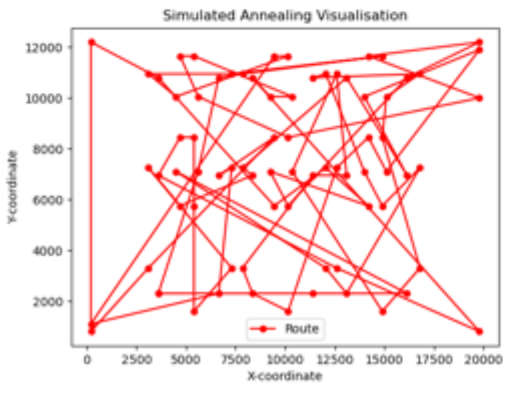
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 23: Pr76 Route Visualisation

The most efficient route can be seen from the christofides algorithm and the elements of that can be seen in the hill climbing and nearest neighbour algorithms routes by having similar patterns in the path. The simulated annealing is clearly the longest route as can be seen by having the longest lines connecting the nodes meaning more distance is travelled.

Ch150 Test

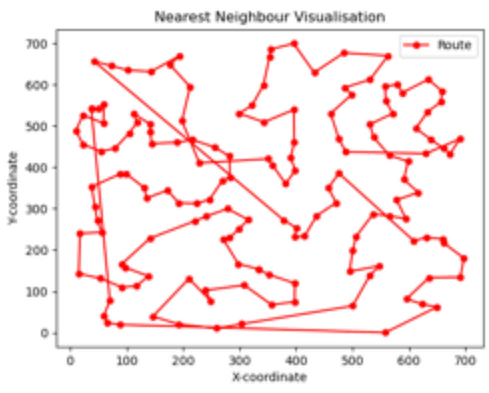
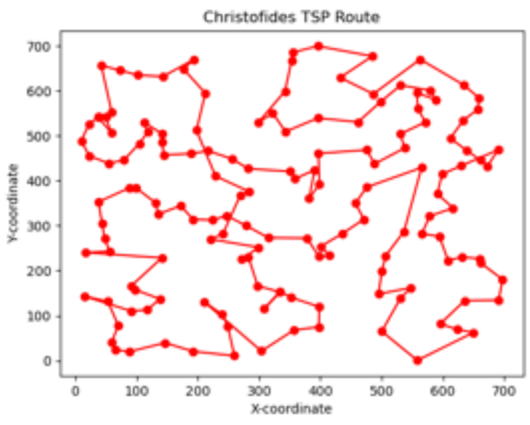
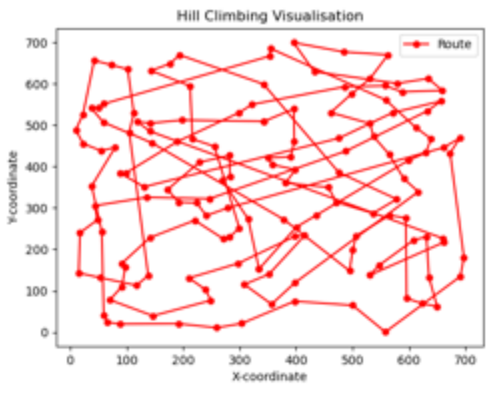
Name of Algorithm	Minimum Cost	Time Taken (seconds)

Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	8191	<1
Hill Climbing	13151	286
Christofides	7135	2
Simulated Annealing	39229	11

Table 24: Ch150 Results Table

The Ch150 data set has an optimal route cost of 6258, the closes algorithm to meet this was the Christofides algorithm with a cost of 7135 which is within 1000 of the optimal cost. However, this finally took more than under a second to complete for this algorithm as it took 2 seconds for the algorithm to complete the route. The hill climbing algorithm completed the route with the third most optimal path and taking the longest time of 286 seconds which is just under 5 minutes. The nearest neighbour algorithm also completed the route however did it in the least amount of time still under 1 second completion time

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*

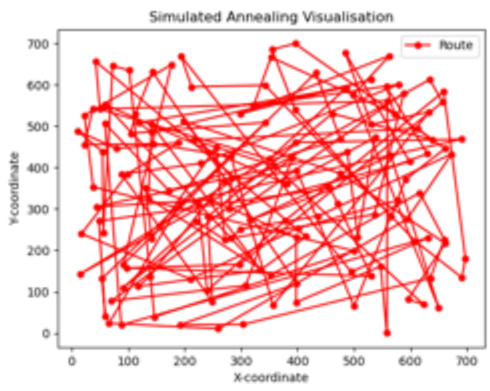
 *Simulated Annealing Visualisation*	

Table 25: Ch150 Route Visualisations

The most efficient route can be seen in the results table above, the christofides algorithm providing the most efficient route can be seen by having the clearest path taken. Whereas on the other hand the simulated annealing algorithm clearly has the least efficient route because there is no clear path taken in the visualisation.

Gr202 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	49336	1
Hill Climbing	68876	1082
Christofides	43256	11
Simulated Annealing	209201	27

Table 26: Gr202 Results Table

The Gr202 problem has an optimal cost of 40160, the closest algorithms follow the same pattern as the previous problems which is the Christofides algorithm completing the most efficient route of the approximation algorithms and the nearest neighbour algorithm producing the second most efficient. The hill climbing algorithm completed the third most efficient route however taking 1082 seconds which is around 18 minutes to complete so this is clearly the maximum bound for the hill climbing algorithm as it cannot complete any larger problem within a reasonable time. There was a greater range of times as all the algorithms aren't completing the problem in under a second and is starting to take time to solve the problem.

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*

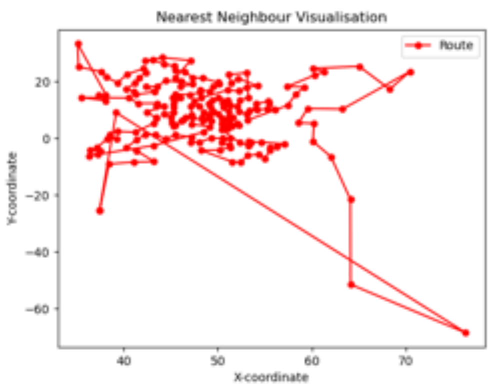
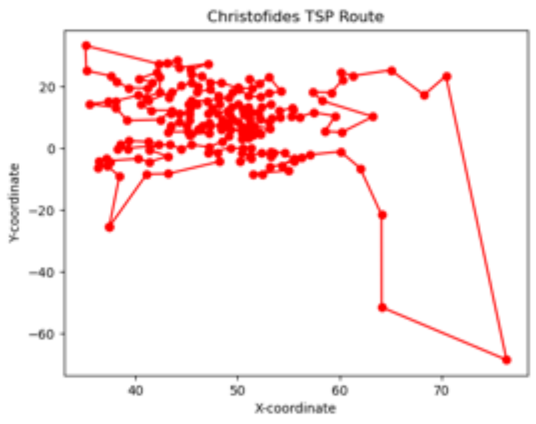
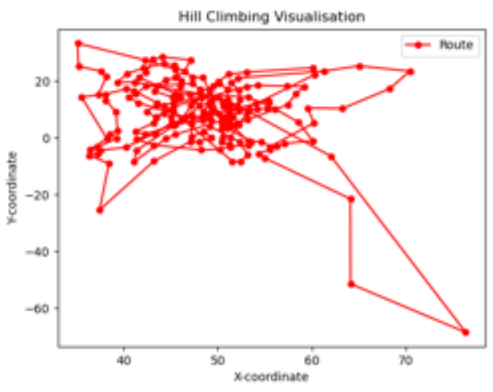
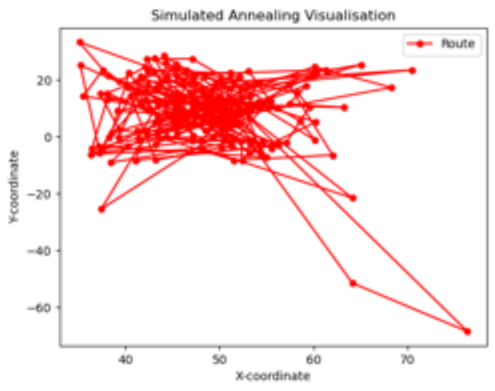
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	 *Hill Climbing Visualisation*
 *Simulated Annealing Visualisation*	

Table 27: Gr202 Route Visualisations

This visualisation table above shows the different paths taken it is clear to see the difference between the efficient routes and the inefficient routes due the clearness of the visualisation. A flaw in the nearest neighbour algorithm can be seen by the long diagonal line across the visualisation where the algorithm has become trapped at that bottom right node and needed to traverse across the entire route to the nearest point which will have increased the cost by a large number.

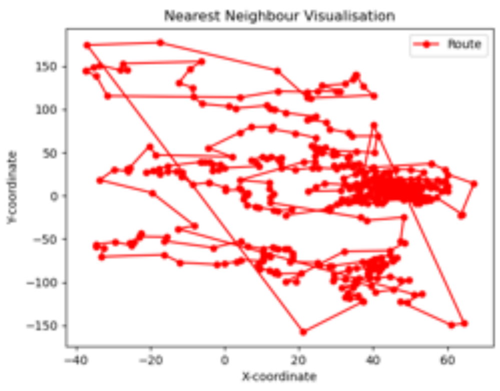
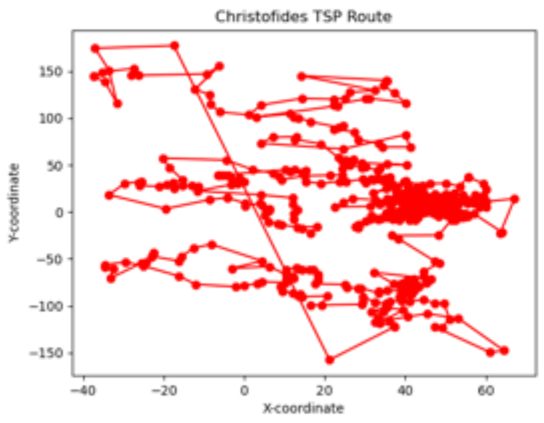
Ali535 Test

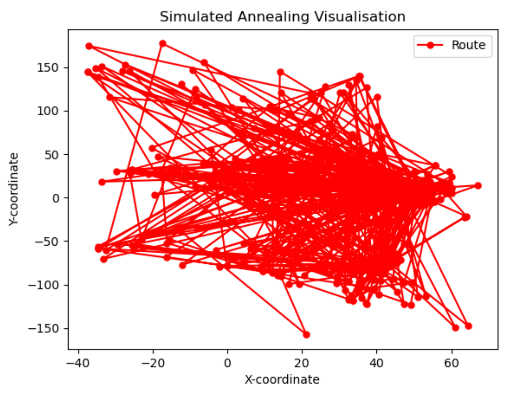
Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	253127	2
Hill Climbing	N/A	N/A
Christofides	229053	94
Simulated Annealing	3188442	615

Table 28: Ali535 Results Table

The optimal cost if this route is 202339, only the nearest neighbour and christofides simulated annealing algorithms could complete this problem in a reasonable time. Christofides completed the more optimal route of the two however taking 94 seconds which is 92 more than the nearest neighbour algorithm. And the simulated annealing reached the upper bound of

The visualisations for these routes look like:

No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
 *Christofides Visualisation*	No route visualisation as the route could not be completed within a reasonable time. *Hill Climbing Visualisation*



Simulated Annealing Visualisation

Nrw1379 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	68964	20
Hill Climbing	N/A	N/A
Christofides	63703	1110
Simulated Annealing	N/A	N/A

Table 29: Nrw1379 Results Table

The optimal cost of this problem is 56638, the Christofides algorithm provided the most efficient route of the two however this took 1110 seconds to complete which is a staggering comparison to the nearest neighbour algorithm which took 20 seconds to complete. This means that the Christofides algorithm has reached the maximum time for execution time as anything longer is an unreasonable time.

U2152 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	79260	235
Hill Climbing	N/A	N/A
Christofides	N/A	N/A
Simulated Annealing	N/A	N/A

Table 30: U2152 Results Table

The only algorithm able to complete this route within a reasonable time is the nearest neighbour algorithm which completed the route with a cost of 79260. This is more than the optimal route which is 64253. However, the nearest neighbour algorithm finally started to take time to complete the route as the algorithm required 235 seconds to complete the route.

The visualisations for these routes look like:

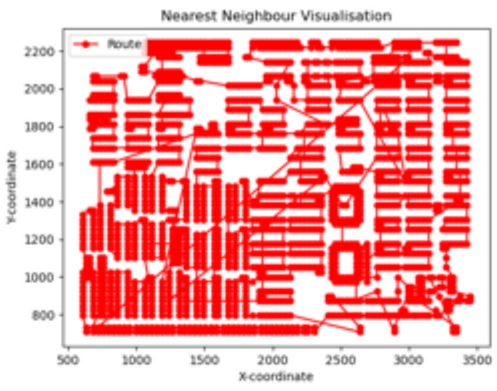
No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Christofides Visualisation*	No route visualisation as the route could not be compelted in a reasonable time. *Hill Climbing Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Simulated Annealing Visualisation*	

Table 31: U2152 Route Visualisations

The table shows the route visualisation from the nearest neighbour algorithm and also shows the flaw to the nearest neighbour algorithm as there are long lines from the bottom corners of the visualisation where the algorithm has gotten trapped without any neighbours near to it and needs to move to the other side of the problem which can be shown with the long diagonal lines across the visualisation and a reason why the cost is higher than the most efficient route.

Pla7397 Test

Name of Algorithm	Minimum Cost	Time Taken (seconds)
Brute-Force	N/A	N/A
Held-Karp	N/A	N/A
Branch and Bound	N/A	N/A
Nearest Neighbour	28107289	964
Hill Climbing	N/A	N/A
Christofides	N/A	N/A
Simulated Annealing	N/A	N/A

Table 32: Pla7379 Results Table

The results table shows that the only algorithm able to complete the problem in a reasonable time is the nearest neighbour algorithm

The visualisation for this route looks like:

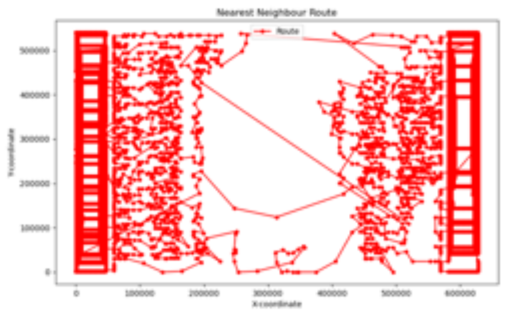
No route visualisation as the route could not be completed in a reasonable time. *Brute Force Visualisation*	No route visualisation as the route could not be completed in a reasonable time *Held Karp Visualisation*
No route visualisation as the route could not be completed in a reasonabl time. *Branch and Bound Visualisation*	 *Nearest Neighbour Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Christofides Visualisation*	No route visualisation as the route could not be completed in a reasonable time. *Hill Climbing Visualisation*
No route visualisation as the route could not be completed in a reasonable time. *Simulated Annealing Visualisation*	

Table 33: Pla7394 Route Visualisations

This table shows the nearest neighbour route visualisation. Although there are a lot of nodes, the route was traversed efficiently, as can be seen, due to the lack of long travelling lines

Data Analysis and Interpretation

This data can be used to analyse the approximation rates of the algorithms, measure the efficiencies and visualise the strengths and weaknesses of the algorithms.

Algorithm Approximations

By using the solution's optimal cost and the cost from the approximation algorithms, the approximation rates of the algorithms can be determined. This can be done by dividing the algorithm route cost by the optimal cost.

Approximation Table

The table used to calculate these approximations looks like:

Problem	optimal	NN Approx	HC Approx	C Approx	SA Approx
d4	594	1	1	1	1
att8	5919	1.08	1.04	1.05	1
burma10	3114	1.16	1	1.05	1.01
burma14	3323	1.22	1	1.09	1.27
ulysses16	6859	1.46	1.02	1.02	1.26

ulysses22	7013	1.51	1.02	1.06	1.29
att48	10628	1.21	1.61	1.19	2.67
eil51	426	1.2	1.53	1.08	2.41
berlin52	7542	1.19	1.42	1.14	2.45
brazil58	25395	1.21	1.37	1.07	2.94
pr76	108158	1.42	1.33	1.08	3.23
ch150	6528	1.25	2.01	1.09	6.01
gr202	40160	1.23	1.72	1.08	5.21
ali535	202339	1.25	0	1.13	0
nrw1379	56638	1.22	0	1.12	0
u2152	64253	1.23	0	0	0
pla7397	23260728	1.21	0	0	0

Table 34: Approximation Table

This table shows a range of different approximations for the different algorithms, what will be measured is the upper bound, lower bound, average approximation and the standard deviation of the approximations.

Nearest Neighbour

- Upper Bound – 1.51 ulysses22 problem
- Lower Bound – 1.16 burma10 problem
- Average approximation – 1.24
- Standard Deviation – 0.125

The nearest neighbour algorithm had the second lowest standard deviation of the algorithms as well as being able to handle the largest number of problems this means that it is a highly versatile and there is low risk in the algorithm producing a high-cost solution. There are exceptions to this as can be seen in Table 31 and 32 where the algorithm gets stuck in a corner and must traverse across the problem to find an unvisited node which will be increasing the cost.

