6CCS3PRJ Final Year
Vehicle Routing Problem

Final Project Report

Author: Nathaniel Tesfaye

Supervisor: Dimitrios Letsios

Student ID: 21059795

April 2024

Abstract

With the recent rise in e-commerce, urbanisation, and concerns over global warming, the need for efficient vehicle routing has never been more critical. This sets the stage for the Vehicle Routing Problem (VRP); a complex problem that seeks to optimise the route a vehicle takes to serve a set of customers. Given the multitude of algorithms that attempt to 'solve' the VRP, our project aims to perform a comprehensive exploration of several of these approaches. We aim to not only research, implement, benchmark, and analyse them but to introduce unique integrations thereof. Our goal is to introduce novel combinations of these algorithms that enhance their overall efficiency (both time, distance, and resource-wise), hence assessing their viability and eventually laying the groundwork for a future Vehicle Routing application that leverages these new solutions.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Nathaniel Tesfaye

April 2024

Acknowledgements

I would like to express my gratitude to my supervisor, Dr Dimitrios Letsios, for providing his guidance and expertise throughout the duration of the project. His enthusiasm and insightful feedback has not only made the project a pleasant experience, but has also fostered an environment for learning.

Contents

Chapter 1

Introduction

1.1 Thesis

This report aims to document the research, implementation, benchmarking, and subsequent analysis of various existing Vehicle Routing algorithms. We will then utilise the insights gained from this analysis to guide the development and subsequent proposition of novel Vehicle Routing algorithms. These novel algorithms aim to leverage the advantages of several of the existing algorithms studied to produce more consistent and effective solutions for the VRP. Ultimately, we will lay the groundwork for a prospective Vehicle Routing application by discussing how our newly implemented algorithms could be implemented within such an application.

1.2 Introduction

As it stands, a significant majority of businesses (72%) still carry out manual vehicle route planning. Carrying out manual planning results in routes that are, on average, 20.5% longer than the true optimal route[1]. In the world of business, where time is money, such inefficient routing can translate into significant financial losses, amounting to thousands of dollars in lost revenue for smaller companies. The larger the company is, the greater the ramifications are, with routing inefficiencies resulting in hundreds of thousands, if not millions, of dollars lost annually. Getting a hold of this issue should therefore become one of the top priorities for businesses, not just for the financial health of the business but also as a strategic move in a world where every second, dollar and customer counts.

[1] https://www.routific.com/route-optimization.

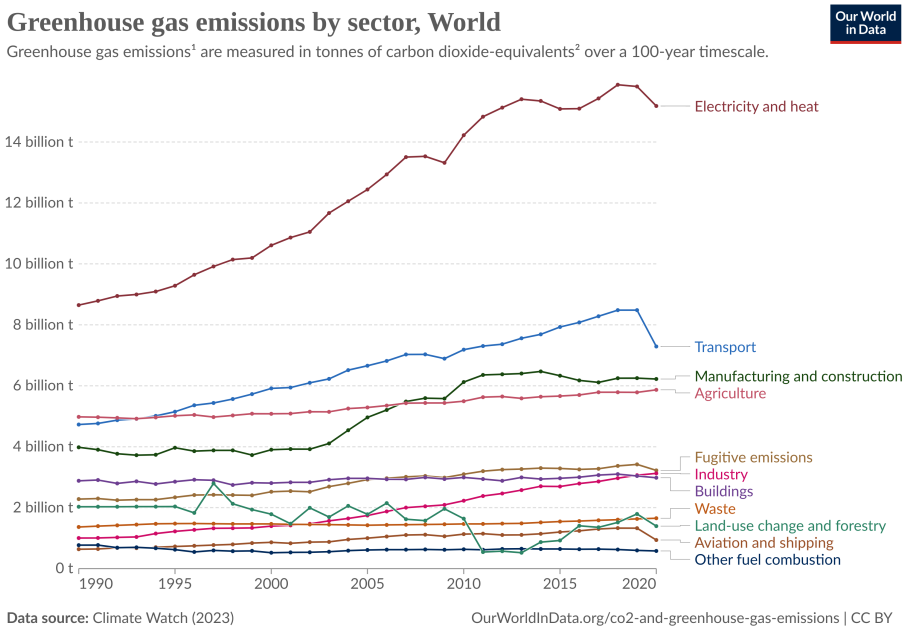

Figure 1.1: Greenhouse gas emissions as a byproduct of the Transportation Sector continue to trend upwards more than almost any other sector and it shows no signs of slowing down any time soon. *The trend downwards at the end is due to the reduction of vehicle usage during the height of the COVID Pandemic and Lockdowns, 2020.* Image: Our World In Data

However, e-commerce and the rest of the business landscape are not the only realms that face severe repercussions as a result of inefficient vehicle routing. The environment is another prominent victim. Transportation as a whole is a significant culrpit for CO_2 emissions globally, accounting for around one-fifth[2]. Road travel, which includes both passenger vehicles and freight trucks, account for the majority of these emissions. Passenger vehicles alone contribute 45.1%, while freight trucks are responsible for another 29.4%. In fact, in the United States, the transportation sector contributes to global warming more than almost any other sector at a whopping 30%[3]. All in all, this indicates that inefficiencies in routing, which result in longer travel distances and more time spent on the road, significantly exacerbate greenhouse gas emissions worldwide. Therefore, any steps taken to rectify these inefficiencies would represent

[2]https://ourworldindata.org/co2-emissions-from-transport
[3]https://www.ucsusa.org/resources/car-emissions-global-warming

a huge leap in the direction of reducing the sector's environmental footprint.

