

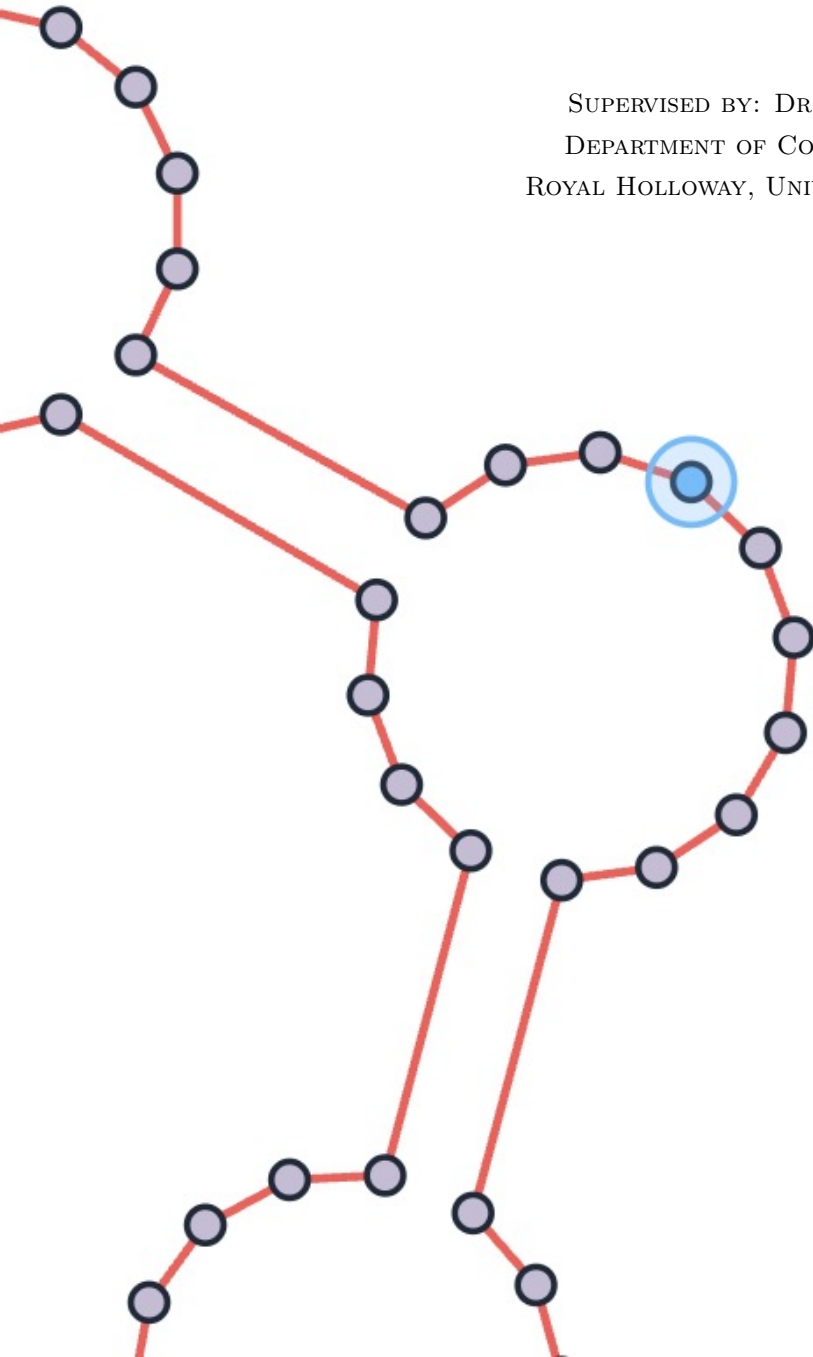
ANALYSING REAL-WORLD INSTANCES OF TRAVELLING SALESPERSON PROBLEM HEURISTICS

FINAL REPORT

DANIEL JUDD

CS3821 - BSc FINAL YEAR PROJECT

SUPERVISED BY: DR ALED WILLIAMS
DEPARTMENT OF COMPUTER SCIENCE
ROYAL HOLLOWAY, UNIVERSITY OF LONDON



Abstract

The Travelling Salesperson Problem (TSP) is one of the most famous problems in computational mathematics, with applications in vehicular routing, manufacturing efficiency, and many other industries. This report provides a comparative analysis of construction and improvement heuristics, evaluated against real-world problem instances.

We examine the performance of greedy construction methods, including Nearest Neighbour and Multi-fragment, alongside the incremental approach of insertion heuristics. Furthermore, the Christofides-Serdyukov algorithm is evaluated as a bounded approximation method. To assess the impact of local search and metaheuristics, 2-Opt and Simulated Annealing are applied as improvement heuristics.

Our results demonstrate that while greedy heuristics offer superior computational speed, local improvement methods consistently reduce tour lengths by approximately 9.6%, while Simulated Annealing has the potential to produce the best results on complex instances at the cost of higher execution time. This report highlights the critical trade-off between solution quality and computational overhead in heuristic selection.

Contents

1	Introduction	2
1.1	The Travelling Salesperson Problem	2
1.2	Aim and Scope	2
2	Background and Literature Review	3
2.1	Practical Applications	3
2.2	Computational Complexity	4
2.3	Heuristic Algorithms	4
2.4	Evaluation Techniques	4
3	Software Development	10
3.1	Requirement Analysis	10
3.2	Design Decisions	11
3.3	Version Control System	11
3.4	Diagrams	12
4	Greedy Heuristics	13
4.1	Nearest Neighbour Heuristic	13
4.2	Repeated Nearest Neighbour Heuristic	17
4.3	Multi-fragment Heuristic	22
5	Vertex Insertion Heuristics	27
5.1	Base Algorithm	27
5.2	Nearest Insertion	28
5.3	Cheapest Insertion	32
5.4	Farthest Insertion	36
6	Approximation Heuristics	41
6.1	Christofides-Serdyukov Heuristic	41
7	Improvement Heuristics	46
7.1	2-Opt Heuristic	46
7.2	Simulated Annealing Heuristic	49
8	Comparative Analysis	53
8.1	Construction Heuristics	53
8.2	Improvement Heuristics	55
8.3	Overall Comparison	56
8.4	Conclusion	57
A	Additional Techniques	58
A.1	Randomised Heuristic	58
B	Diary Entries	60
B.1	Term 1	60
B.2	Term 2	62
C	Full Results Tables	66

1 Introduction

1.1 The Travelling Salesperson Problem

The Travelling Salesperson Problem (TSP) is one of the most famous NP-hard problems in computational mathematics. The TSP is defined as follows: a travelling salesperson must visit n cities exactly once before returning to the initial city. This route is called a *tour*. In graph-theoretic terms, a tour corresponds to a Hamiltonian cycle on a complete weighted graph. Let c_{ij} denote the distance between city i and city j , for all $1 \leq i, j \leq n$.

Throughout this report, we consider the *metric* TSP, where distances are symmetric and satisfy the triangle inequality. Therefore, $c_{ij} = c_{ji}$, and for any three nodes i , j and k , the direct distance c_{ij} must not be greater than travelling indirectly via k , or:

$$c_{ij} \leq c_{ik} + c_{kj} \quad (1.1)$$

When referring to a city in this report, we use *city* and *node* interchangeably. The goal of the TSP is to determine the order in which all nodes should be visited to minimise the total length of the tour.

1.2 Aim and Scope

This report analyses a range of heuristic approaches for solving real-world instances of the metric TSP. These heuristics include greedy construction heuristics, vertex insertion heuristics, approximation heuristics, and improvement heuristics. Each heuristic is defined and evaluated in terms of computational efficiency, tour quality, and behaviour on both synthetic and real-world instances. The report also presents a software implementation developed to visualise, compare, and experimentally analyse these heuristics.

2 Background and Literature Review

2.1 Practical Applications

There are many practical real-world applications of the TSP. The classical definition of the TSP assumes just one traveller visiting all cities alone. However, real-world use cases often include constraints. These constraints could relate to the traveller, the cities, or the goal of the problem itself. Below we describe a few variations on the classical TSP and their applications in industry.

2.1.1 Vehicle Routing Problem

The most direct variation of the TSP is the Vehicle Routing Problem (VRP) (Toth and Vigo, 2014). Unlike the classical TSP, which assumes a single traveller completing one tour, VRP considers multiple vehicles departing from a common depot node. Each vehicle must visit a designated subset of customer nodes, and the objective is to optimise the collection of delivery routes rather than a single tour.

VRP is widely applicable in real-world logistics, particularly for delivery and freight companies. In these settings, reducing the total tour length can lower fuel consumption, travel times, and operating costs. For example, companies such as UPS and Amazon rely heavily on route optimisation to improve efficiency across large delivery fleets. Amazon also launched a routing research challenge in 2021 to find the best route optimisation algorithms, where there was \$175,000 in prize money, highlighting the practical and commercial importance of this problem (Merchan et al., 2022). Although this project focuses on the classical metric TSP, VRP demonstrates how the same core routing ideas extend to more realistic operational settings involving multiple tours, vehicles and a central depot.

2.1.2 TSP with Time Windows

Another variant on the key problem is TSP with Time Windows (TSPTW) which constrains travellers to visit cities within predefined time windows (López-Ibáñez et al., 2013). The objective is to minimise the sum of the travel times while adhering to these constraints. For every city i , we define a time window $[a_i, b_i]$, where a_i is the earliest arrival time and b_i is the latest service time. If the vehicle arrives before a_i , it must wait, increasing the total travel time. Arriving after b_i is often considered infeasible. Most implementations include a service time s_i , the time required to complete the task at each stop.

TSPTW has applications in many scheduling and logistical operations, such as bus routing, field service scheduling or production line sequencing. In these settings, a short tour alone is not sufficient as visits must also occur within permitted service intervals. Although this report does not consider temporal constraints, TSPTW shows how real-world routing problems often need to consider feasibility as well as short tour length.

2.1.3 The Pickup and Delivery Problem

The Pickup and Delivery Problem (PDP) is a variant on VRP in which a fleet of capacitated vehicles must be routed to satisfy a set of transport requests (Wolfinger and Salazar-González, 2021). Each request consists of collecting a load from an origin location and delivering it to a destination location. Several requests may be transported simultaneously by the same vehicle as long as the combined load does not exceed the vehicle capacity. Therefore, unlike the classical TSP or standard VRP, visits are not independent.

Applications of PDP include many transport, logistics and manufacturing settings where goods or passengers must be moved between locations. Common examples include courier services, where parcels are collected from one customer and delivered to another, as well as taxi systems, where passengers must be picked up before being dropped off. PDP also appears in warehouse environments, where materials are transferred between storage areas, machines or distribution points. Although this project focuses on unconstrained tours, PDP illustrates how practical routing problems often include dependencies between each visit.

2.1.4 Dynamic TSP

Dynamic TSP (DTSP) refers to a variant of the TSP in which part of the problem may change over time (Li et al., 2006). Unlike the classical static TSP, where all nodes and connection costs are fixed

in advance, dynamic variants may involve changes such as moving targets, time-dependent travel times, stochastic delays or newly revealed requests.

These dynamic features are common in real-world routing environments. For example, delivery systems must adapt to uncertain travel times or updated customer information during operation. Recent work has studied DTSP variants with time-dependent and stochastic travel times (DTSP-TDS), where routes must be updated in response to changing traffic conditions (Chen et al., 2025). More specialised DTSP applications include in settings such as surveillance and defence where targets may move over time.

Although this report evaluates static instances, DTSP highlights how practical routing problems may need to be repeatedly updated as the underlying instance changes over time.

2.2 Computational Complexity

Given its prevalence across a wide range of domains, the TSP has been extensively researched in the community (Goh et al., 2024). It is known that the TSP is classified as NP-hard. This classification means that no polynomial-time algorithm is currently known for solving the TSP optimally. This makes brute force or other exhaustive methods impractical even for moderately sized instances. Brute force methods involve finding the optimal tour by extensively comparing the total distance of every possible tour to find the shortest tour. While this method is guaranteed to find the optimal tour, the number of possible tours, $|T|$, in a symmetric instance with n nodes can be calculated using the following equation:

$$|T| = \frac{(n-1)!}{2} \quad (2.1)$$

This equation comes from the fact there are $(n-1)!$ possible permutations of traversing every node in a Hamiltonian cycle, from a given starting node. However, since two tours traversed in opposite directions are considered equivalent for a symmetric instance, we can divide this number by 2. Applying Equation 2.1, an instance with 10 nodes has 181,440 possible tours. While this number of solutions could be computed within a reasonable time, an instance with just 20 nodes has 60,822,550,204,416,000 possible tours. With 100 nodes, there are over 4.7×10^{155} possible tours - more tours than atoms in the observable universe (Oswald et al., 2002). This demonstrates that the use of brute force or other exhaustive methods is infeasible, even for instances with a smaller number of nodes. Because exact solution methods quickly become infeasible, this report focuses on heuristic algorithms.

2.3 Heuristic Algorithms

Due to the intractability of finding optimal TSP solutions, we must instead rely on heuristic methods. The word *heuristic* comes from the Greek verb *heuriskein*, which means “to find”. In the context of the TSP, heuristics refer to methods of finding solutions using reasonable computational cost but with no guarantee of optimality (Salcedo-Sanz, 2016). This can greatly reduce the computational effort, and therefore time, required to find a solution to the TSP.

In this report, the heuristics we consider can be categorised as greedy construction heuristics, vertex insertion heuristics, approximation heuristics and improvement heuristics. Each type of heuristic has different benefits and drawbacks. To read about an intuitive but impractical heuristic implementation, see Appendix A.1.

2.4 Evaluation Techniques

Within this report, we observe and evaluate the quality, efficiency, and overall effectiveness of several heuristics. Below, we describe a few common evaluation techniques which support the conclusions drawn later in the report.

2.4.1 Big O Notation

Big O notation can be used to contextualise the theoretical and visual behaviour of the heuristics examined in this report (Mala and Ali, 2022). Big O notation compares, and classifies, algorithmic time complexities according to how quickly their runtimes increase as the number of nodes, n , increases. Big O is part of the broader Bachmann–Landau family of notations, alongside Big Θ (*Theta*) and Big Ω (*Omega*). While Big O provides an upper bound on the growth of a function, Big Ω provides a lower bound. When used

together they allow us to characterise the tightness of a complexity estimate. Big Θ is used when an algorithm's running time is bounded both above and below by the same growth function, meaning it grows 'on the order of' that function (Knuth, 1976). Big O focuses on the overall growth, by using dominant terms and abstracting away any constant factors or lower-order terms. This allows these classifications to focus on the factors that influence performance most.

Put formally, let f be a valued function defined on a domain D , and let g be a non-negative valued function on the same domain D . f is the function to be estimated. g is the comparison function. For this report, the domain D is defined as time. We can then write

$$f(x) = O(g(x)) \quad (2.2)$$

if there exists a positive real constant M , such that

$$|f(x)| \leq M \cdot g(x) \quad \text{for all } x \in D. \quad (2.3)$$

We can call the constant M an implied constant as it is not specified. Furthermore, despite the existence of the equality sign ($=$) in Equation 2.2, the expression refers to an inequality relating f and g (Avigad and Donnelly, 2004). It may be helpful to review the commonly encountered classifications, ordered from lowest to highest growth:

$$O(1) \rightarrow O(\log n) \rightarrow O(n) \rightarrow O(n \log n) \rightarrow O(n^2) \rightarrow O(n^3) \rightarrow O(2^n) \rightarrow O(n!) \quad (2.4)$$

For example, the runtime of an algorithm with a Big O notation of $O(n^2)$ grows no faster than n^2 operations for sufficiently large n , even if the exact runtime is smaller in practice. Conversely, the runtime of an algorithm with a Big Ω notation of $\Omega(n^2)$ grows no slower than n^2 operations, meaning its growth is at least quadratic for larger n . Finally, the runtime of an algorithm with a Big Θ notion of $\Theta(n^2)$ performs at exactly quadratic order, growing proportionally to n^2 .

Big O notation allows us to compare each heuristic using a mathematical designation. This removes any dependence on hardware or implementation details. Before conducting empirical tests, we perform a theoretical analysis of each heuristic to determine its asymptotic complexity. This involves a structural inspection of the algorithm's logic to infer its Big O time growth. By identifying any dominant operations, such as nested loops or external functions, we can characterise how the computation time scales as the nodes increase. By finding a theoretical baseline, we are able to predict the feasibility of running heuristics on large instances. Additionally, theoretical analysis allows us to validate our empirical results by checking that the implementation adheres to the expected growth trend.

2.4.2 Empirical Analysis

Throughout this project we collate evidence to fairly evaluate and compare each heuristic. Firstly, we evaluate the computation time of different heuristics over multiple instances to observe time growth. Additionally, we evaluate the tour distance found by different heuristics over multiple instances to observe tour quality.

We collate metrics using a Java implementation of each heuristic. This implementation tests heuristics on an instance of randomly distributed nodes within a grid with a width and height of 1,000. All timing experiments were conducted on the same device (Device: MacBook Pro, Processor: Apple M4, Memory: 16 GB) and software environment (OS: macOS Tahoe 26.2, Java: 17) to ensure consistency between measurements. An experiment consists of a heuristic and an instance with n nodes.

To achieve more consistent and reliable results, we repeat the same experiment over 10 trials. Repeating each experiment setup over 10 trials provides enough repetition to observe growth trends and identify volatility, while keeping the total number of experiments manageable across all heuristics and values of n . With 10 heuristic setups, 99 sizes of n and 10 trials, we can expect to conduct 9,900 experiments. Once exported from the implementation, we can evaluate the results. First, we find the median computation time ($\tilde{t}$) and relative standard deviation (RSD).

Results from the Java implementation may be affected by a multitude of external factors: OS interruptions, high-frequency timing variance, or random instance specifics. By using the median computation time we are able to remove these outliers. Given a sorted set of k times $t_{(0)}, t_{(1)}, \dots, t_{(k-1)}$, we can calculate the median computation time using the following equation:

$$\tilde{t} = \begin{cases} t_{(\frac{k-1}{2})} & \text{if } k \text{ is odd} \\ \frac{t_{(\frac{k-1}{2})} + t_{(\frac{k}{2})}}{2} & \text{if } k \text{ is even} \end{cases} \quad (2.5)$$

To verify the validity of the exported data, we calculate the RSD. This shows us the consistency of the heuristic's performance across different random instances. A low RSD ($\leq 10\%$) indicates that the median results are representative of the algorithm's typical behaviour. A high RSD, however, can suggest that the heuristic is sensitive to a specific instance. We can calculate the RSD with the standard deviation (σ) and the mean ($\bar{t}$) using the following equation:

$$\text{RSD} = \left(\frac{\sigma}{\bar{t}} \right) \times 100\% = \left(\frac{\sqrt{\frac{1}{k} \sum_{i=0}^{k-1} (t_i - \bar{t})^2}}{\frac{1}{k} \sum_{i=0}^{k-1} t_i} \right) \times 100\% \quad (2.6)$$

In addition to evaluating the computation time, we also want to evaluate the quality of tours produced by different heuristics by evaluating tour lengths. To do this, we apply the Beardwood-Halton-Hammersley (BHH) theorem (Beardwood et al., 1959). The BHH theorem states that for n nodes distributed independently and identically within a region of area V , the length of the optimal TSP tour L satisfies the limit:

$$\lim_{n \rightarrow \infty} \frac{L}{\sqrt{n \cdot V}} = \beta \quad (2.7)$$

where β is a constant. The modern, widely accepted, value for the BHH constant is $\beta \approx 0.7120$ (Percus and Martin, 1996). Applying this to the data exported from the Java implementation, we can calculate $\hat{\beta}$ using the tour length, number of nodes and instance area. This can be referred to as the normalised tour length. To analytically assess the tour lengths, we can compare the observed $\hat{\beta}$ to β and find the optimality gap δ , defined as:

$$\delta = \frac{\hat{\beta} - \beta}{\beta} \times 100\% \quad (2.8)$$

These empirical techniques provide a consistent and fair evaluation of heuristic performance as n increases. We can use this data to plot graphs and compare heuristic performance. To improve visual clarity and highlight trends in the resulting plots, a rolling mean (window size $w = 5$) is applied to the data. This smoothing technique reduces high-frequency noise, caused by specific random node distributions or external factors, without altering the underlying performance characteristics of the heuristics.

2.4.3 Instances

When we refer to an 'instance' within this report, we are referring to an array of nodes/cities with coordinates: x and y . These nodes, and their associated coordinates, are generated through functions using predetermined options, such as 'Type', 'Spread' or 'Seed'. Specifically, the 'Seed' option can be used to precisely reproduce instances. Throughout this report, we observe how the heuristics behave when solving various instances. For consistency, we evaluate the heuristics using a defined set of instance types. These types are defined as follows:

- **Real-world instance** – A map-based layout of 100 cities derived from real-world cities of the British Isles and mainland Europe. This instance is not adjustable and can be used for consistent comparison between different heuristics. For consistency, the initial starting city for the real-world instance, if not amended, is *London*.

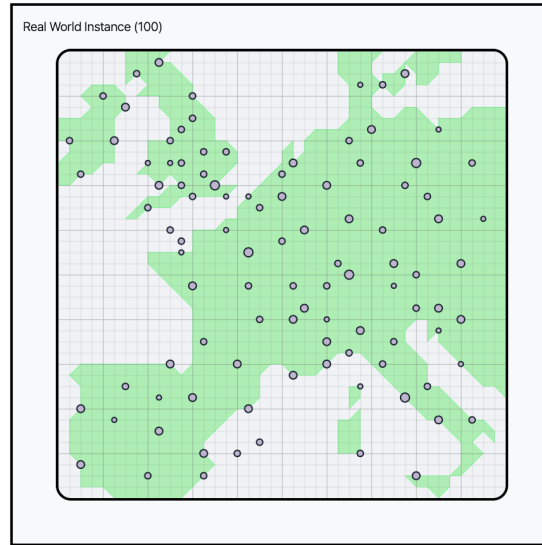


Figure 2.1: The real-world instance with 100 cities.

- **Grid instances** – Nodes laid out in a grid pattern. Options like ‘Width’, ‘Spread’, ‘Seed’ and ‘Density’ can be adjusted to change the behaviour of the instance. For consistency, the initial starting node for grid instances, if not amended, is the top left node: *P1-1*.

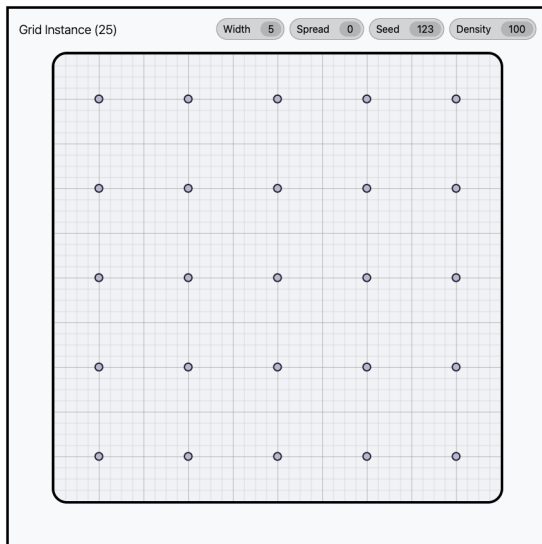


Figure 2.2: A grid instance with 25 evenly spaced nodes.

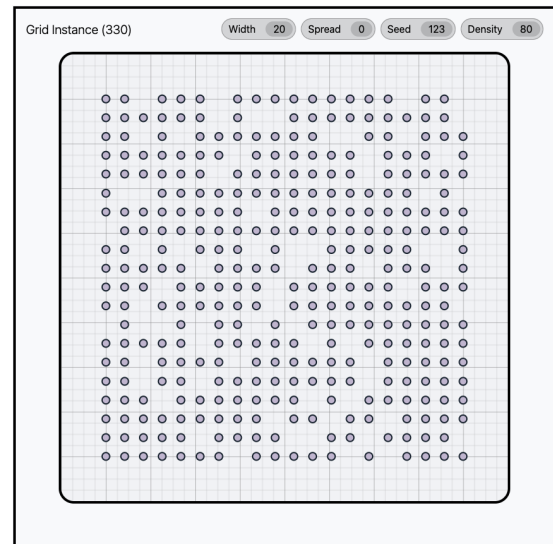


Figure 2.3: A grid instance with 25 evenly spaced, then randomly adjusted, nodes.