Hill Climbing

- Upper Bound – 2.01 ch150 problem
- Lower Bound – 1 burma10 and burma14 problems
- Average Approximations – 1.31
- Standard Deviation – 0.334

The hill climbing algorithm had the third highest standard deviation, this means that the algorithm is slightly more unpredictable than the nearest neighbour and christofides algorithm when it comes to producing a route with high optimality. The algorithm did produce the lowest bound out of all the algorithms however producing two routes that are fully optimal which was not seen by any other algorithm. The biggest downside to this algorithm is that it solved the lowest number of TSP problems out of the approximation algorithms only solving 12 of the problems.

Christofides

- Upper Bound – 1.19 att48 problem
- Lower Bound – 1.02 ulysses16
- Average – 1.08
- Standard Deviation – 0.05

The christofides algorithm was the best performing approximation algorithm out of the four having the lowest average and deviation it is the most reliable when it comes to consistently producing near-efficient routes. Interestingly the christofides algorithm performed best against the two Ulysses problems which is where the nearest neighbour algorithm performed the worst. Furthermore, this algorithm completed the second most TSP problems of all the algorithms completing 14 out of 16 problems.

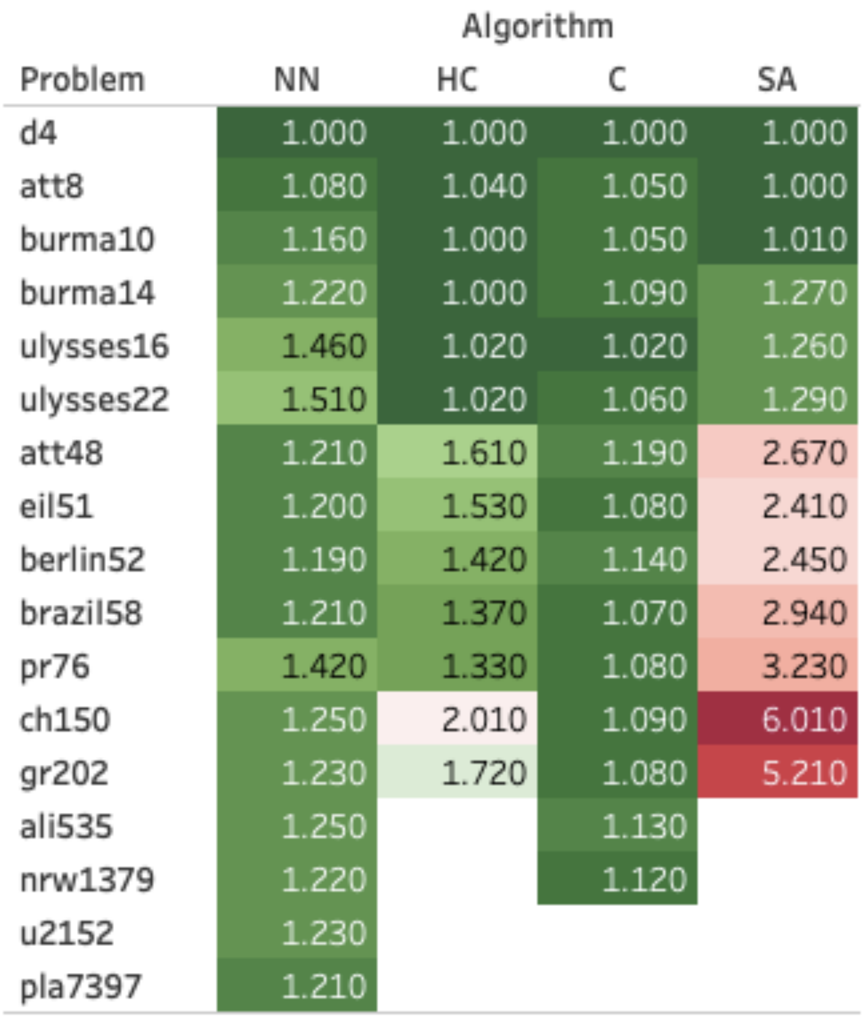
Simulated Annealing

- Upper Bound – 6.01
- Lower Bound – 1.01
- Average – 2.44
- Standard Deviation – 1.62

The simulated annealing algorithm was the most inconsistent algorithm out of all that were tested. The algorithm was initialised to find a route as quickly as possible regardless of optimality, it did this however not as efficiently as expected it only completed the same number of problems as the hill climbing algorithm which made it inefficient as the hill climbing algorithm found the same number of solutions. It had the highest average approximation and the highest deviation between the approximations. Although it started well on the smaller algorithms the approximation jumped massively from 1.29 on the Ulysses22 problem to 2.67 on the att48 problem and then continued to grow massively to 6.01.

Result Visualisations

Approximation Heat Map



	Algorithm			
Problem	NN	HC	C	SA
d4	1.000	1.000	1.000	1.000
att8	1.080	1.040	1.050	1.000
burma10	1.160	1.000	1.050	1.010
burma14	1.220	1.000	1.090	1.270
ulysses16	1.460	1.020	1.020	1.260
ulysses22	1.510	1.020	1.060	1.290
att48	1.210	1.610	1.190	2.670
eil51	1.200	1.530	1.080	2.410
berlin52	1.190	1.420	1.140	2.450
brazil58	1.210	1.370	1.070	2.940
pr76	1.420	1.330	1.080	3.230
ch150	1.250	2.010	1.090	6.010
gr202	1.230	1.720	1.080	5.210
ali535	1.250		1.130	
nrw1379	1.220		1.120	
u2152	1.230			
pla7397	1.210			

Figure 78: Approximation Heat Map

This heat map shows the approximation of the different exact algorithms on different problems. It clearly shows the approximation levels from dark green which is an optimal or near optimal route changing colour to red with the approximation increasing.

This shows two things:

- Strength and weaknesses of the approximation algorithms, hill climbing and simulated annealing work very well on small data sets and gradually perform worse and worse as data sets increase. Performing poorly on data sets above 22 nodes.
- Consistency of the algorithm: christofides algorithm performs consistently well across every problem, nearest neighbour performs consistently average across every algorithm and the other two are not consistent at all.

Stacked Bar Chart

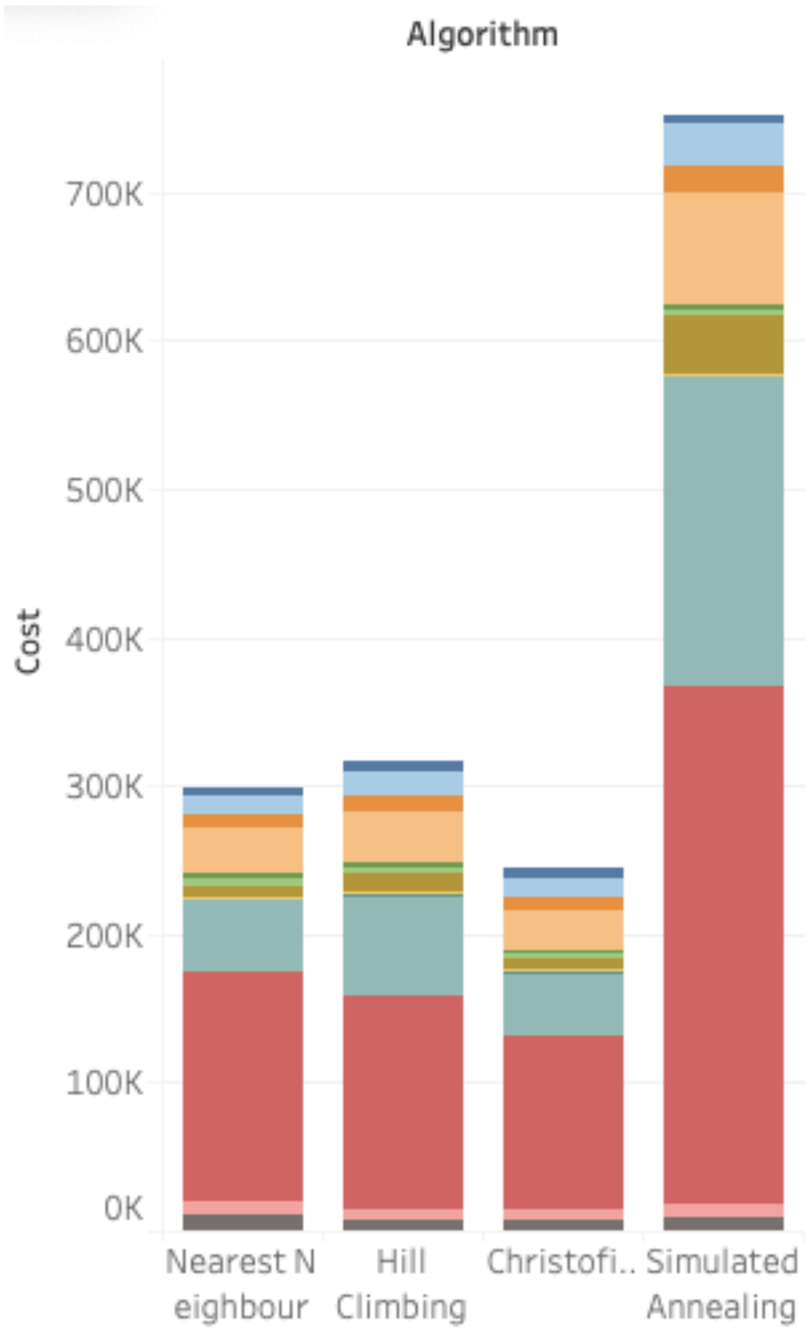


Figure 79: Approximation Algorithm Stacked Bar Charts

This stacked bar chart shows the total cost for each problem for each approximation algorithm. This figure shows that:

- Hill Climbing was the most optimal algorithm when it comes to solving the tsp problems.
- Simulated annealing was as optimal as the other algorithms until it reached a large problem then was outperformed by the other algorithms.

Bar Chart with Upper and Lower Bound Lines

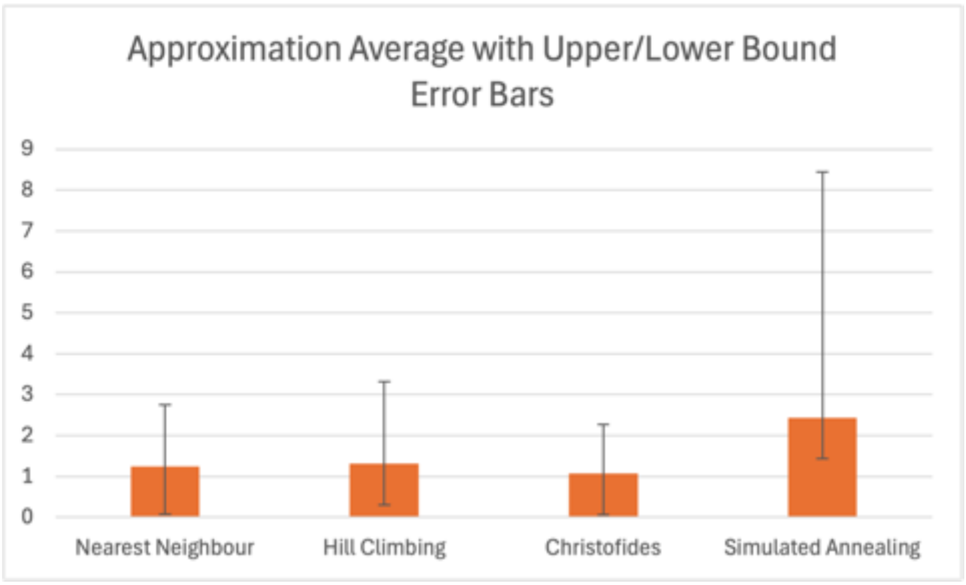


Figure 80: Approximation Averages with Upper/Lower Bound Error Bars

This is a graph that shows the average of approximation rates for the algorithms. The orange bars represent the average, and the error bars represent the upper and lower bounds of approximation.
This graph shows that:

- The christofides algorithm is the most consistent because of the smallest error bar.
- Despite having similar averages, the hill climbing algorithm is less consistent than the nearest neighbour algorithm due to having a larger error bar.

Exact Algorithms

The exact algorithms all performed as intended all producing the most optimal route, the main differences between the algorithms were execution time. The brute force algorithm is the least time effective algorithm the largest solvable problem was the burma10 algorithm which took 34 seconds to complete, the next problem after that the burma14 problem would take an estimated time of 816000 seconds which is approximately 9.5 days. Whereas the other two algorithms completed the problem in under a minute. The branch and bound was able to complete that problem before reaching its upper bound on time meaning that it was able to complete one more problem over the brute force.

The most time efficient algorithm was the held karp algorithm which was able to complete problems of sizes up to 22 nodes. Which is a large value to find with an exact algorithm something the other two algorithms could not do. This is nowhere near what the approximation algorithms could achieve as the slowest approximation algorithm the hill climbing algorithm was able to complete 535 node problems.

Comparison of Exact and Approximate Algorithms

The exact algorithms were completed across six problems with the held – karp algorithm performing the best out of the three algorithms when it came to speed of completing the problem and size of the problem that can be completed in a reasonable time.

The approximate algorithms were completed across seventeen problems, the nearest neighbour algorithm which was able to complete every TSP problem. However, the algorithm that performed the best was the Christofides algorithm which had the lowest approximation rates out of all the algorithms and performed consistently well across all problems.

Bar Chart of Number of Cities Solved Per algorithm

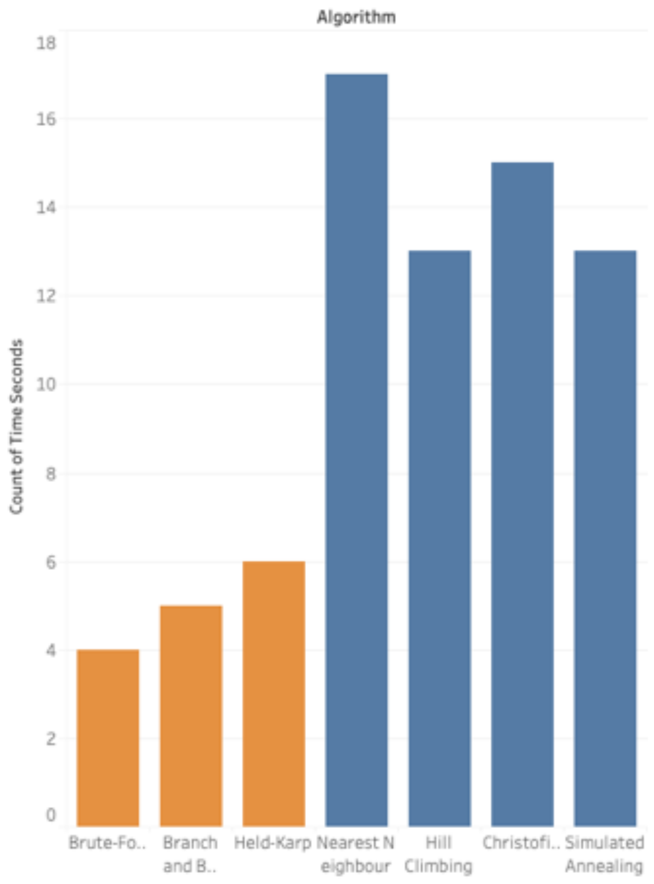


Figure 81: Number of Problems Solved per Algorithm

This graph shows the number of TSP problems completed within a reasonable time. With the exact algorithms coloured in orange and the approximate algorithms shown in blue. The number of problems that could be solved by the worst performing approximate algorithms was more than double that of the best performing exact algorithm.

The approximate algorithms are more much more versatile than the exact algorithms with the brute force being the least time efficient.

Cost vs Number of Cities

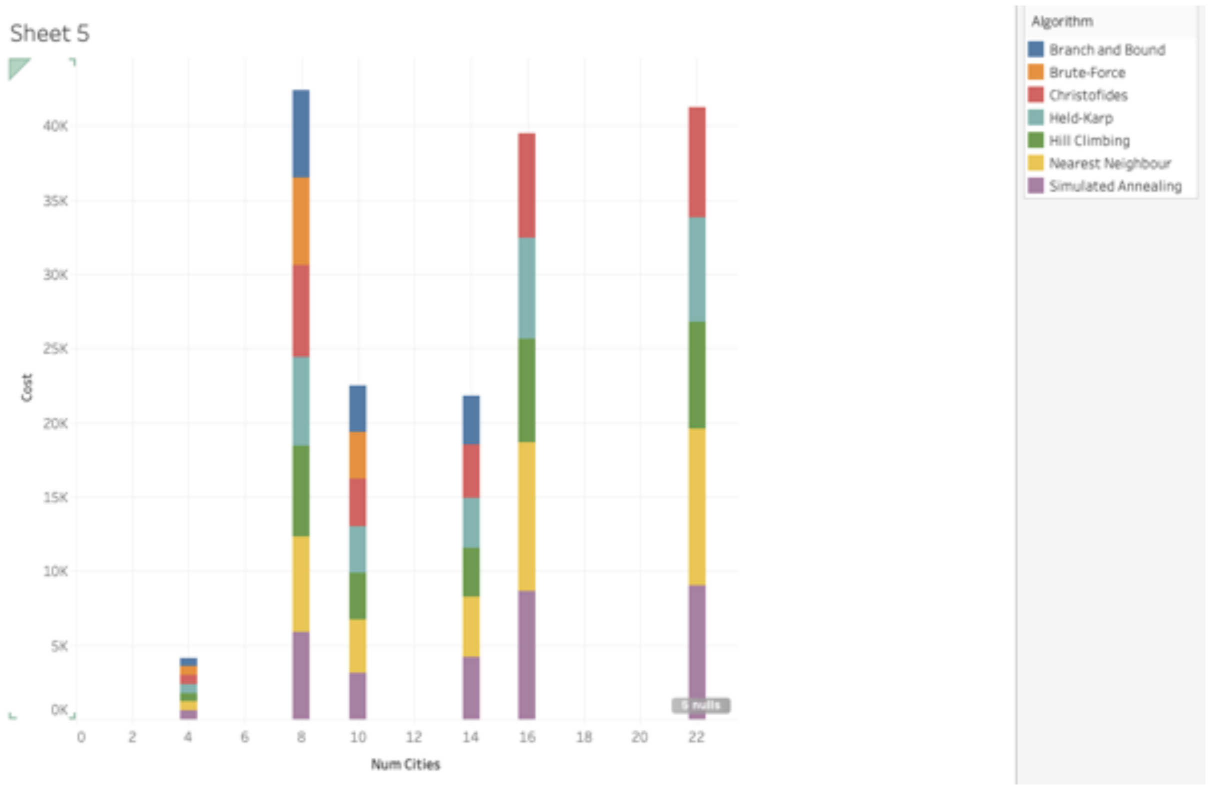


Figure 82: Cost by Size of Problem

This graph is a stacked bar chart that shows the cost each algorithm takes for the number of cities in the problem. This is taken from the first six nodes so to measure efficiencies of the approximate algorithms against the exact algorithms.