The shift towards a more urbanised yet sustainable world has also been made much more difficult by inefficient vehicle routing. According to a study by *Yuan Gao* and *Jiaxing Zhu* [7], economic growth, urban development and the boom of the automobile industry has resulted in the widespread adoption of private and public vehicles, which are now far more common modes of transport for daily commutes than walking or cycling. Modern cities, especially those heavily reliant on automobiles for travel, are unsure as to how they can balance their needs for modern infrastructure with issues like commuting distances, increased congestion and environmental degradation.

1.3 Project Motivation & Importance

So, by now the dire consequences of vehicle routing inefficiencies should be clear. These aforementioned issues have given rise to a new type of problem - the Vehicle Routing Problem (VRP). The VRP is a class of problems that aim to determine the optimal route that a vehicle should take to serve a set of customers. 'Optimal' can mean different things based on the context of the problem. For example, looking at each of the aforementioned issues, we can deduce the following:

- **Business & E-Commerce** - Businesses generally aim to keep profit margins as high as possible. Consequently, they aim to cut out unnecessary expenditure wherever possible. An 'optimal' route to a business would therefore typically be a route which minimises factors like travel time, travel distance, or fuel usage. The VRP can therefore be tailored to generate routes that serve all desired customers whilst also aiming to reduce travel time, travel distance, fuel usage (or any combination of the three) as much as possible.
- **Environmental Impact** - People concerned with environmental impact desire more sustainable operations worldwide through reduced carbon footprints, greenhouse gas emissions, noise pollution, air pollution, etc. To them, an 'optimal' route would be a route which mitigates the impact of all of these factors to the greatest possible degree. The VRP could be leveraged in this scenario to generate routes that serve all customers while reducing these negative environmental impacts.
- **Urbanisation** - While urbanisation has had its benefits, it has introduced several challenges for local and national authorities including (but not limited to) increased route

distances, congestion, pollution, etc. Therefore, an 'optimal' route from the perspective of these authorities would be a route that keeps these factors in mind. The VRP can be used to generate routes that keep the impact of these factors at a minimum.

As we've seen, vehicle routing has become more challenging yet more relevant as of late. In fact, the route optimisation software market is estimated to reach $12,416,000,000 by 2030[4]. Contributing a small part to this critical field through the introduction of my own algorithm(s) is an exciting prospect. Embarking on this project gives me an opportunity to contribute to this field, while helping me to develop a deeper understanding of what it takes to provide a solution to a real-world problem of this nature.

1.4 Real-World Applications

There are several practical applications of VRP. Here are some of the most common:

1. Courier Services
2. Public Transport Services
3. School Bus Services
4. Humanitarian Aid Distribution Services

1.5 Case Study

A case study which illustrates the tremendous influence of efficient vehicle routing algorithms can be found in a 2020 paper by *Giovanni Calabrò*, *Vincenza Torrisi*, *Giuseppe Inturri* and *Matteo Ignaccolo* [3]. In the paper, they document the improvement of a large-scale routing problem through the introduction of a novel ant-colony simulation-based optimisation algorithm (one of the many algorithms that tackle the VRP). The case study focuses on routing inefficiencies faced by a freight transport and logistics company in South Italy in their transportation of palletised fruit and vegetables from their various farm locations to the main depot. The goal was to introduce a vehicle routing algorithm to optimse the process by reducing the total distance travelled by the trucks, as well as minimising wasted space in the trucks, hence reducing the operational overhead for the company. This problem ishat are a bit more outside the boxn't classified solely as the Vehicle Routing Problem, but instead the Capacitated Vehicle

[4]https://www.upperinc.com/blog/route-optimization-trends-statistics/

Routing Problem (CVRP); an extension of the regular VRP that not only aims to minimise route distances, but also aims to optimise the load carried by each vehicle in a company's fleet.

The former inefficiency of the company's transportation process stemmed from their reliance on empirical experience to carry out their operations, hence missing out on opportunities to minimise route distances and wasted space. According to the study, this was the primary cause of the unnecessarily high operational costs and poor service timings.

The study remedies this issue through the introduction of an ant-colony simulation-based optimisation algorithm; a meta-heuristic algorithm which is renowned for its ability to generate routes close to the true optimal routes by simulating the behaviour of ants in finding the shortest possible path to food sources.

Although no specific numerical data was provided, implementing and utilising this more efficient vehicle routing algorithm had several notable impacts on the company's operations:

- **Reduced Total Distance Travelled** - The optimisations resulted in routes that were shorter in both time and distance, hence increasing the number of operations that could be carried out over time, reducing fuel costs, reducing vehicle wear and tear, etc.
- **Improved Load Factor**[5] - The optimisations resulted in more optimal vehicle loading and better usage of vehicle capacity, hence contributing to cost savings.
- **Reduced Number of Vehicles Used** - The study highlighted that the fact that load factor was improved meant that less of the company's vehicles had to be on the road due to more optimal load distribution, hence reducing operational costs further.
- **Other** - It can also be assumed that several other factors took a turn for the better, such as customer satisfaction, greenhouse gas emissions, etc.