- **Circle instances** – Nodes laid out along a circle of fixed radius. Options like ‘Nodes’, ‘Spread’ and ‘Seed’ can be adjusted to change the behaviour of the instance. For consistency, the initial starting node for circle instances, if not amended, is the top node: *P1*.

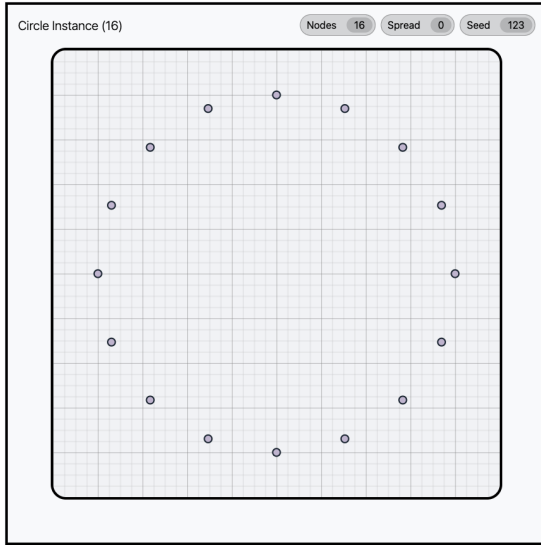


Figure 2.4: A circle instance with 16 evenly spaced nodes.

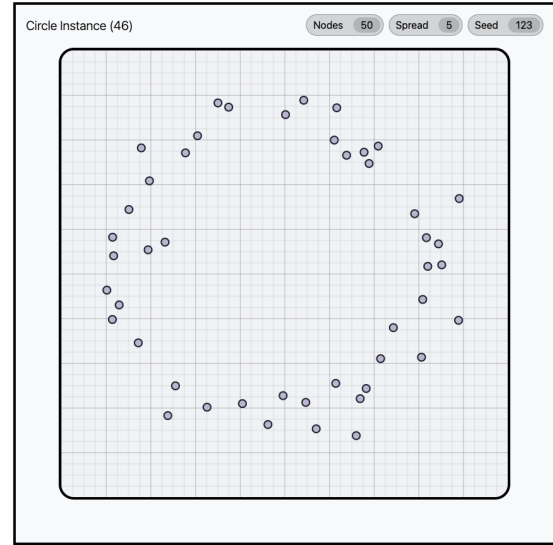


Figure 2.5: A circle instance with 43 evenly spaced, then randomly adjusted, nodes.

- **Cluster instances** – Clusters of nodes laid out along a circle of fixed radius. Options like ‘Nodes’, ‘Clusters’, ‘Spread’ and ‘Seed’ can be adjusted to change the behaviour of the instance. For consistency, the initial starting node for cluster instances, if not amended, is the first node in the first cluster: *P1-1*.

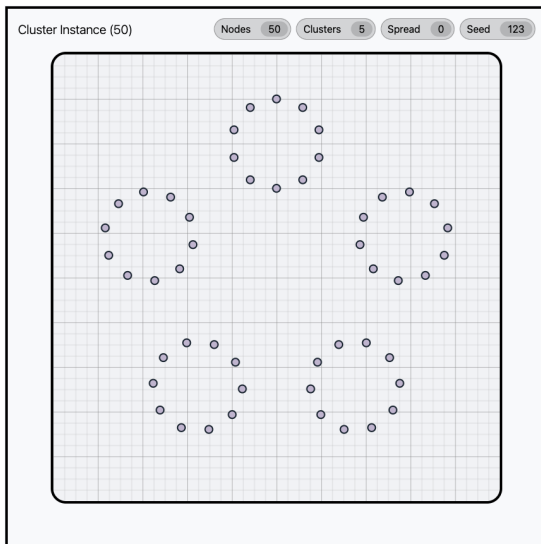


Figure 2.6: A cluster instance with 5 evenly spaced clusters with 50 evenly spaced and distributed nodes.

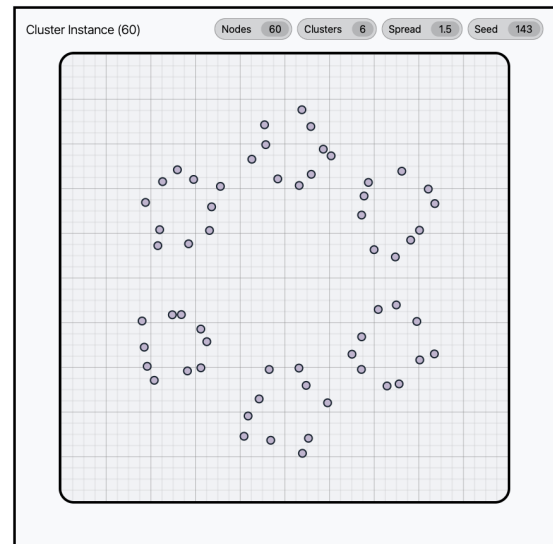


Figure 2.7: A cluster instance with 6 clusters with 60 evenly spaced, then randomly adjusted, nodes.

- **Random instances** – Randomly distributed nodes with no constraints. Options like ‘Nodes’, ‘Spread’ and ‘Seed’ can be adjusted to change the behaviour of the instance. For consistency, the initial starting node for random instances, if not amended, is *P1*.

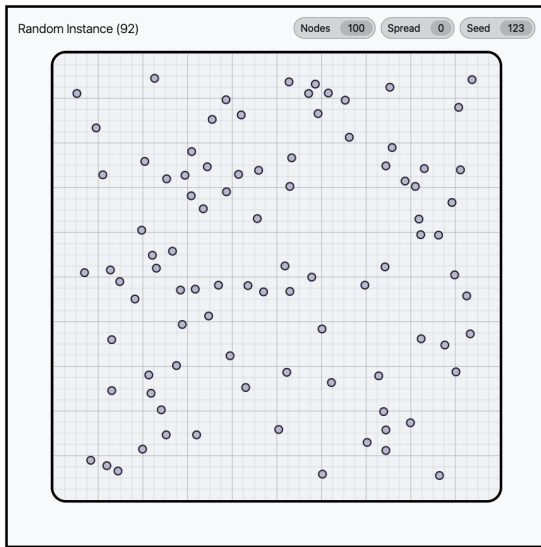


Figure 2.8: A random instance with 92 randomly distributed nodes.

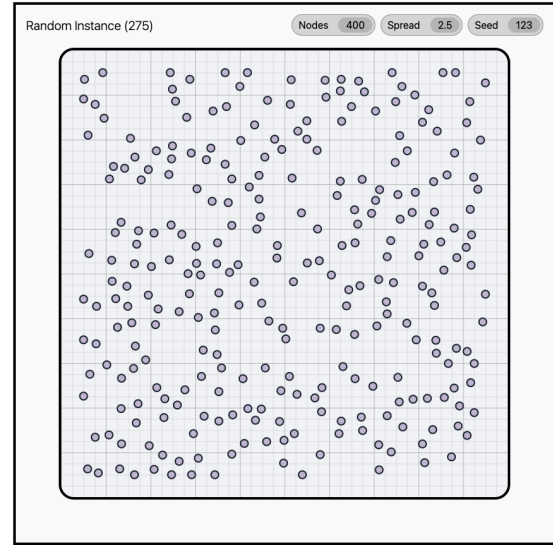


Figure 2.9: A random instance with 275 randomly distributed, then randomly adjusted, nodes.

- **Test instances** – Imported layouts from the TSPLIB collection. These instances have solved optimal tours, which can be used to evaluate tour quality. For consistency, the initial starting node for test instances, if not amended, is *P1*.

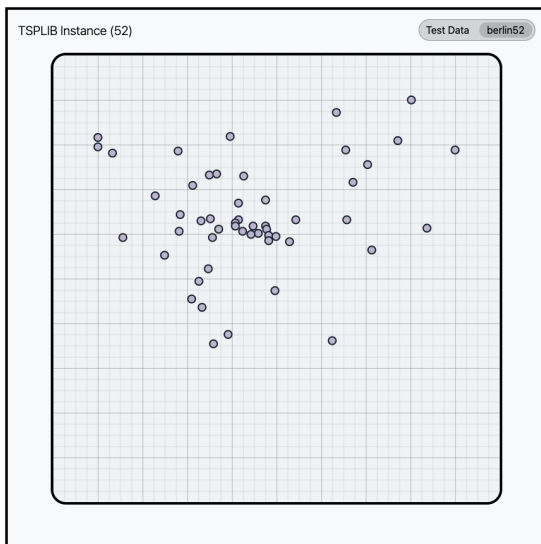


Figure 2.10: The Djibouti instance, *dj38*, with 38 cities and an optimal length of 6,656 (Reinelt, 1991).

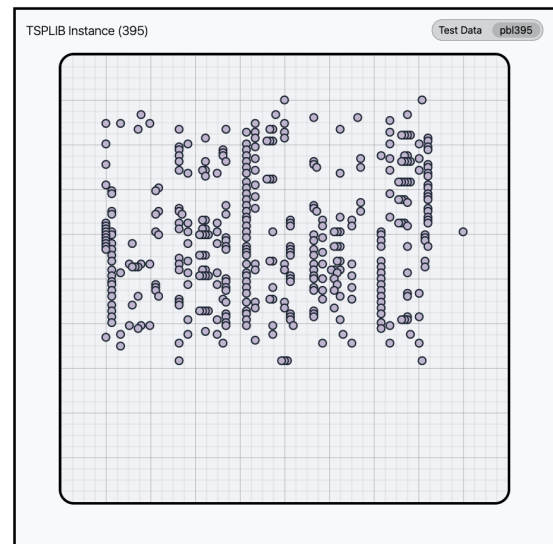


Figure 2.11: The Western Sahara instance, *wi29*, with 29 cities and an optimal length of 27,603 (Reinelt, 1991).

3 Software Development

3.1 Requirement Analysis

3.1.1 Functional Requirements

- **Requirement 1:** The project shall load predefined and imported TSP instances.
- **Requirement 2:** The project shall allow the user to select a heuristic.
- **Requirement 3:** The project shall allow the user to select a starting node (when applicable).
- **Requirement 4:** The project shall be able to compute the tour of the given instance using the given heuristic and starting node.
- **Requirement 5:** The project shall produce the steps taken to solve an instance, enabling step-by-step playback in the UI.
- **Requirement 6:** The project shall present computed metrics, such as the tour length, step count, or computation time.

3.1.2 Non-functional Requirements

- **Requirement 1:** Heuristics shall solve in under 1 second for instances with up to 1,000 nodes.
- **Requirement 2:** Computations which could take long periods of time must not affect the performance of the UI.
- **Requirement 3:** Distance calculations must be consistent across all heuristics.
- **Requirement 4:** The GUI must clearly represent TSP instances, solutions and controls.
- **Requirement 5:** Heuristic solving functions shall be modular and consistent to better support adding new ones.
- **Requirement 6:** Heuristic solving should not block the main UI thread, and must be delegated to a Web Worker.

3.1.3 Constraints

- **Constraint 1:** The project must be implemented with Vue.js and TypeScript, as laid out in the project plan.
- **Constraint 2:** All heuristic algorithms must agree with commonly accepted implementations.
- **Constraint 3:** The project, and all required computation, shall be limited to client-side only, no need for a backend.
- **Constraint 4:** To preserve browser performance, TSP instances shall be limited to a reasonable size.
- **Constraint 5:** The project shall support importing external TSP instances (e.g. from TSPLIB (Reinelt, 1991)).

3.1.4 Success Criteria

- **Criteria 1:** The project must correctly compute the tours for all supported instances and heuristics.
- **Criteria 2:** The user must be able to playback any generated solutions.
- **Criteria 3:** Performance metrics must align with derived growth trends for each heuristic.

3.2 Design Decisions

3.2.1 Architectural Decisions

The project was developed as a client-side interactive GUI using Vue.js and TypeScript. This architecture compartmentalises the user interface from computational logic. The implementation computes heuristic solutions using a Web Worker to separate computation away from the UI. Java was also used alongside the GUI for large-scale and iterative experimental evaluation. Python was used to generate visual materials for this report. The project was intentionally kept client-side to reduce unnecessary complexity. This was acceptable because problem instances and heuristic computations were localised and self-contained. This also aligned with the project constraints.

The decision to solve instances using a Web Worker allowed the GUI to remain responsive while solutions were being computed. Without this, larger instances and playback generation could block rendering and harm usability. Additionally, instances were solved in full before being displayed to the user to allow for full playback. This architecture supported modularity and reproducibility throughout development, with support for additional heuristics and instance types.

3.2.2 User Interface Design

The development of the project's GUI required careful thought of various design decisions. This discipline ensured that the GUI was intuitive and easy to use, as well as being informative and highly capable. An important factor was the generation of the instances. The project needed to allow the user to generate a wide range of instances, while not overcrowding the UI. To achieve this, the project uses procedural parametric instance generation. This abstracted the complexity of the underlying coordinate logic, allowing the user to create tailored instances through parameters without manual plotting. The inclusion of a 'Seed' parameter was critical for experimental accuracy, allowing for the reproduction of specific instances for experiments and debugging.

Another factor considered in the GUI design was the process of viewing heuristic solutions. The project maintains a logical hierarchy for the actions required to solve an instance: instance creation, heuristic choice, and start node selection. This linear workflow prevented state errors and reduced the complication on the user. Furthermore, the project features additional capabilities, such as comparison of multiple heuristics and downloadable results, to provide a more informative user experience.

Finally, one of the most impactful factors was solution presentation. The system tracked the steps taken by the heuristic algorithms while solving individual instances. This unlocked the ability to playback frames of a solution. The development of playback controls to allow for unrestricted analysis enabled the user to inspect the heuristic's logic at specific frames rather than simply seeing the final tour. However, it was crucial how this information was interpreted and shown to the user. Details such as grouping comparison steps to reduce the number of frames, or the order of selection steps were vital in establishing a coherent display of a solution. This technique highlights the macro behaviour of the heuristic and prevents the core logic from being obscured by redundant details, while also displaying the necessary algorithmic steps.

3.3 Version Control System

Throughout development of the project, the departmental version control system (VCS), GitLab, was used. GitLab was used not only for backup, but to also support safe iteration throughout development. Making use of branches allowed new heuristics and refactors to be developed independently before merging into the main codebase, reducing the risk of existing functionality being broken. This was particularly useful when modifying shared components such as instance generators or heuristics interfaces. In addition to storing the GUI, the same repository stored the Java program, Python script and additional documents. This improved traceability between implementation, experimentation and reports.

3.4 Diagrams

3.4.1 UML Class Diagram

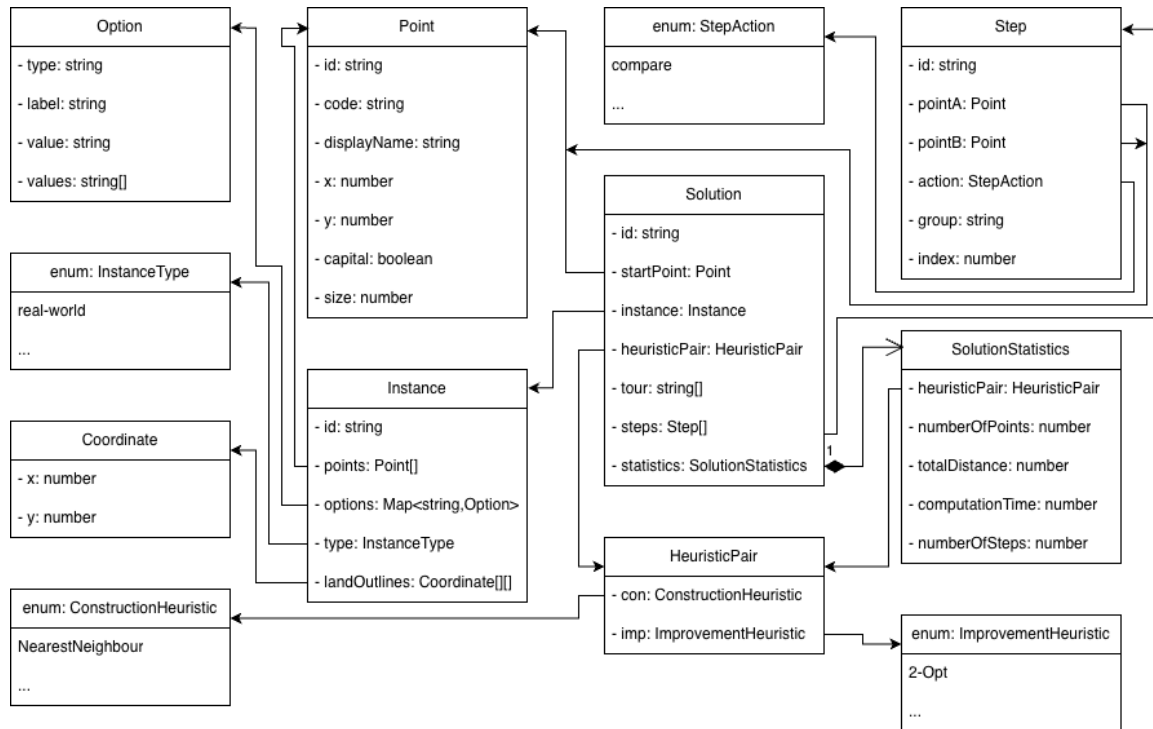


Figure 3.1: A UML diagram of the classes implemented in the project.

The UML class diagram shown in Figure 3.1 shows how the project isolates core concepts such as instances, points (nodes), solutions and playback steps. In particular, the design distinguishes between the problem data (*Instance*, *Point*), the solving configuration (*HeuristicPair*), and the output (*Solution*, *Step*). This separation supported modular development, making it easier to add new heuristics and instance types, and keep the GUI logic consistent across different solving methods.

3.4.2 UML Sequence Diagram

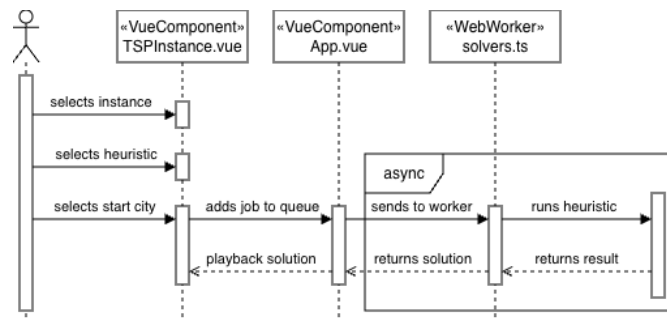


Figure 3.2: A UML sequence diagram showing the flow of viewing a solution.

Figure 3.2 shows a UML sequence diagram that illustrates how a user-triggered solve request flows through the program. From instance, heuristic and start node selection to asynchronous execution in the Web Worker. This reflects an important architectural decision of the project, where heuristic computation is delegated away from the main UI thread so that large instances can be solved without freezing the interface. This demonstrates how the implementation satisfies the non-functional requirement of responsiveness.

4 Greedy Heuristics

4.1 Nearest Neighbour Heuristic

4.1.1 Algorithm Explained

The Nearest Neighbour (NN) algorithm (Rosenkrantz et al., 1977) is one of the simplest methods of computing a tour. The NN algorithm determines the distance from a given node to every unvisited node to find its nearest neighbour. This nearest neighbour is then selected and considered visited. These previous two steps are repeated until every node has been visited and the tour is complete.

Put formally, let n_s be the starting node, and all the nodes in the instance be $N = n_1, \dots, n_m$. We can then define a set of unvisited nodes as $U = N \setminus \{n_s\}$. We then repeat the following step until U is empty ($\emptyset$): from the most recently selected node, n_r , we compute the distance to every node in U , to find the nearest neighbouring unvisited node, which is then added to T and removed from U . Finally, we add n_s to the end of the tour, T , to close the loop and complete the tour.

4.1.2 Pseudocode Representation

Below is the pseudocode implementation of the Nearest Neighbour algorithm. The parameters of the function are the start (and end) node, **startNode**, as well as the list of all nodes, **nodes**. The function runs the NN algorithm on the given list of nodes, and returns an ordered list of nodes, **tour**.

Algorithm 1 Nearest Neighbour Algorithm

```
1: procedure RUNNEARESTNEIGHBOUR(nodes, startNode)
2:   tour  $\leftarrow$  [startNode]
3:   unvisitedNodes  $\leftarrow$  nodes
4:   currentNode  $\leftarrow$  startNode
5:   remove currentNode from unvisitedNodes ▷ Mark the start node as visited
6:   while there are unvisited nodes do
7:     nearestNode  $\leftarrow$  null
8:     nearestDistance  $\leftarrow$   $\infty$ 
9:     for each node in unvisitedNodes do ▷ Find the nearest unvisited node to currentNode
10:      distance  $\leftarrow$  HYPOT(node, currentNode)
11:      if distance < nearestDistance then
12:        nearestDistance  $\leftarrow$  distance
13:        nearestNode  $\leftarrow$  node
14:      end if
15:    end for
16:    add nearestNode to tour
17:    remove nearestNode from unvisitedNodes
18:    currentNode  $\leftarrow$  nearestNode ▷ Update currentNode
19:  end while
20:  add startNode to tour ▷ Close tour
21:  return tour
22: end procedure
```

4.1.3 Examples

To better visualise and understand NN, we can look at how the algorithm performs in different scenarios. Below are the final solutions of various instances computed using the NN algorithm.

Best Case

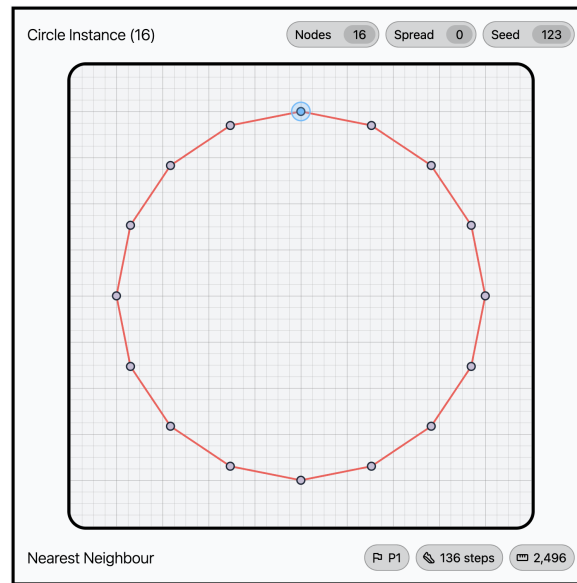


Figure 4.1: The Nearest Neighbour solution of a circle TSP instance with 16 nodes

The best case scenario for the Nearest Neighbour algorithm is an instance where there is a clear route with no need to backtrack during the tour. One such instance could be a perfect circle of nodes. An example of this can be seen in Figure 4.1 where we can see that the algorithm found the optimal tour by simply travelling around the circle and returning to the start node.

General Case

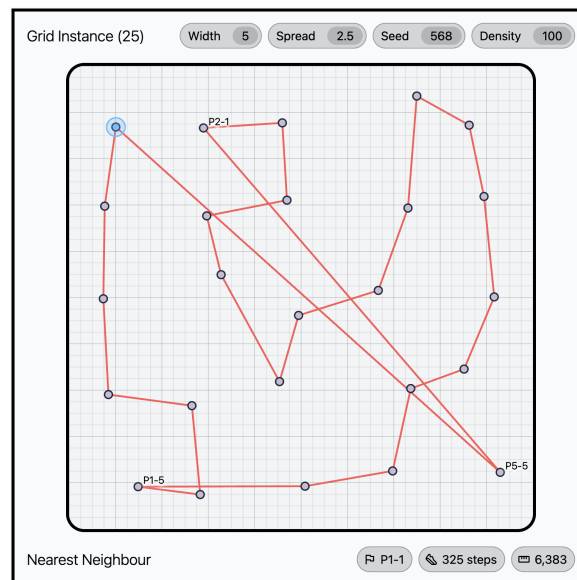


Figure 4.2: The Nearest Neighbour solution of a grid TSP instance with 25 nodes

However, the general case shown in Figure 4.2 showcases a common pitfall of the NN algorithm. As NN traverses through unvisited nodes, some nodes are often missed. In Figure 4.2 we can see the algorithm skips node *P5-5* and is then forced to go back and pick this node up later from node *P2-1*. This detour adds significant distance to the tour by adding two large edges. These commonly missed nodes, caused by the algorithms greedy methodology, save minor distance in the short term by taking shortcuts, but

cost major distance in the long term. This becomes even more evident in the real-world instance where NN skips multiple cities and even regions.