- All seven of the algorithms have the same performance on the first algorithm.
- All seven algorithms have a similar performance on the eight node problems as well.
- On the 10-node problem the approximate algorithms start to become inefficient in route covering more distance with the nearest neighbour algorithm covering the most distance.
- Some of the approximate algorithms are still producing near optimal routes.
- On the 14-node problem the gap between the exact and approximate algorithms start to grow the nearest neighbour and simulated annealing algorithm increase more than the exact algorithms.
- The 22-node problem is the last completable by an exact algorithm, the gap grows between the two sets of algorithms although not as large as expected on these smaller problems.

Tree Map

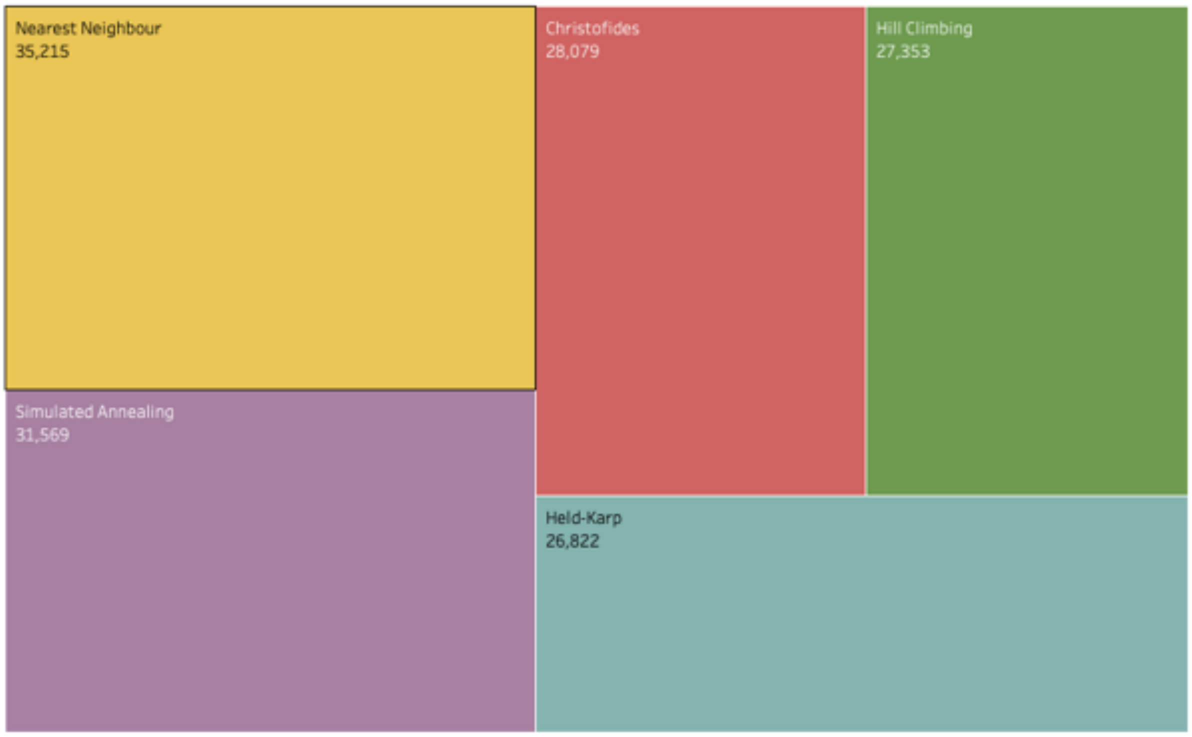


Figure 83: Tree Map of Total Route Cost

This tree map shows the total costs for approximate algorithms and the held karp algorithm for the first six problems.

The biggest takeaway from this tree map is that over six problems there is only a cost difference between the hill climbing algorithm and held karp difference of 531, which is over six algorithms is 88.5. Which is much closer and much more effective than initially anticipated in chapter one.

The least efficient approximation algorithm the nearest neighbour had a total cost of 35,215 which over six problems is an average of 1400 more than the Held-Karp algorithm which is less efficient however is not a large gap.

Time Taken Table

Problem	BF	HK	BB	NN	HC	C	SA
D4	1	1	1	1	1	1	1
ATT8	1	1	1	1	1	1	1
BURMA10	34	1	1	1	1	1	1
BURMA14	N/A	2	53	1	1	1	1
ULYSSES16	N/A	11	N/A	1	1	1	1
ULYSSES22	N/A	596	N/A	1	1	1	1
ATT48	N/A	N/A	N/A	1	3	1	1
EIL51	N/A	N/A	N/A	1	3	1	1
BERLIN52	N/A	N/A	N/A	1	4	1	1
BRAZIL58	N/A	N/A	N/A	1	4	1	1
PR76	N/A	N/A	N/A	1	20	1	2

CH150	N/A	N/A	N/A	1	286	2	11
GR202	N/A	N/A	N/A	1	1082	11	27
ALI535	N/A	N/A	N/A	2	N/A	94	N/A
NRW1379	N/A	N/A	N/A	20	N/A	1110	N/A
U2152	N/A	N/A	N/A	235	N/A	N/A	N/A
PLA7939	N/A	N/A	N/A	964	N/A	N/A	N/A

Table 35: Table of Time Taken to Complete Problems

This table shows the time taken to complete the TSP problems.

- Only once all exact algorithms took more than ten minutes to complete a problem did it take more than a second for an approximate algorithm to complete the problem.
- The hill climbing algorithm was the slowest approximation algorithm and could still complete 76 node problems in 20 seconds which is faster than a brute force could complete a 10-node problem.

For the first six problems the held karp and hill climbing algorithms had a cost difference of 531; so, although the hill climbing algorithm has produced a slightly larger route, it completed all six problems in six seconds, whereas the held karp completed these problems in 607 seconds which is just over 10 minutes. Which means that the approximation algorithm solved the problem 101 times faster than the exact algorithm.

If the brute force algorithm required 34 seconds to complete a 10-node problem, it would require approximately 3.5 days to complete the 14-node problem. To calculate how long it would take to solve pla7939 is impossible because 7939! is too large of a number to calculate, so the nearest neighbour algorithm can complete a problem in 16 minutes that is impossible for brute force to try and attempt. The time differences between the two sets of algorithms are monumental.

The christofides algorithm had the smallest approximation rate of any of the approximate algorithms, having an average approximation rate of 1.08 that of the optimal solution, this means that the average christofides algorithm is 8% less optimal than the optimal route. This algorithm also completed the second most TSP problems within the given time and required problems in the triple figures to require 2 seconds to solve.

Limitations of the Testing

The greatest limitation of the testing is due to hardware, devices with greater memory and processing powers will have been able to process and execute the algorithms more efficiently and quickly meaning that more data can be recorded within the reasonable time set out in the design chapter. To solve this more powerful machinery could've been used however this may not be possible due to university resources.

The limit on hardware to test these problems make a complete comparison between the exact and approximate algorithms hard because although an optimal cost is provided by the TSPLIB library, an optimal route or execution time is not provided this means that it is hard to do a direct comparison when it comes to time and route visualisation between the two.

Another limitation is the TSP problems that are testing the algorithms, only using TSPLIB data files may create patterns in the data that may be different if multiple TSP problems are used from different sources for example using coordinate data from OpenStreetMaps. The TSPLIB data was the correct choice however as this contains multiple data types such as two-dimensional Euclidean data, geographical data and explicit matrix data meaning that the algorithms have been tested in different ways.

A way the testing of the algorithms could have been improved is with reproducibility tests, if the tests on the data types are running 3-5 different times on either the same device or different devices any anomalies produced in this round of tests can easily be identified and will make the results stronger due to the reproducibility of the results. This was not possible due to time constraints on this project and if given more time would be possible to strengthen the results.

7. Project Evaluation

Problem Statement

This report was to investigate the:

The Comparison of Exact and Approximation Algorithms for Completing the Travelling Salesman Problem

This was done by using exact and approximate algorithms to solve the same sets of TSP problems to view the strengths and weaknesses of the different algorithms and to get a comparison between the two sets.

Methods use to compare the two sets of algorithms include, number of problems solved in a reasonable time, cost of the problems solved, time taken to solve the problem and efficiency of the algorithm.

Direct Comparison

Exact algorithms always guarantee an optimal solution for the problems it solves as seen in the initial six problems; however, this comes at the cost of significant processing time to solve the problems especially as the problem size increases. For small scale problems between 4-10 nodes exact algorithms are optimal as problems can be solved optimally in reasonable time; however, this will very quickly become unsustainable as seen in this project.

On the other hand, the approximate algorithms completed the problems much faster but there was a trade-off in route cost and efficiency, while they do not guarantee the optimal solution and on larger solutions offer approximations can offer approximations of 10% in the best-case scenario. This means they are more suitable for problems over ten nodes as these problems can be solved in seconds where it may take the exact algorithms minutes to complete.

Algorithm Performance Summary

- Brute Force – worst performing exact algorithm in terms of time and problems completed. This guarantees the optimal solution but only optimal, but it is only feasible on problems of ten nodes and under.
- Branch and Bound – good for small to medium problems of up to 15 nodes and will also provide optimal route. Time and space complexity is still exponential in worst case scenarios so can be on level with brute force.
- Held – Karp – the best performing exact algorithm in terms of execution time and number of problems completed. Guarantees the optimal route and can solve faster than the other two exact algorithms but times can still be very large for problems over 20 nodes.
- Nearest Neighbour – the best performing approximate algorithm in terms of completion time, however randomisation of starting city can cause algorithm to be trapped and travel long distances to escape leading to suboptimal routes.
- Hill – Climbing – provides very optimal routes on smaller problems however starts to go very sub optimal after 50 nodes and was the slowest of the approximation algorithms.

- Simulated Annealing – offers a balance between route quality and speed and allows for customisation in terms of processing but eventually behaves like the hill climbing route.
- Christofides – the best TSP solving algorithm in the project offers the best overall balance between optimality and execution speed. Completed a 1379-node problem in 18 minutes with the smallest average approximation and the least amount of deviation in approximations.

When to use Exact or Approximate Algorithms.

Approximate algorithms scaled far better than the exact algorithms. Problems such as Ch150, Nrw1379 and Rl5934 would take years to solve using exact algorithms but minutes with approximate algorithms. So exact are far more suitable for problems of 20 nodes and under.

Use exact algorithms when there are problems of 20 nodes and under and accuracy and optimality of the routes is critical and there are no time constraints.

Use approximate algorithms on a large scale and time sensitive problems of nodes of 20+ where a good enough solution is acceptable.

Functional Requirement Fulfilment

In the methodology chapter four bullet points were set out in the functional requirements to have a complete comparison between the algorithms these are:

- Use at least three different exact and three different approximate algorithms.
- Calculate the path taken by the algorithm, the cost of the route, the time taken to complete the algorithm and create a visualisation of the route taken.
- Present the data collected in tables and use graphs and visualisations as an easier way of presenting the data.
- Present a comparison between exact and approximate algorithms.

The first functional requirement set out in the methodology chapter was to use a range of at least three different exact algorithms and three approximate algorithms so there was more data to use in the comparison. This was achieved by using seven different algorithms. Three exact algorithms which were the Brute-Force algorithm, the Held-Karp algorithm and the Branch and Bound algorithm. And four approximate algorithms the Christofides algorithm, the Nearest Neighbour algorithm, the Hill Climbing algorithm and the Simulated Annealing algorithm. More approximate algorithms were chosen over exact algorithms because they all have different features and functions whereas an exact algorithm will always find the most efficient route. Three different exact algorithms were still chosen to see if there were any time differences in these algorithms and what is the largest problem that can be completed with an exact algorithm within a reasonable time.

The second functional requirement was to calculate different metrics from the routes taken by the different algorithms. This was done and can be seen in the design, development and testing chapters. The path taken, cost of the route, time taken and visualisations of the path were all collected, presented and analysed.

The third functional requirement was to present the data in tables and use graphs and other visualisations to present this data for analysis. This was completed in chapter 6 testing and evaluation of artefact; this was useful in helping to explain data and allows for the data to be summarised quickly.

The fourth functional requirement was to present a comparison between exact and approximate algorithms. Which was completed in this chapter.

The nonfunctional requirements albeit limited were also fulfilled as there was no need for user security, the code is properly commented and the functions are properly outlined and kept modular in order to keep the presentation neat.

8. Conclusion and Future Work

Conclusion

In conclusion this project successfully compared different exact and approximate algorithms, using different TSP problems and algorithms strengths and weaknesses for both was found and they were compared. For smaller scale problems an exact algorithm is perfect for solving and finding the optimal route.

These results can be used in broader applications for scheduling algorithms as well as route planning as different algorithms can be selected based off size of problem as seen by the different results on different size problems. Furthermore, these results can also be used to research on robots and autonomous movement as the different algorithms can influence the robots path finding and scheduling which will be great for future use in warehouse picking. Another way these results can be used is by selecting the correct algorithm for use in natural disaster and emergency response, as this can help with rescue routes and evacuation plans.

Time Management

The overall time management of the project was good, all goals that were set out in the introduction and in the project, specification were met and nothing had to be reduced in size or skipped to meet any deadlines. Monthly and weekly meetings set with the project supervisor had section deadlines that had been set by me for these meeting to present work completed. The project will be submitted in full and on time. Confirmation of supervisor meeting and monthly meeting notes can be found in the appendix chapter.

Issues Faced and Shortcomings

An issue faced was due to the processing power and memory of the hardware used to complete these tests, because of this there is limited data for the exact algorithms making a comparison more difficult especially when creating visualisations of the data when comparing time taken as the exact algorithms are all over ten minutes long before the approximate algorithms begin to take two seconds to process.

To continue this another issue faced was by setting the reasonable time to run the algorithms at ten minutes, this could have been longer extending to thirty minutes to gain more data and allow for a better comparison between the two sets of algorithms.

One of the issues faced in this project was to do with the loading tsp file function when used on especially large files as was seen in the nearest neighbour algorithm. The function builds the graph library before the algorithm starts running so will build the dictionary for every city and will build another one to represent the next city so is always building the square value of the number of nodes. This will add significant delay to large tsp problems like the 7000-node example used earlier because it means a 49 million distance dictionary needs to be loaded before the algorithm is ran and the algorithm will only take seconds to run causing a massive delay that are unreported. An on demand look up could be used as used in other nearest neighbour examples to only build graphs for whatever data is needed. Or using iteration to store this data in temporary variables so it can be iterated through and built more quickly.

A potential shortcoming is that there is no user interaction with the program whilst it is running this can be inconvenient and would look more professional if the user could select a problem or an algorithm when the program is running instead of hardcoding the problem into the main code of the program itself.

Potential Improvements

One way this experiment would have been repeated differently is regular interval TSP problems. The TSP problems available in the TSP library are limited when it comes to smaller problems with the smallest ones all being selected and tested. To improve the data collected one improvement would be to have TSP problems in intervals of 2 for example: 4, 6, 8, 10, 12, 14, 16, 18, 20 node problems to gain more data on the exact algorithms and produce a better comparison with the approximate algorithms.

Additionally using only TSPLIB data can be misleading to the data collected for the comparison so more data sources and more TSP problems could have been used to ensure a fairer spread of data and potentially more reliable results.

Another improvement that would be made is repeating the experiment multiple times to get more consistent data and outline any anomalies in the data as running an experiment once does not provide the most accurate data and any outliers cannot be identified until it has been ran twice.

Future Work

Future work that can be taken to expand the project includes, using a graphical user interface to allow users more control of the program for example a GUI can allow users to select a TSP problem and algorithm as well as select options for route visualisation such as colour of visualisation. As well as allowing users to see all the data produced by the algorithm inside this interface, meaning that the command line interface is replaced, and the program can be used more interactively. This program could be expanded from a CLI comparison of seven algorithms to an interface to help user's plan the fastest route to places and run tests on the most efficient routes possible, this could help with travelling, storage or even planning a grocery list to get every item in a store in the correct way.