In summary, the case study shows that employing a vehicle routing algorithm, specifically the ant-colony algorithm, resulted in significant advancements in vehicle travel distances, vehicle travel times, vehicle load factors and the number of vehicles deployed. As a result, the company was able to significantly upgrade their logistics operations and reduce costs, hence leaving them with a much more sustainable operational model. This case study therefore clearly demonstrates the profound impact that efficient vehicle routing algorithms can have in the real-world.

[5]Load Factor - Amount of Load carried by a Vehicle as a Percentage of its overall Capacity

1.6 Project Aims

In the forthcoming sections, we will explore a range of existing algorithms designed to provide solutions to the regular Vehicle Routing Problem. I aim to begin this project by documenting my research, implementation, testing and benchmarking of these existing algorithms. This will be followed by a thorough analysis of these benchmarked results, examining the advantages of disadvantages of each. I then aim to leverage the insights gained from this examination to exploit the advantages of some of the algorithms to form my own novel vehicle routing algorithm. Finally, I will put all the pieces together in an attempt to lay the groundwork for a prospective Vehicle Routing application that could be built in the future.

1.7 Report Structure

In the next chapter (Background), we'll take a closer look into the technical details of the Vehicle Routing Problem.

In chapter 3, we'll take a high level look at 6 carefully chosen algorithms that currently exist to provide solutions to instances of the VRP. Chapters 4 & 5 will consist of the requirements, specifications and design of both the existing algorithms and the novel algorithm(s).

Chapter 6 will begin with the documentation of the actual implementation, testing, verification and benchmarking of the existing algorithms. It will involve an analysis of the benchmarked results obtained upon running the existing algorithms, they will be compared against each other, and conclusions will be drawn as to how we're going to implement the novel algorithm. The documentation of the implementation, testing and benchmarking of the novel algorithm will then follow.

The penultimate chapter (chapter 7) will involve the evaluation of both my code deliverables and the quality of the process I followed throughout the project, as well as any potential future work that could be carried out. The final chapter (chapter 8) will be a brief chapter discussing the relevant professional standards, social and ethical implications.

Chapter 2

Background

2.1 Introduction to Vehicle Routing Problem

2.1.1 Definition and Aims

The Vehicle Routing Problem is an optimisation problem that aims to find the most efficient route for a vehicle to travel, visiting a specific set of locations[1] before returning to the starting location. By the most 'efficient' route, we mean the route which minimises the total cost[2], which includes not only common factors like distance travelled and time taken, but also factors such as fuel consumption and environmental impact. Take Amazon's daily delivery operations as an example. Amazon has several warehouses that need to dispatch numerous vehicles to deliver to thousands of addresses in the area. For Amazon and all of its stakeholders, leveraging the VRP to navigate the situation is crucial. By applying VRP solutions, customers are able to receive orders in the most timely and cost-effective manner, and Amazon are able to maximise profit margins.

2.1.2 Origin

The VRP is widely recognised to have been first formally introduce in a 1959 paper by *George B. Dantzig* and *John H. Ramser* [6]. It discusses the optimal routing of a fleet of gasoline trucks to several service stations. The goal is to minimise vehicle mileage while still meeting the stations' service demands.

[1] These 'locations' may be referred to as 'nodes', 'waypoints', 'cities', 'customers', 'vertices', or 'points' throughout this paper.

[2] The notion of 'cost' will be explored in more detail in section 2.2.

Both authors then go on to relate the VRP to the Travelling Salesman Problem (TSP), describing the VRP as a 'generalisation' of the TSP[3]. They mention that this is because while the TSP aims to visit every point on the graph and return to the starting point, the VRP does the same thing but replaces 'points' with 'customers' and often simultaneously works under certain capacity constraints (the gasoline trucks in the study being an example). Dantzig and Ramser then proceed to provide one of the seminal linear-programming solutions to the VRP.

This solution was considered innovative at the time, and laid the groundwork for several future VRP algorithms, solutions, and studies which followed. Despite the limited computational resources available then, the study showcased the potential for using the VRP to solve complex, real-world logistical problems. For this reason, Dantzig and Ramser's work is considered to be one of, if not the, most influential pieces in the history of the vehicle routing field. As an example, the work was utilised and iterated on by *G Clarke* and *J W. Wright* in their 1964 paper *"Scheduling of Vehicle Routing Problem from a Central Depot to a Number of Delivery Points [5]."* Essentially, Wright and Clarke improved on the algorithm proposed by Dantzig and Ramser by introducing their own algorithm called the *"Savings Algorithm"*, which worked to provide a solution to the VRP much more efficiently than that of Dantzig and Ramser.

2.2 Technical Foundations

2.2.1 Preliminaries

There are several preliminary concepts that must be understood before we can proceed to exploring the VRP in more technical depth.