Real-World Case

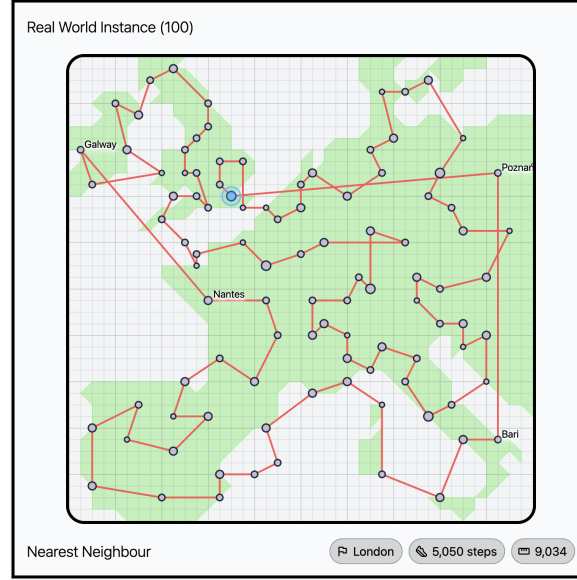


Figure 4.3: The Nearest Neighbour solution of the real-world TSP instance

Figure 4.3 is the tour given when the NN algorithm is run on the real-world instance, starting at *London*. We can observe that the algorithm begins reasonably well, picking up cities across the east of the British Isles. However, the algorithm then chooses to head towards Denmark via *Dover*. By greedily skipping the rest of the British Isles early on, the algorithm is required to go back there multiple times, introducing a jump between *Galway* and *Nantes*. Finally, the algorithm skips past *Poznań* early on in the traversal, causing big jumps between *Bari*, *Poznań* and *London* when closing the tour.

4.1.4 Evaluation

Time Growth

We can theoretically derive the time growth of NN by analysing its algorithm. From the start node, we check all other nodes to find the nearest node. If we have an instance with n nodes, then the first node check $(n - 1)$ nodes, the second checks $(n - 2)$ nodes, and so on for all nodes. This equates to:

$$\frac{n(n-1)}{2} = \frac{n^2 + n}{2} \quad (4.1)$$

For Big O notation, we can drop the lower order terms of Equation 4.1. As a result, we derive the time growth of NN as $O(n^2)$.

n	Median Time (ms)
10	0.0
100	0.0
250	0.1
500	0.4
750	0.9
1,000	1.6

Table 4.1: The median computation times for different values of n between 10 and 1,000 using the Nearest Neighbour algorithm. For full results, see Table C.1.

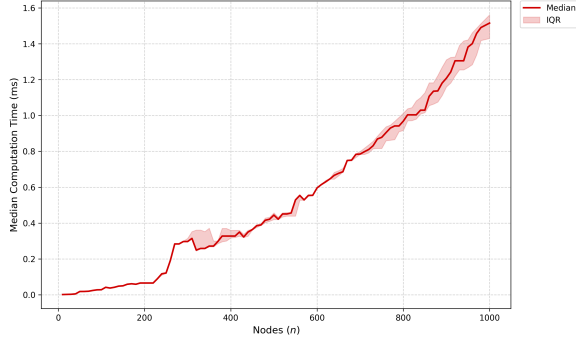


Figure 4.4: A graph showing the median computation time against nodes (n).

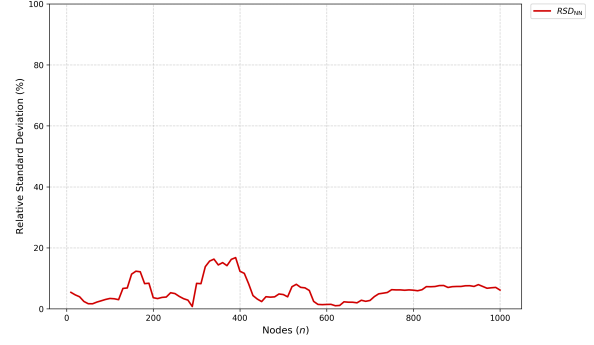


Figure 4.5: A graph showing the relative standard deviation in computation time against nodes (n).

Table 4.1 shows how the computation time of the NN algorithm grows as n increases. We can observe from this table that when the number of nodes is doubled from 500 to 1,000 nodes, the median computation time increases by an exact factor of 4. This can also be seen in Figure 4.4, where the median computation time increases polynomially as n increases. We can trust the precision of these results as Figure 4.5 shows that the relative standard deviation remains below 20% for the full range of conducted experiments. These results show that the NN algorithm can find instance tours quickly, able to solve 1,000 nodes in 1.6ms. However, there is a slight degradation in performance as n increases.

Using both the empirical and theoretical analysis, we are able to derive that the NN heuristic follows a quadratic time growth of $O(n^2)$. While this is significantly better than the factorial growth of the brute force approach, this confirms that NN will still struggle to scale for very large instances.

Tour Quality

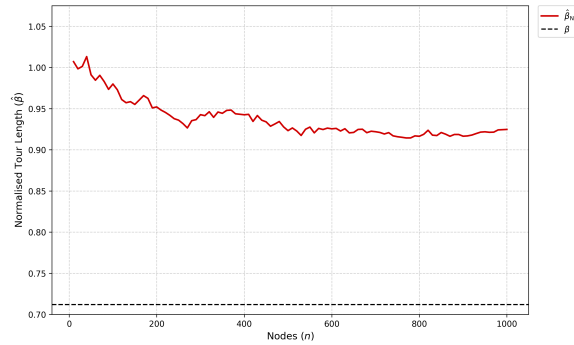


Figure 4.6: A graph showing the normalised tour length ($\hat{\beta}_{NN}$) against nodes (n).

Figure 4.6 shows the normalised tour length achieved by the NN algorithm for $10 \leq n \leq 1,000$. For $n \leq 200$, we find that $\hat{\beta}_{NN}$ reaches as high as 1.05. An optimality gap of over 47.4%. However, for larger instances of $n \geq 600$, we find that $\hat{\beta}_{NN}$ settles at around 0.93. This large fluctuation in $\hat{\beta}_{NN}$ shows us that NN can often perform unreliably, due to lacking global context. Extrapolating, we can define $\hat{\beta}_{NN} = 0.93$. When compared to the theoretical optimal limit of $\beta \approx 0.7120$ this represents an optimality gap of $\delta = 30.6\%$.

4.1.5 Conclusion

Overall, we can observe that the NN heuristic scales well but lacks the ability to reliably produce coherent tours. We see large jumps at the end of the majority of tours, creating large discrepancies. As well as its greedy behaviour, one of the main causes of these inconsistencies is the algorithm's dependency on the start node.

4.2 Repeated Nearest Neighbour Heuristic

4.2.1 Algorithm Explained

The Repeated Nearest Neighbour (RNN) algorithm is an extension of NN. The RNN algorithm adopts the NN algorithm to determine which starting node produces the best tour. To do so, the algorithm applies NN while attempting every node as a starting node. By keeping track of the best tour, and its length, the RNN algorithm finds the shortest possible NN tour of an instance.

Put formally, let all the nodes in the instance be $N = n_1, \dots, n_m$ and let T_B be the best tour. For every starting node, n_s , in N , repeat the following steps. Let T be the tour found using the NN algorithm with N and n_s . If the total distance of T is less than the total distance of T_B , then set T_B to T . Once we iterate through every start node n_s , we know T_B to be the optimum NN tour.

4.2.2 Pseudocode Representation

Below is the pseudocode implementation of the Repeated Nearest Neighbour algorithm. The parameters of the function are the list of all nodes, **nodes**. The function runs the RNN algorithm on the given list of nodes, and returns an ordered list of nodes, **tour**.

Algorithm 2 Repeated Nearest Neighbour Algorithm

```
1: procedure RUNREPEATEDNEARESTNEIGHBOUR(nodes)
2:   shortestTour  $\leftarrow$  null
3:   shortestTourLength  $\leftarrow \infty$ 
4:   for each startNode in nodes do                                 $\triangleright$  Try every node as the start node
5:     tour  $\leftarrow$  RUNNEARESTNEIGHBOUR(nodes, startNode)            $\triangleright$  Call the NN algorithm
6:     tourLength  $\leftarrow$  CALCULATETOURLENGTH(tour)
7:     if tourLength < shortestTourLength then                       $\triangleright$  Check to update the shortest pointers
8:       shortestTourLength  $\leftarrow$  tourLength
9:       shortestTour  $\leftarrow$  tour
10:    end if
11:  end for
12:  return shortestTour                                              $\triangleright$  Return the shortest tour
13: end procedure
```

4.2.3 Examples

To better visualise and understand RNN, we can look at how the algorithm performs in different scenarios. As RNN is a direct extension of NN, we can compare their solutions. Below are the final solutions of various instances computed using the RNN algorithm.

Best Case

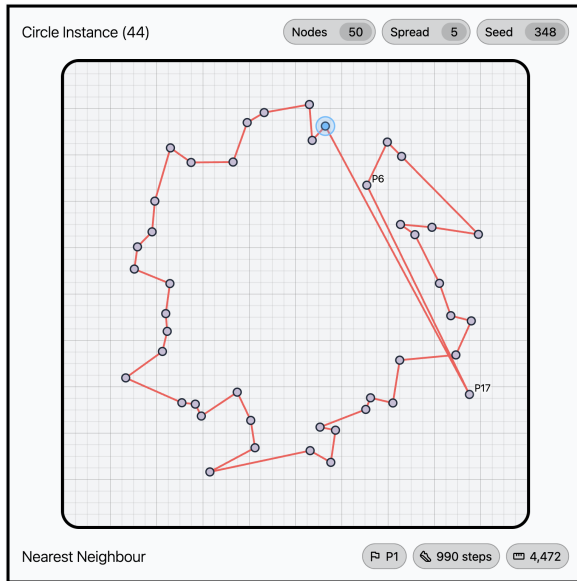


Figure 4.7: The Nearest Neighbour solution of a circle TSP instance with 44 nodes

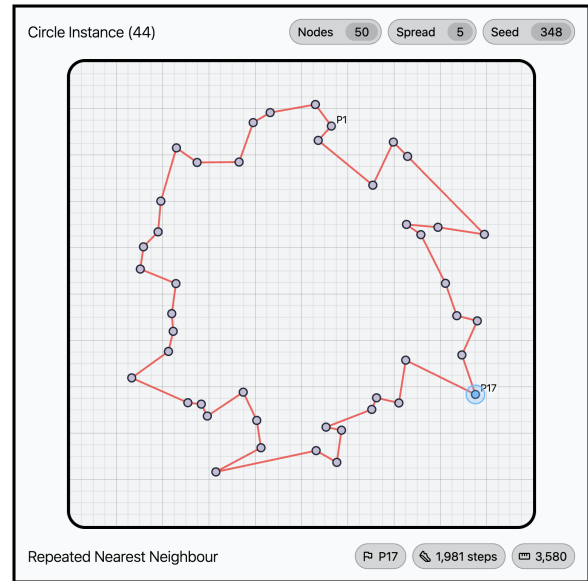


Figure 4.8: The Repeated Nearest Neighbour solution of a circle TSP instance with 44 nodes

Above we can compare two solutions to the same circle instance, found using the RNN algorithm (Figure 4.8) and the NN algorithm (Figure 4.7). We can observe that NN misses node $P17$ in the middle of its traversal. This requires the algorithm to go back and clean up the node at the end. However, despite RNN making use of the same underlying logic, since the algorithm was able to check every possible starting node, it had more opportunities to find a tour that picked up every node. For this instance, this happened to be the tour starting at $P17$. The thorough examination of this instance by the RNN heuristic reduced the tour length by approximately 19.9% compared to the NN heuristic.

General Case

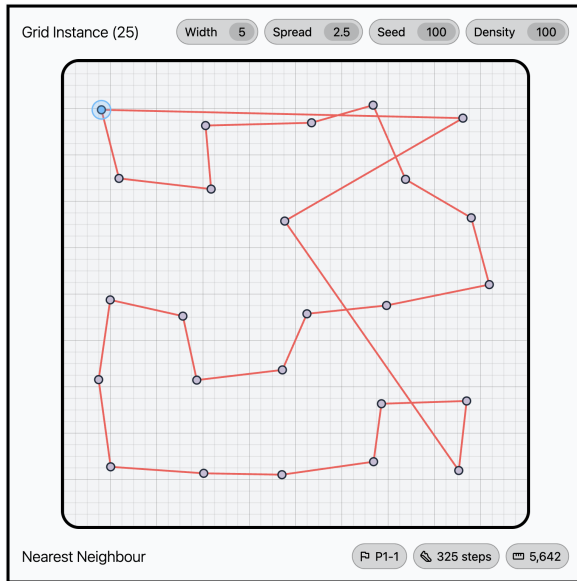


Figure 4.9: The Nearest Neighbour solution of a grid TSP instance with 25 nodes

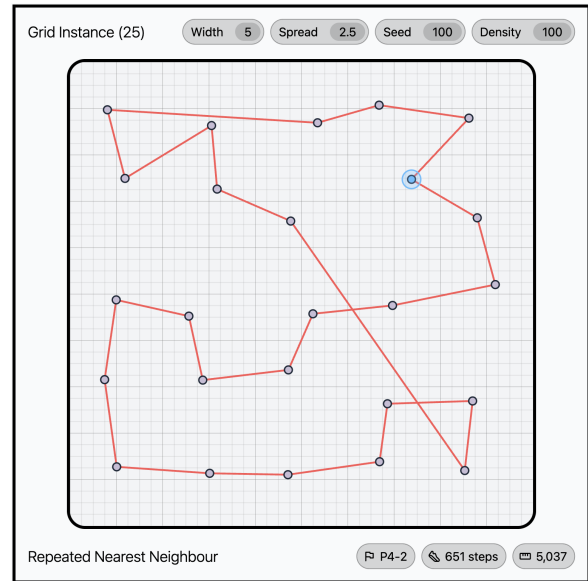


Figure 4.10: The Repeated Nearest Neighbour solution of a grid TSP instance with 25 nodes

We can state that RNN is an improvement on NN as it always finds the best possible NN solution for any given instance. By definition, the tour found by RNN will always be at least equal, if not better, than the tour found by NN. We can see that Figure 4.10 shows a distinctly better tour than Figure 4.9. However, RNN is still undermined by the shortcomings of NN. We can see how both the RNN and NN solutions miss nodes, or groups of nodes, and have to make inefficient jumps.

Real-World Case

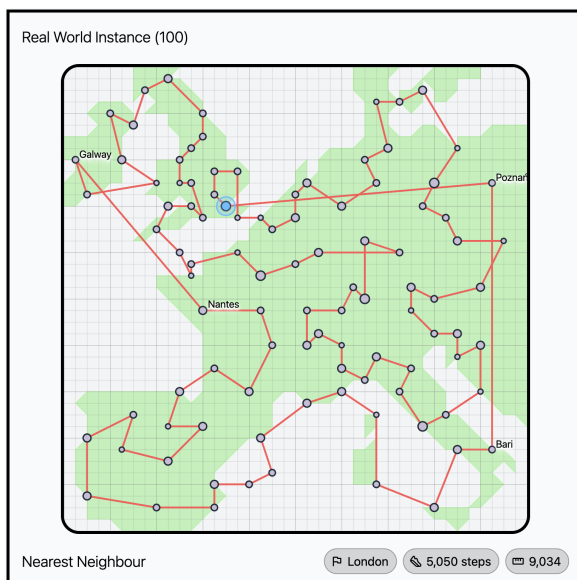


Figure 4.11: The Nearest Neighbour solution of the real-world TSP instance

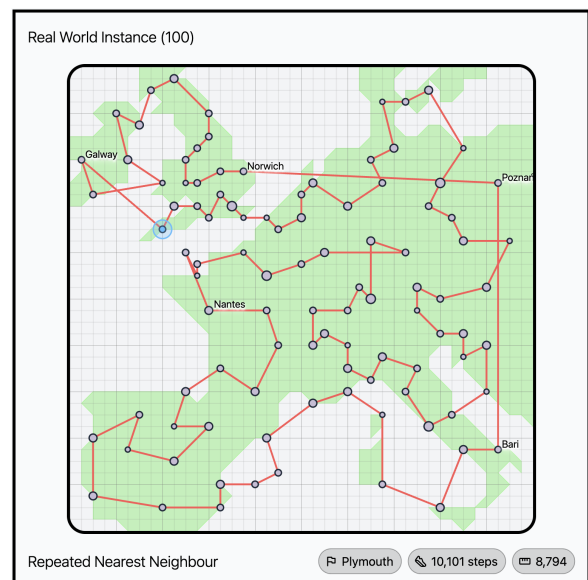


Figure 4.12: The Repeated Nearest Neighbour solution of the real-world TSP instance

In Figure 4.12, we can observe that the RNN algorithm determines that the optimal starting city is *Plymouth*. This alteration in the starting city improves the tour, removing one of the largest jumps made

by NN between *Galway* and *Nantes*. However, we can also note that the majority of the remaining tour stays the same. This is most evident when comparing the identical sub-tour formed between *Nantes* and *Bari*. This also means RNN makes similar mistakes caused by the lack of foresight from NN. For example, the RNN tour is still forced to make large jumps between *Bari*, *Poznań* and *Norwich*. Overall, RNN reduces the tour length from 9,034 to 8,794, a reduction of approximately 2.7%.

4.2.4 Evaluation

Time Growth

We can infer the time growth algorithmically by analysing the structure of the RNN algorithm. We can start by inheriting the Big O notation for NN of $O(n^2)$. Then, for an instance with n nodes, the RNN algorithm runs the NN algorithm n times, for every possible start node. We can therefore infer a time growth complexity for RNN as:

$$O(n^2 \cdot n) = O(n^3) \quad (4.2)$$

n	Median Time (ms)
10	0.0
100	1.8
250	13.5
500	92.5
750	301.7
1,000	698.8

Table 4.2: The median computation times for different values of n between 10 and 1,000 using the Repeated Nearest Neighbour algorithm. For full results, see Table C.2.

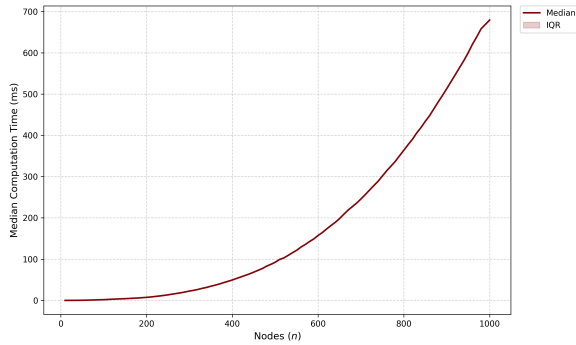


Figure 4.13: A graph showing the median computation time against nodes (n).

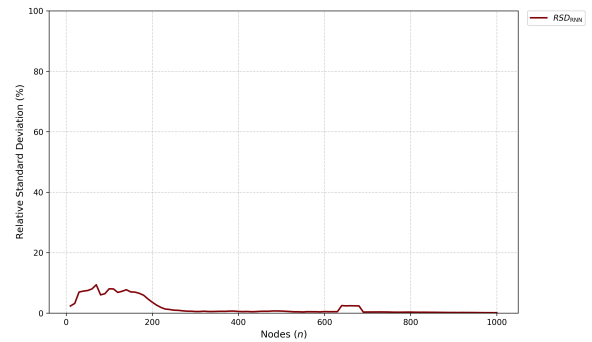


Figure 4.14: A graph showing the relative standard deviation in computation time against nodes (n).

The empirical results shown in Figure 4.13 and Table 4.2 show a significant polynomial growth in computation time as n increases. By applying a power-law analysis between two of the data points ($n = 250$ and $n = 750$), an exponential growth factor of approximately 2.83 is observed ($\log_3 22.3$). This strongly suggests that the implementation scales at a rate approaching $O(n^3)$. In Figure 4.14, we can observe a consistently low RSD for $10 \leq n \leq 1,000$.

Putting together the theoretical and empirical analyses, we can derive that the time growth of the RNN algorithm has a cubic $O(n^3)$ complexity. This is an increase in complexity from $O(n^2)$ for NN. While still quick, able to solve an instance with 1,000 nodes in less than 700ms, there is a clear, unsustainable degradation in performance. For example, the runtime for an instance with 1,000 nodes takes approximately 436 times longer to compute.

Tour Quality

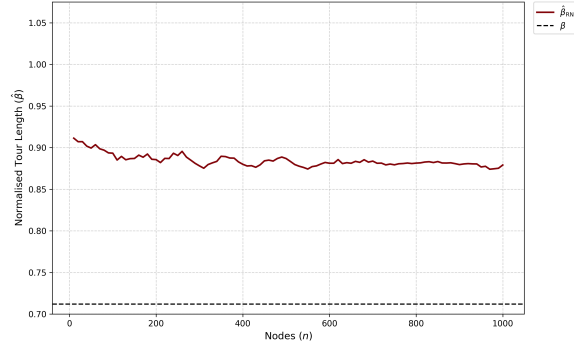


Figure 4.15: A graph showing the normalised tour length ($\hat{\beta}_{RNN}$) against nodes (n).

In Figure 4.15, we can see the normalised tour length achieved by the RNN algorithm for $10 \leq n \leq 1,000$. We can observe a consistent line that hovers around, and eventually settles on, approximately 0.88. This consistency can be attributed towards the extensive search the algorithm performs for the best possible NN tour. By defining $\hat{\beta}_{RNN} = 0.88$, we find an optimality gap of $\delta = 23.6\%$ when compared to $\beta \approx 0.7120$.

4.2.5 Conclusion

Overall, we can see a clear improvement in tours produced by RNN compared to NN. Furthermore, we can see a reduction in major inconsistencies that were present in NN tours. However, the algorithm ultimately inherits NN's limitations. This causes the algorithm to almost inevitably miss nodes. Therefore, the severe increase in runtime is hard to justify.

4.3 Multi-fragment Heuristic

4.3.1 Algorithm Explained

Unlike other greedy heuristics, such as NN or RNN, which use vertex selection, the Multi-fragment (MF) heuristic (Gutin and Yeo, 2007) uses edge selection. To build a tour, the MF algorithm repeatedly selects the shortest unused edge in the instance. However, the selection of this edge must not violate the structural constraints of a Hamiltonian cycle.

Put formally, let an edge be a set of two vertices, (u, v) . Let E be a set of all possible edges of an instance with n nodes. Let T be the tour. To start, we sort the elements of E by increasing length. Then we iterate through each edge in E . As we go, we add an edge, e_k , to T if it meets specific criteria. This criterion is defined as follows: neither u nor v in e_k have a degree of 2, and adding the edge does not create a cycle (unless it completes T). We stop iterating through E when we have n edges in T , forming a single Hamiltonian cycle.

4.3.2 Pseudocode Representation

Below is the pseudocode implementation of the Multi-fragment algorithm. The parameters of the function are the list of all nodes, **nodes**. The function runs the MF algorithm on the given list of nodes, and returns an ordered list of nodes, **tour**.