Another way the project could be expanded is by taking user input to build the TSP problem for example using data from OpenStreetMaps or map API's the program could allow users to enter geographical locations and create a list of places and the different algorithms can be run on. This gives the user more control of the program and means the comparison between the exact and approximation algorithms can be made tailor specific towards the user needs.

Thirdly more approximation algorithms could be added such as the greedy algorithm and ant colony optimisation for an even bigger comparison. And giving the users more information based off the best route to take.

A way that the outputs could be improved is highlighting the most used paths and most used connections across the algorithms in the route paths that are printed out like how the ant colony method would work this could help users with visualising how routes are less efficient than other's if they are taking the wrong path at certain places.

9. Bibliography

Agarwal, S. et al., 2020. E-Commerce Delivery Routing System Using Bellman-Held-Karp Algorithm. In: S. Books, ed. *Advances in Electromechanical Technologies.* Singapore: Springer, pp. 277-285.
Amazon Web Services, 2025. *What is Python?.* [Online]
Available at: https://aws.amazon.com/what-is/python/#:~:text=Python%20is%20a%20programming%20language,systems%2C%20and%20increases%20development%20speed.
[Accessed 3 Apr 2025].
Cambridge Assessment International Education, 2023. *Pseduocode Guide For Teachers.* [Online]
Available at: https://www.cambridgeinternational.org/Images/697401-2026-pseudocode-guide-for-teachers.pdf
[Accessed 28 Mar 2025].
Chase, C., Chen, H., Neoh, A. & Wilder-Smith, M., 2020. *An Evaluation of the Travelling Salesman Problem.* [Online]
Available at: https://scholarworks.calstate.edu/downloads/xg94hr81q?locale=en
[Accessed 24 Feb 2025].
Christofides, N. & Whitlock, C., 1977. An Algorithm for Two-Dimensional Cutting Problems. *Operations Research,* 25(1), pp. 30-44.
Cook, W., 2018. *UK49687 Shortest Possible Tour to Nearly Every Pub in the United Kingdom.* [Online]
Available at: https://www.math.uwaterloo.ca/tsp/uk/index.html
[Accessed 12 Mar 2025].
Eyelit Technologies, 2023. *The Travelling Salesman Problem in Manufacturing.* [Online]
Available at: https://eyelit.ai/the-traveling-salesman-problem/
[Accessed 11 Mar 2025].
Fortnow, L., 2013. *The golden ticket : P, NP and the search for the impossible.* Course Book ed. Princeton: Princeton University Press.
GeeksForGeeks, 2024. *Hill Climbing and Simulated Annealing for the Travelling Salesman Problem.* [Online]
Available at: https://www.geeksforgeeks.org/hill-climbing-and-simulated-annealing-for-the-traveling-salesman-problem/
[Accessed 11 Mar 2025].
GeeksForGeeks, 2024. *Hill Climbing and Simulated Annealing for the Travelling Salesman Problem.* [Online]
Available at: https://www.geeksforgeeks.org/hill-climbing-and-simulated-annealing-for-the-traveling-salesman-problem/
[Accessed 06 Apr 2025].
GeeksForGeeks, 2024. *Travelling Salesman Problem Using Branch and Bound.* [Online]
Available at: https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/
[Accessed 6 Apr 2025].
Ghosh, P., 2025. *Christofides Algorithm: The Secret Weapon for Route Optimization.* [Online]
Available at: https://priyadarshanghosh26.medium.com/christofides-algorithm-the-secret-weapon-for-route-optimization-d2b9ec68d66e
[Accessed 06 Apr 2025].

Grant, R., 2020. *tsplib95.* [Online]
Available at: https://pypi.org/project/tsplib95/#data
[Accessed 20 Mar 2025].

Grant, R., 2022. *TSPLIB 95 Documentation.* [Online]
Available at: https://tsplib95.readthedocs.io/_/downloads/en/latest/pdf/
[Accessed 30 Mar 2025].

Gutin, G. & Punnen, A. P., 2002. *The travelling salesman problem and its variations.* 1st ed. 2007 ed. Boston: Kluwer Academic Publishers.

Howell, E., 2023. *Hill Climbing Optimisation Algorithm: A Simple Beginner's Guide.* [Online]
Available at: https://towardsdatascience.com/hill-climbing-optimization-algorithm-simply-explained-dbf1e1e3cf6c/#:~:text=The%20general%20flow%20of%20the,as%20a%20number%20of%20iterations.
[Accessed 28 Feb 2025].

John D. Hunter, M. D., 2025. *Matplotlib.* [Online]
Available at: https://pypi.org/project/matplotlib/#data
[Accessed 20 Mar 2025].

Koether, R. T., 2016. *The Travelling Salesman Problem Nearest-Neighbour Algorithm.* [Online]
Available at: https://people.hsc.edu/faculty-staff/robbk/Math111/Lectures/Fall%202016/Lecture%2033%20-%20The%20Nearest-Neighbor%20Algorithm.pdf
[Accessed 28 Feb 2025].

Liang, D., 2024. *Intro - Python Algorithms: Travelling Salesman Problem.* [Online]
Available at: https://medium.com/@davidlfliang/intro-python-algorithms-traveling-salesman-problem-ffa61f0bd47b
[Accessed 03 Apr 2025].

matplotlib, 2024. *Simple Plot.* [Online]
Available at: https://matplotlib.org/stable/gallery/pyplots/pyplot_simple.html#sphx-glr-gallery-pyplots-pyplot-simple-py
[Accessed 31 Mar 2025].

matplotlib, 2024. *Using Matplotlib.* [Online]
Available at: https://matplotlib.org/stable/users/explain/quick_start.html#parts-of-a-figure
[Accessed 31 Mar 2025].

Microsoft, 2025. *Visual Studio Code.* [Online]
Available at: https://visualstudio.microsoft.com/#vscode-section
[Accessed 03 Apr 2025].

Ming Tan, C., 2008. *Simulated Annealing.* s.l.:IntechOpen.

NetworkX, 2024. *Minimum Spanning Tree.* [Online]
Available at: https://networkx.org/documentation/stable/auto_examples/graph/plot_mst.html
[Accessed 06 Apr 2025].

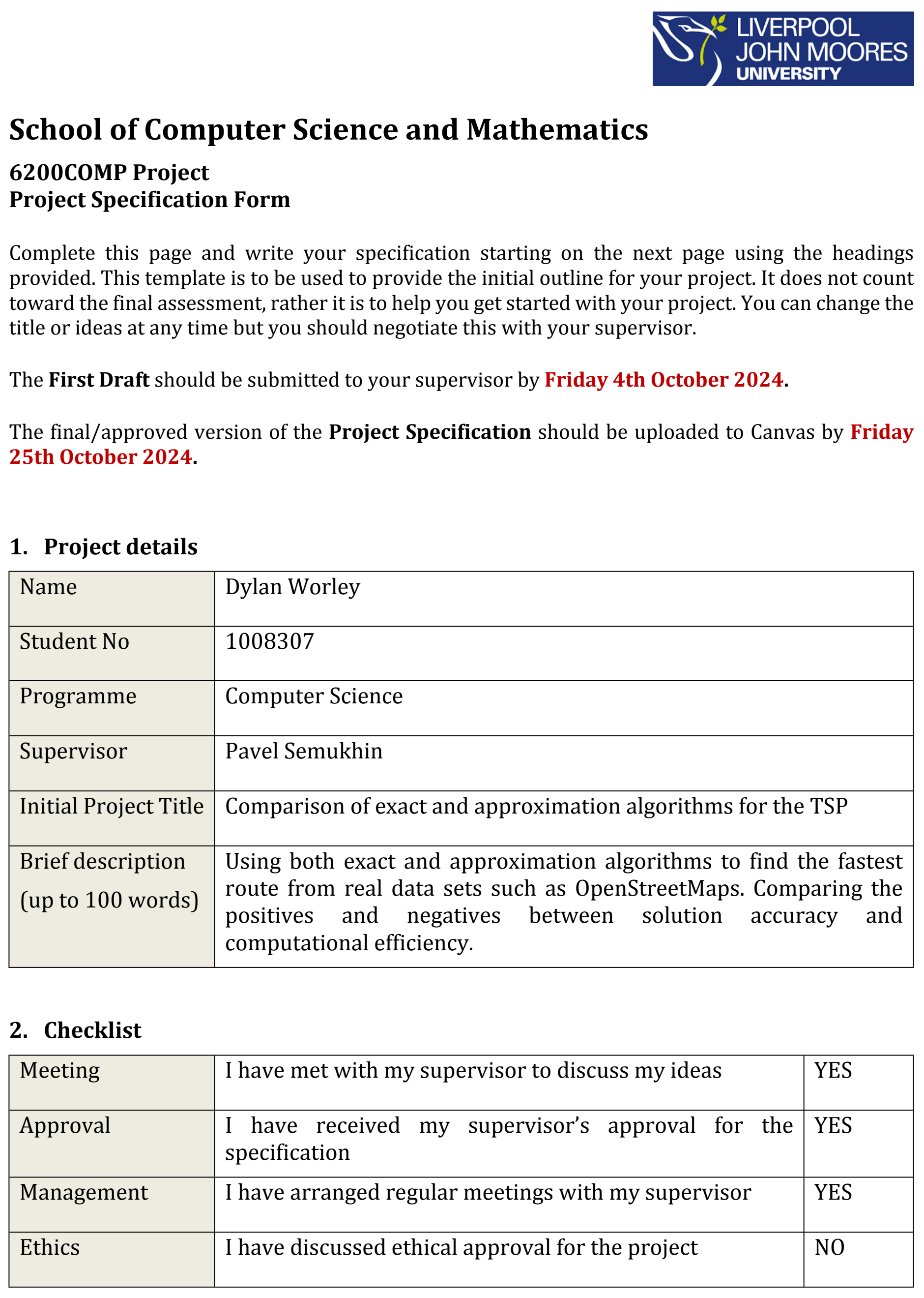
NetworkX, 2024. *Minimum Weight Matching.* [Online]
Available at: https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.matching.min_weight_matching.html
[Accessed 06 Apr 2025].

Nguyen, Q. N., 2020. *Travelling Salesman Problem and Bellman-Held-Karp Algorithm.* [Online]
Available at: https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf
[Accessed 27 Feb 2025].

Oliphant, T. E., 2025. *NumPy.* [Online] Available at: https://pypi.org/project/numpy/#data [Accessed 20 Mar 2025].
Plummer, H., n.d. *The Travelling Salesman Problem: An Introduction.* [Online] Available at: https://users.cs.cf.ac.uk/C.L.Mumford/howard/Introduction.html [Accessed 10 Jan 2025].
Python, 2023. *Itertools.* [Online] Available at: https://pypi.org/project/itertools-permutations/#data [Accessed 20 Mar 2025].
Reinelt, G., 2013. *TSPLIB.* [Online] Available at: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ [Accessed 12 Mar 2025].
Routoura, n.d. *Routoura Intro.* [Online] Available at: http://routora.com/web-intro [Accessed 11 Mar 2025].
Saiyed, A. R., 2012. *The Travelling Salesman Problem.* [Online] Available at: https://cs.indstate.edu/~zeeshan/aman.pdf [Accessed 24 Feb 2025].
Smith, A. P., Munoz, C. A., Narkawicz, A. J. & Markevicius, M., 2015. *An Implementation Framework for Branch and Bound Algorithms.* Hampton, Virginia: National Aeronautics and Sapce Administration.
Watts, S., 2024. *SOLID Principles in Object Oriented Design.* [Online] Available at: https://www.bmc.com/blogs/solid-design-principles/ [Accessed 03 Apr 2025].
Weisstein, E., 2025. *Eulerian Graph.* [Online] Available at: https://mathworld.wolfram.com/EulerianGraph.html [Accessed 06 Apr 2025].
Worley, D., 2025. *TSP Project.* [Online] Available at: github.com/DylWorley04/TSP [Accessed 10 April 2025].
Yu, J., 2014. *Travelling Salesman Problem - History.* [Online] Available at: https://optimization.cbe.cornell.edu/index.php?title=Traveling_salesman_problem [Accessed 24 Feb 2024].

10. Appendix

Project Specification



School of Computer Science and Mathematics

6200COMP Project
Project Specification Form

Complete this page and write your specification starting on the next page using the headings provided. This template is to be used to provide the initial outline for your project. It does not count toward the final assessment, rather it is to help you get started with your project. You can change the title or ideas at any time but you should negotiate this with your supervisor.

The **First Draft** should be submitted to your supervisor by **Friday 4th October 2024**.

The final/approved version of the **Project Specification** should be uploaded to Canvas by **Friday 25th October 2024**.

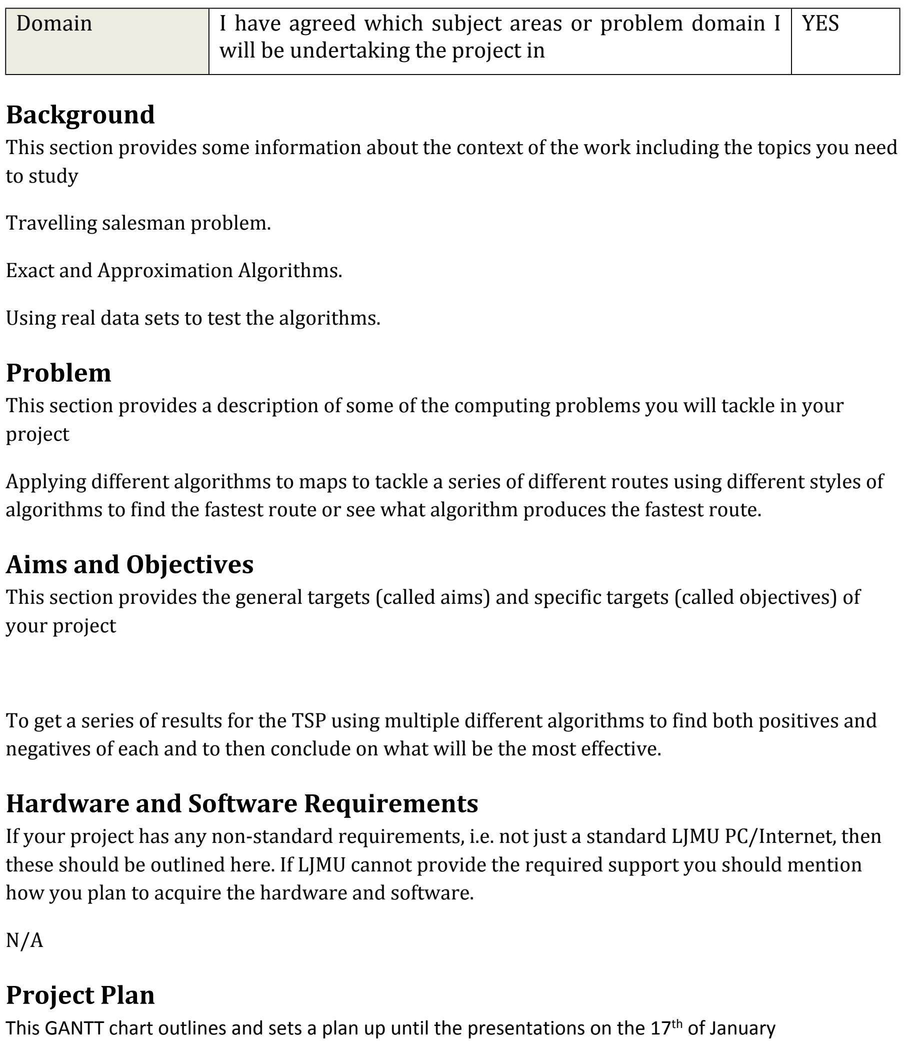
1. Project details

Name	Dylan Worley
Student No	1008307
Programme	Computer Science
Supervisor	Pavel Semukhin
Initial Project Title	Comparison of exact and approximation algorithms for the TSP
Brief description (up to 100 words)	Using both exact and approximation algorithms to find the fastest route from real data sets such as OpenStreetMaps. Comparing the positives and negatives between solution accuracy and computational efficiency.

2. Checklist

Meeting	I have met with my supervisor to discuss my ideas	YES
Approval	I have received my supervisor's approval for the specification	YES
Management	I have arranged regular meetings with my supervisor	YES
Ethics	I have discussed ethical approval for the project	NO

Figure 84: Project Specification Page 1



Domain	I have agreed which subject areas or problem domain I will be undertaking the project in	YES

Background

This section provides some information about the context of the work including the topics you need to study

Travelling salesman problem.

Exact and Approximation Algorithms.

Using real data sets to test the algorithms.

Problem

This section provides a description of some of the computing problems you will tackle in your project

Applying different algorithms to maps to tackle a series of different routes using different styles of algorithms to find the fastest route or see what algorithm produces the fastest route.

Aims and Objectives

This section provides the general targets (called aims) and specific targets (called objectives) of your project

To get a series of results for the TSP using multiple different algorithms to find both positives and negatives of each and to then conclude on what will be the most effective.