Graph Theory

Graphs are pivotal in modelling and solving the TSP, so must be deeply understood. After all, a map with a set of locations and roads can be abstracted down to a graph.

- **Graph** - A graph G is a pair $G = (V, E)$, where V represents a set of vertices and E represents a set of edges. Each edge e connects a pair of vertices $v1$ and $v2$, where $e \in E$ and $v1, v2 \in V$. In VRP, these vertices typically represent customers, and an edge typically represents the route to travel between two customers.
- **(Un)directed Graph** - A directed graph is a graph G where each edge e has a direction.

[3]The relationship between the VRP and the TSP will be explored in more depth in section 2.2.

Each edge e in a directed graph is represented as a pair ($v1$, $v2$), where $v1, v2 \in V$, and the edge e connecting $v1$ and $v2$ starts at vertex $v1$ and ends at vertex $v2$. On the other hand, edges in undirected graphs are considered to have no direction. Both directed and undirected graphs are often seen in VRP.

- **Complete Graph** - In a complete graph, every unique pair of vertices is connected by one individual edge. Assuming a complete directed graph has n vertices, the total number of edges would be $n \cdot (n-1)$. Assuming the graph is complete and undirected, that number would be $\frac{n \cdot (n-1)}{2}$. Complete graphs are an important notion in VRP because they indicate that every location is reachable from every other location, which is a common assumption in VRP formulations.

- **Weighted Graph** - A weighted graph is a graph in which every edge has an associated 'weight'. These weights are often referred to as 'costs'. These 'costs' represent the distance, time, fuel consumption, or any other applicable metric that's expended when travelling from one vertex to another through that specific edge. We will explore the notion of 'cost' later in this section. Weights form the cornerstone of an optimal VRP solution; by minimising the summated weight of traversed edges, we minimise the overall 'cost', hence allowing for a more optimal solution.

- **Subgraph** - A subgraph S consists of a set of vertices V_s that is a subset of the set of vertices V of the larger graph G, denoted by $V_s \subset V$. The subgraph's set of edges E_s is also a subset of the set of edges E of the larger graph G, denoted by $E_s \subset V$. Subgraphs are important in VRP for multiple reasons, for example when only a subset of customers need to be visited.

- **Path** - A path is a sequence of vertices $v1, v2, v3, ..., vx$, where each consecutive vertex is connected by an edge in the set E. The length of a path is usually computed as the summation of the weights of each edge in the path. Paths are crucial in VRP because they represent the route that a vehicle must take to serve a set of customers.

- **Cycle** - A cycle is a path that starts at vertex $v1$, goes through a unique set of vertices and then returns to vertex $v1$. In the VRP, cycles represent routes both starting and ending at the same location, which is extremely common.

- **Hamiltonian Cycle** - A cycle that visits every vertex in the graph exactly once (barring the starting vertex which is visited twice; once at the start and once at the end).

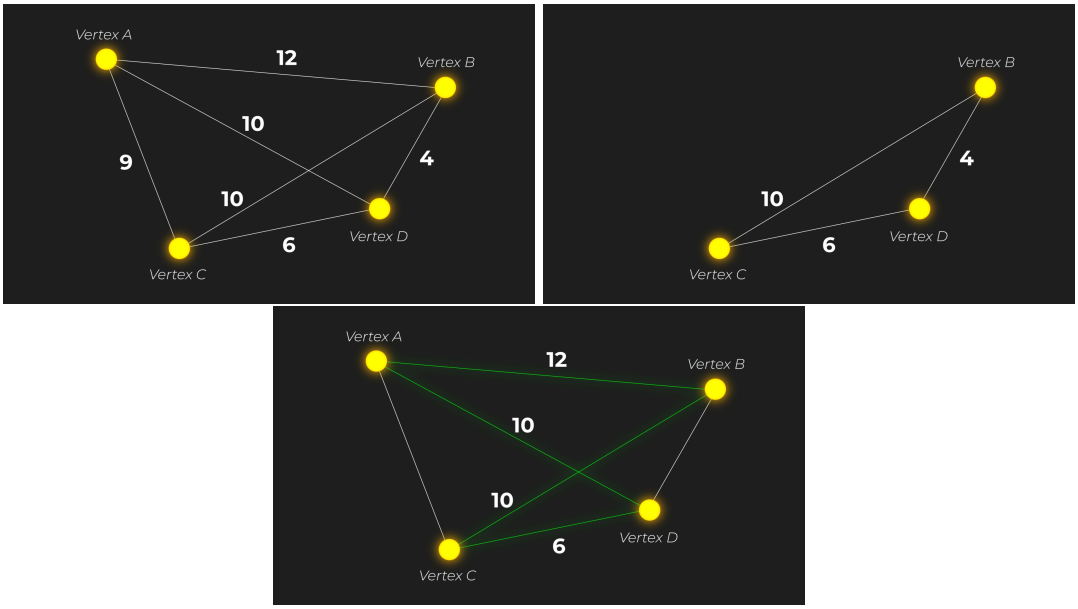

Figure 2.1: The first image (top left) depicts a complete, undirected, weighted graph G. The weights of each edge are labelled. In this case, the weights represent the distances between each pair of vertices (not to scale). The second image (top right) depicts a subgraph of the graph G. The third image (bottom center) depicts one possible Hamiltonian Cycle of the graph G. The edges in the Hamiltonian Cycle are highlighted in green. The cost of this Hamiltonian Cycle is $12 + 10 + 10 + 6 = 38$. Images: Self

.