Algorithm 3 Multi-fragment Algorithm

```
1: procedure RUNMULTIFRAGMENT(nodes)
2:   edges  $\leftarrow$  GENERATEEDGES(nodes) ▷ Generate all edges between nodes
3:   unvisitedEdges  $\leftarrow$  SORTBYDISTANCE(edges)
4:   n  $\leftarrow$  LENGTH(nodes)
5:   Initialise nodeDegrees[n]  $\leftarrow$  0 for all n  $\in$  nodes ▷ Initialise states
6:   Initialise adjacencyList[n]  $\leftarrow$  [] for all n  $\in$  nodes
7:   disjointSet  $\leftarrow$  NEWDISJOINTSET(nodes)
8:   edgeCount  $\leftarrow$  0
9:   while edgeCount < n do
10:    selectedEdge  $\leftarrow$  null
11:    for each edge in unvisitedEdges do ▷ Find the smallest edge that meets criteria
12:      if nodeDegrees[edge.A] < 2 and nodeDegrees[edge.B] < 2 then
13:        if disjointSet.find(edge.A)  $\neq$  disjointSet.find(edge.B) or edgeCount = n - 1 then
14:          selectedEdge  $\leftarrow$  edge
15:          break
16:        end if
17:      end if
18:    end for
19:    if selectedEdge  $\neq$  null then ▷ Update states relating to the selected edge
20:      add selectedEdge.B to adjacencyList[selectedEdge.A]
21:      add selectedEdge.A to adjacencyList[selectedEdge.B]
22:      remove selectedEdge from unvisitedEdges
23:      increment nodeDegrees[selectedEdge.A]
24:      increment nodeDegrees[selectedEdge.B]
25:      increment edgeCount
26:      disjointSet.union(selectedEdge.A, selectedEdge.B)
27:    end if
28:  end while
29:  tour  $\leftarrow$  CONSTRUCTTOUR(nodes, adjacencyList) ▷ Construct tour using adjacency list
30:  return tour
31: end procedure
```

4.3.3 Examples

To better visualise and understand MF, we can look at how the algorithm performs in different scenarios. Below are the final solutions of various instances computed using the MF algorithm.

Best Case

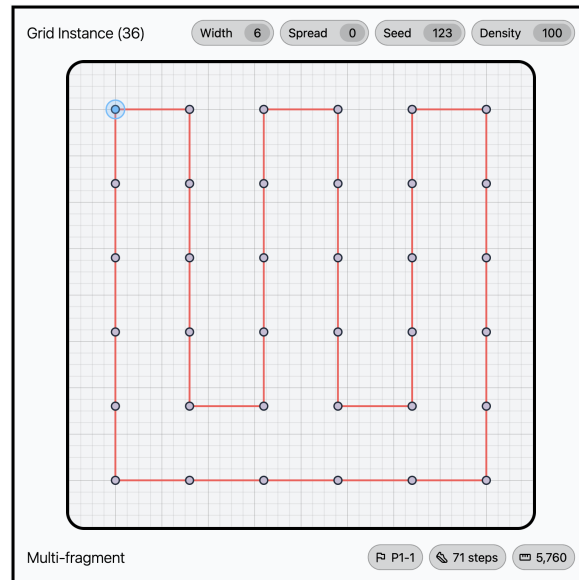


Figure 4.16: The Multi-fragment solution of a grid TSP instance with 36 nodes

We can observe that the best case instance for the MF algorithm is a uniformly distributed grid of nodes. MF often finds a near optimal solution for these instances due to all edges in the optimal tour having an equal length. These can be referred to as unit edges. This results in the algorithm following a serpentine path around the grid due to the logical constraints. We can also find that the algorithm performs better for even-width grid instances over ones with an odd-width. This is because a grid graph $G_{m,n}$ has a Hamiltonian circuit, consisting of unit edges, if and only if $m \times n$ is even and $m, n > 1$ (Itai et al., 1982). As a consequence, if $m \times n$ is odd the MF algorithm is mathematically forced to make a large jump at the end of the tour. In Figure 4.16, since m and $n = 6$ we know there is a Hamiltonian circuit of just unit edges. We can observe that the MF algorithm finds this circuit.

General Case

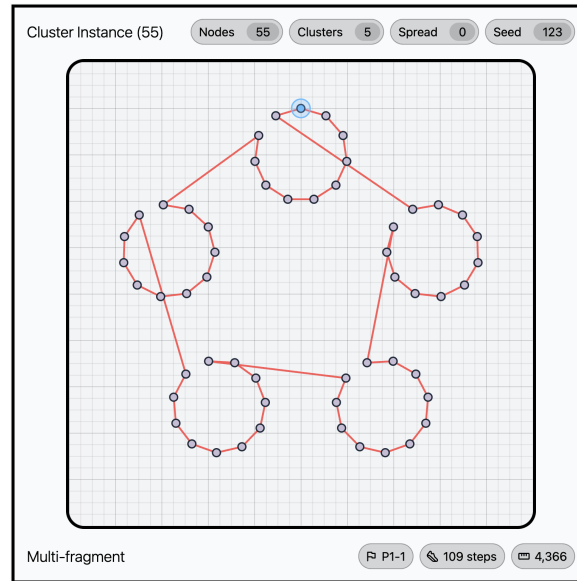


Figure 4.17: The Multi-fragment solution of a cluster TSP instance with 55 nodes

For non-grid instances, however, MF struggles to close the tour efficiently. Instead, the algorithm favours traversing localised segments of nodes due to prioritising smaller edges. By the end of the tour, with very few edges that maintain the logical constraints remaining, the algorithm is forced to frantically connect these segments. With the cluster instance in Figure 4.17, we can observe how the algorithm builds the tour within the local clusters with little forethought of how to traverse between them. This causes the algorithm to connect the clusters with inefficient jumps that often overlap with existing edges.

Real-World Case

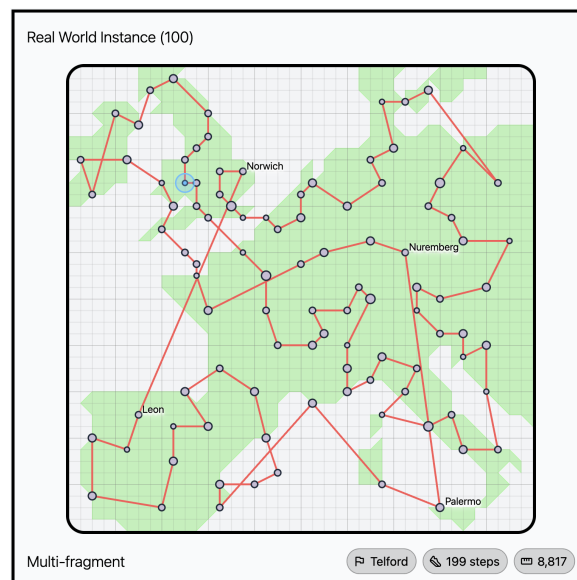


Figure 4.18: The Multi-fragment solution of the real-world TSP instance

These drawbacks are evident in the MF solution of the real-world instance in Figure 4.18. The algorithm performs well in localised clusters, like the UK and around Spain. However, as the number of available valid edges decreases throughout the run the algorithm is forced to take large jumps, like between *Norwich*

and *Leon* and between *Nuremberg* and *Palermo*. This further demonstrates how greedy algorithms fail to anticipate the structural requirements of an instance, focusing instead on immediate local gains.

4.3.4 Evaluation

Time Growth

To derive the time growth theoretically, we can look at the MF algorithm. We start by computing every possible edge between all nodes in the instance. For an instance with n nodes, every node can visit every other node ($n - 1$ nodes). Therefore, the number of possible edges $|E|$ can be found using:

$$|E| = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} \quad (4.3)$$

For Big O notation, we can simplify Equation 4.3 to $O(n^2)$. To efficiently iterate through the edges, we must sort the edges by distance from shortest to largest. Sorting a list with E elements takes $O(E \log E)$ time. By substituting $E = n^2$, we get $O(n^2 \log n^2)$. By apply logarithmic rules, which state that $\log n^2 \equiv 2 \log n$, we get $O(n^2 \cdot 2 \log n)$.

Finally, the algorithm must iterate through these edges and select the edges. At worst, this takes n^2 iterations, therefore we get $O(n^2)$. Combining these three phases of the MF algorithm, we get:

$$O(n^2) + O(n^2 \cdot 2 \log n) + O(n^2) = O(3n^2 \cdot 2 \log n) = O(n^2 \log n) \quad (4.4)$$

n	Median Time (ms)
10	0.0
100	0.3
250	2.2
500	5.3
750	11.3
1,000	21.2

Table 4.3: The median computation times for different values of n between 10 and 1,000 using the Multi-fragment algorithm. For full results, see Table C.3.

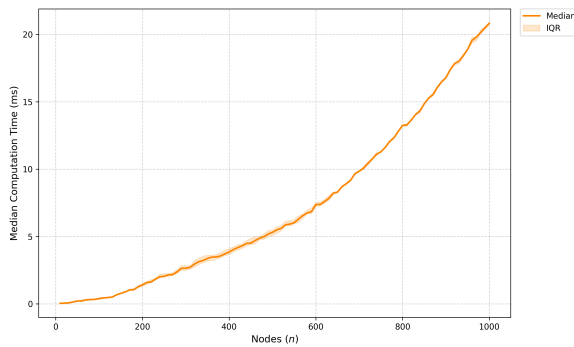


Figure 4.19: A graph showing the median computation time against nodes (n).

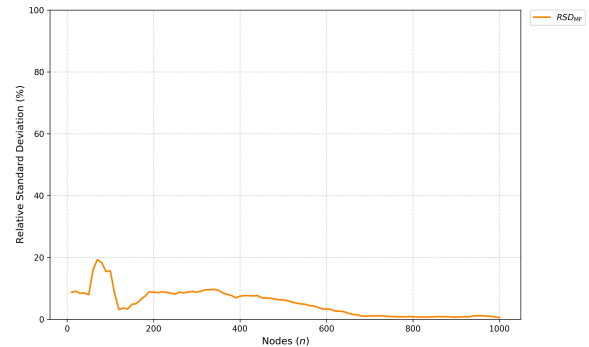


Figure 4.20: A graph showing the relative standard deviation in computation time against nodes (n).

Using empirical data from Table 4.3, we can observe how the median computation time for the MF algorithm increases with n . From 500 to 1,000 nodes, the table shows how the computation time almost quadruples. Equivalently, Figure 4.19 shows a clear polynomial time growth. Despite a slight variation in the results for $n \leq 100$ shown in Figure 4.20, we can see the RSD remains below 20% throughout.

These experimental observations match our theoretical analysis of the MF algorithm. We can therefore determine the Big O notation of the multi-fragment heuristic to be $O(n^2 \log n)$.

Tour Quality

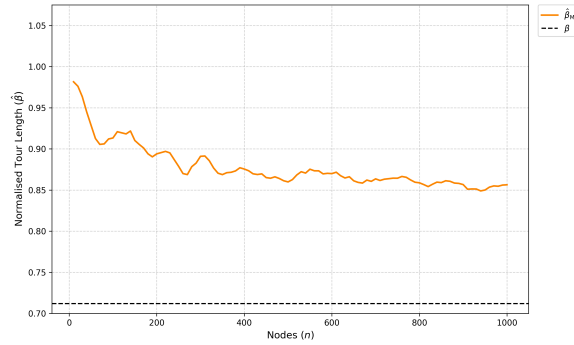


Figure 4.21: A graph showing the normalised tour length ($\hat{\beta}_{MF}$) against nodes (n).

Figure 4.21 shows the normalised tour length for the MF algorithm ($\hat{\beta}_{MF}$) for $10 \leq n \leq 1,000$. Although $\hat{\beta}_{MF}$ starts high, it decreases significantly early on, showing that the algorithm performs better on denser instances with more edges to choose from. We can note that as n increases, $\hat{\beta}_{MF}$ tends towards a value of around 0.86. When compared to the theoretical BHH constant ($\beta \approx 0.7120$), we can determine an optimality gap of $\delta = 20.8\%$.

4.3.5 Conclusion

We can recognise that MF is the best performing greedy heuristic. The algorithm uses its global context to overcome the main pitfall of NN, being limited by a poorly selected start node. Furthermore, it improves on RNN (for larger instances) by prioritising the tour structure for local clusters of nodes.

However, we must consider the computational cost of generating, sorting and examining every possible edge of an instance. Additionally, despite the algorithm's global context, we can still observe how an early edge selection can harm the final structure of the tour. These potential initial slip ups still impair the final tours produced by MF, as the algorithm lacks the ability to amend selected edges.

5 Vertex Insertion Heuristics

5.1 Base Algorithm

5.1.1 Algorithm Explained

So far, we have built tours that close the loop of the tour on the very last iteration. Insertion heuristics, however, start with a small sub-tour and subsequently insert a node in between two other nodes with every iteration, expanding like a rubber band. The node to be inserted depends on the node selection method of the heuristic's algorithm. The three node selection methods we will focus on are Nearest Insertion, Farthest Insertion, and Cheapest Insertion.

Put formally, let $N = n_1, \dots, n_m$ be the set of all nodes. We start the tour as a small sub-tour of $T = \{n_1, n_2, n_1\}$. Therefore, the set of unvisited nodes can be defined as, $U = N \setminus \{n_1, n_2\}$. For every node, u_x , in U , we use the node selection method to determine when u_x is inserted into T , until $U = \emptyset$.

5.1.2 Pseudocode Representation

Below is the pseudocode implementation of the Base Insertion algorithm. The parameters of the function are the list of all nodes, **nodes**, and a function used to select nodes, **selectionMethod**. The function runs the base Insertion algorithm on the given list of nodes, and returns an ordered list of nodes, **tour**. See Sections 5.2.2, 5.4.2 and 5.3.2 for Pseudocode representations of selection methods which can be provided for the **selectionMethod**.

Algorithm 4 Base Insertion Algorithm

```
1: procedure RUNINSERTION(nodes, selectionMethod)
2:   firstNode  $\leftarrow$  nodes[0]
3:   secondNode  $\leftarrow$  nodes[1]
4:   tour  $\leftarrow$  [firstNode, secondNode, firstNode] ▷ Initialise a sub-tour
5:   unvisited  $\leftarrow$  nodes
6:   remove firstNode from unvisited
7:   remove secondNode from unvisited
8:   while unvisited is not  $\emptyset$  do
9:     (nextNode, targetIndex)  $\leftarrow$  SELECTIONMETHOD(unvisited, tour) ▷ Use selection method
10:    insert nextNode into tour at targetIndex
11:    remove nextNode from unvisited
12:  end while
13:  return tour
14: end procedure
```

5.1.3 Time Growth

The base insertion algorithm starts with a small sub-tour, and performs one insertion per iteration until all n nodes have been added. Therefore, the total time growth of a specific insertion heuristic depends on the cost of selecting each node and determining where it should be inserted. Since the selection process is repeated for all remaining nodes, the overall complexity is obtained by combining the cost of one selection step with the n iterations of the base algorithm.

5.2 Nearest Insertion

5.2.1 Algorithm Explained

The Nearest Insertion (NI) selection method adapts the principles of the Nearest Neighbour (NN) heuristic to an incremental construction process. While NN seeks the node that has the minimum distance to the last node added to the sub-tour, NI identifies the unvisited node that has the minimum distance to any node in the sub-tour.

To do this, the selection algorithm iterates over each unvisited node and then loops through every node in the sub-tour. It then calculates the distance between the two, maintaining a reference to the global minimum distance. Once all pairs of nodes have been evaluated, the node yielding the minimum distance is selected. The algorithm also records the position of the closest sub-tour node so that the selected node can be inserted into the sub-tour at the appropriate position.

Put formally, let U be the set of unvisited nodes and let T be the current sub-tour. For each unvisited node $u \in U$, we compute its distance to every node $v \in T$. We then find the pair (u, v) that minimises this distance out of all pairs. The selected node u is therefore the unvisited node nearest to any node already in the sub-tour. Once selected, u is then inserted into T immediately after the sub-tour node v . This process is repeated until $U = \emptyset$, at which point T is a complete tour.

5.2.2 Pseudocode Representation

Below is the pseudocode implementation of the Nearest Insertion selection algorithm. The parameters of the function are the list of unvisited nodes, **unvisitedNodes**, and the list of nodes already in the sub-tour, **tour**. The function returns the unvisited node that is nearest to any node already in the sub-tour, **nearestNode**, with the index at which it should be inserted, **bestIndex**.

Algorithm 5 Nearest Insertion Selection Algorithm

```
1: procedure SELECTNEARESTNODE(unvisitedNodes, tour)
2:   nearestNode  $\leftarrow$  null
3:   bestIndex  $\leftarrow$  -1
4:   minimumDistance  $\leftarrow$   $\infty$ 
5:   for each node in unvisitedNodes do
6:     for each (tourNode, index) in tour do                                 $\triangleright$  Find the nearest sub-tour node
7:       dist  $\leftarrow$  HYPOT(node, tourNode)
8:       if dist < minimumDistance then                                     $\triangleright$  Update nearest pointers
9:         nearestNode  $\leftarrow$  node
10:        bestIndex  $\leftarrow$  index + 1
11:        minimumDistance  $\leftarrow$  dist
12:      end if
13:    end for
14:  end for
15:  return (nearestNode, bestIndex)
16: end procedure
```

5.2.3 Examples

To better visualise and understand NI, we can look at how the algorithm performs in different scenarios. Below are the final solutions of various instances computed using the NI algorithm.

Best Case

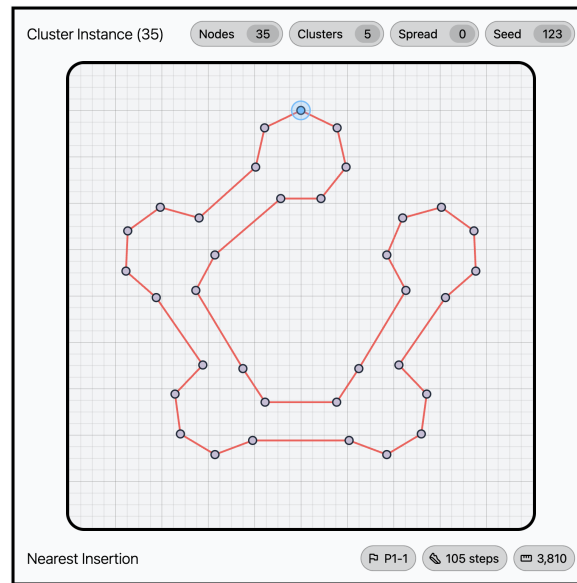


Figure 5.1: The Nearest Insertion solution of a cluster TSP instance with 35 nodes and 5 clusters

The tour shown in Figure 5.1 demonstrates how the algorithm progressively expands. We can observe how it begins at *P1-1* and adds all nodes in the first cluster before proceeding to the next. NI continues to explore each cluster completely before moving on to another, creating a clean traversal of each cluster. This produces a visibly structured tour with no backtracking or overlapping edges because the nearest unvisited node is usually in the same local cluster as the current sub-tour. The tour shown in Figure 5.1 is especially good because the clusters are evenly separated with no spread.

General Case

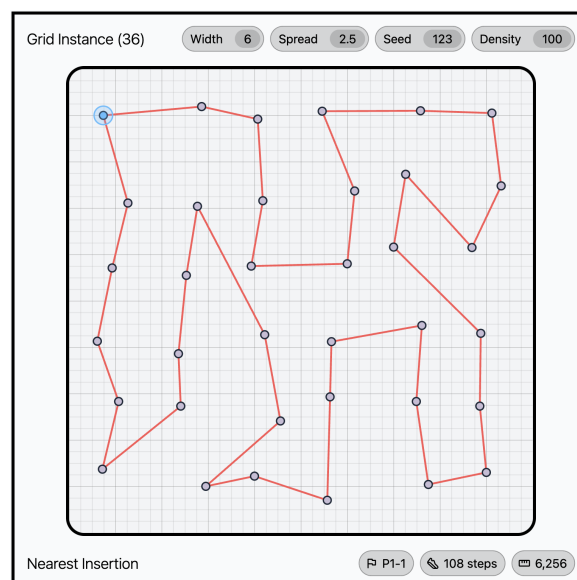


Figure 5.2: The Nearest Insertion solution of a grid TSP instance with 25 nodes

Although the tour seen in Figure 5.2 appears visually coherent, this is a consequence of the insertion framework itself. By gradually expanding a single sub-tour, NI often avoids the obvious overlaps or detours seen in other heuristics. However, this visual neatness can be misleading. Because the NI algorithm only

considers which unvisited node is nearest to the current sub-tour, rather than which insertion would minimise the overall tour structure, it can still produce an inefficient overall structure.

Real-World Case

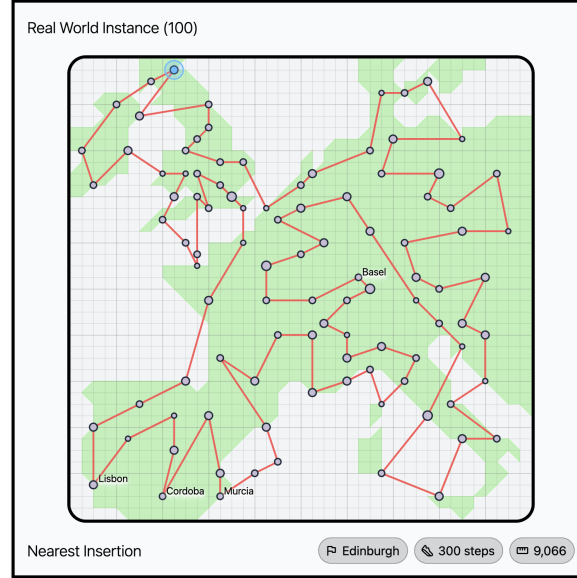


Figure 5.3: The Nearest Insertion solution of the real-world TSP instance

In Figure 5.3 we can see another visually coherent tour produced by NI. The route contains no overlapping edges or jumps to close the tour. However, we can definitely observe an inefficient traversal for the center of the instance, around *Basel*. We can also observe how the sub-tour expands in a branching manner, reaching out for individual cities as it expands. This is most noticeable as the tour reaches out to *Lisbon*, *Cordoba* and *Murcia*.

5.2.4 Evaluation

Time Growth

By analysing the logical steps of the NI selection method alongside the base insertion framework, we can derive the overall time growth of the NI heuristic. As discussed in Section 5.1.3, the base insertion algorithm performs one insertion per iteration until all n nodes have been added, so the overall complexity depends on the cost of each selection step.

The selection logic suggests that we must compare each unvisited node with every node already in the sub-tour. However, the Java implementation optimises this process by using a cache of the minimum distances from each unvisited node to the current sub-tour. This allows the next node to be selected in linear time. The best insertion position and cache updates can then also be determined in linear time. Therefore, each iteration runs in $O(n)$ time, and since the insertion process is repeated for all n nodes, the implemented NI heuristic exhibits an overall time growth of $O(n^2)$.

n	Median Time (ms)
10	0.0
100	0.2
250	0.5
500	1.7
750	2.9
1,000	3.7

Table 5.1: The median computation times for different values of n between 10 and 1,000 using the Nearest Insertion algorithm. For full results, see Table C.4.

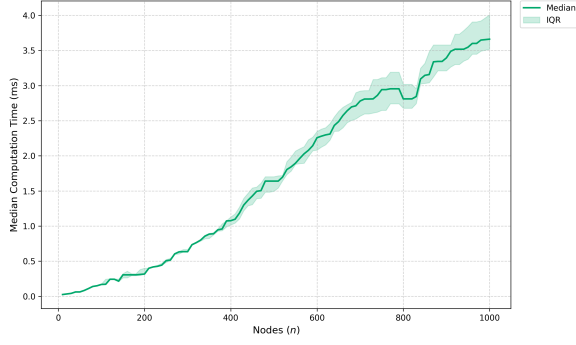


Figure 5.4: A graph showing the median computation time against nodes (n).

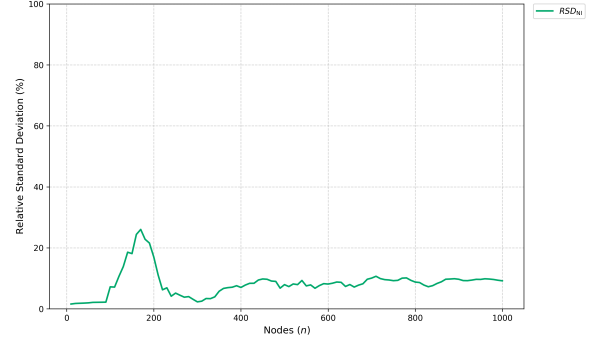


Figure 5.5: A graph showing the relative standard deviation in computation time against nodes (n).