Hardware and Software Requirements

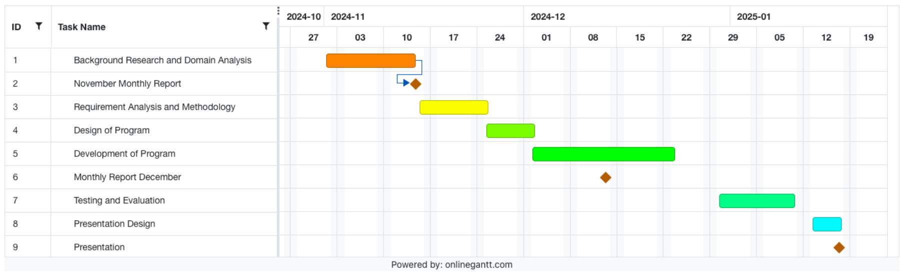
If your project has any non-standard requirements, i.e. not just a standard LJMU PC/Internet, then these should be outlined here. If LJMU cannot provide the required support you should mention how you plan to acquire the hardware and software.

N/A

Project Plan

This GANTT chart outlines and sets a plan up until the presentations on the 17th of January

Figure 85: Project Specification Page 2



ID	Task Name
1	Background Research and Domain Analysis
2	November Monthly Report
3	Requirement Analysis and Methodology
4	Design of Program
5	Development of Program
6	Monthly Report December
7	Testing and Evaluation
8	Presentation Design
9	Presentation

References

Some of the resources I will be using are:

TSP Wikipedia : https://en.wikipedia.org/wiki/Travelling_salesman_problem

Different types of TSP Algorithms: https://www.routific.com/blog/travelling-salesman-problem

Using dynamic programming for the TSP: https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/

A collection of TSP samples:

- http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
- https://github.com/mastqe/tsplib

Signature (student)	[illegible]	Date	25/10/2024
Signature (supervisor)	[illegible]	Date	25/10/2024

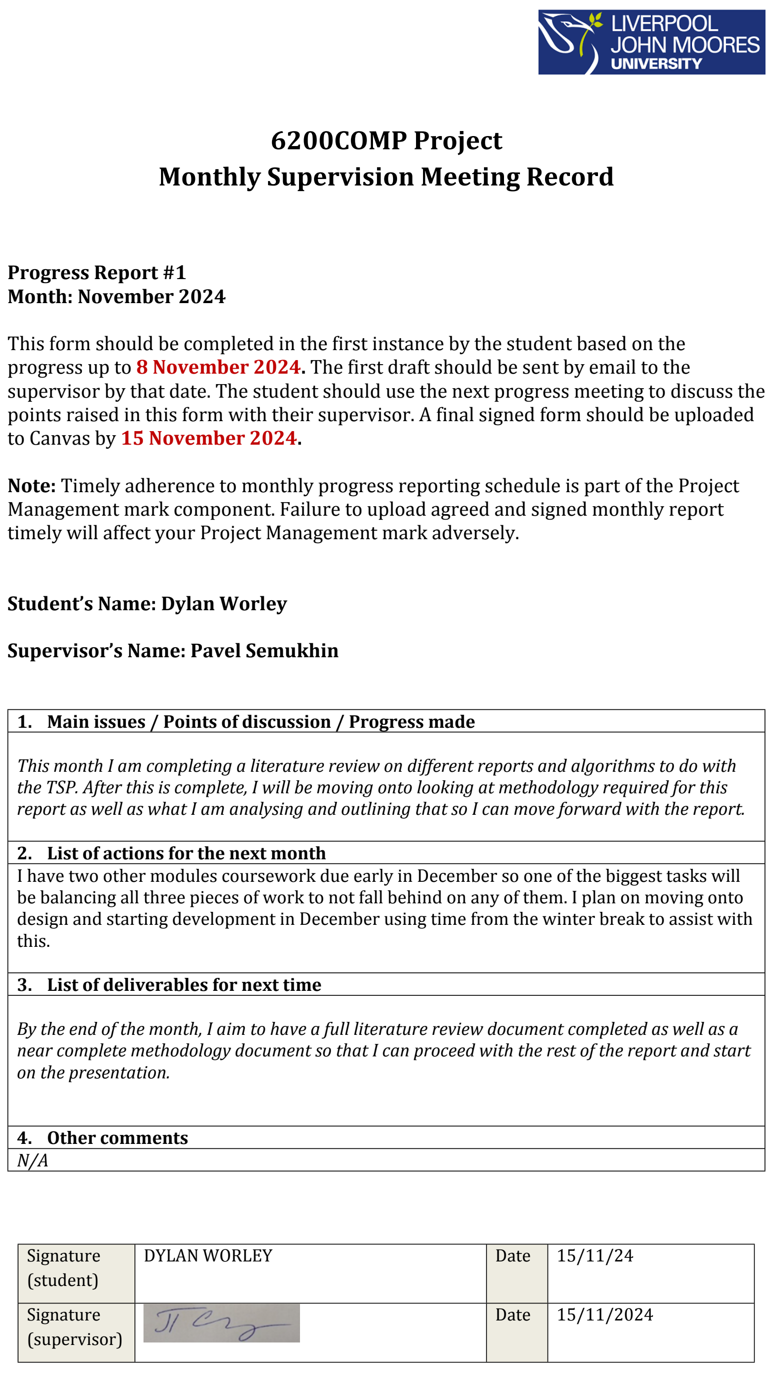
Figure 86: Project Specification Page 3

This is the original project specification from the start of the project, initially outlining the report to be using data types such as OpenStreetMaps which would not have been as effective as TSPLIB. The original Gantt chart also was slightly wrong in terms of dates due to a deadline misunderstanding which means the chart needs to be readjusted.

Signed Monthly Reports and Monthly Meetings

Each monthly report was also followed with a monthly in person meeting with my project supervisor to discuss monthly progress. Towards the end of March and start of April meeting became weekly instead of monthly to ensure product is staying in correct direction and ensuring that everything is being done correctly.

November 2024



LIVERPOOL JOHN MOORES UNIVERSITY

6200COMP Project
Monthly Supervision Meeting Record

Progress Report #1
Month: November 2024

This form should be completed in the first instance by the student based on the progress up to **8 November 2024**. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by **15 November 2024**.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Dylan Worley

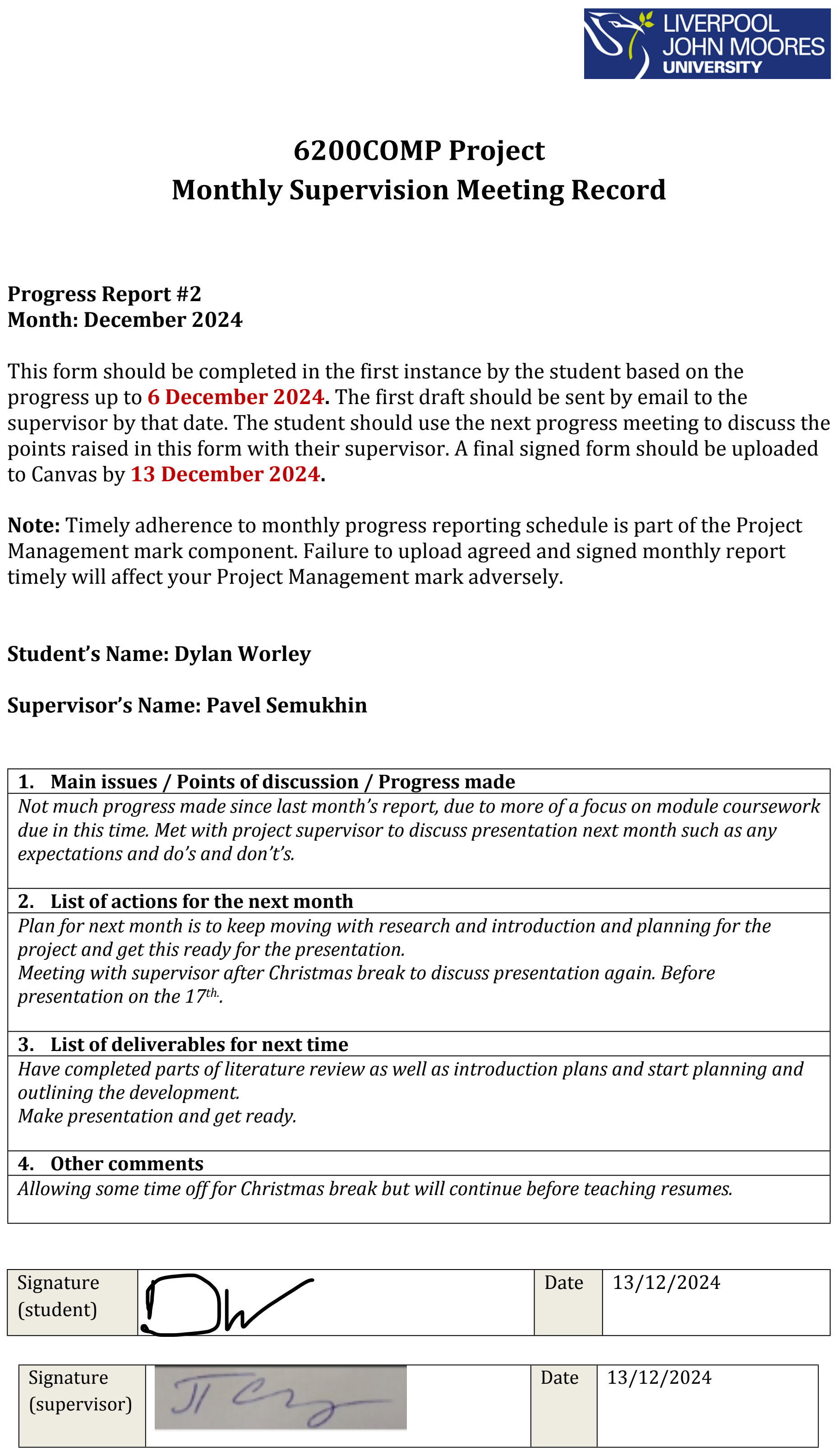
Supervisor's Name: Pavel Semukhin

1. Main issues / Points of discussion / Progress made
This month I am completing a literature review on different reports and algorithms to do with the TSP. After this is complete, I will be moving onto looking at methodology required for this report as well as what I am analysing and outlining that so I can move forward with the report.
2. List of actions for the next month
I have two other modules coursework due early in December so one of the biggest tasks will be balancing all three pieces of work to not fall behind on any of them. I plan on moving onto design and starting development in December using time from the winter break to assist with this.
3. List of deliverables for next time
By the end of the month, I aim to have a full literature review document completed as well as a near complete methodology document so that I can proceed with the rest of the report and start on the presentation.
4. Other comments
N/A

Signature (student)	DYLAN WORLEY	Date	15/11/24
Signature (supervisor)	[illegible]	Date	15/11/2024

Figure 87: November 2024 Meeting Notes

December 2024



6200COMP Project
Monthly Supervision Meeting Record

Progress Report #2
Month: December 2024

This form should be completed in the first instance by the student based on the progress up to **6 December 2024**. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by **13 December 2024**.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Dylan Worley

Supervisor's Name: Pavel Semukhin

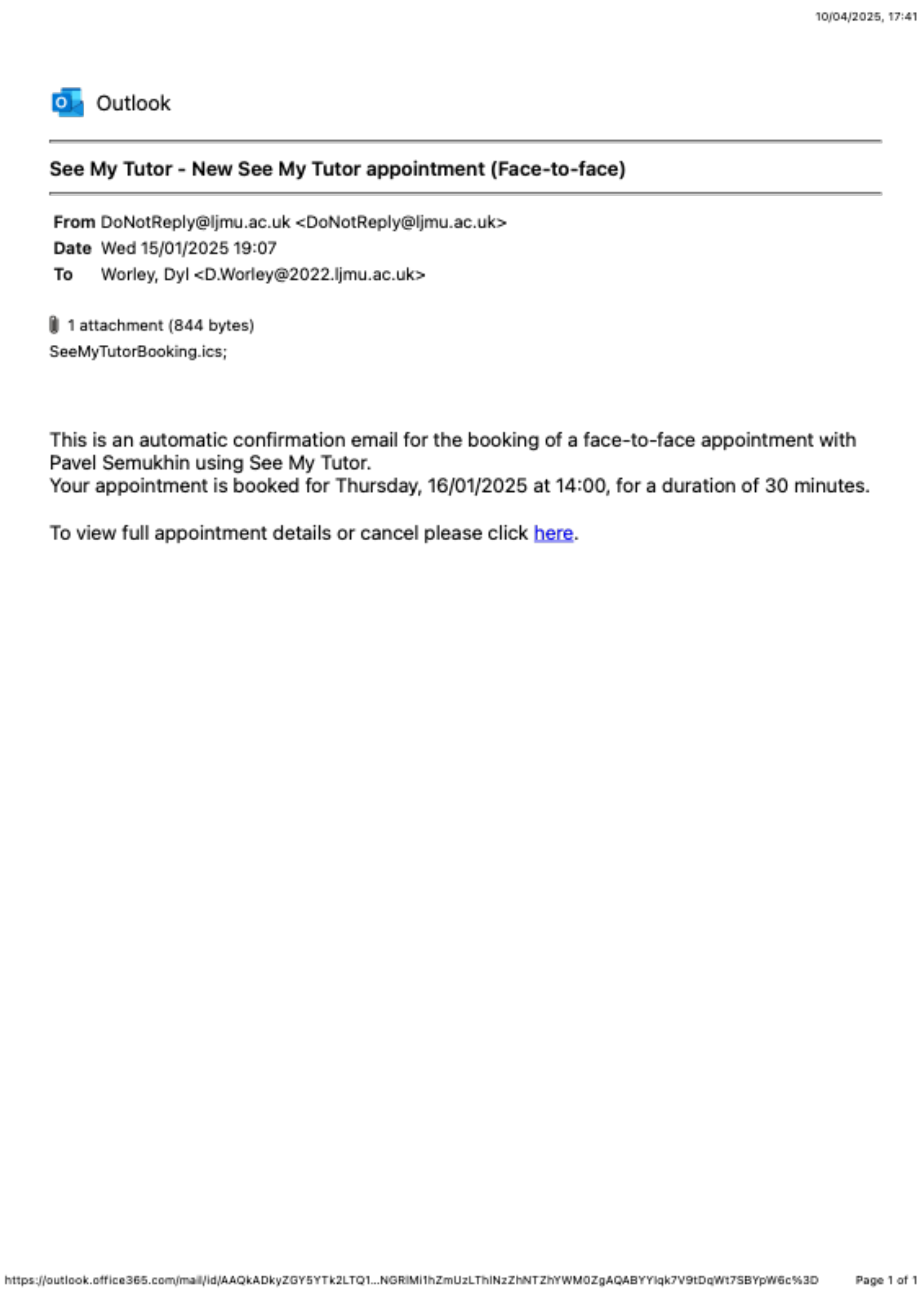
1. Main issues / Points of discussion / Progress made
Not much progress made since last month's report, due to more of a focus on module coursework due in this time. Met with project supervisor to discuss presentation next month such as any expectations and do's and don't's.
2. List of actions for the next month
Plan for next month is to keep moving with research and introduction and planning for the project and get this ready for the presentation. *Meeting with supervisor after Christmas break to discuss presentation again. Before presentation on the 17th.*
3. List of deliverables for next time
Have completed parts of literature review as well as introduction plans and start planning and outlining the development. *Make presentation and get ready.*
4. Other comments
Allowing some time off for Christmas break but will continue before teaching resumes.