VRP-specific Notions

- **Depot** - The starting/ending point of a vehicle's route.
- **Route** - A sequence of waypoints that a vehicle passes through on it's route.
- **Customer** - Each individual waypoint that the vehicle passes through to serve on its route.
- **Total Cost** - The total expense accumulated by a vehicle on its route, such as distance travelled, time elapsed, fuel expended, etc.
- **Objective Function** - Determines the goal of the problem. This typically involves minimising cost while maximising service.
- **Feasible Solution** - A solution that does not violate any problem constraints.
- **Optimal Solution/Optimal Route** - The feasible solution (route) that minimises cost to the greatest degree.
- **Runtime** - The amount of time it takes for a VRP algorithm to execute.

- **Solution Quality** - How well the solution satisfies the objective function.
- **Computational Complexity** - How much resources are required to run an algorithm.

Cost

Cost is a critical metric in the context of VRP. It is the way in which we can evaluate the effectiveness of each feasible solution obtained by a vehicle routing algorithm by representing the resources expended as a single numerical value. In the VRP, cost is typically one of, but not limited to: distance travelled, time taken and fuel consumed.

If we were to take distance travelled as the metric of cost of a route, then the route's cost would be calculated as the sum of the distances of all edges traversed. For example, in Figure 2.1, the bottom center image depicts a feasible TSP solution to the graph provided. There are 4 edges traversed in the route. These 4 edges have weights of 12, 10, 10 and 6, where these weights are considered to be the distance between each pair of nodes. The 'cost' of this route would therefore be $12 + 10 + 10 + 6 = 38$.

Solving the VRP in a way that minimises cost is vital for several reasons, many of which we have seen in Section 1 of this paper.

2.2.2 Relationship to Travelling Salesman Problem (TSP)

The Vehicle Routing Problem is an extension of the classic optimisation problem, the Travelling Salesman Problem (TSP). A definition of TSP is as follows:

- **Formulation** - TSP is formulated as a complete graph, defined as $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges. In terms of VRP, this could be thought of as a road network, with roads connecting different locations together.
- **Edge Cost** - There is an associated cost with each edge $e \in E$, representing the cost of moving from vertex $v1 \in V$ to vertex $v2 \in V$ via e. In terms of VRP, this cost could potentially be the distance travelled, time taken, or fuel expended to travel from one location to another via a road.
- **Objective** - The objective of TSP is to find a Hamiltonian Cycle in the graph G such that the total cost of all edges traversed is minimised. In terms of VRP, let's take courier companies as an example. This route starts at the depot, visits all designated customers

for that journey via a route that aims to minimise the total cost, and then returns to the depot.

From this point forward, we will approach solving the VRP as solving the TSP. We have seen that this is a sound approach when we looked at *The Truck Dispatching Problem, 1959.*

2.2.3 Computational Complexity Classes

TSP is part of a class of computational problems known as NP-hard problems; problems for which there is no known polynomial-time algorithm that exists for solving all instances of the problem optimally. As the number of nodes increases, the number of possible cycles grows factorially with it. More specifically, for a complete, undirected graph with n vertices, there are $\frac{(n-1)!}{2}$ possible cycles(2). This factorial growth makes it computationally infeasible to compute all possible cycles and simply select the most optimal one for a large number of vertices. In other words, the *Brute Force* algorithm for solving TSP is computationally infeasible for large values of n.

Figure 2.2: Attempting to solve the Vehicle Routing Problem for the 8 largest cities in the UK via Brute Force - there are 2,520 possible routes to examine. Simply adding in the next 5 largest cities brings this number up to 239,500,800, which could be considered computationally infeasible. Images: Self

In fact, this is typical for other algorithms as well. There are several algorithms that can find optimal tours for smaller instances of TSP quite easily, but none of these algorithms are suitable for larger instances because they grow exponentially.

So, given the NP-hard nature of TSP, finding a polynomial-time algorithm that solves all TSP instances optimally remains impossible. Consequently, we often instead resort to methods that provide sufficiently effective solutions within reasonable timeframes, as opposed to solutions of absolute optimality. This way, we can get down to polynomial growth [11]. This pragmatic approach will be explored in more depth in section 2.3.

2.3 Solution Approaches

As we've just seen, finding a polynomial-time algorithm that solves all TSP instances for any value of n remains impossible. We'll now take a high-level look at 3 different types of algorithms that navigate this problem by leveraging their own unique strengths and weaknesses: Exact Algorithms, Heuristics and Meta-Heuristics. These different algorithms are employed in varying scenarios based on the size of the problem, the need for solution quality and the need for speed.