From both Table 5.1 and Figure 5.4, we can see how the NI heuristic performs for $10 \leq n \leq 1,000$. We can observe a polynomial time growth in the graph. While Figure 5.5 features a clear bump in RSD for $100 \leq n \leq 250$, we can attribute this to the exceptionally low computation time of less than 0.5ms. We can take our theoretical analysis alongside our empirical evaluation to conclude that the NI heuristic has a time growth complexity of $O(n^2)$.

Tour Quality

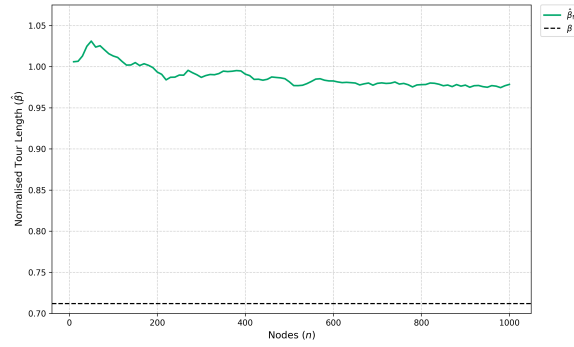


Figure 5.6: A graph showing the normalised tour length ($\hat{\beta}_{\text{NI}}$) against nodes (n).

Figure 5.6 shows the normalised tour length ($\hat{\beta}_{\text{NI}}$) as the number of nodes n increases. We can observe a relatively flat plot that settles at around 0.98 for $n = 1,000$. When compared to the theoretical constant ($\beta \approx 0.7120$), we can calculate an optimality gap for NI of $\delta = 37.6\%$.

5.2.5 Conclusion

Overall, the Nearest Insertion heuristic matches the time complexity of heuristics such as NN, despite using a more complex construction approach. However, its selection logic still prioritises local decisions over global tour structure, which often leads to inefficient routes.

5.3 Cheapest Insertion

5.3.1 Algorithm Explained

The Cheapest Insertion (CI) selection method extends the base insertion algorithm by selecting the node that causes the smallest increase in the length of the current sub-tour upon insertion. Unlike Nearest Insertion, which chooses the unvisited node closest to any node already in the sub-tour, CI evaluates the cost of inserting each unvisited node between every pair of consecutive nodes already in the sub-tour.

To achieve this, the selection algorithm iterates over each unvisited node and every edge currently in the sub-tour. For each possible insertion, it calculates the increase in tour length that would result from replacing an edge (v_i, v_{i+1}) with two new edges (v_i, u) and (u, v_{i+1}) . The node and position yielding the smallest increase are then selected.

Put formally, let U be the set of unvisited nodes and let $T = (v_1, \dots, v_k, v_1)$ be the current sub-tour. For each unvisited node $u \in U$, and for each pair of consecutive nodes (v_i, v_{i+1}) in T , we can calculate the cost of insertion:

$$\Delta(u, i) = d(v_i, u) + d(u, v_{i+1}) - d(v_i, v_{i+1}) \quad (5.1)$$

Where $d(x, y)$ denotes the Euclidean distance between two nodes x and y . We then select the pair (u, i) that minimises $\Delta(u, i)$ out of all valid insertions. The node u is then inserted between v_i and v_{i+1} . This process is repeated until $U = \emptyset$, at which point T is a complete tour.

5.3.2 Pseudocode Representation

Below is the pseudocode implementation of the Cheapest Insertion selection algorithm. The parameters of the function are the list of unvisited nodes, **unvisitedNodes**, and the list of nodes already in the sub-tour, **tour**. The function returns the unvisited node that is cheapest to add between any of the nodes already in the sub-tour, **cheapestNode**, with the index at which it should be inserted, **bestIndex**.

Algorithm 6 Cheapest Insertion Selection Algorithm

```
1: procedure SELECTCHEAPESTNODE(unvisitedNodes, tour)
2:   cheapestNode  $\leftarrow$  null
3:   bestIndex  $\leftarrow$  -1
4:   minimumIncrease  $\leftarrow$   $\infty$ 
5:   for each nodeA in unvisitedNodes do  $\triangleright$  Find the cheapest node to insert between tour nodes
6:     for  $i \leftarrow 0$  to length(tour) - 1 do
7:       nodeB  $\leftarrow$  tour[ $i$ ]
8:       nodeC  $\leftarrow$  tour[( $i + 1$ ) mod LENGTH(tour)]
9:       distAB  $\leftarrow$  HYPOT(nodeA, nodeB)
10:      distAC  $\leftarrow$  HYPOT(nodeA, nodeC)
11:      distBC  $\leftarrow$  HYPOT(nodeB, nodeC)
12:      increase  $\leftarrow$  distAB + distAC - distBC  $\triangleright$  Calculate insertion cost ( $\Delta$ )
13:      if increase < minimumIncrease then  $\triangleright$  Update cheapest pointers
14:        cheapestNode  $\leftarrow$  nodeA
15:        bestIndex  $\leftarrow$   $i + 1$ 
16:        minimumIncrease  $\leftarrow$  increase
17:      end if
18:    end for
19:  end for
20:  return (cheapestNode, bestIndex)
21: end procedure
```

5.3.3 Examples

To better visualise and understand CI, we can look at how the algorithm performs in different scenarios. Below are the final solutions of various instances computed using the CI algorithm.

Best Case

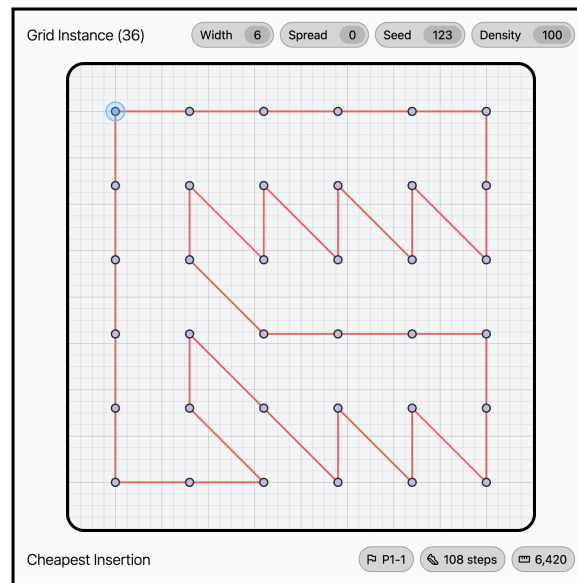


Figure 5.7: The Cheapest Insertion solution of a grid TSP instance with 25 nodes

From Figure 5.7 we can observe that the CI algorithm performs well on a structured grid instance. The structured grid means many potential insertions along the sub-tour boundary produce only a small increase in the total distance. This allows the sub-tour to expand in a controlled and relatively compact manner. As a result, the final tour appears well organised, with no unnecessarily long edges.

General Case

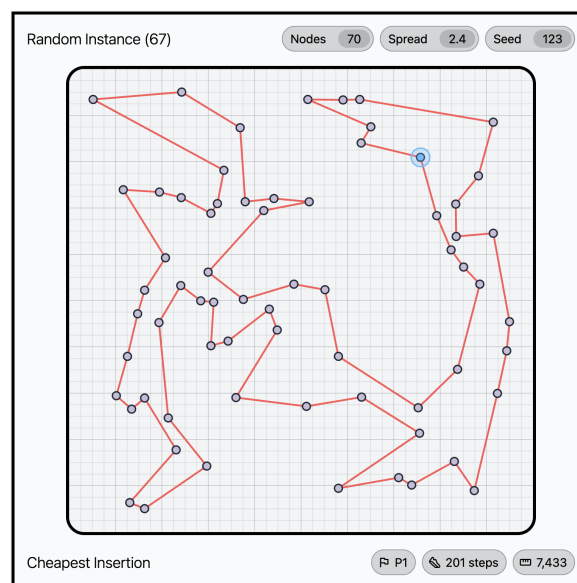


Figure 5.8: The Cheapest Insertion solution of a random TSP instance with 67 nodes

The tour shown in Figure 5.8 demonstrates the more typical behaviour of the CI heuristic. Since the algorithm still selects the insertion with the smallest immediate increase in tour length, it tends to favour locally inexpensive additions. In a random instance, this often leads to coherent local structure across separate groups of nodes. However, these individually cheap choices do not necessarily support the best global route. As a result, the final tour can still contain inefficient connections, despite appearing more deliberate than tours produced by NI.

Real-World Case

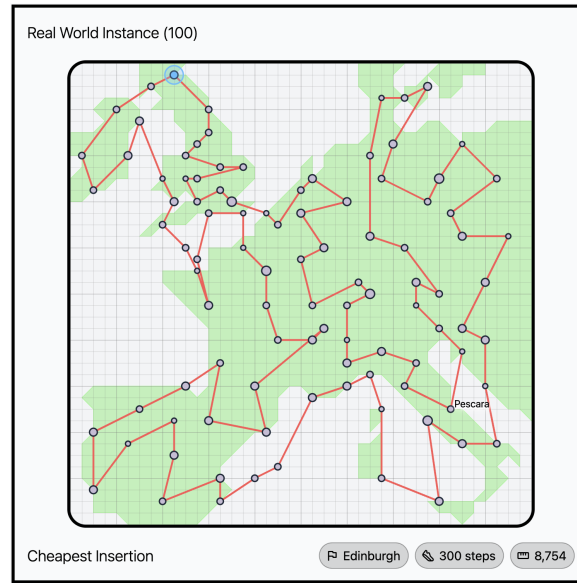


Figure 5.9: The Cheapest Insertion solution of the real-world TSP instance

In Figure 5.9, the CI heuristic produces a tour of the real-world instance that appears structured and well controlled. Some of the impulsive growth seen in NI is reduced because the selection method evaluates the increase in tour length before every insertion. However, we can see that evaluating local insertion cost is not capable of global planning. Although the tour contains fewer visibly erratic branches, inefficiencies remain in the wider structure of the route. For example, the tour takes a long route around central Europe down to *Pescara*. This shows that a sequence of individually cheap insertions does not necessarily lead to an optimal tour.

5.3.4 Evaluation

Time Growth

By analysing the CI selection logic alongside the base insertion framework, we can derive the overall time growth of the CI heuristic. As described in Section 5.1.3, the base insertion algorithm performs one insertion per iteration until all n nodes have been added. The overall complexity, therefore, depends on the cost of each selection step.

For each iteration, the CI selection logic evaluates every unvisited node and every possible insertion position in the current sub-tour to determine the minimum increase in tour length. This gives a quadratic $O(n^2)$ complexity per selection step. Since this insertion process is repeated for all n nodes, the overall time growth of the CI heuristic is cubic, giving $O(n^3)$.

n	Median Time (ms)
10	0.0
100	0.8
250	5.8
500	43.4
750	148.6
1,000	351.8

Table 5.2: The median computation times for different values of n between 10 and 1,000 using the Cheapest Insertion algorithm. For full results, see Table C.5.

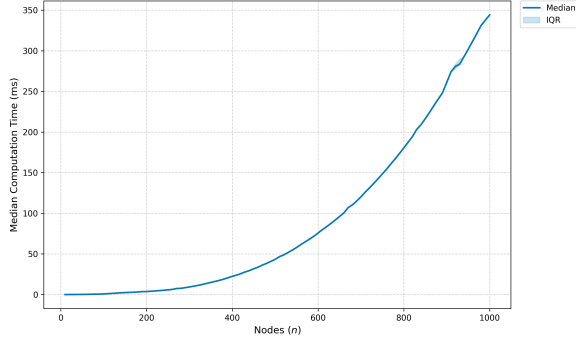


Figure 5.10: A graph showing the median computation time against nodes (n).

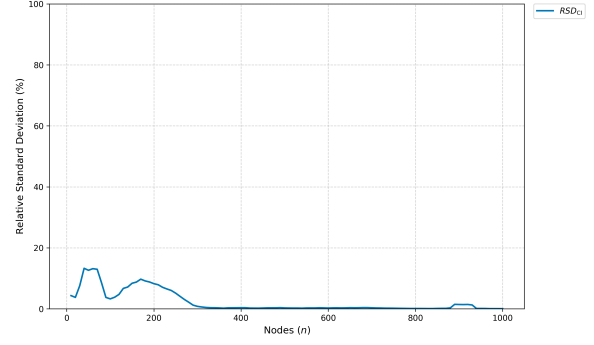


Figure 5.11: A graph showing the relative standard deviation in computation time against nodes (n).

The data shown in Table 5.2 and Figure 5.10 support this observation. The graph shows a strong polynomial time growth curve, reaching a computation time of over 300ms for $n \geq 950$. We can perform a power-law analysis of two of the data points ($n = 250$ and 1,000), to find an exponential growth factor of approximately 2.96 ($\log_4 60.66$). Figure 5.11 supports this analysis, showing a consistently low RSD. We can combine this power-law analysis with the theoretical evaluation to conclude that the CI heuristic has a time growth complexity of $O(n^3)$.

Tour Quality

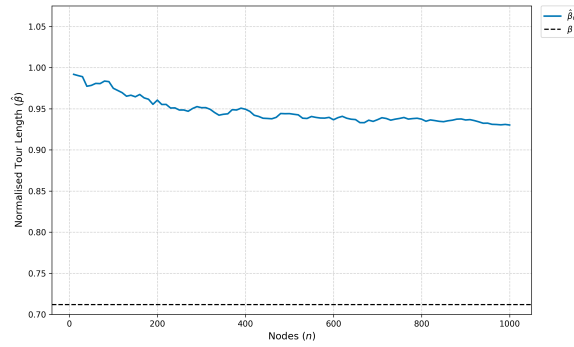


Figure 5.12: A graph showing the normalised tour length ($\hat{\beta}_{CI}$) against nodes (n).

We can see the normalised tour length $\hat{\beta}_{CI}$ in Figure 5.12. The results show a stable plot that tends towards 0.93 as n increase. This consistency could be traced to the selection methods exhaustive search for the cheapest node to insert. This represents an optimality gap of $\delta = 30.6\%$ when compared with the theoretical constant $\beta \approx 0.7120$.

5.3.5 Conclusion

Overall, the Cheapest Insertion heuristic produces more controlled tours than the more naive NI approach, reducing the erratic branching seen in NI solutions. However, this improvement comes at a substantial computation cost, as added selection logic increases the implementation's time growth from $O(n^2)$ to $O(n^3)$. Consequently, at $n = 1,000$, CI takes approximately 95 times longer than NI. Despite this additional cost, the heuristic still prioritises marginal local gains and therefore does not always produce the best global tour structure.

5.4 Farthest Insertion

5.4.1 Algorithm Explained

So far, each heuristic has aimed to build the tour using the smallest edges available. For Farthest Insertion (FI), the opposite is true. Counterintuitively, the algorithm works by selecting the unvisited node that is farthest from its nearest node already in the sub-tour. To do this, the selection algorithm loops through each unvisited node and then loops through each node in the tour. It then subsequently calculates the distance between the two, maintaining a reference to the global maximum distance. Once all node pairs have been evaluated, the node yielding the maximum distance is returned to the base insertion algorithm to be inserted into the tour.

Put formally, let U be the set of unvisited nodes and let T be the current sub-tour. For each unvisited node $u \in U$, and for each node $v \in T$, we compute the Euclidean distance $d(u, v)$. For each u , we define its distance to the sub-tour as:

$$\min_{v \in T} d(u, v) \quad (5.2)$$

We then select the node u such that this value is maximised over all $u \in U$. That is, FI chooses the unvisited node that is farthest from its nearest node in the current sub-tour. This selected node is then inserted into T immediately after the corresponding nearest sub-tour node. This process is repeated until $U = \emptyset$, at which point T is a complete tour.

5.4.2 Pseudocode Representation

Below is the pseudocode implementation of the Farthest Insertion selection algorithm. The parameters of the function are the list of unvisited nodes, `unvisitedNodes`, and the list of nodes already in the sub-tour, `tour`. The function returns the unvisited node that is farthest to any node already in the sub-tour, `farthestNode`, with the index at which it should be inserted, `bestIndex`.

Algorithm 7 Farthest Insertion Selection Algorithm

```
1: procedure SELECTFARTHESTNODE(unvisitedNodes, tour)
2:   farthestNode  $\leftarrow$  null
3:   bestIndex  $\leftarrow$  -1
4:   maximumDistance  $\leftarrow$   $-\infty$ 
5:   for each node in unvisitedNodes do
6:     minDistToTour  $\leftarrow$   $\infty$ 
7:     minIndex  $\leftarrow$  -1
8:     for each (tourNode, index) in tour do                                 $\triangleright$  Find the nearest sub-tour node
9:       dist  $\leftarrow$  HYPOT(node, tourNode)
10:      if dist  $<$  minDistToTour then
11:        minDistToTour  $\leftarrow$  dist
12:        minIndex  $\leftarrow$  index
13:      end if
14:    end for
15:    if minDistToTour  $>$  maximumDistance then                                 $\triangleright$  Choose the farthest
16:      farthestNode  $\leftarrow$  node
17:      bestIndex  $\leftarrow$  minIndex + 1
18:      maximumDistance  $\leftarrow$  minDistToTour
19:    end if
20:  end for
21:  return (farthestNode, bestIndex)
22: end procedure
```

5.4.3 Examples

To better visualise and understand FI, we can look at how the algorithm performs in different scenarios. Below are the final solutions of various instances computed using the FI algorithm.

Best Case

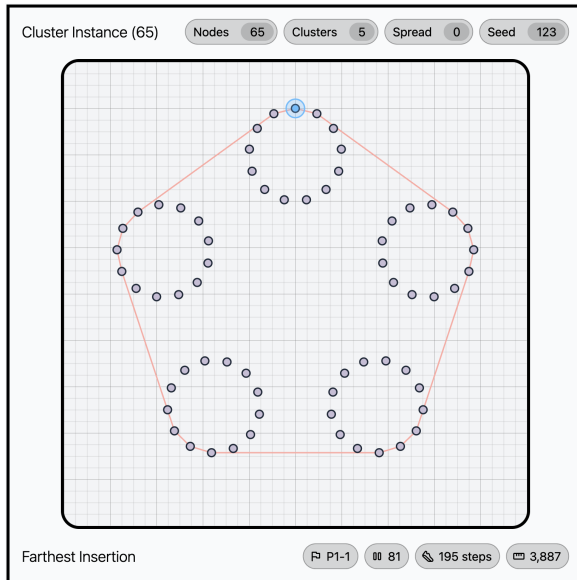


Figure 5.13: Step 81 of the Farthest Insertion solution of a cluster TSP instance with 65 nodes

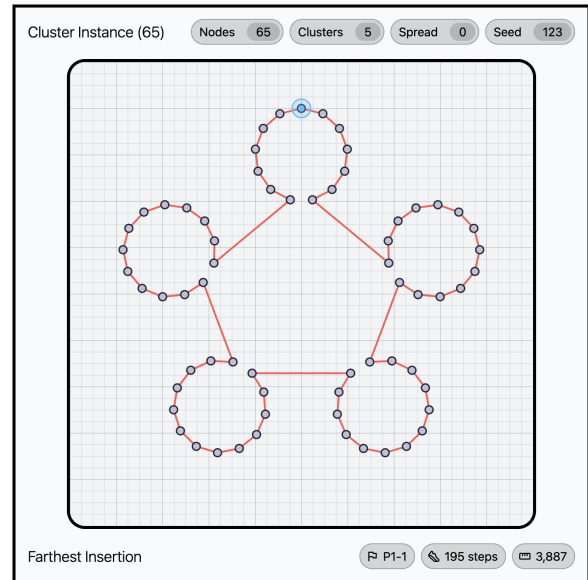


Figure 5.14: The Farthest Insertion solution of a cluster TSP instance with 65 nodes

Figures 5.13 and 5.14 show how the FI heuristic performs on a cluster instance. In the instance, nodes in clusters not yet incorporated lie far from the initial sub-tour. As a result, FI selects nodes from distant clusters early in the construction process. In Figure 5.13 we can observe how the heuristic begins by wrapping around the outer structure of the instance, forming a broad framework that captures all five clusters. We can then see how the remaining nodes are filled in from here. This gives the algorithm a strong global awareness early on. Consequently, the final tour shown in Figure 5.14 is highly structured, with efficient connections between clusters and no large detours.

General Case

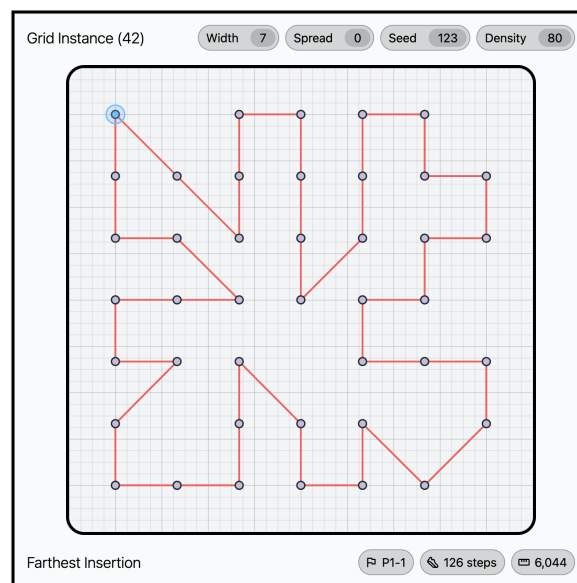


Figure 5.15: The Farthest Insertion solution of a grid TSP instance with 42 nodes

The tour shown in Figure 5.15 demonstrates how the FI selection logic behaves on a grid instance. As with the cluster instance, the algorithm begins by enclosing the broader structure of the instance before

reaching inward towards the remaining nodes. This produces a route that appears orderly, but the early emphasis on distant nodes can lead to awkward internal connections once the outer framework has been established. As a result, although the final tour is visually coherent, it may still contain inefficient interior edges.

Real-World Case

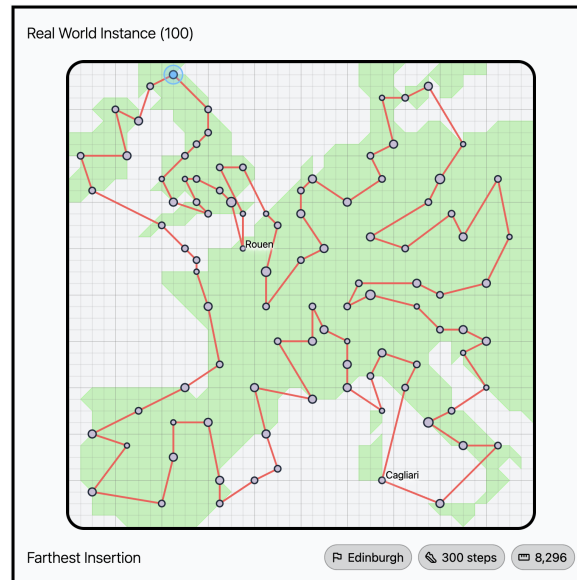


Figure 5.16: The Farthest Insertion solution of the real-world TSP instance

In Figure 5.16, we can observe that FI produces a strong tour of the real-world instance. By prioritising cities that are farthest from the current sub-tour, the heuristic expands quickly across the wider geography of the instance. This prevents the tour from growing through narrow local branches. This allows the algorithm to establish a broad structure early on, which reduces the fragmented growth and purely local insertion choices seen in NI and CI. Although the final tour is not optimal, with inefficient traversals to *Rouen* or *Cagliari*, it is visually coherent and makes effective use of the major regional groupings within the instance.

5.4.4 Evaluation

Time Growth

We can analyse the logical steps taken by the FI selection logic with the base insertion framework, to derive the overall time growth of the FI heuristic. As outlined in Section 5.1.3, the base insertion algorithm performs one insertion per iteration until all n nodes have been added, so the overall complexity depends on the cost of each selection step.