Signature (student)	[illegible]	Date	13/12/2024

Signature (supervisor)	[illegible]	Date	13/12/2024

Figure 88: December 2024 Monthly Meeting Notes

January 2025



10/04/2025, 17:41

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Wed 15/01/2025 19:07
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

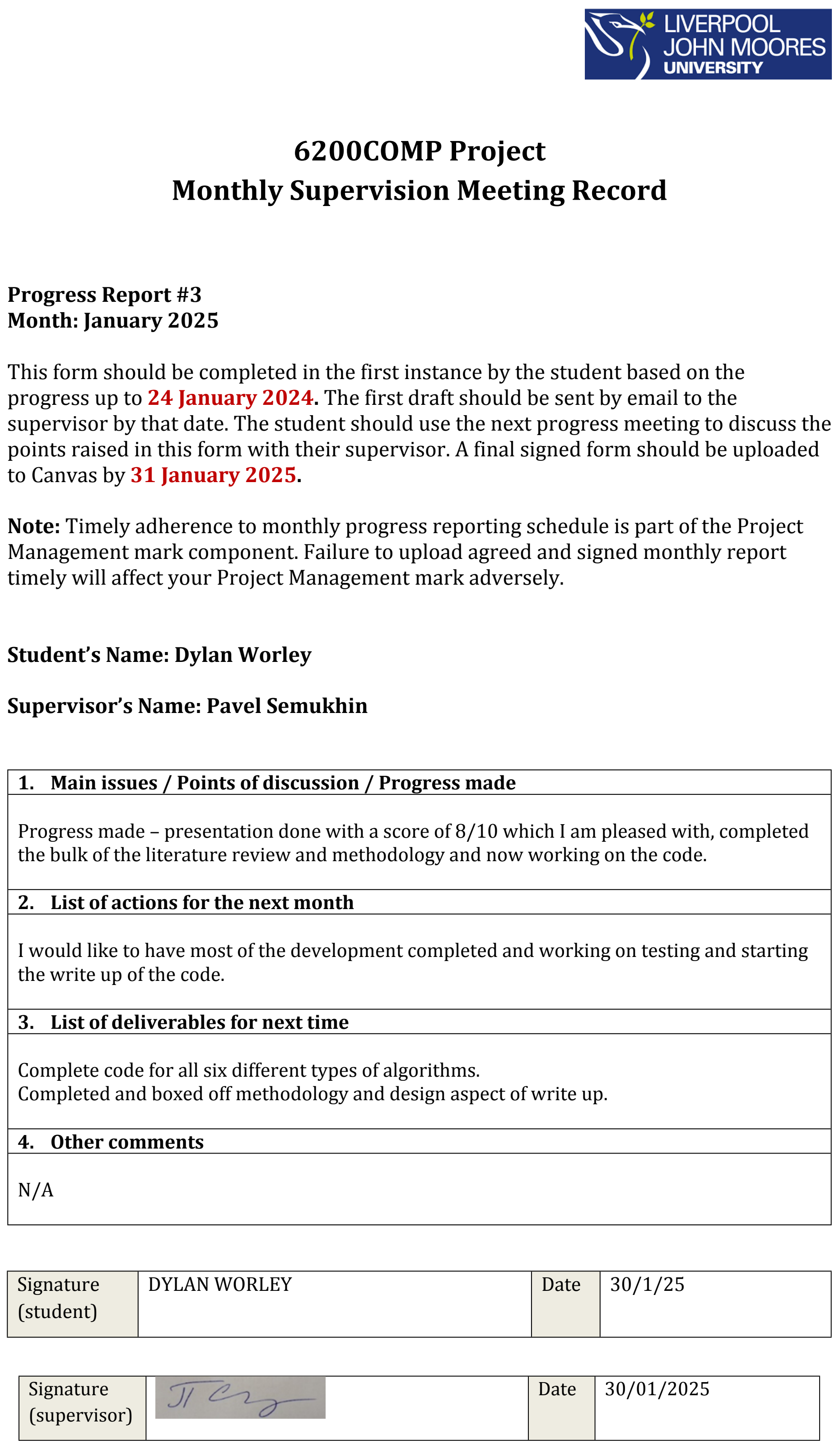
1 attachment (844 bytes)
SeeMyTutorBooking.ics;

This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 16/01/2025 at 14:00, for a duration of 30 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ1...NGRlMi1hZmUzLThlNzZhNTZhYWM0ZgAQABYYIqk7V9tDqWt7SBYpW6c%3D Page 1 of 1

Figure 89: January 2025 Monthly Supervisor Meeting



6200COMP Project
Monthly Supervision Meeting Record

Progress Report #3
Month: January 2025

This form should be completed in the first instance by the student based on the progress up to **24 January 2024**. The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by **31 January 2025**.

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Dylan Worley

Supervisor's Name: Pavel Semukhin

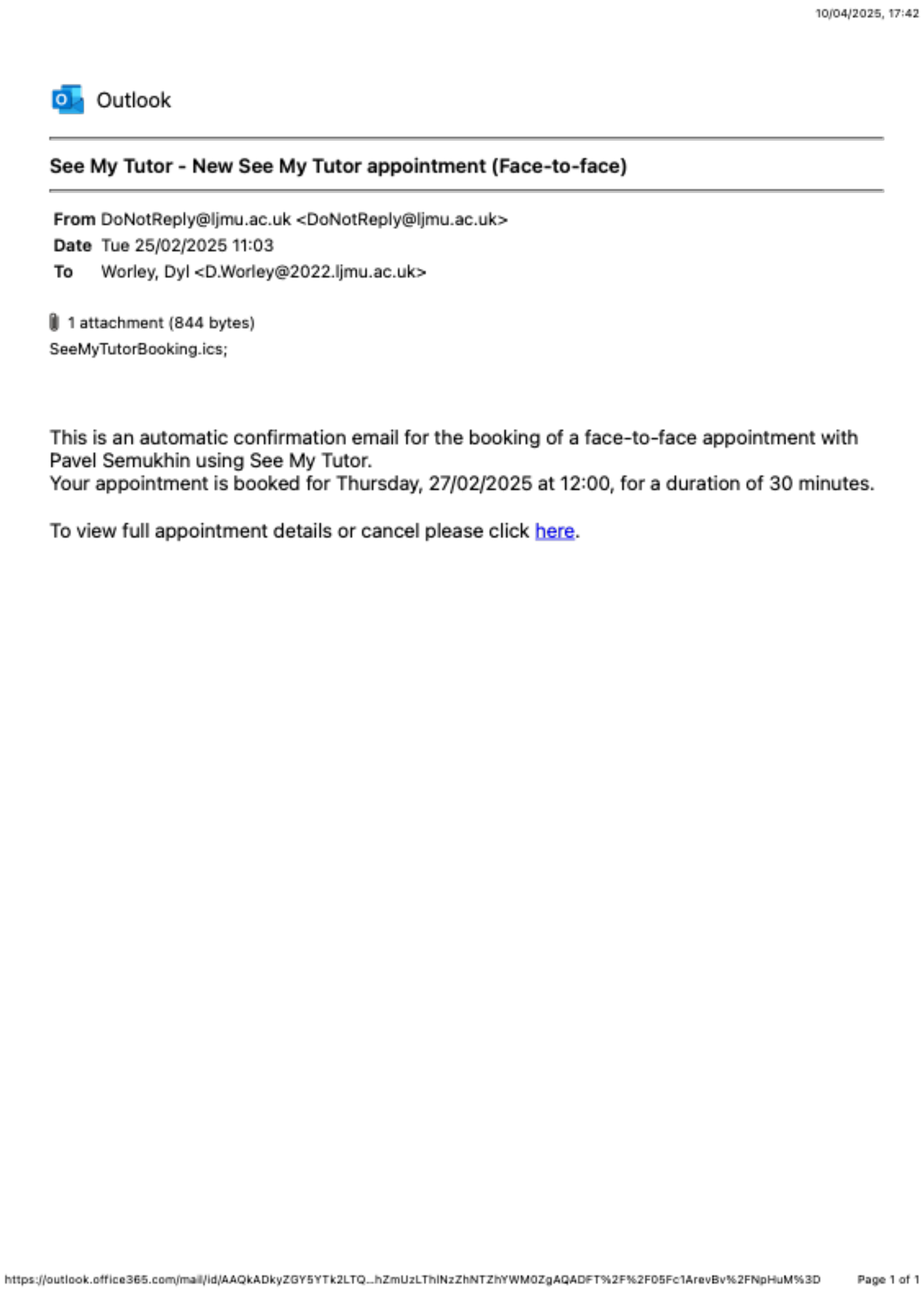
1. Main issues / Points of discussion / Progress made
Progress made – presentation done with a score of 8/10 which I am pleased with, completed the bulk of the literature review and methodology and now working on the code.
2. List of actions for the next month
I would like to have most of the development completed and working on testing and starting the write up of the code.
3. List of deliverables for next time
Complete code for all six different types of algorithms. Completed and boxed off methodology and design aspect of write up.
4. Other comments
N/A

Signature (student)	DYLAN WORLEY	Date	30/1/25

Signature (supervisor)	[illegible]	Date	30/01/2025

Figure 90: January 2025 Monthly Meeting Notes

February 2025



10/04/2025, 17:42

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Tue 25/02/2025 11:03
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

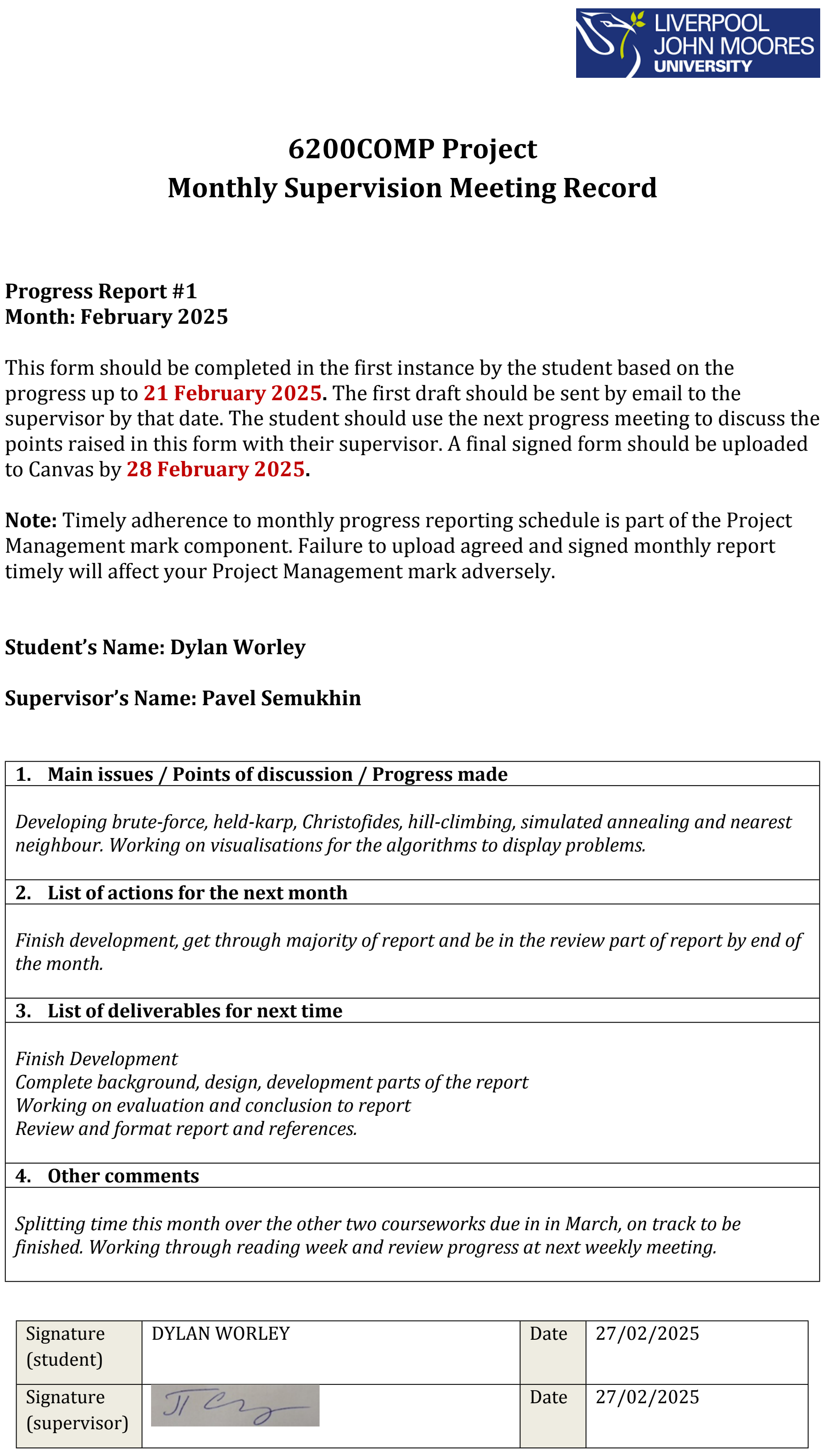
1 attachment (844 bytes)
SeeMyTutorBooking.ics;

This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 27/02/2025 at 12:00, for a duration of 30 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ...hZmUzLThlNzZhNTZhYWM0ZgAQADFT%2F%2F05Fc1ArevBv%2FNpHuM%3D Page 1 of 1

Figure 91: February 2025 Monthly Supervisor Meeting



6200COMP Project
Monthly Supervision Meeting Record

Progress Report #1
Month: February 2025

This form should be completed in the first instance by the student based on the progress up to **21 February 2025.** The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by **28 February 2025.**

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

Student's Name: Dylan Worley

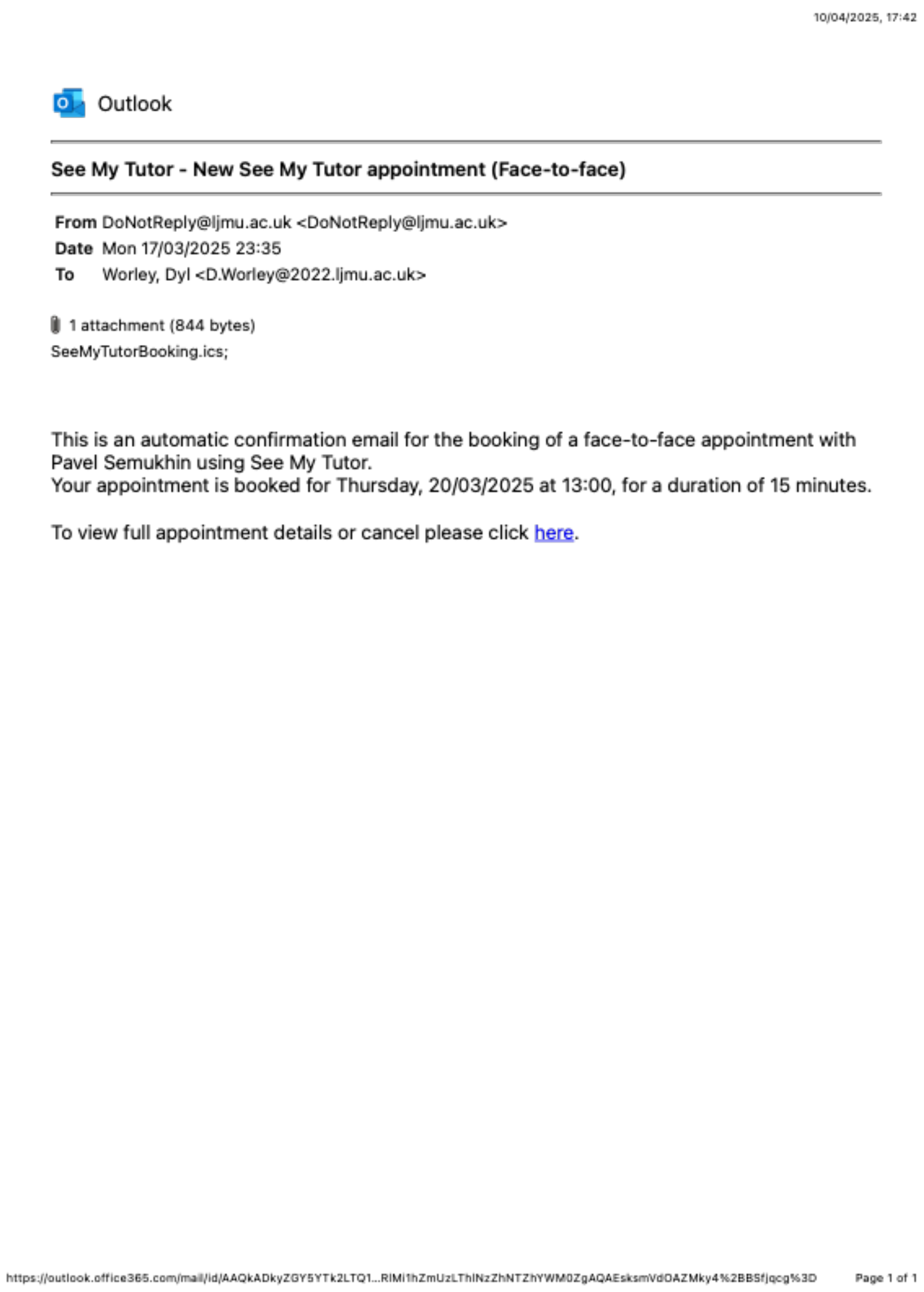
Supervisor's Name: Pavel Semukhin

1. Main issues / Points of discussion / Progress made
Developing brute-force, held-karp, Christofides, hill-climbing, simulated annealing and nearest neighbour. Working on visualisations for the algorithms to display problems.
2. List of actions for the next month
Finish development, get through majority of report and be in the review part of report by end of the month.
3. List of deliverables for next time
Finish Development *Complete background, design, development parts of the report* *Working on evaluation and conclusion to report* *Review and format report and references.*
4. Other comments
Splitting time this month over the other two courseworks due in in March, on track to be finished. Working through reading week and review progress at next weekly meeting.