2.3.1 Exact Algorithms

Exact algorithms are algorithms that can guarantee the solution provided will be optimal. They use formal mathematical models to provide a meticulous framework by which they will attack the problem instance[4]. They usually require both an objective function and a set of constraints to be passed to them. They then proceed to systematically explore the search space to retrieve the optimal solution subject to these constraints.

The idea of guaranteeing an optimal solution sounds too good to be true... and for small instances it is indeed typically flawless! However, exact methods generally fall into the NP-hard complexity class, with its runtime generally increasing exponentially with n (sometimes reaching $O(2^n)$ or worse[5]!), hence making it nigh on impossible to use for TSP instances with high numbers of nodes.

Examples of Exact Algorithms include:

- Mixed-Integer Linear Programming
- Branch-and-bound
- Branch-and-price

2.3.2 Heuristics

Heuristics are methods that provide quick, satisfactory solutions to complex problems[6]. They operate without exploring the entire search space. Unlike exact methods, they are relatively rapid, but this comes at the cost of optimality. The gulf between the two becomes increasingly evident as n is escalated.

[4]https://towardsdatascience.com/exact-algorithm-or-heuristic-20d59d7fb359
[5]https://www.baeldung.com/cs/tsp-exact-solutions-vs-heuristic-vs-approximation-algorithms
[6]https://softjourn.com/insights/heuristic-programming

Examples of Heuristics include:

- Nearest Neighbour
- Insertion Heuristic
- Two-Opt

2.3.3 Meta-Heuristics

Meta-Heuristics are higher-level heuristics that employ more strategic approaches to finding near-optimal solutions. They are more flexible than regular heuristics, and are capable of providing higher quality solutions with more ease [9]. Meta-heuristics are far less prone to converging to local optima in comparison to regular heuristics, which is a common pitfall of regular heuristics for larger scale problem instances.

Examples of Meta-Heuristics include:

- Genetic Algorithm
- Large Neighbourhood Search
- Ant-Colony Optimisation

We will inspect some of these algorithms more deeply from section 4 onwards.

2.4 Metrics & Trade-offs

There are several metrics by which a TSP algorithm can be measured. The most notable are runtime and solution quality, although other viable metrics include CPU time, memory usage, solution conistency, etc.

2.4.1 Runtime vs Solution Quality

The main trade-off to consider in the development and selection of a TSP algorithm is often the runtime of the algorithm vs the desire for solutions of higher quality. It is practically a rule of thumb that higher quality solutions induce higher runtimes. We've seen this with exact methods for example, where the pro of obtaining an exact solution is starkly contrasted by the con of exponentially increasing runtimes. On the contrary, algorithms with lower runtimes (such as Heuristics) tend to also produce solutions of lower quality.

The circumstances surrounding the problem instance heavily influence which of the two metrics to lean further towards.

A situation where Solution Quality would be prioritised:

- **Interplanetary Route Planning** - Interplanetary Expeditions are undertaken relatively infrequently. The stakes are high and the consequences of getting things wrong can be dire. Therefore, it makes sense to sacrifice runtime in favour of higher solution quality.

A situation where Runtime would be prioritised:

- **Courier Services** - Courier companies like Amazon handle extraordinary numbers of orders daily. In 2022, this amounted to 13.13 million orders per day in the US, which equates to roughly 9,116 orders per minute[7]. This astronomical volume of demand, paired with the dynamic nature of ordering online, means it would make more sense for Amazon to favour runtime slightly ahead of solution quality.

The main metrics we will be focusing our benchmarking in section 6 on will be runtime and solution quality, although other metrics may be explored too.

2.5 VRP Variants

Given the vast array of logistical challenges raised in the real-world, VRP has evolved to adopt several different variants. If required, several different variants can be combined. Here's a brief outline of some of them:

- **Regular VRP (VRP)** - at its core, solving the regular VRP is the same as solving the regular TSP.
- **Capacitated VRP (CVRP)** - CVRP introduces the constraint of the vehicle(s) having a capacity, factoring in the goods that a vehicle has to carry and deliver to specific customers.
- **VRP w/ Time Windows (VRP-TW)** - VRP-TW introduces the constraint of vehicles only being able to serve specific customers at a specific time.
- **Multi-Depot VRP (MD-VRP)** - MD-VRP includes multiple depots, where potentially several vehicles have to be routed between the customers and these various depots.

[7] https://capitaloneshopping.com/research/amazon-logistics-statistics/

In this project, we will be focussing on the regular VRP, and treating it in a similar way to the TSP. This is strategic; the regular VRP forms the foundation upon which all other variants are built. Developing a deeper understanding of the regular VRP can therefore serve as a springboard to adopting the other variants in the future. Furthermore, the regular VRP is relatively simpler, hence allowing for more clarity in the investigation and proposition of the algorithms associated with it.

Chapter 3

Algorithms

3.1 Introduction

There are 6 base algorithms that will be implemented and benchmarked in this project. They are:

- Nearest Neighbour
- Insertion Heuristic (w/ Convex Hull)
- Mixed-Integer Linear Programming
- Two-Opt
- Large Neighbourhood Search
- Genetic Algorithm

The process used to select these 6 algorithms was strategic. Considering the project's timeframe, 6 seemed like a reasonable and rational number. The selected algorithms span a comprehensive spectrum, including heuristics, meta-heuristics and exact methods; this provides a chance to have a thorough look at the trade-offs between runtime, solution quality and scalability all across the board. Furthermore, this assortment has great potential for facilitating hybrid approaches (which we saw was a very promising avenue in the Literature Review). Overall, this appears to be a sound methodology to follow.