The selection logic suggests that we must compare each unvisited node with every node already in the sub-tour. However, the Java implementation optimises this process by using a cache of the maximum distances from each unvisited node to the current sub-tour. We can therefore select the next node in linear time. The best insertion position and cache updates can also be determined in linear time. Therefore, each iteration runs in $O(n)$ time, and since the insertion process is repeated for all n nodes, the implemented FI heuristic exhibits an overall time growth of $O(n^2)$.

n	Median Time (ms)
10	0.0
100	0.1
250	0.3
500	1.4
750	2.7
1,000	3.5

Table 5.3: The median computation times for different values of n between 10 and 1,000 using the Farthest Insertion algorithm. For full results, see Table C.6.

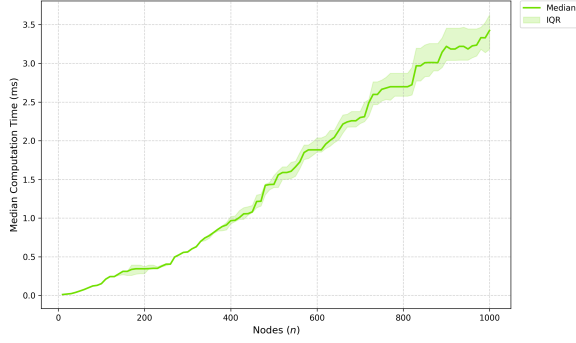


Figure 5.17: A graph showing the median computation time against nodes (n).

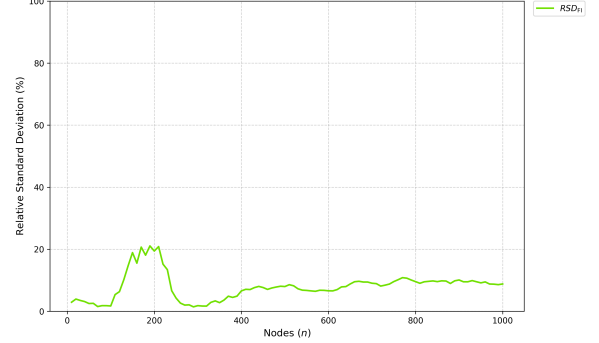


Figure 5.18: A graph showing the relative standard deviation in computation time against nodes (n).

From both Table 5.3 and Figure 5.17, we can see how the FI heuristic performs for $10 \leq n \leq 1,000$. We can observe a polynomial time growth in the graph. While Figure 5.18 features a clear bump in RSD for $100 \leq n \leq 250$, we can attribute this to the exceptionally low computation time of less than 0.3ms. We can combine our theoretical analysis alongside our empirical evaluation to derive that the FI heuristic has a time growth complexity of $O(n^2)$.

Tour Quality

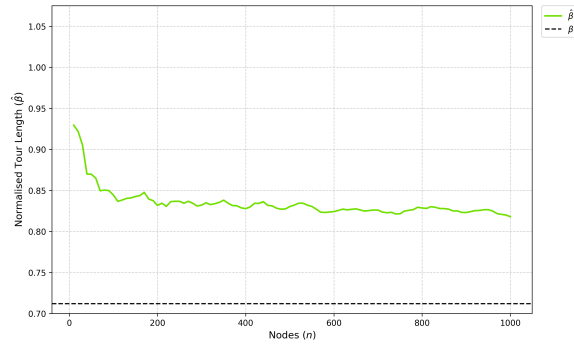


Figure 5.19: A graph showing the normalised tour length ($\hat{\beta}_{\text{FI}}$) against nodes (n).

Figure 5.19 shows the normalised tour length $\hat{\beta}_{\text{FI}}$ for $10 \leq n \leq 1,000$. We can observe a steep decrease for $n \leq 100$. On smaller instances, the selection logic can make a few early choices disproportionately influential, leading to less efficient tours. As the number of nodes increases, however, the instance becomes denser and the heuristic has more opportunity to establish a broad outer structure before filling in the remaining interior nodes. The data then settles at approximately 0.82. When compared with the theoretical constant ($\beta \approx 0.7120$), this represents an optimality gap of just $\delta = 16.6$.

5.4.5 Conclusion

Overall, the Farthest Insertion heuristic is able to consistently construct tours with a global structure. By expanding rapidly across the instance, the heuristic gains a global overview of the problem and can make more effective selections than NI or CI. Furthermore, it maintains a quadratic time growth of $O(n^2)$, making it both efficient and competitive. However, FI still remains a constructions heuristic driven by incremental local decisions.

6 Approximation Heuristics

6.1 Christofides-Serdyukov Heuristic

6.1.1 Algorithm Explained

The Christofides-Serdyukov (Christofides or CS) heuristic is a bounded approximation algorithm for the metric TSP. Unlike greedy or insertion heuristics which construct tours directly, Christofides builds a sequence of graph structures whose properties guarantee that the final tour is no more than $\frac{3}{2}$ times the optimal tour length. The algorithm was published by Christofides in 1976, and was independently discovered by Serdyukov in the USSR and later published in 1978 (van Bevern and Slugina, 2020). Hence, Christofides-Serdyukov.

The algorithm begins by constructing a Minimum Spanning Tree (MST) over all nodes in the instance. Since every tree contains an even number of odd-degree vertices, the odd-degree nodes of the MST can be paired using minimum-weight perfect matching. Combining the MST with these matching edges produces a connected multigraph in which every vertex has even degree, and therefore one that admits an Eulerian circuit. The algorithm constructs this circuit using a stack-based traversal. It repeatedly follows unused edges, removing them as they are traversed, and backtracks whenever no further unused edge is available. Finally, repeated visits to the same node are removed by shortcutting the circuit, yielding a Hamiltonian cycle (Christofides, 1976).

Put formally, let N be the set of all nodes in a complete metric graph $G = (N, E)$, where each edge $e \in E$ has weight $w(e)$. First, compute a minimum spanning tree M of G . Let $O \subseteq N$ be the set of vertices with odd degree in M . Next, compute a minimum-weight perfect matching P on the subgraph induced by O . The multigraph $H = M \cup P$ is then connected and all of its vertices consequently have an even degree, so it admits an Eulerian circuit C . By traversing C and shortcutting previously visited vertices, we obtain a Hamiltonian cycle T . Since the graph is metric, shortcutting does not increase the total tour length, and T is the final Christofides tour.

6.1.2 Pseudocode Representation

Below is the pseudocode implementation of the Christofides-Serdyukov algorithm. The parameters of the function are the start (and end) node, `startNode`, and the list of all nodes, `nodes`. The function runs the Christofides algorithm on the given list of nodes, and returns an ordered list of nodes, `tour`.

Algorithm 8 Christofides-Serdyukov Algorithm

```
1: procedure RUNCHRISTOFIDES(nodes, startNode)
2:   mst  $\leftarrow$  NEWMINIMUMSPANNINGTREE(nodes, startNode)
3:   oddDegreeNodes  $\leftarrow$  []
4:   for each [node, neighbours] in mst do                                 $\triangleright$  Calculate odd-degrees
5:     if LENGTH(neighbours) mod 2  $\neq$  0 then
6:       add node to oddDegreeNodes
7:     end if
8:   end for                                                                 $\triangleright$  Generate graph structures
9:   matchingEdges  $\leftarrow$  MINIMUMWEIGHTPERFECTMATCHING(oddDegreeNodes, nodes)
10:  multigraph  $\leftarrow$  COMBINE(mst, matchingEdges)
11:  rawCircuit  $\leftarrow$  FINDEULERIANCIRCUIT(multigraph, startNode)
12:  visited  $\leftarrow$  {}
13:  tour  $\leftarrow$  []
14:  for each node in rawCircuit do                                        $\triangleright$  Construct tour using raw circuit
15:    if node is not in visited then
16:      add node to visited
17:      add node to tour
18:    end if
19:  end for
20:  add startNode to tour
21:  return tour
22: end procedure
```

6.1.3 Examples

To better visualise and understand Christofides, we can look at how the algorithm performs in different scenarios. However, as a globally informed heuristic, it is less meaningful to distinguishing between the best case and general case scenarios. Therefore, we observe how the heuristic behaves on the real-world instance for comparative analysis. Below is the final solution of the real-world instance computed using the Christofides algorithm.

Real-World Case

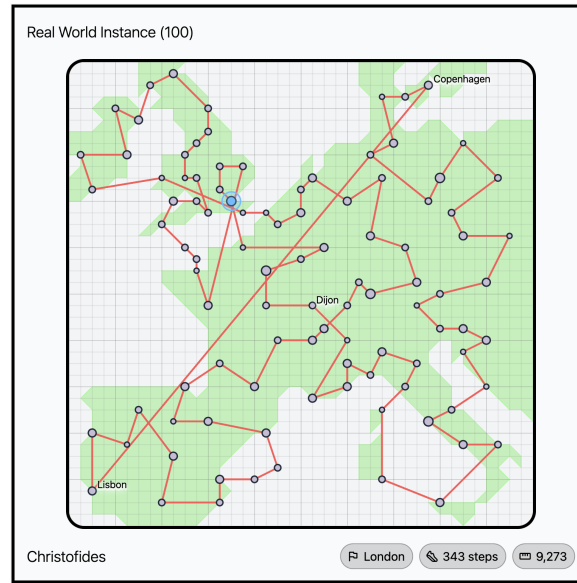


Figure 6.1: The Christofides-Serdyukov solution of the real-world TSP instance

Figure 6.1 shows the Christofides tour of the real-world instance. The route is broadly coherent, exhibiting the advantage of constructing the tour through an MST, odd-degree matching, and Eulerian shortcutting rather than direct selection. This gives the algorithm more structural awareness than NN, NI, or CI. However, the final tour is still undermined by a number of expensive long-range edges. In particular, the matching process can introduce costly edges when only a limited set of valid pairings remain. Most notably, this is visible in the large jump between *Lisbon* and *Copenhagen*. Additionally, the algorithm creates a clear crossover in the route around *Dijon*. This demonstrates that although Christofides provides a theoretical $\frac{3}{2}$ -approximation guarantee, the intermediate graph structures can still lead to poor geometric decisions in practice.

6.1.4 Evaluation

Time Growth

To derive the time growth of the Christofides algorithm theoretically, we can analyse its major stages. First, we construct an MST using Prim's algorithm. For this implementation, Prim's algorithm repeatedly scans all nodes to select the next vertex and then updates the minimum value for each edge, which gives a time complexity of $O(n^2)$. The odd-degree vertices of the MST are then found in linear time with $O(n)$ time complexity.

Next, a complete weighted graph is formed over k odd-degree vertices, taking $O(k^2)$. The minimum-weight perfect matching is the most computationally expensive part of the algorithm and dominates the overall runtime. Combining the MST and matching edges into a multigraph, constructing an Eulerian circuit and shortcutting the repeated vertices, are all polynomial operations. We can therefore determine that the time growth of the Christofides implementation is dominated by the minimum-weight perfect matching stage.

n	Median Time (ms)
10	0.0
100	1.1
250	3.4
500	10.5
750	24.8
1,000	53.4

Table 6.1: The median computation times for different values of n between 10 and 1,000 using the Christofides-Serdyukov algorithm. For full results, see Table C.7.

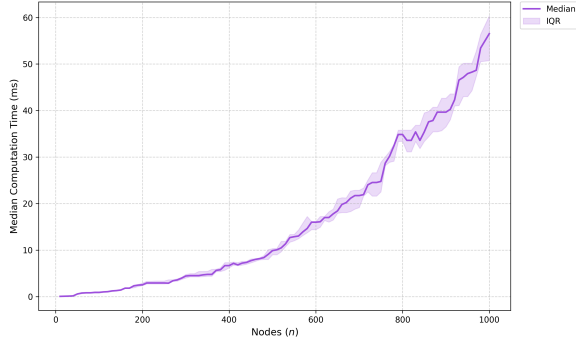


Figure 6.2: A graph showing the median computation time against nodes (n).

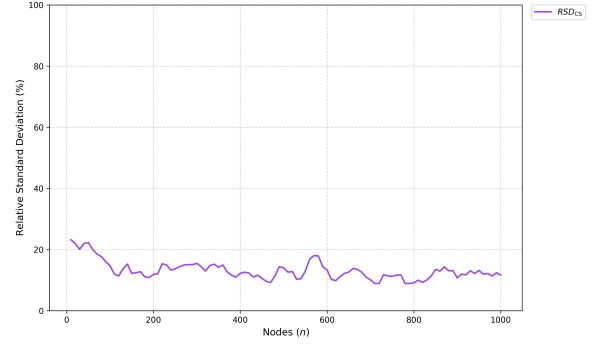


Figure 6.3: A graph showing the relative standard deviation in computation time against nodes (n).

The empirical results in Table 6.1 and Figure 6.2 show a clear polynomial growth in the computation time as n increases. The curve is noticeably steeper than those seen for NN, FI or MF, which is attributable to the minimum-weight perfect matching stage. Using a power-law comparison of two data points from Table 6.1 ($n = 500$ and $n = 1,000$), we obtain an observed growth exponent of approximately 2.34 ($\log_2 5.1$). This lies between quadratic and cubic growth, suggesting that the implementation scales at a super-quadratic rate in practice. This supports the theoretical analysis and shows that Christofides is significantly more computationally expensive than other construction heuristics.

Figure 6.3 shows a moderate but consistent RSD of around 10%-15% for $10 \leq n \leq 1,000$. This is likely due to the minimum-weight perfect matching stage, where both the number and arrangement of odd-degree vertices can affect the required computation. As a result, Christofides is more sensitive to instance structure than other construction heuristics.

Tour Quality

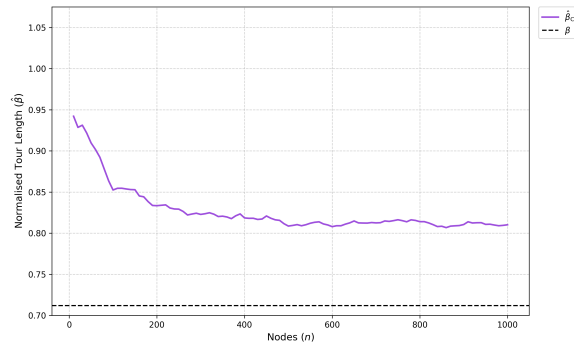


Figure 6.4: A graph showing the normalised tour length ($\hat{\beta}_{CS}$) against nodes (n).

We can observe the normalised tour length $\hat{\beta}_{CS}$ of the Christofides heuristic in Figure 6.4. As with several other heuristics, $\hat{\beta}_{CS}$ declines rapidly for $n \leq 200$, suggesting that the heuristic benefits from denser instances where the intermediate graph structures can better reflect the wiser geometry of the instance. The results then stabilise towards $\hat{\beta}_{CS} = 0.81$, leading to an optimality gap of just $\delta = 13.8\%$.

6.1.5 Conclusion

Overall, the Christofides-Serdyukov heuristic uses multiple graph structures to build a stronger global awareness of the instance, allowing it to produce better organised tours than the greedy or vertex insertion heuristics. Although this comes at a noticeable computational cost, the resulting tour quality is the strongest of the pure construction heuristics.

7 Improvement Heuristics

Unlike construction heuristics, improvement heuristics require an initial tour to refine. This combination of multiple heuristics can be referred to as a *setup*. In this report, the improvement methods are evaluated using tours initially constructed by the Multi-Fragment heuristic (See Section 4.3). This decision aligns with the project specification, which identifies Multi-Fragment+2-Opt as an extension. Furthermore, using a weaker heuristic such as NN would risk overstating the effect of improvement, while using a stronger baseline such as FI or CS would make it harder to detach the contribution of the local search stage.

Unlike construction heuristic performance which can be evaluated independently, improvement heuristic performance is coupled to the structure of the provided initial tour. For example, while improving a Randomised (Appendix A.1) tour can yield admirable results, this would require significantly more effort than improving a Farthest Insertion (Section 5.4) tour. For this reason, the empirical behaviours of computation time or tour quality are more meaningfully evaluated in comparison to other improvement heuristics, as in Section 8.2.

7.1 2-Opt Heuristic

7.1.1 Algorithm Explained

2-Opt (2O) is a local search improvement heuristic that iteratively refines a given tour. At each step, the algorithm examines pairs of non-adjacent edges and swaps them when doing so produces a shorter tour. This process is repeated until a full pass yields no further improvement (Croes, 1958).

Put formally, let the given tour over an instance with n nodes be $T = (v_1, v_2, \dots, v_n, v_1)$. 2-Opt selects two non-adjacent edges, (v_{i-1}, v_i) and (v_j, v_{j+1}) , where $1 < i < j < n$. The change in tour length Δ caused by swapping these two edges can be defined as:

$$\Delta = (d(v_{i-1}, v_j) + d(v_i, v_{j+1})) - (d(v_{i-1}, v_i) + d(v_j, v_{j+1})) \quad (7.1)$$

where $d(v_x, v_y)$ is the Euclidean distance between v_x and v_y . If $\Delta < 0$, then we replace these edges with (v_{i-1}, v_j) and (v_i, v_{j+1}) . This operation reverses the order of the sub-tour between v_i and v_j . The algorithm repeats this process until no improving 2-Opt swap remains.

7.1.2 Pseudocode Representation

Below is a pseudocode implementation of the 2-Opt algorithm which uses an immediate-improvement methodology. The parameter of the function is an initial tour, **initialTour**. The function runs the 2-Opt algorithm on the given tour and returns an improved tour.

Algorithm 9 2-Opt Algorithm

```
1: procedure RUNTWOOPT(initialTour)
2:   tour  $\leftarrow$  initialTour
3:   remove last node from tour ▷ Open the tour for amendments
4:   improved  $\leftarrow$  true
5:   while improved do ▷ Continue iterating while improvements remain
6:     improved  $\leftarrow$  false
7:     for  $i \leftarrow 1$  to LENGTH(tour) - 2 do
8:       for  $j \leftarrow i + 1$  to LENGTH(tour) - 1 do
9:         distAB  $\leftarrow$  HYPOT(tour[i-1], tour[j]) ▷ Calculate delta ( $\Delta$ )
10:        distCD  $\leftarrow$  HYPOT(tour[i], tour[j+1])
11:        distAC  $\leftarrow$  HYPOT(tour[i-1], tour[j])
12:        distDB  $\leftarrow$  HYPOT(tour[j], tour[j+1])
13:        delta  $\leftarrow$  (distAB + distCD) - (distAC + distDB)
14:        if delta < 0 then ▷ Check if this swap will improve the tour
15:          reverse the sub-tour from tour[i] to tour[j]
16:          improved  $\leftarrow$  true
17:        end if
18:      end for
19:    end for
20:  end while
21:  add first node to tour ▷ Close the tour
22:  return tour
23: end procedure
```

7.1.3 Examples

To better visualise and understand 2-Opt, we can look at how the algorithm performs in different scenarios. However, for an improvement heuristic with full access to the structure of a given tour, distinguishing between the best case and general case scenarios is less meaningful than for construction heuristics. Therefore, we observe how the heuristic behaves on the real-world instance for comparative analysis. Below is the final solution of the real-world instance computed using the Multi-fragment and 2-Opt algorithms.

Real-World Case

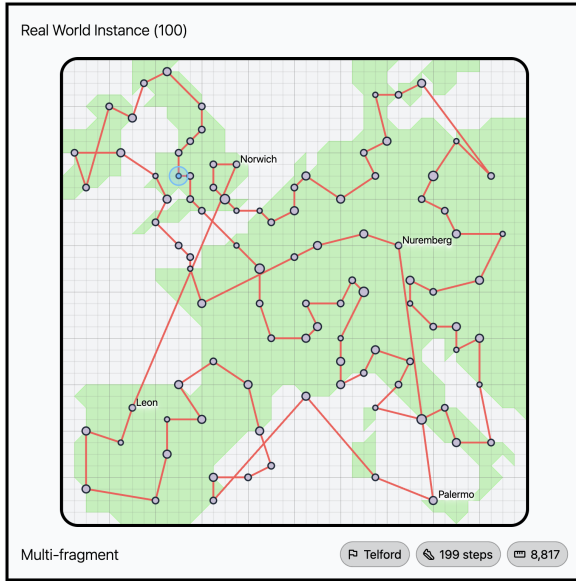


Figure 7.1: The Multi-fragment solution of the real-world TSP instance

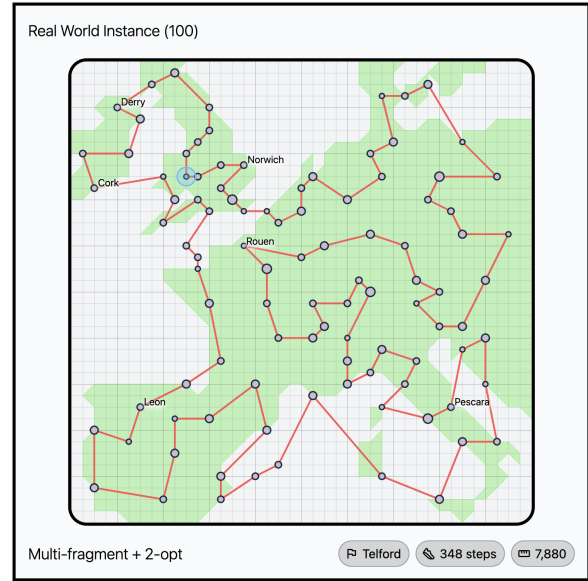


Figure 7.2: The Multi-fragment + 2-Opt solution of the real-world TSP instance

By comparing Figure 7.2 with the original MF tour in Figure 7.1, we can observe the effectiveness of 2-Opt. The most significant change is the removal of large jumps created by MF, such as the edge between *Leon* and *Norwich*. These large connections are replaced by a more geographically coherent traversal through neighbouring regions. Furthermore, the algorithm removes overlapping edges by reversing sub-tours, as seen in the area between *Derry* and *Cork*. At the same time, 2-Opt preserves the original structure of the MF tour where it already performed well, namely the sub-tour from *Rouen* to *Pescara*. Overall, 2-Opt reduces the tour length from 8,817 to 7,880, a reduction of approximately 10.6%.

7.1.4 Evaluation

Time Growth

We can theoretically analyse the time growth of 2-Opt by considering the cost of one complete improvement pass. For a tour with n nodes, the algorithm examines pairs of non-adjacent edges using two nested loops, giving $O(n^2)$ potential swaps per pass. Since the change in tour length Δ can be computed using the four affected nodes, each swap evaluation is constant time. However, the total number of accepted swaps and the number of full passes required before termination depend on the structure of the initial tour. As a result, while the cost of a single pass is clearly quadratic, the overall runtime of 2-Opt is dependent on the given input and is therefore more meaningfully evaluated empirically as part of a complete setup.

7.1.5 Conclusion

Overall, we can observe that 2-Opt is an effective improvement heuristic for refining an existing tour. By repeatedly removing inefficient pairs of edges, the algorithm can remove overlaps and large jumps while preserving the stronger areas of the initial tour. This makes it a practical and computationally efficient method to improve heuristic tours. However, as 2-Opt terminates as soon as no further improving swap can be found, the final tour may remain only locally optimal.

7.2 Simulated Annealing Heuristic

7.2.1 Algorithm Explained

Simulated Annealing (SA) is a meta-improvement heuristic that extends local search by allowing occasional non-improving swaps. Similar to 2-Opt, the algorithm considers swapping two edges in a tour by reversing a sub-tour. However, unlike 2-Opt, which only accepts improving swaps, SA may also accept a worsening swap with some probability.

The process of SA is inspired by the physical behaviour of annealing in metallurgy, where a material is heated and then cooled to a lower-energy state. In the heuristic setting, this idea is modelled using a temperature schedule defined by the current temperature T_k , cooling rate α and minimum temperature $T_{\min}$. The temperature is updated geometrically using $T_{k+1} = \alpha \cdot T_k$. While T_k remains high, the algorithm is more willing to accept worsening swaps, encouraging exploration of the search space. As T_k decreases due to α , the algorithm becomes more rigid and increasingly favours improving swaps, until $T_k \leq T_{\min}$.