Signature (student)	DYLAN WORLEY	Date	27/02/2025
Signature (supervisor)	[illegible]	Date	27/02/2025

Figure 92: February 2025 Monthly Meeting Notes

March 2025



10/04/2025, 17:42

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Mon 17/03/2025 23:35
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

1 attachment (844 bytes)
SeeMyTutorBooking.ics;

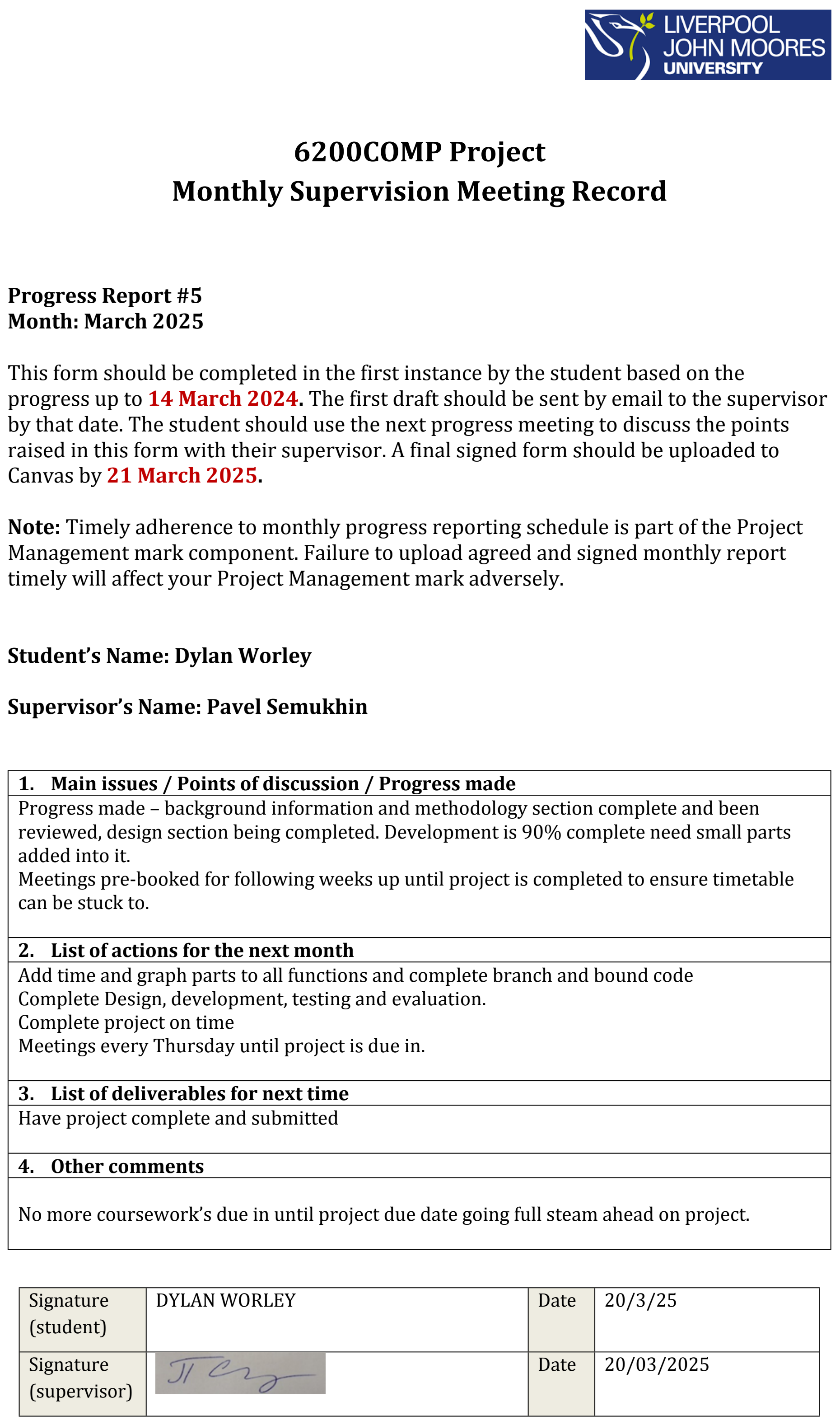
This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 20/03/2025 at 13:00, for a duration of 15 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ1...RlMi1hZmUzLThlNzZhNTZhYWM0ZgAQAEsksmVdOAZMky4%2BBSfjqcg%3D Page 1 of 1

Figure 93: March 2025 First Meeting

This is the confirmation of the first supervisor meeting in March, due to being close to the deadline the number of meetings increased to ensure work was on track.



6200COMP Project
Monthly Supervision Meeting Record

Progress Report #5
Month: March 2025

This form should be completed in the first instance by the student based on the progress up to **14 March 2024.** The first draft should be sent by email to the supervisor by that date. The student should use the next progress meeting to discuss the points raised in this form with their supervisor. A final signed form should be uploaded to Canvas by **21 March 2025.**

Note: Timely adherence to monthly progress reporting schedule is part of the Project Management mark component. Failure to upload agreed and signed monthly report timely will affect your Project Management mark adversely.

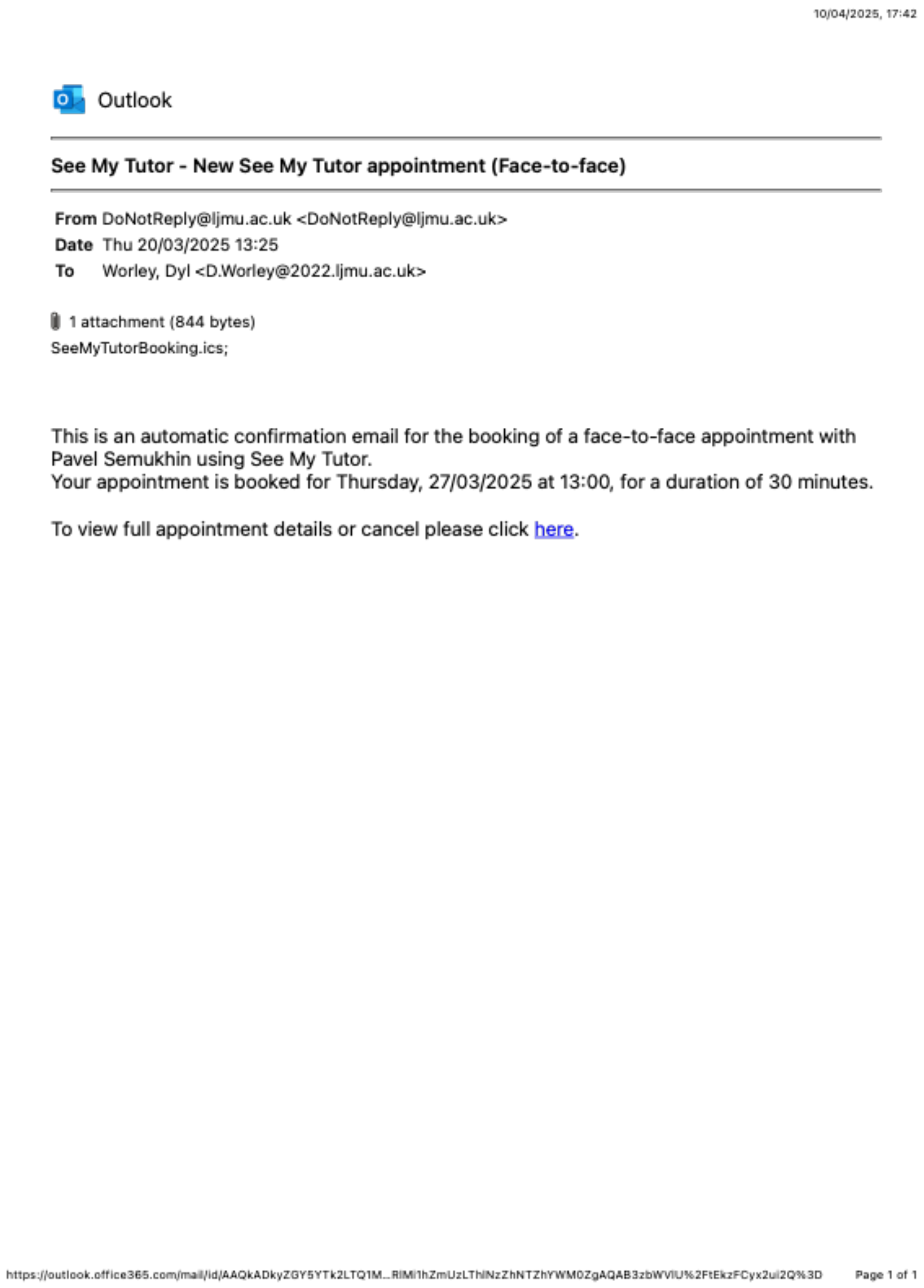
Student's Name: Dylan Worley

Supervisor's Name: Pavel Semukhin

1. Main issues / Points of discussion / Progress made
Progress made – background information and methodology section complete and been reviewed, design section being completed. Development is 90% complete need small parts added into it. Meetings pre-booked for following weeks up until project is completed to ensure timetable can be stuck to.
2. List of actions for the next month
Add time and graph parts to all functions and complete branch and bound code Complete Design, development, testing and evaluation. Complete project on time Meetings every Thursday until project is due in.
3. List of deliverables for next time
Have project complete and submitted
4. Other comments
No more coursework's due in until project due date going full steam ahead on project.

Signature (student)	DYLAN WORLEY	Date	20/3/25
Signature (supervisor)	[illegible]	Date	20/03/2025

Figure 94: March 2025 Monthly Meeting Notes


10/04/2025, 17:42

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Thu 20/03/2025 13:25
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

1 attachment (844 bytes)
SeeMyTutorBooking.ics;

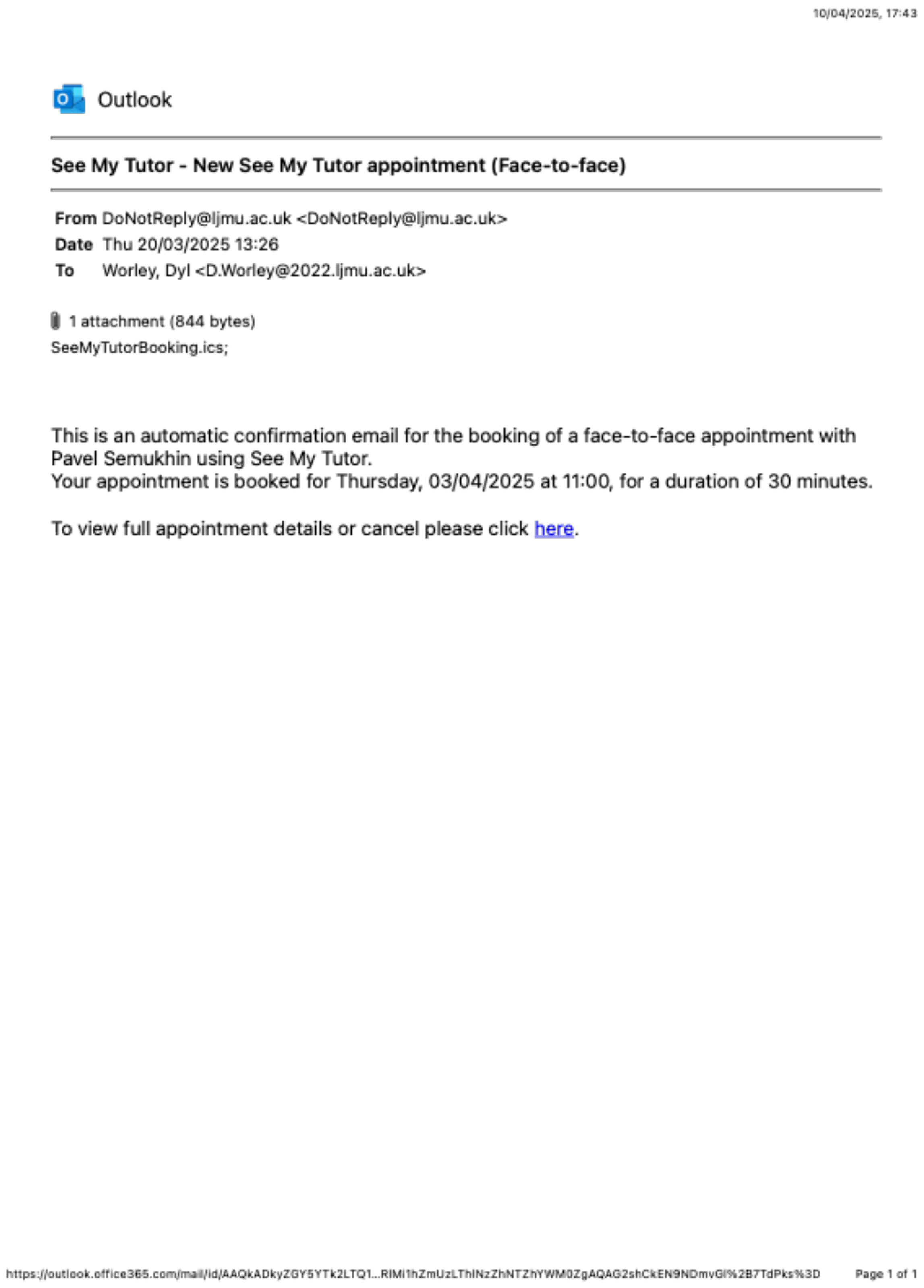
This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 27/03/2025 at 13:00, for a duration of 30 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ1M...RlMi1hZmUzLThlNzZhNTZhYWM0ZgAQAB3zbWVlU%2FtEkzFCyx2ui2Q%3D Page 1 of 1

Figure 95: March 2025 Second Supervisor Meeting

April 2025



10/04/2025, 17:43

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Thu 20/03/2025 13:26
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

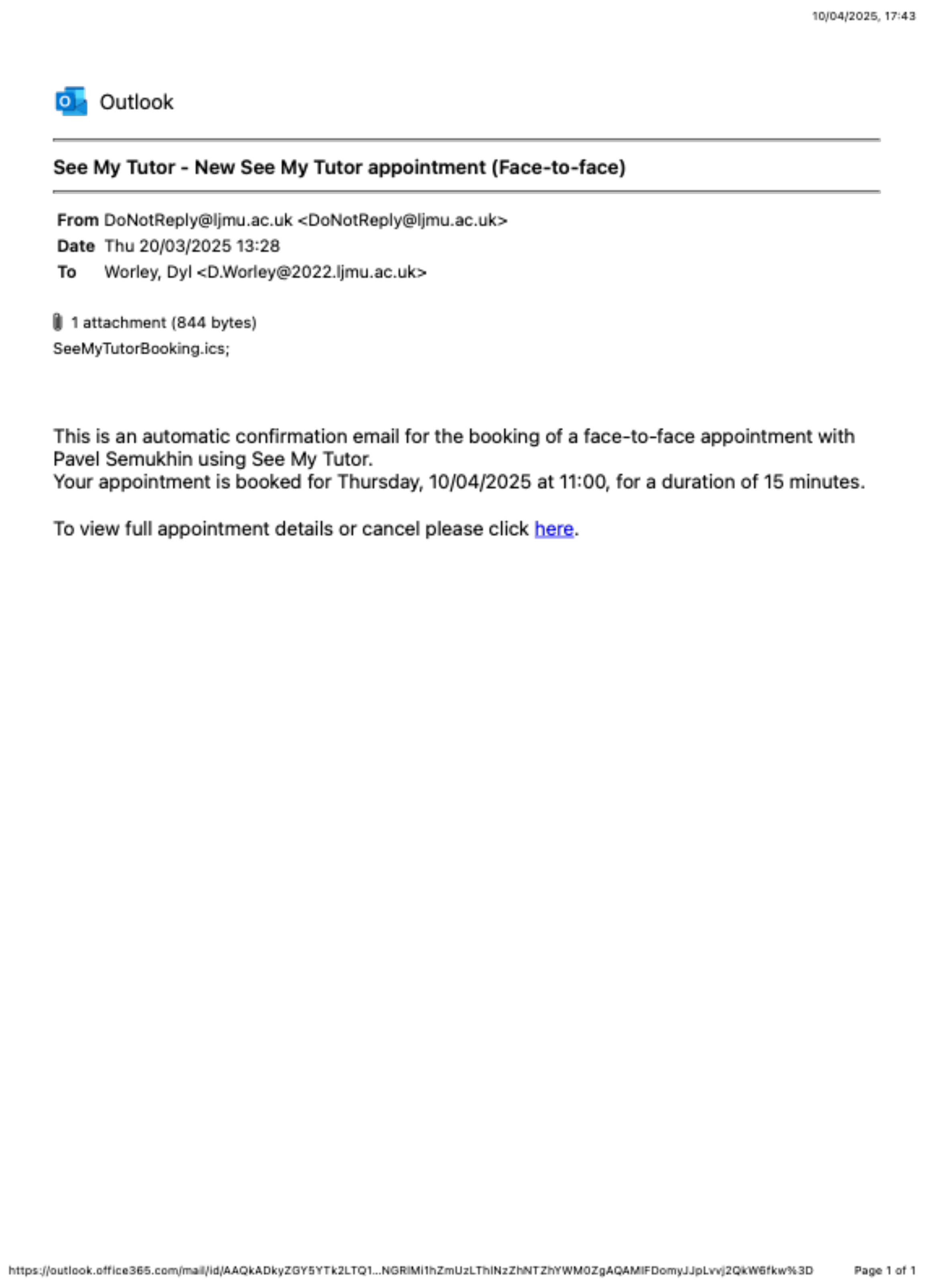
1 attachment (844 bytes)
SeeMyTutorBooking.ics;

This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 03/04/2025 at 11:00, for a duration of 30 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ1...RlMi1hZmUzLThlNzZhNTZhYWM0ZgAQAG2shCkEN9NDmvGl%2B7TdPks%3D Page 1 of 1

Figure 96: April 2025 First Supervisor Meeting



10/04/2025, 17:43

Outlook

See My Tutor - New See My Tutor appointment (Face-to-face)

From DoNotReply@ljmu.ac.uk <DoNotReply@ljmu.ac.uk>
Date Thu 20/03/2025 13:28
To Worley, Dyl <D.Worley@2022.ljmu.ac.uk>

1 attachment (844 bytes)
SeeMyTutorBooking.ics;

This is an automatic confirmation email for the booking of a face-to-face appointment with Pavel Semukhin using See My Tutor.
Your appointment is booked for Thursday, 10/04/2025 at 11:00, for a duration of 15 minutes.

To view full appointment details or cancel please click here.

https://outlook.office365.com/mail/id/AAQkADkyZGY5YTk2LTQ1...NGRlMi1hZmUzLThlNzZhNTZhYWM0ZgAQAMIFDomyJJpLvvj2QkW6fkw%3D Page 1 of 1

Figure 97: April 2025 Second Supervisor Meeting

This is the last supervisor meeting as it was on the 10th of April one day before the project deadline.