3.2 Individual Algorithms Overview

In chapters 4 and 5, we will run through the requirements and specifications on how we're going to implement, test and benchmark these algorithms. However, in this section (3.2) we're going to run through how these algorithms work, their strengths/weaknesses, and any other elements that should be made known.

3.2.1 Nearest Neighbour

The Nearest Neighbour (NN) algorithm is a heuristic that is arguably the most intuitive and straightforward TSP heuristic. It works by starting at an arbitrary vertex and iteratively visiting the nearest unvisited vertex. When all vertices have been exhausted, it returns back to the starting vertex.

Algorithm 1: Nearest Neighbour

Data: A symmetric, complete graph G
Result: A list of vertices representing the route to take

```
1 Select a random vertex s from V; //this is our starting vertex
2 while there are unvisited vertices do
3 |  Append the nearest unvisited vertex to the end of route;
4 Append s to the end of route;
5 return route;
```

As is often the case with heuristics, NN prioritses efficiency over optimality. In NN's case though, this is extreme. NN has a runtime complexity of $O(n^2)$ [11], where n represents the number of vertices. The reason for this is because the heuristic works by stopping at the current vertex and searching through all unvisited vertices from that point in order to find the next vertex to move to. Mathematically, this means that we'd have $n \cdot \frac{(n-1)\cdot(n-2)\ldots}{2}$, which, when expanded out, simplifies to $O(n^2)$ (because n^2 would be the highest order term of the expression). This represents a quadratic rate of growth with respect to n. Due to NN's runtime efficiency, it can be executed quite a bit faster than other algorithms. This is especially true for larger values of n.

However, this is speed is at the cost of the optimality of the routes which NN produces, especially for larger values of n. One reason for this is NN's tendency to converge to local optima. As we've seen, NN iteratively selects the nearest vertex to the vertex it is currently examining, without considering the context of the current route being formed as a whole. This becomes especially evident when there are a set of vertices clustered together that NN leaves to

the end of the route, which makes it even more likely that a suboptimal route will be produced.

3.2.2 Insertion Heuristic (w/ Convex Hull)

Insertion Heuristic w/ Convex Hull (IHCV) works by first finding the Convex Hull of the initial graph G. This Convex Hull serves as our initial subtour[1]. We then find the 'cheapest' vertex (that currently isn't in the subtour) to insert into the subtour, such that the cost of adding this vertex to the subtour is minimal. We repeat this insertion of the 'cheapest' vertex (that isn't in the subtour) until all vertices from G are part of the subtour.

Algorithm 2: Insertion Heuristic (w/ Convex Hull)

Data: A symmetric, complete graph G
Result: A list of vertices representing the route to take

```
1  Compute the Convex Hull of G as a list of vertices and assign this to a new variable
     route;
2  Obtain a list of all vertices not inside route and assign this to a new variable
     unvisited_vertices;
3  while len(unvisited_vertices) > 0 do
4      Initialise cheapest_vertex as an empty variable;
5      Initialise cheapest_cost as infinity;
6      Initialise cheapest_position as an empty variable;
7      for each vertex in unvisited_vertices do
8          for each possible position in route do
9              Calculate the cost of inserting the vertex at this position;
10             if cost < cheapest_cost then
11                 Update cheapest_cost and cheapest_vertexaccordingly;
12     Delete cheapest_vertex from unvisited_vertices;
13     Insert cheapest_vertex into route at cheapest_position;
14 return route;
```

Due to the computation of the Convex Hull, Insertion Heuristic begins with a more 'global' subtour upon which it will build its route. It is not abnormal for several of the Convex Hull's edges to align with the true optimal route. The vertices that are then inserted into the route are on the basis of the least cost induced (which avoids the pitfalls of NN by not showing bias to optimising the route locally), hence producing more optimal solutions for the most part.

However, this tendency towards heightened accuracy results in a less efficient and more computationally expensive process. Insertion Heuristic w/ Convex Hull exhibits a runtime complexity of $O(n^2 log(n))$ [11]. Assuming that the Convex Hull is formed using Graham Scan

[1] We'll soon begin to notice a pattern emerging with the algorithms that produce higher quality solutions; they almost exclusively begin with some sort of initial solution, then begin to iterate on/optimise that. This is one proven way to improve the likelihood of escaping local optima.

(as it will do in our implementation of the heuristic), we can deduce that Insertion Heuristic w/ Convex Hull's runtime complexity stems from the $O(nlogn)$ [15] time complexity of Convex Hull and the assumed $O(n^2)$ time complexity of the insertion process. All in all, while the heuristic (again) isn't guaranteed to produce an optimal solution, it typically does produce more optimal solutions than NN at the cost of computational efficiency. This is especially true as the value of n increases.