The SA algorithm starts with an initial tour and repeatedly chooses two random indices that define a candidate 2-Opt swap. If swapping these edges improves the tour, the swap is accepted. If the swap worsens the tour, it may still be accepted according to a probability P , linked to the increase in tour length Δ and T_k :

$$P = e^{\frac{-\Delta}{T_k}} \quad (7.2)$$

Since $\Delta > 0$ for a worsening swap and $T_k > 0$, Equation 7.2 returns a value $0 < P < 1$, which can be interpreted as the probability of accepting that move. As T_k is reduced towards $T_{\min}$, this probability reduces with it. This means that, while 2-Opt stops when no improving swap remains, SA may continue exploring beyond these local optima. However, this does not guarantee a better final tour in practice, as the algorithm's performance relies heavily on the chosen temperature schedule, iteration count and termination conditions.

7.2.2 Pseudocode Representation

Below is a pseudocode implementation of the Simulated Annealing algorithm. The parameter of the function is an initial tour, `initialTour`. The function runs the Simulated Annealing algorithm on the given tour and returns an improved tour.

Algorithm 10 Simulated Annealing Algorithm

```
1: procedure RUNSIMULATEDANNEALING(initialTour)
2:   tour  $\leftarrow$  initialTour
3:   remove last node from tour ▷ Open the tour for amendments
4:   iterationsPerTemp  $\leftarrow$  100 ▷ Define constants
5:   initialTemp  $\leftarrow$  1000
6:   coolingRate  $\leftarrow$  0.99
7:   minimumTemp  $\leftarrow$  1
8:   maxSolidSteps  $\leftarrow$  100
9:   currentTemp  $\leftarrow$  initialTemp ▷ Initialise temperature
10:  solidSteps  $\leftarrow$  0
11:  while currentTemp > minimumTemp do
12:    solidStep  $\leftarrow$  true
13:    for  $i \leftarrow 1$  to iterationsPerTemp do
14:      Initialise two random indices  $a$  and  $b$ , where  $1 \leq a, b < |\text{tour}|$  and  $a \neq b$ 
15:      distAB  $\leftarrow$  HYPOT(tour[a-1], tour[b]) ▷ Calculate delta ( $\Delta$ )
16:      distCD  $\leftarrow$  HYPOT(tour[a], tour[b+1])
17:      distAC  $\leftarrow$  HYPOT(tour[a-1], tour[a])
18:      distDB  $\leftarrow$  HYPOT(tour[b], tour[b+1])
19:      delta  $\leftarrow$  (distAB + distCD) - (distAC + distDB)
20:      if delta < 0 then ▷ If this swap will improve tour, approve it
21:        reverse the sub-tour from tour[i] to tour[j]
22:        solidStep  $\leftarrow$  false
23:      else if  $e^{-\text{delta}/\text{currentTemp}} > \text{RANDOM}(0, 1)$  then ▷ Otherwise, probabilistically approve it
24:        reverse the sub-tour from tour[i] to tour[j]
25:        solidStep  $\leftarrow$  false
26:      end if
27:    end for
28:    if solidStep then ▷ Check if the tour has stopped improving
29:      increment solidSteps
30:      if solidSteps  $\geq$  maxSolidSteps then
31:        break
32:      end if
33:    else
34:      solidSteps  $\leftarrow$  0
35:    end if
36:    currentTemp  $\leftarrow$  coolingRate  $\times$  currentTemp
37:  end while
38:  add first node to tour ▷ Close the tour
39:  return tour
40: end procedure
```

7.2.3 Examples

To better visualise and understand Simulated Annealing, we can look at how the algorithm performs in different scenarios. However, for an improvement heuristic with full access to the structure of a given tour, distinguishing between the best case and general case scenarios is less meaningful than for construction heuristics. Therefore, we observe how the heuristic behaves on the real-world instance for comparative analysis. Below is the final solution of the real-world instance computed using the Multi-fragment and Simulated Annealing algorithms.

Real-World Case

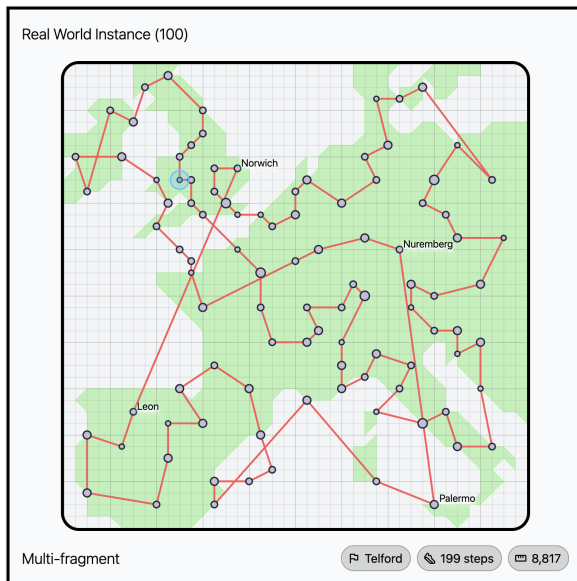


Figure 7.3: The Multi-fragment solution of the real-world TSP instance

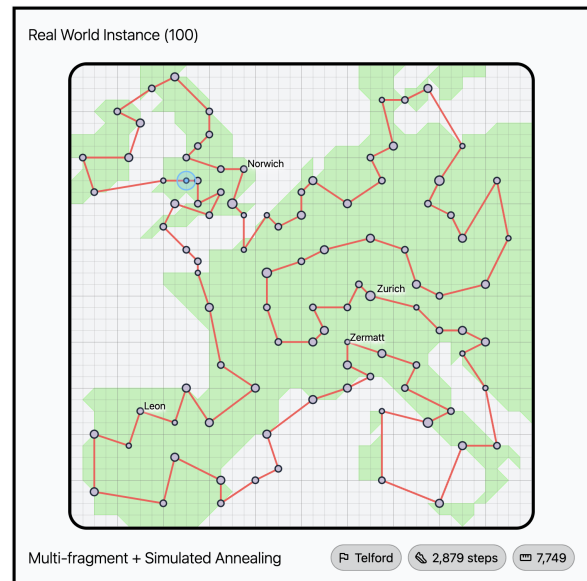


Figure 7.4: The Multi-fragment + Simulated Annealing solution of the real-world TSP instance

We can compare Figure 7.4 with the original MF tour shown in Figure 7.3 to observe how SA performs on the real-world instance. Similar to 2-Opt, SA removes major inefficiencies introduced by the MF algorithm, such as the large edge between *Leon* and *Norwich*. These edges are removed as SA immediately accepts improving swaps. However, unlike 2-Opt, SA is able to accept non-improving swaps during the search. This allows the heuristic to reconstruct parts of the tour that would otherwise be preserved by a strictly improving local search. One example of this can be seen in the disconnection of the edge between *Zurich* and *Zermatt*, which contributes to improving the route through Italy and back towards Spain. Overall, SA reduces the tour length from 8,817 to 7,749, a reduction of approximately 12.1%.

7.2.4 Evaluation

Time Growth

The runtime growth of the SA algorithm can be theoretically analysed by considering the cost of its temperature schedule and the work performed within each temperature step. For a given temperature, the algorithm performs a fixed number of potential 2-Opt swaps. Each potential swap requires selecting two indices, computing the associated change in tour length Δ , and then, if accepted, reversing the corresponding sub-tour. Since the value of Δ can be calculated using only the four affected nodes, the cost of evaluating one potential move is constant.

Unlike 2-Opt, the runtime growth of SA is strongly dependent on the chosen temperature schedule. The number of temperature levels is determined by the initial temperature, cooling rate and minimum temperature, while the amount of work at each level is set by the chosen number of iterations. More advanced implementations modify the temperature schedule based off the size of the initial tour for more accurate annealing, increasing the complexity of runtime growth further. In addition, because worsening moves may be accepted probabilistically, the number of successful swaps depends on the

stochastic behaviour of the search. As a result, while the cost of an individual step is simple to analyse, the full runtime of SA is determined by both the schedule parameters and the initial tour.

7.2.5 Conclusion

Overall, Simulated Annealing employs a more exploratory improvement strategy than 2-Opt. By allowing occasional non-improving swaps, the heuristic can escape local optima and restructure parts of the tour that a purely greedy local search would maintain. This gives SA the potential to produce stronger tours than a deterministic improvement method when the temperature schedule is well matched to the problem. However, this flexibility also makes the heuristic more sensitive to parameter choice and introduces greater uncertainty into both its runtime and final tour quality.

8 Comparative Analysis

The comparative tables and graphs in this chapter provide an overall ranking of heuristic performance on random instances. By conducting experiments on instances of randomly distributed nodes, it is possible to collect consistent large-scale results for comparison. However, earlier examples in this report have also shown that the structure of instances can strongly affect heuristic behaviour. We can observe that regular geometric layouts, such as grids or circles, may favour simpler heuristics like NN or MF, whereas more complex layouts, such as clustered or real-world instances, can expose weaknesses in algorithms driven purely by local selection. Therefore, heuristic choice should depend not only on average runtime or tour quality, but also on the structural characteristics of the target problem.

8.1 Construction Heuristics

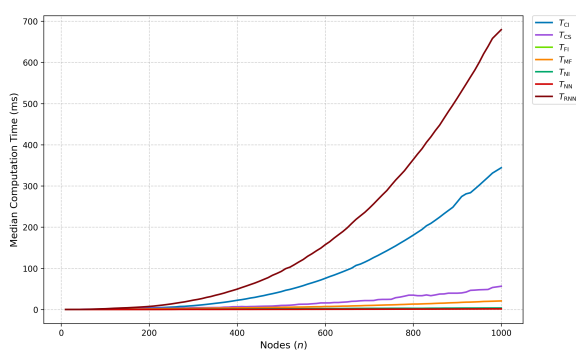


Figure 8.1: Comparative graph showing the median computation time of construction heuristics for $10 \leq n \leq 1,000$.

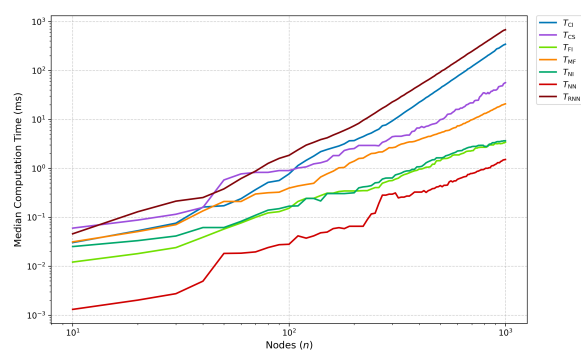


Figure 8.2: Comparative graph showing the median computation time of construction heuristics for $10 \leq n \leq 1,000$ on a logarithmic scale.

From Figures 8.1 and 8.2, we can compare the median computation times for all implemented construction heuristics as n increases. We can observe that NN remains the fastest heuristic for all values of n , which is consistent with its simple greedy selection process. Slightly slower are NI and FI, which follow almost identical runtime curves, suggesting that their different selection rules lead to similar computational cost. However, Figure 8.1 emphasises that RNN and CI scale much worse, with their additional checks becoming increasingly expensive as n grows. Despite CS and MF performing similarly to CI for smaller values of n , both heuristics scale slightly better for larger instances, suggesting that the cost of their initial setup becomes less significant as n increases.

As n increases from 10 to 1,000, Figure 8.2 shows that the separation in computation time widens as the heuristics scale. The logarithmic graph also shows a much clearer distinction among the faster heuristics, which is less visible in the regular-scale graph. Overall, these comparisons in median computation time align with earlier empirical and theoretical time growth analysis.

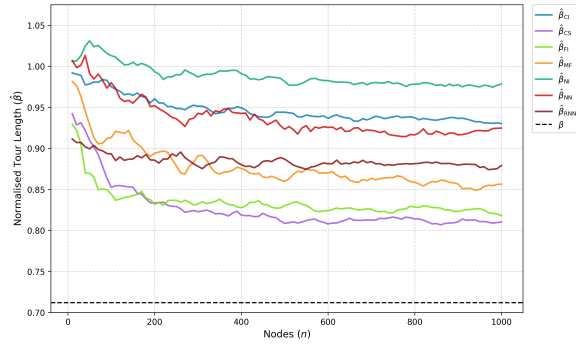


Figure 8.3: Comparative graph showing the normalised tour length of construction heuristics compared with the theoretical constant ($\beta \approx 0.7120$) for $10 \leq n \leq 1,000$.

Heuristic	$\hat{\beta}$	δ
CS	0.81	13.8%
FI	0.82	16.6%
MF	0.86	20.8%
RNN	0.88	23.6%
NN	0.93	30.6%
CI	0.93	30.6%
NI	0.98	37.6%

Figure 8.4: Comparative table showing the normalised tour length ($\hat{\beta}$) and optimality gap (δ) of construction heuristics relative to $\beta \approx 0.7120$ ($n = 1,000$).

We can compare the normalised tour lengths of the construction heuristics using Figure 8.3. Across all heuristics, there is a general downward trend as n increases, with most curves stabilising for $n \geq 200$. This suggests that each heuristic becomes more consistent on denser random instances, some more so than others. Table 8.4 summarises the endpoint values at $n = 1,000$ with their optimality gap.

From Table 8.4, we can see NI performs worst overall, with an optimality gap of $\delta = 37.6\%$. NN and CI also remain relatively weak, both settling at approximately $\hat{\beta} = 0.93$, despite CI applying a much greater computational cost. RNN and MF both improve upon NN and CI, but still remain noticeably above the best performing construction heuristics: FI and CS, with $\hat{\beta}_{FI} = 0.82$ and $\hat{\beta}_{CS} = 0.81$. These results show a clear separation between heuristics that are driven by immediate local gain and those that use an instance's global structure to construct a more coherent tour.

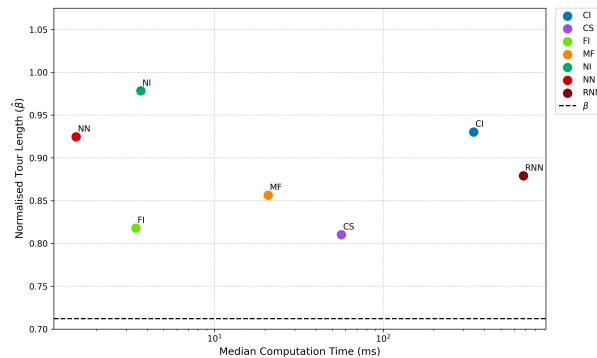


Figure 8.5: Comparative graph showing the trade-offs between median computation time and normalised tour length for construction heuristics ($n = 1,000$).

Figure 8.5 combines both computation time and tour quality for the construction heuristics at $n = 1,000$, allowing direct comparison of their practical trade-offs. The strongest overall compromises are FI and CS. FI achieves near-best tour quality while remaining among the fastest heuristics, whereas CS yields the best tour quality overall at the cost of a higher computation time. MF also occupies a middle ground, offering a clear improvement over NN, NI and CI, while also costing significantly less than CI and RNN.

CI and RNN, however, are hard to justify. While RNN improves on NN and CI improves on NI in tour quality, both gains come at a very large computational cost. NI is also unappealing as its runtime is nearly identical to FI while producing weaker tours. Overall, this graph shows that NN, FI and CS are the non-dominated construction heuristics, as none is both slower and less effective than another method.

8.2 Improvement Heuristics

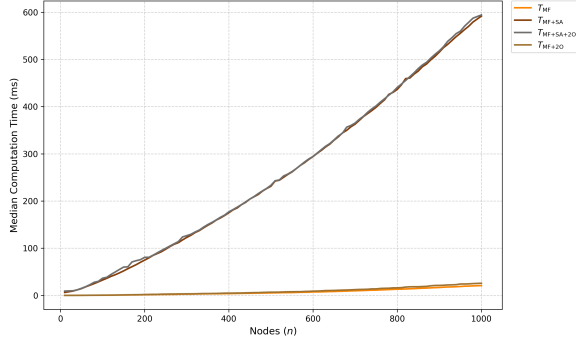


Figure 8.6: Comparative graph showing the median computation time of improvement heuristics for $10 \leq n \leq 1,000$.

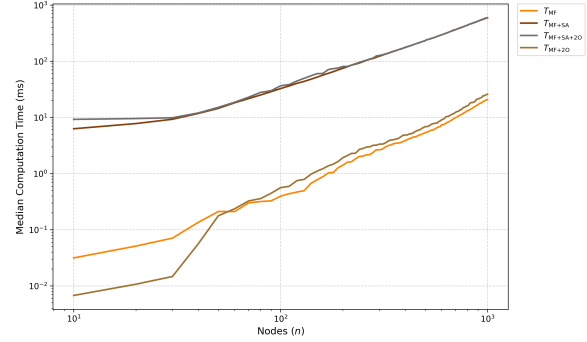


Figure 8.7: Comparative graph showing the median computation time of improvement heuristics for $10 \leq n \leq 1,000$ on a logarithmic scale.

From Figures 8.6 and 8.7, we can observe the median computation times for all implemented improvement setups compared with their construction heuristic, MF. We can see that the improvement setups that include SA require significantly more computation, even for instances with small n . This can be attributed to the high number of temperature steps and subsequent number of iterations that are performed by the SA algorithm. In Figure 8.7, we can also observe that the setups with 2-Opt are only marginally more expensive to run, with MF+SA+2O incurring very little extra computational cost relative to MF+SA.

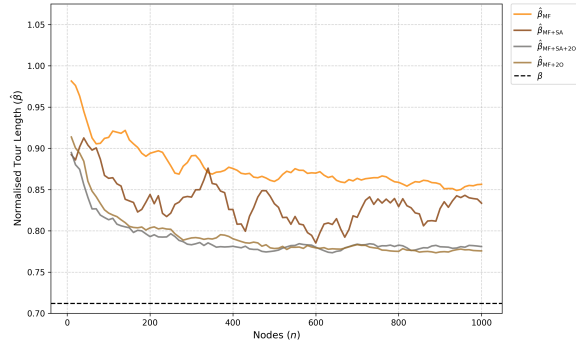


Figure 8.8: Comparative graph showing the normalised tour length of improvement heuristics compared with the theoretical constant ($\beta \approx 0.7120$) for $10 \leq n \leq 1,000$.

Heuristic	$\hat{\beta}$	δ
MF+2O	0.78	9.6%
MF+SA+2O	0.78	9.6%
MF+SA	0.83	16.6%
MF	0.86	20.8%

Figure 8.9: Comparative table showing the normalised tour length ($\hat{\beta}$) and optimality gap (δ) of improvement setups relative to $\beta \approx 0.7120$ ($n = 1,000$).

In Figure 8.8, we can compare the normalised tour lengths of the improvement setups. Similarly to the construction heuristics, there is a rapid decline in the normalised tour length as n increases for $n \leq 200$ as for smaller instances, early decisions are more influential. We can observe that the setups that include 2-Opt exhibit a very stable endpoint quality, consistent with the algorithm terminating only once no improving local swap remains. However, for the setup which only includes SA (MF+SA), we can observe a curve which fluctuates excessively, with no reliable data point below a 2-Opt setup. Table 8.9 summarises the endpoint values at $n = 1,000$ with their optimality gap.

From Table 8.9, we can observe that every improvement setup improved upon the MF construction heuristic with an optimality gap of $\delta = 20.8\%$. By applying just SA, we see some improvement but struggle to trust the value of $\delta = 16.6\%$. The two setups with the best performance are those that include 2-Opt: MF+2O and MF+SA+2O. Both setups achieve an optimality gap of just $\delta = 9.6\%$. While the data shows 2-Opt consistently accomplishing better tour quality than SA, this likely reflects SA's dependence on random moves and schedule parameters, meaning its performance may be more sensitive to tuning than 2-Opt. Further tuning of parameters could improve SA, but under the present configuration it does not outperform setups with 2-Opt.

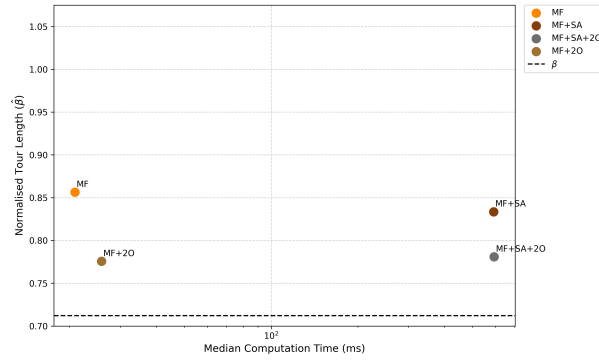


Figure 8.10: Comparative graph showing the median computation time against normalised tour length of improvement heuristics ($n = 1,000$).

Figure 8.10 shows a direct comparison of practical trade-offs for the improvement setups at $n = 1,000$, by combining both computation time and tour quality. Unlike the construction heuristic comparison shown in Figure 8.5, this graph shows that the clear dominant setup is MF+2O. It achieves the shortest tours while only requiring a moderate increase in computation time over the MF heuristic. Although both MF+SA and MF+SA+2O improve upon the MF tour quality, these improvements come at a far greater computational cost, and they are therefore hard to justify. Overall, this reinforces that 2-Opt is the most effective improvement heuristic, delivering a large reduction in tour length for relatively little additional cost.

8.3 Overall Comparison

Heuristic	Time	Δ Time
NN	1.6	0%
FI	3.5	120%
NI	3.7	131%
MF	21.2	1,237%
MF+2O	26.0	1,542%
CS	53.4	3,272%
CI	351.8	22,093%
MF+SA+2O	603.2	37,957%
MF+SA	604.7	38,053%
RNN	698.8	43,990%

Table 8.1: Comparative table showing the median computation time of all evaluated heuristics ($n = 1,000$) relative to the fastest (NN).

Heuristic	Dist	Δ Dist
MF+2O	24,451	0.0%
MF+SA+2O	24,620	0.7%
CS	25,592	4.7%
FI	25,986	6.3%
MF+SA	27,033	10.6%
MF	27,046	10.6%
RNN	27,944	14.3%
CI	29,320	19.9%
NN	29,453	20.5%
NI	30,849	26.2%

Table 8.2: Comparative table showing the median tour distance of all evaluated heuristics ($n = 1,000$) relative to the shortest (MF+2O).

Table 8.1 shows the median computation time, and the percentage difference relative to the fastest heuristic, for every evaluated setup at $n = 1,000$. We can observe that NN is the fastest overall, with FI and NI the only other heuristics that reach close to its computational efficiency. MF and MF+2O occupy a middle ground, with the 2-Opt only adding a small overhead to the original MF construction. While CS is slower still, it remains substantially cheaper than CI, RNN and SA-based improvement setups which all obtain a very high computational cost.

Table 8.2 shows the corresponding median tour distance, and the percentage difference relative to the shortest heuristic. We can observe that MF+2O produces the shortest tour, with MF+SA+2O only 0.7% behind. Among the construction heuristics, CS and FI perform the best, both outperforming MF, RNN, CI and NN, with NI clearly performing the worst.

These tables emphasise that the heuristics that produce the best tours are not always the fastest, and vice versa, but some manage to provide a stronger balance than others. In particular, MF+2O stands out as the strongest compromise, while FI appears to be the most attractive purely construction heuristic when both computation time and tour quality are considered together.

8.4 Conclusion

Overall, this report analysed a range of heuristic approaches for solving real-world instances of the metric TSP. We observed and evaluated the behaviour of each setup theoretically and empirically by investigating various experiments with the GUI and Java implementations. We found that no single heuristic is best at all metrics, with a clear trade-off dependent on tour quality, computation time or both. Additionally, we found that the structure of instance can effect heuristic choice.

For instances of randomly distributed nodes, we found that the fastest heuristics, by some margin, were NN, FI and NI. Alternatively, we discovered the heuristics that produced the best tours were setups that included 2-Opt, such as MF+2O and MF+SA+2O. We can conclude that the heuristic to choose should be based off a balance between the required speed and quality, and its suitability to a particular instance. Greedy heuristics remain useful for applications where speed is critical. However, where tour quality is prioritised, advanced construction methods such as Vertex Insertion or Approximation heuristics are more viable. Furthermore, where more computation can be utilised, a combination of a fast Greedy heuristic and an effective Improvement heuristic may be applicable.