Full Code

Here is the full code for every algorithm to assist with viewing the development section.

Brute Force

```
import tsplib95
import numpy as np
import random
import itertools
import matplotlib.pyplot as plt
import time

# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem

# Function to calculate the cost of the route
def calculate_cost(route, graph):
    total_cost = 0
    num_cities = len(route)
    for i in range(num_cities):
        current_city = route[i]
        next_city = route[(i + 1) % num_cities]  # Wrap around to the start of the
route
        total_cost += graph[current_city][next_city]
    return total_cost

def brute_force(cities, graph):
    begin_time = time.time() #start the timer

    # Generate all permutations of the cities and initialsie variables
    all_permutations = itertools.permutations(cities) # a permutation
    min_cost = float('inf') #inf used to
    optimal_route = None

    # Iterate over all permutations and calculate costs
    for perm in all_permutations:
        cost = calculate_cost(perm, graph)
        if cost < min_cost:
            min_cost = cost
            optimal_route = perm
    end_time = time.time() #end the timer
    return optimal_route, min_cost, begin_time, end_time


# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords

    # Plot the route
    route_coords = [city_coords[city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])

    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("Brute Force Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()

# Main code
if __name__ == "__main__":

    filename = "./tsplib-master/burma14.tsp"  # Replace with your TSP file path
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance, begin_time, end_time = brute_force(cities, graph)
    print("Best Route: " , best_route)
    print("Number of Cities: ", len(best_route))
    print("Total Cost: ", best_distance)
    print("Execution Time: ", (round(end_time - begin_time)), "seconds")
    # Plot the best route
    plot_route(cities, best_route, problem)
```

Figure 98: Brute Force Full Code

Held – Karp

```
def tsp_dynamic_programming(filename):
    begin_time = time.time() #start the timer
    problem, cities, distances = load_tsp_file(filename)

    #Step 1: Initialise the DP table
    n = len(distances)
    dp = [[math.inf] * n for _ in range(1 << n)]
    parent = [[None] * n for _ in range(1 << n)]

    dp[1][0] = 0

    #Step 2: Fill the DP table
    for mask in range(1 << n):
        for last_visited in range(n):
            if not (mask & (1 << last_visited)):
                continue
            for next in range(n):
                if mask & (1 << next):
                    continue
                new_mask = mask | (1 << next)
                new_dist = dp[mask][last_visited] + distances[last_visited][next]
                if new_dist < dp[new_mask][next]:
                    dp[new_mask][next] = new_dist
                    parent[new_mask][next] = last_visited

    #Step 3: Find the minimum cost and reconstruct the path
    min_cost = math.inf
    end_city = None
    full_mask = (1 << n) - 1
    for last_visited in range(1, n):
        cost = dp[full_mask][last_visited] + distances[last_visited][0]
        if cost < min_cost:
            min_cost = cost
            end_city = last_visited

    #Step 4: Reconstruct the optimal path
    tour = []
    mask = full_mask
    last_visited = end_city
    while last_visited is not None:
        tour.append(cities[last_visited])
        new_last_visited = parent[mask][last_visited]
        mask ^= (1 << last_visited)
        last_visited = new_last_visited

    # Step 5: Reverse the path to get the correct order
    tour = tour[::-1]
    tour.append(cities[0])

    # Step 6: Return the result and end timer
    end_time = time.time() #end the timer
    return problem, cities, tour, min_cost, begin_time, end_time

# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    city_coords = problem.node_coords

    route_coords = [city_coords[city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    plt.figure(figsize=(8, 6))
    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")

    plt.title("Held Karp Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()

# Main execution
if __name__ == "__main__":
    filename = "./tsplib-master/ulysses16.tsp"
    problem,    cities,    best_path,    min_cost,    begin_time,    end_time    =
tsp_dynamic_programming(filename)
    print("Best path:", best_path)
    print("Number of cities:", len(cities))
    print("Total cost:", min_cost)
    print("Execution time:", (end_time - begin_time), "seconds")
    plot_route(cities, best_path, problem)
```

Figure 99: Held-Karp Full Code

Branch and Bound

```
import numpy as np
import tsplib95
import matplotlib.pyplot as plt
import time


# Load TSPLIB file using tsplib95
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    adj_matrix = np.array([[problem.get_weight(i, j) for j in cities] for i in
cities]) # an adjacency matrix is a dis
    return adj_matrix, cities, problem

# Branch and Bound Functions
def first_min(adj, i): #finds the first minimum edge distance from i to any other
node
    min_val = np.inf
    for k in range(len(adj)):
        if adj[i][k] < min_val and i != k:
            min_val = adj[i][k]
    return min_val

def second_min(adj, i): #finds the second minimum edge distance from i to any other
node
    first, second = np.inf, np.inf
    for j in range(len(adj)):
        if i == j:
            continue
        if adj[i][j] <= first:
            second = first
            first = adj[i][j]
        elif adj[i][j] <= second:
            second = adj[i][j]
    return second

def  tsp_rec(adj,  current_bound,  current_weight,  level,  current_path,  visited,
final_res, final_path): #recursive function to solve by pruning branches
    N = len(adj)

    if level == N:
        if adj[current_path[level - 1]][current_path[0]] != 0:
            current_res    =    current_weight    +    adj[current_path[level    -
1]][current_path[0]]
            if current_res < final_res[0]:
                final_path[:N + 1] = current_path[:]
                final_path[N] = current_path[0]
                final_res[0] = current_res
        return

    for i in range(N):
        if adj[current_path[level-1]][i] != 0 and not visited[i]:
            temp = current_bound
            current_weight += adj[current_path[level - 1]][i]

            if level == 1:
                current_bound  -=  (first_min(adj,  current_path[level  -  1])  +
first_min(adj, i)) / 2
            else:
                current_bound  -=  (second_min(adj,  current_path[level  -  1])  +
first_min(adj, i)) / 2

            if current_bound + current_weight < final_res[0]:
                current_path[level] = i
                visited[i] = True

                tsp_rec(adj,   current_bound,   current_weight,   level   +   1,
current_path, visited, final_res, final_path)

            current_weight -= adj[current_path[level - 1]][i]
            current_bound = temp

            visited = [False] * N
            for j in range(level):
                if current_path[j] != -1:
                    visited[current_path[j]] = True

# Main TSP solver using Branch and Bound
def solve_tsp_branch_bound(adj):
    begin_time = time.time() #start the timer
    N = len(adj)
    current_bound = 0
    current_path = [-1] * (N + 1)
    visited = [False] * N

    final_res = [np.inf]
    final_path = [-1] * (N + 1)

    for i in range(N):
        current_bound += (first_min(adj, i) + second_min(adj, i))

    current_bound = np.ceil(current_bound / 2)

    visited[0] = True
    current_path[0] = 0

    tsp_rec(adj,  current_bound,  0,  1,  current_path,  visited,  final_res,
final_path)
    end_time = time.time() #end the timer
    return final_res[0], final_path, begin_time, end_time

# Plotting the solution using matplotlib
# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords

    # Plot the route
    route_coords = [city_coords[cities[city]] for city in route]

    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])

    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("Branch and Bound Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()

# Example usage
if __name__ == "__main__":
    filename = "./tsplib-master/ulysses16.tsp"  # Replace with your TSP file path
    adj_matrix, cities, problem = load_tsp_file(filename)
    best_cost,       best_route,       begin_time,       end_time       =
solve_tsp_branch_bound(adj_matrix)
    print("Total Cost:", best_cost)
    print("Number of Cities:", len(best_route))
    print("Optimal Path:", [cities[i] for i in best_route])
    print("Execution Time:", end_time - begin_time)
    plot_route(cities, best_route, problem)
```

Figure 100: Branch and Bound Full Code

Nearest Neighbour

```
import tsplib95
import itertools
import matplotlib.pyplot as plt
import time

def load_tsp_file(filename): #function to define file name
    problem = tsplib95.load(filename) #loads file into problem variable
    cities = list(problem.get_nodes()) #gets all the nodes storing thme in the
cities variables
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem #returns variables for cities and graph

def tsp_nearest_neighbour(graph, cities): #defines function for nearest neighbour
algorithm
    begin_time = time.time() #start time
    num_cities = len(cities) #gets the number of cities from the tsp file and then
stores it in variable
    start_city = cities[0]
    unvisited_cities = set(cities)
    unvisited_cities.remove(start_city)

    current_city = start_city #starts at start city
    path = [start_city]
    total_cost = 0

    while unvisited_cities: #while there are unvisited cities
        nearest_city      =      min(unvisited_cities,      key=lambda      city:
graph[current_city][city])
 #creates a temporary variable called lambda and sets it to the nearest city
        total_cost += graph[current_city][nearest_city]
        path.append(nearest_city)
        unvisited_cities.remove(nearest_city)
        current_city = nearest_city

    # Return to the starting city
    total_cost += graph[current_city][start_city]
    path.append(start_city)
    end_time = time.time() #end time
    return path, total_cost, begin_time, end_time


# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords


    # Plot the route
    route_coords = [city_coords [city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])

    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("Nearest Neighbour Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()


if __name__ == "__main__": #main function
    filename = "./tsplib-master/att8.tsp"
    cities, graph, problem = load_tsp_file(filename)

    best_path,  min_cost,  begin_time,  end_time   =  tsp_nearest_neighbour(graph,
cities)
    print("Best path: ", best_path)
    print("Cities Visited: ", len(best_path))
    print("Minimum cost: ", min_cost)
    print("Execution Time: ", (end_time - begin_time))
    # Plot the best route
    plot_route(cities, best_path, problem)
```

Figure 101: Nearest Neighbour Full Code

Christofides

```
import tsplib95
import networkx as nx
import matplotlib.pyplot as plt
import time

# Load TSPLIB file
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem, G
```

```
# Cost calculation
def calculate_cost(route, graph):
    total_cost = 0
    for i in range(len(route)):
        current = route[i]
        next_city = route[(i + 1) % len(route)]  # wrap around
        total_cost += graph[current][next_city]
    return total_cost

# Christofides Algorithm
def christofides_tsp(G):
    begin_time = time.time()
    # Step 1: Minimum Spanning Tree
    mst = nx.minimum_spanning_tree(G)

    # Step 2: Find odd degree nodes
    odd_nodes = [v for v, d in mst.degree() if d % 2 == 1]

    # Step 3: Minimum Weight Perfect Matching among odd degree nodes
    subgraph = G.subgraph(odd_nodes)
    matching        =        nx.algorithms.matching.min_weight_matching(subgraph,
maxcardinality=True)

    # Step 4: Combine MST and Matching
    eulerian_graph = nx.MultiGraph(mst)
    eulerian_graph.add_edges_from(matching)

    # Step 5: Find Eulerian Circuit
    euler_circuit = list(nx.eulerian_circuit(eulerian_graph))

    # Step 6: Shortcutting to TSP Tour
    tour = []
    visited = set()
    for u, v in euler_circuit:
        if u not in visited:
            tour.append(u)
            visited.add(u)
    tour.append(tour[0])
    end_time = time.time()
    return tour, begin_time, end_time

# Plotting function
def plot_route(route, problem):
    coords = problem.node_coords
    route_coords = [coords[city] for city in route]
    x_vals = [x for x, y in route_coords]
    y_vals = [y for x, y in route_coords]

    x_vals.append(x_vals[0])
    y_vals.append(y_vals[0])
    plt.plot(x_vals, y_vals, 'r-', marker='o')
    plt.title("Christofides TSP Route")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.show()

# Main code
if __name__ == "__main__":
    filename = "./tsplib-master/nrw1379.tsp"  # Change path accordingly
    cities, graph, problem, G = load_tsp_file(filename)
    route, begin_time, end_time = christofides_tsp(G)
    cost = calculate_cost(route, graph)
    print("Best Route:", route)
    print("Number of Cities:", len(set(route)))
    print("Total Cost:", cost)
    print("Execution Time:", round(end_time - begin_time), "seconds")
    plot_route(route, problem)
```

Figure 102: Christofides Full Code

Hill Climbing

```
import numpy as np
import random
import tsplib95
import itertools
import matplotlib.pyplot as plt
import time



# Set seed for reproducibility
np.random.seed(42)
random.seed(42)


# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem

def calculate_cost(route, graph):
    total_cost = 0
    n = len(route)
    for i in range(n):
        current_city = route[i]
        next_city = route[(i + 1) % n]  # Wrap around to the start of the route
        total_cost += graph[current_city][next_city]
    return total_cost


# Function to create a random initial route
def create_initial_route(cities):
    return random.sample(list(cities), len(cities))

# Function to create neighboring solutions
def get_neighbors(route):
    neighbors = []
    for i in range(len(route)):
        for j in range(i + 1, len(route)):
            neighbor = route.copy()
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

#hill climbing function
def hill_climbing(cities, graph):
    begin_time = time.time()
    current_route = create_initial_route(cities)
    current_distance = calculate_cost(current_route, graph)

    while True:
        neighbors = get_neighbors(current_route)
        next_route = min(neighbors, key= lambda x: calculate_cost(x,graph)) # lamb
        next_distance = calculate_cost(next_route, graph)

        if next_distance >= current_distance:
            break

        current_route, current_distance = next_route, next_distance
    end_time = time.time()
    return current_route, current_distance, begin_time, end_time



# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords

    # Plot the route
    route_coords = [city_coords[city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])
    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("Hill Climbing Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()


# Main Code
if __name__ == "__main__":
    filename = "./tsplib-master/gr202.tsp"  # Replace with your TSP file path
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance, begin_time, end_time = hill_climbing(cities, graph)
    print("Best Route: ", best_route)
    print("Number of Cities:", len(set(route)))
    print("Total Cost: ", best_distance)
    print("Execution Time: ", ((end_time - begin_time)))
    # Plot the best route
    plot_route(cities, best_route, problem)
```

Figure 103: Hill Climbing Full Code

Simulated Annealing

```
import numpy as np
import random
import tsplib95
import itertools
import matplotlib.pyplot as plt
import time


# Set seed for reproducibility
np.random.seed(42)
random.seed(42)


# Function to load a TSPLIB file and extract cities and distances
def load_tsp_file(filename):
    problem = tsplib95.load(filename)
    cities = list(problem.get_nodes())
    graph = {i: {j: problem.get_weight(i, j) for j in cities} for i in cities}
    G = problem.get_graph()
    return cities, graph, problem

def calculate_cost(route, graph):
    total_cost = 0
    n = len(route)
    for i in range(n):
        current_city = route[i]
        next_city = route[(i + 1) % n]  # Wrap around to the start of the route
        total_cost += graph[current_city][next_city]
    return total_cost


# Function to create a random initial route
def create_initial_route(cities):
    return random.sample(list(cities), len(cities))

# Function to create neighboring solutions
def get_neighbors(route):
    neighbors = []
    for i in range(len(route)):
        for j in range(i + 1, len(route)):
            neighbor = route.copy()
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def     simulated_annealing(cities,     graph,     initial_temp,     cooling_rate,
max_iterations):
    begin_time = time.time()
    current_route = create_initial_route(cities)
    current_distance = calculate_cost(current_route, graph)
    best_route = current_route.copy()
    best_distance = current_distance
    temperature = initial_temp

    for _ in range(max_iterations):
        neighbors = get_neighbors(current_route)
        next_route = random.choice(neighbors)
        next_distance = calculate_cost(next_route, graph)

        if    next_distance    <    current_distance    or    random.random()    <
np.exp((current_distance - next_distance) / temperature):
            current_route, current_distance = next_route, next_distance

            if current_distance < best_distance:
                best_route, best_distance = current_route, current_distance

        temperature *= cooling_rate
    end_time = time.time()
    return best_route, best_distance, begin_time, end_time


# Function to plot the route using matplotlib
def plot_route(cities, route, problem):
    # Get the coordinates of the cities from the problem
    city_coords = problem.node_coords

    # Plot cities as red dots
    #plt.figure(figsize=(10, 8))
    #for city, (x, y) in city_coords.items():
        #plt.scatter(x, y, color='red', zorder=5)
        #plt.text(x + 20, y + 20, str(city), fontsize=12, color='black')

    # Plot the route
    route_coords = [city_coords[city] for city in route]
    route_x = [x for x, y in route_coords]
    route_y = [y for x, y in route_coords]

    # Add the return to the starting city to complete the loop
    route_x.append(route_x[0])
    route_y.append(route_y[0])

    plt.plot(route_x, route_y, 'r-', marker='o', markersize=5, label="Route")
    plt.title("Simulated Annealing Visualisation")
    plt.xlabel("X-coordinate")
    plt.ylabel("Y-coordinate")
    plt.grid(False)
    plt.legend()
    plt.show()

# Parameters for Simulated Annealing
initial_temp = 500
cooling_rate = 0.95
max_iterations = 200

# Set name of file
if __name__ == "__main__":
    filename = "./tsplib-master/ali535.tsp"  # Replace with your TSP file path
    cities, graph, problem = load_tsp_file(filename)
    best_route, best_distance, begin_time, end_time = simulated_annealing(cities,
graph, initial_temp, cooling_rate, max_iterations)
    print("Best Route: " , best_route)
    print("Number of Cities:", len(set(route)))
    print("Total Cost: ", best_distance)
    print("Execution Time: ", ((end_time - begin_time)))
    plot_route(cities, best_route, problem)
```

Figure 104: Simulated Annealing Full Code