Final Report word count: 13,583

Professional Issues Report word count: 1,255

Total word count: 14,838

A Additional Techniques

A.1 Randomised Heuristic

A.1.1 Algorithm Explained

The Randomised heuristic provides a fast but naive method for constructing a TSP tour. The randomised heuristic selects the next node at random from the remaining unvisited nodes. This process continues until all nodes have been visited, at which point the tour returns to the starting node.

Put formally, let the starting node be n_s , and let the set of unvisited nodes be $U = N \setminus \{n_s\}$. At each iteration, the algorithm selects a node, n_k , at random, appends it to the tour, T , and removes it from U . This repeats until $U = \emptyset$, after which n_s is appended to the end of the tour.

A.1.2 Pseudocode Representation

Below is the formal pseudocode implementation of the Randomised Algorithm. The inputs for the function are the start (and end) node, **startNode**, as well as the list of all nodes, **nodes**. The function runs the Randomised algorithm on the given list of nodes, and returns an ordered list of nodes, **tour**.

Algorithm 11 Randomised Algorithm

```
1: procedure RUNRANDOMISED(nodes, startNode)
2:   tour  $\leftarrow$  [startNode]
3:   unvisitedNodes  $\leftarrow$  nodes
4:   currentNode  $\leftarrow$  startNode
5:   remove currentNode from unvisitedNodes
6:   while there are unvisited nodes do
7:     randomNode  $\leftarrow$  random element from unvisitedNodes
8:     add randomNode to tour
9:     remove randomNode from unvisitedNodes
10:    currentNode  $\leftarrow$  randomNode
11:  end while
12:  add startNode to tour
13:  return tour
14: end procedure
```

A.1.3 Effectiveness and Evaluation

To better visualise and understand the Randomised heuristic, we can look at how the algorithm performs on the real-world instance. Below is the final solution of the real-world instance computed using the Randomised algorithm.

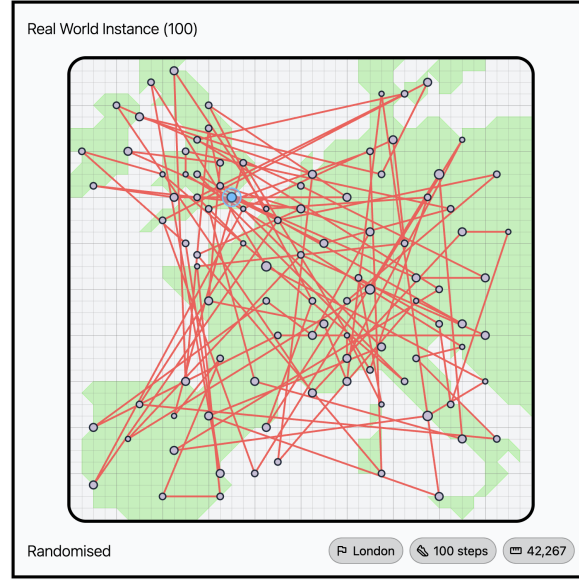


Figure A.1: The Randomised solution of the real-world TSP instance

Figure A.1 shows how the Randomised heuristic provides a chaotic solution to the real-world instance. With no rationale in choosing the next city after another, the algorithm makes large jumps across Europe. This leads to a total distance of 42,267, over 367% worse than any other heuristic we evaluated. Additionally, for the real-world instance with 100 nodes the search space consists of 4.7×10^{155} possible tours. With a truly random approach, we can observe that the probability of yielding a competitive result is effectively zero.

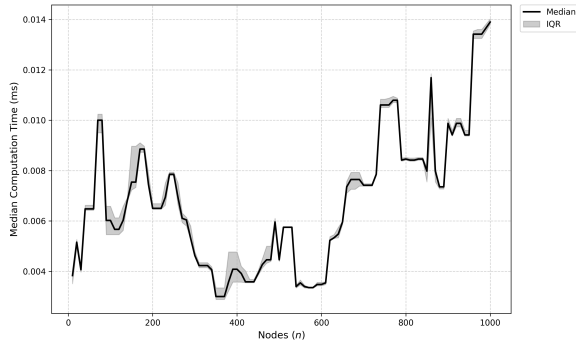


Figure A.2: A graph showing the median computation time against nodes (n).

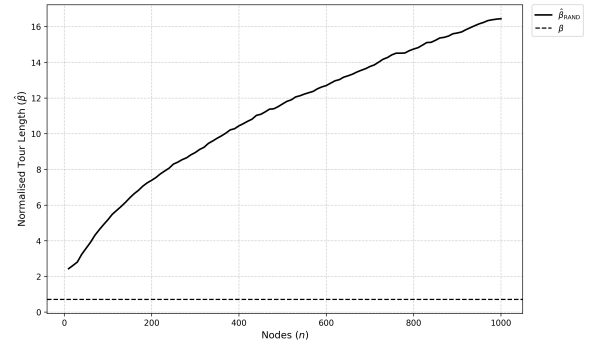


Figure A.3: A graph showing the normalised tour length ($\hat{\beta}_R$) against nodes (n).

As the algorithm performs exactly one random selection per node, the total number of operations scales linearly with the number of nodes, n . Therefore, the randomised technique has a run time complexity of $O(n)$. Due to this extreme efficiency, the Randomised heuristic can quickly find solutions for significantly large instances. From Figure A.2, we can see the algorithm is able to solve instances with 1,000 nodes in just 0.014ms. However, the implementation lacks any logic or global context to select each node. We can see this causes the normalised tour length $\hat{\beta}_R$ shown in Figure A.3 to increase with n (instead of converging on a value near $\beta \approx 0.7120$ as expected).

To conclude, while the Randomised heuristic rapidly terminates, the constructed tour lacks organisation. We can therefore determine there to be very few industry applications of the Randomised heuristic without use of improvement heuristics.

B Diary Entries

B.1 Term 1

- **Week 1 (22nd September)**

What have I done?

I thought through different ideas for the project and planned which languages I'd like to use. I created an initial Vue project and tested concepts of using a Scalable Vector Graphic (SVG) to represent the graphs. I chose to use SVGs over other web graphic renderers, such as Canvas, because SVGs provide easier control of edges and nodes, and work more fluidly in the web environment that I will be developing in.

What am I going to do next?

I plan to expand on the SVG concept to check that different ideas work. I also plan to meet with my supervisor, Aled.

What's holding me back?

I'd like to hold a meeting with my supervisor to find out certain details surrounding my project and to verify that the ideas I'm having are allowed before I go any further.

- **Week 2 (29th September)**

What have I done?

I created the initial *real-world* graph, with around 100 cities/nodes from around Europe. I also researched more aspects of the project like different heuristics, ways of testing them and how I could implement them in practice.

I held my first meeting with my supervisor, Aled, to discuss TSP proofs and definitions.

Finally, I began working on my project plan, looking at the given template to understand what the requirements are.

What am I going to do next?

I will complete my project plan. Making sure to research for the abstract, finalise my timeline and consider risks and mitigations.

What's holding me back?

Other obligations.

- **Week 3 (6th October)**

What have I done?

I finished and submitted my project plan. I did further research into the TSP with papers to elevate my abstract and gain a clearer understanding of heuristic definitions and performance expectations. I did some additional work on my GUI implementation.

What am I going to do next?

I plan to begin implementing my first heuristics and initial data representation (tables and graphs).

What's holding me back?

I need to research into the best ways of collecting data for, and representing, graphs and tables.

- **Week 4 (13th October)**

What have I done?

I held my second meeting with my supervisor, to clarify the project scope and workflow. We also discussed which heuristics to prioritise this term, as well as areas I should focus on over the next few weeks. I verified that my ideas will lead to meaningful results.

What am I going to do next?

I aim to implement the first heuristic, over a complete instance (an instance with no connection constraints).

What's holding me back?

Other obligations.

- **Week 5 (20th October)**

What have I done?

I implemented Nearest Neighbour (NN) for complete instances, with fast performance holding up to 1,000 nodes (where it solves in under 800ms and computes over 500,000 steps).

I added the ability to playback how solutions were computed (using tracked comparison and selection steps). This feature will allow me to visualise how each heuristic behaves and traverses instances.

I also improved rendering performance: decreasing the rendered elements from 1,562 to 686 (for an instance with 225 nodes).

What am I going to do next?

I intend to research into further heuristics to implement and investigate how I can improve performance of my NN implementation.

What's holding me back?

Nothing.

- **Week 6 (27th October)**

What have I done?

I refactored some of my implementation and made minor improvements to UI, to add the capability for future heuristics and improve readability. Researched into subjects required for interim reports.

What am I going to do next?

I want to begin working on interim report(s).

What's holding me back?

Other obligations and modules.

- **Week 7 (3rd November)**

What have I done?

I implemented the capability to compare multiple heuristics (NN, RNN, Greedy, etc.) on the same instance at once. This is my first step towards evaluating performance differences between heuristics for the interim report. Yet to implement more heuristics other than NN and Randomised (See Appendix A.1).

I overhauled the layout of the GUI, while also improving rendering performance.

What am I going to do next?

I want to meet with my supervisor again to discuss reports, presentation, and data collection and evaluation.

What's holding me back?

Other module coursework.

- **Week 8 (10th November)**

What have I done?

Following on from my meeting with my supervisor, I began working on my interim report. I outlined the core section headings. In doing so, I wrote the pseudocode for NN and RNN, and added some examples for NN (See Section 4.1.3).

What am I going to do next?

I want to start work on exportable GUI data which can be used in my interim report.

As well as further work on my interim report.

What's holding me back?

Other obligations.

- **Week 9 (17th November)**

What have I done?

I continued work on my report, deriving the formulae for step growth for both NN and RNN.

I implemented Batch Mode in the GUI and functionality to export table of results to *.csv* to allow for time growth evaluation.

What am I going to do next?

I intend to improve my interim report introductions and continue most sections.

I also plan to have a meeting with Aled to discuss my presentation and interim report progress.

What's holding me back?

Nothing.

- **Week 10 (24th November)**

What have I done?

I researched the Greedy heuristic for my interim report.

I also, implemented the ability to import test data from the TSP Library (TSPLIB) into my GUI, to allow tour quality evaluation. The addition of the TSPLIB test data will allow for comparison against established benchmarks.

What am I going to do next?

I want to research Vertex Insertion, and finish my presentation.

What's holding me back?

Nothing.

- **Week 11 (1st December)**

What have I done?

I researched Vertex Insertion heuristics, like Nearest Insertion and Farthest Insertion, for my interim report.

Began work on my retrospective report.

I also finalised the content and script of my presentation and sent it off to my supervisor for feedback, before submission.

What am I going to do next?

I need to do final work on interim report and retrospective.

As well as practice for my presentation, before presenting.

What's holding me back?

Nothing.

- **Week 12 (8th December)**

What have I done?

I conducted final practice for my presentation, and presented my presentation.

I completed my retrospective and interim report. Due to time constraints and an effort to submit high quality work, I decided to remove the Vertex Insertion sections for the interim report. I also made final changes based off feedback from proofreaders and my supervisor, Aled.

Finally, I created the required video.

What am I going to do next?

In term 2, I intend to evaluate the progress made so far, and refer to my updated timeline and plan.

What's holding me back?

Nothing.

B.2 Term 2

- **Week 1 (19th January)**

What have I done?

Received grade and feedback for my interim review and presentation. Held meeting with my supervisor to discuss grade breakdown and areas of minor improvement. Researched into more heuristics. Finalised my list of 9 heuristics algorithms I aim to implement: NN, RNN, Randomised, Greedy, VI, NI, FI, and Christofides, as well as 2-Opt and potentially Simulated Annealing.

What am I going to do next?

Research further into each heuristic, and subsequently implement the first (Greedy).

What's holding me back?

Finding concrete literature on each heuristic.

- **Week 2 (26th January)**

What have I done?

Implemented the Multi-fragment (was Greedy) algorithm. Changed the name of the Greedy algorithm to Multi-fragment to better differentiate away from the class of Greedy algorithms. Refined

the way steps are added to improve consistency and reduce repeated code.

What am I going to do next?

Research into and implement the first insertion heuristic. Refactor some functionality to work better with insertion heuristics.

What's holding me back?

Nothing.

- **Week 3 (2nd February)**

What have I done?

Implemented three insertion heuristics: Nearest Insertion, Cheapest Insertion, Farthest Insertion. Developed a generalised insertion solver that takes a relevant strategy (e.g. 'Cheapest') as a parameter to prevent code duplication. Improved and re-worked how SVG Paths are built from steps, to allow multiple steps to be rendered in one 'frame'. This allows: representation of multiple steps in one frame (e.g. Showing a lot of compare steps in one frame), and step removal and insertion in one frame (e.g. useful for updating the path in insertion heuristics).

What am I going to do next?

Research into and implement improvement heuristics, such as 2-Opt and Simulated Annealing.

What's holding me back?

Nothing.

- **Week 4 (9th February)**

What have I done?

Implemented Christofides and 2-Opt heuristics. Added a new step type: 'Improvement'. Improved user experience and how SVG paths are built.

What am I going to do next?

Keeping to my term 2 plan, I aim to finalise the GUI functionality and style.

What's holding me back?

Other obligations.

- **Week 5 (16th February)**

What have I done?

Completely reworked the layout of the GUI. Added multiple pages: Home, Analysis, Results. This resulted in a better, more segmented user experience. Fixed numerous bugs.

Began work on my Java implementations. I've got NN working for over 67,000,000 points in under a minute, and I'm seeing much more consistent graph plots and tables.

What am I going to do next?

I now believe I am ready to conduct multiple experiments on all heuristics and instances. I can then produce the visual materials to go in my report.

What's holding me back?

Nothing.

- **Week 6 (23rd February)**

What have I done?

Made minor adjustments to GUI and underlying code. Researched more into the best ways to experiment and analyse each heuristic.

Implemented a new instance type: Cluster, which generates a number of clusters, c , and distributes a number of points, n , across these clusters. A random spread can be added to reduce uniformity. Improved improvement heuristic step rendering to show progress over the original tour rather than individual improvements.

What am I going to do next?

Conduct and export experiments using my GUI. Implement Christofides in Java to produce consistent timing results for all construction heuristics. Conduct and export experiments using the Java implementations.

What's holding me back?

Other obligations.

- **Week 7 (2nd March)**

- **What have I done?**

- Implemented final construction heuristics in Java: Cheapest Insertion, Christofides and Randomised. Reverted NN implementation to match TypeScript implementation (no longer using a KDTree). Started work on my final report by finalising section layout and cleaning up existing writing.

- **What am I going to do next?**

- Produce visual materials using this exported data.

- **What's holding me back?**

- Other obligations.

- **Week 8 (9th March)**

- **What have I done?**

- Finished implementing Python script which takes exported .csv files and creates graphs and LaTeX tables. Finalised graph style and content. Finalised table content: deciding to show only 'milestone' n values rather than every step (unlike graph which shows every step of 10).

- Held a meeting with my supervisor. Continued work on my final report; specifically worked on Vertex Insertion and Multi-fragment sections.

- **What am I going to do next?**

- Finish first draft of report to send to my supervisor, focussing on macro features over micro details.

- **What's holding me back?**

- Nothing.

- **Week 9 (16th March)**

- **What have I done?**

- Imported the visual materials generated from my Python script into my final report. Improved Randomised Heuristic section by justifying why it is not viable. Expanded the following background theory sections: Big O Notation, Practical Applications, Empirical Analysis.

- Finalised Java implementation by developing Simulated Annealing and 2-Opt.

- Sent a draft of my final report to my supervisor for feedback.

- **What am I going to do next?**

- Work on conclusion and comparison sections. Finalise introduction sections. Start professional issues report.

- **What's holding me back?**

- Nothing.

- **Week 10 (23rd March)**

- **What have I done?**

- Began work on professional issues report. Subsequently, researched into UPS ORION case study and linked it to my project.

- Finalised all visual materials and data to be used in my final report. Worked further on Practical Applications sections. Finalised Randomised Heuristic section by adding data backed analysis. Worked on the structure of 2-Opt and Simulated Annealing sections. Finalised Nearest Neighbour, Repeated Nearest Neighbour and Multi-fragment sections.

- Began work on my final video demo. Showcasing my GUI, implemented technology and used dependencies.

- **What am I going to do next?**

- Finalise final report, final demo video, professional issues report and Git repo.

- **What's holding me back?**

- Other obligations.

- **Week 11 (30th March)**

- **What have I done?**

Finalised professional issues reflection section. Finalised all heuristic sections of final report. Finalised comparative analysis and conclusion of final report. Recorded and edited final demo video. Finalised code and contents of Git repo.

Submitted my final report, final demo video, professional issues report and git repo.

What am I going to do next?

Nothing.

What's holding me back?

Nothing.

C Full Results Tables

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
250	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
500	0.5	0.5	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
750	0.9	0.9	0.9	0.9	0.9	0.8	0.8	0.8	0.8	0.8	0.9
1,000	1.7	1.6	1.4	1.4	1.4	1.3	1.9	1.9	1.7	1.6	1.6

Table C.1: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Nearest Neighbour algorithm. For a concise summary, see Table 4.1.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	2.2	2.1	2.0	1.9	1.9	1.8	1.7	1.8	1.8	1.6	1.8
250	13.7	13.6	13.8	13.5	13.2	13.3	13.8	13.3	13.6	13.2	13.5
500	92.5	91.5	90.7	92.4	91.6	93.5	92.0	93.1	93.9	95.5	92.5
750	301.9	301.7	301.7	302.4	301.9	299.3	298.9	300.9	302.1	299.9	301.7
1,000	701.0	692.6	705.1	698.7	699.0	698.2	697.7	694.6	699.1	700.5	698.8

Table C.2: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Repeated Nearest Neighbour algorithm. For a concise summary, see Table 4.2.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	0.4	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
250	2.1	1.9	3.0	2.6	2.5	2.4	2.2	2.2	2.0	1.9	2.2
500	6.5	6.1	5.8	5.5	5.4	5.2	5.0	4.8	4.9	4.8	5.3
750	11.7	11.1	11.3	11.2	11.3	11.1	11.3	11.3	11.4	11.4	11.3
1,000	21.2	21.2	21.3	21.5	21.2	21.3	21.1	21.1	21.2	20.9	21.2

Table C.3: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Multi-fragment algorithm. For a concise summary, see Table 4.3.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	0.1	0.2	0.2	0.2	0.1	0.2	0.1	0.2	0.1	0.1	0.2
250	0.6	0.6	0.6	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
500	1.7	1.6	1.5	1.5	1.4	1.7	2.2	2.1	1.8	1.7	1.7
750	4.3	3.9	3.2	3.1	3.0	2.9	2.6	2.6	2.6	2.3	2.9
1,000	5.0	4.6	4.0	4.1	3.7	3.6	3.6	3.3	3.2	3.1	3.7

Table C.4: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Nearest Insertion algorithm. For a concise summary, see Table 5.1.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	0.7	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8
250	7.2	6.7	6.3	6.1	5.9	5.7	5.6	5.6	5.3	5.3	5.8
500	43.4	43.4	43.1	43.4	43.5	43.5	43.6	43.5	43.0	43.4	43.4
750	148.1	148.9	148.6	150.2	148.5	148.0	148.1	147.7	149.5	149.1	148.6
1,000	352.2	351.6	351.8	352.2	351.7	351.7	352.1	351.7	351.6	352.2	351.8

Table C.5: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Cheapest Insertion algorithm. For a concise summary, see Table 5.2.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
250	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
500	1.5	1.4	1.4	1.3	1.2	1.2	1.9	1.9	1.7	1.6	1.4
750	3.8	3.4	3.2	2.8	2.8	2.5	2.5	2.3	2.2	2.2	2.7
1,000	4.8	4.3	4.0	3.8	3.6	3.4	3.3	3.1	3.1	2.9	3.5

Table C.6: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Farthest Insertion algorithm. For a concise summary, see Table 5.3.

	Computation times (ms)										
n	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	Median
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
100	1.5	1.4	1.0	1.2	1.2	1.0	1.1	1.0	1.0	0.9	1.1
250	5.9	3.7	6.2	3.1	2.4	4.1	3.4	3.3	3.1	3.4	3.4
500	11.6	12.7	7.8	9.3	8.0	15.0	8.9	7.7	12.3	17.3	10.5
750	33.3	18.7	63.4	34.1	19.6	24.5	30.2	25.0	19.7	18.0	24.8
1,000	82.9	44.8	57.2	54.0	46.5	58.4	38.2	50.0	52.9	58.8	53.4

Table C.7: The computation times of all trials, $t_1 - t_{10}$, and their median for different values of n between 10 and 1,000 using the Christofides-Serdyukov algorithm. For a concise summary, see Table 6.1.

Bibliography

- Avigad, J. and Donnelly, K. (2004). Formalizing o notation in isabelle/hol. *Journal of Automated Reasoning*, 32(4):333–347.
- Beardwood, J., Halton, J. H., and Hammersley, J. M. (1959). The shortest path through many points. *Mathematical Proceedings of the Cambridge Philosophical Society*, 55(4):299–327.
- Chen, D., Imdahl, C., Lai, D., and Van Woensel, T. (2025). The dynamic traveling salesman problem with time-dependent and stochastic travel times: A deep reinforcement learning approach. *Transportation Research Part C: Emerging Technologies*, 172:105022.
- Christofides, N. (1976). Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University. Republished in Operations Research Forum, 2022.
- Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812.
- Goh, Y. L., Cao, Z., Ma, Y., Dong, Y., Dupty, M. H., and Lee, W. S. (2024). Hierarchical neural constructive solver for real-world tsp scenarios. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 884–895, New York, NY, USA. Association for Computing Machinery.
- Gutin, G. and Yeo, A. (2007). The greedy algorithm for the symmetric tsp. *Algorithmic Operations Research*, 2(1):33–36.
- Itai, A., Papadimitriou, C. H., and Szwarcfiter, J. L. (1982). Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686.
- Knuth, D. E. (1976). Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24.
- Li, C., Yang, M., and Kang, L. (2006). A new approach to solving dynamic traveling salesman problems. In Wang, T.-D., Li, X., Chen, S.-H., Wang, X., Abbass, H., Iba, H., Chen, G.-L., and Yao, X., editors, *Simulated Evolution and Learning*, pages 236–243, Berlin, Heidelberg. Springer Berlin Heidelberg.
- López-Ibáñez, M., Blum, C., Ohlmann, J. W., and Thomas, B. W. (2013). The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. *Applied Soft Computing*, 13(9):3806–3815.
- Mala, F. and Ali, R. (2022). The big-o of mathematics and computer science. *Journal of Applied Mathematics and Computation*, 6:1–3.
- Merchan, D., Arora, J., Pachon, J., Konduri, K., Winkenbach, M., Parks, S., and Noszek, J. (2022). 2021 amazon last mile routing research challenge: Data set. *Transportation Science*, 58.
- Oswald, M., Reinelt, G., and Theis, D. O. (2002). Traveling salesman problem. <http://comopt.ifl.uni-heidelberg.de/projects/projekttspsir/tspposter.html>. Accessed 7 October 2025.
- Percus, A. G. and Martin, O. C. (1996). Finite size and dimensional dependence in the euclidean traveling salesman problem. *Phys. Rev. Lett.*, 76:1188–1191.
- Reinelt, G. (1991). Tspplib—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384.
- Rosenkrantz, D., Stearns, R., and II, P. (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6:563–581.
- Salcedo-Sanz, S. (2016). Modern meta-heuristics based on nonlinear physics processes: A review of models and design procedures. *Physics Reports*.
- Toth, P. and Vigo, D. (2014). *Vehicle Routing*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

-
- van Bevern, R. and Slugina, V. A. (2020). A historical note on the $3/2$ -approximation algorithm for the metric traveling salesman problem. *Historia Mathematica*, 53:118–127.
- Wolfinger, D. and Salazar-González, J.-J. (2021). The pickup and delivery problem with split loads and transshipments: A branch-and-cut solution approach. *European Journal of Operational Research*, 289(2):470–484.