3.2.3 Mixed-Integer Linear Programming

Mixed-Integer Linear Programming (MILP) is an exact method, and therefore provides optimal solutions to TSP instances. The algorithm works by utilising an **Objective Function**[2], **Constraints**[3] and a set of **Decision Variables**[4].

MILP **Frameworks** act as the vessel through which the Objective Function, Constraints and Decision Variables of the problem can be systematically represented. These, along with the rest of the problem formulation, form what is known as a 'model'. Frameworks pass this model to the **Solver**[5].

Solvers then proceed to compute the optimal solution using a specific method. One of these methods is Branch-and-bound(1). This can be considered a major advantage of MILP approaches to solving the TSP because solving the problem is majorly abstracted; the only thing required of the user is to model the problem correctly!

Algorithm 3: Solving MILP Problems

1: Define Framework
2: Define Objective Function, Variables, Constraints, and the rest of the Model
3: Define Solver
4: Pass Model to Solver
5: *Solver solves Problem*

MILP is an exact method, and therefore is of NP-hard nature. As a result, it is rather challenging to reliably define its runtime complexity. While it can technically take exponential time to execute for larger problem instances, modern solvers have improved to the point where this is often not the case.

MILP is extremely precise, being capable of modelling a vast array of problems in great

[2]An expression which determines what we wish to maximise/minimise.
[3]A set of expressions which limit the values that the decision variables can hold.
[4]Represent the decisions that need to be made in order to maximise/minimise the Objective Function.
[5]Algorithms/Software which, given a model, act to compute the optimal solution to optimisation problems.

Algorithm 4: Example MILP Modelling, Python, Framework: Pyomo, Solver: Gurobi

Define Framework:
import pyomo.environ as pyo
from pyomo.opt import SolverFactory ▷ Imports

Define Model:
model = pyo.ConcreteModel()

Define Variables:
model.x1 = pyo.Var(within=pyo.NonNegativeReals)
model.x2 = pyo.Var(within=pyo.NonNegativeReals)
model.y = pyo.Var(within=pyo.Binary) ▷ Variables: x1 (any positive number), x2 (any positive number), y (binary, 0 or 1)

Define Objective Function:
model.obj = pyo.Objective(expr=5*model.x1 + 8*model.x2 + 10*model.y,
sense=pyo.maximize) ▷ Obj. Function: Maximize 5x1 + 8x2 + 10y

Define Constraints:
model.c1 = pyo.Constraint(expr=2*model.x1 + 4*model.x2 + 5*model.y ¡= 10)
model.c2 = pyo.Constraint(expr=3*model.x1 + 2*model.x2 - 5*model.y ¿= 3)
model.c3 = pyo.Constraint(expr=model.x1 - 3*model.x2 + model.y == 1)

Define Solver:
opt = SolverFactory('gurobi')

Pass Model to Solver, Solve Problem:
result = opt.solve(model)

Print Solution:
print(model.x1.value)
print(model.x2.value)
print(model.y.value) ▷ Optimal Solution: x1 = 3.4, x2 = 0.8, y = 0, obj = 23.4

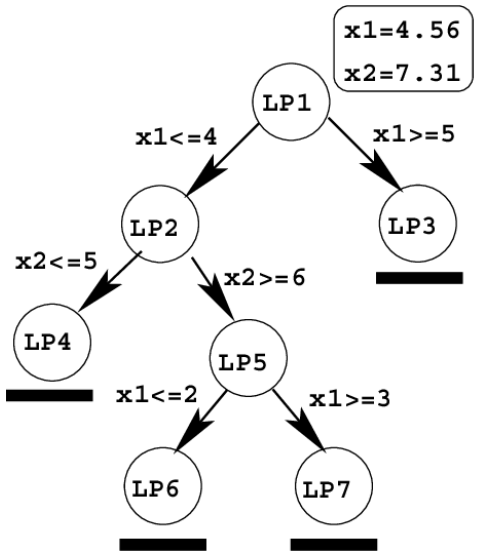

Figure 3.1: An image depicting a high-level view of how Linear Programming uses Branch-and-bound through divide-and-conquer [8]. The decision variables (*x1*, *x2*) are iteratively constrained at each node in an attempt to find the optimal solution (by finding the optimal value for each decision variable). It begins with a relaxed solution at the first node (LP1), before branching down by iteratively adding constraints, hence creating numerous subproblems as it converges closer and closer to the true optimal solution. The thick horizontal lines indicate the termination of a branch at the point of a subproblem being unable to surpass its current best value. Image: https://www.researchgate.net/

detail. As with all exact models, it is capable of producing the true optimal solutions for optimisation problems, assuming that ample time/computational resources are provided. It stands as a stark contrast to heuristics like NN and Insertion Heuristic in that sense, which are on the opposite end of the spectrum. Indeed, the glaring con of MILP is its NP-hard nature. As mentioned previously, this results in it's runtime potentially increasing exponentially as n increases, particularly for large values of n. This makes MILP often only desirable for smaller instances of TSP/VRP, where an optimal solution can be guaranteed in a somewhat satisfactory span of time. So, while heuristic methods can provide rapid solutions of decent quality, MILP and other exact methods are unrivalled when it comes to solution quality.

3.2.4 Two-Opt

Two-Opt is an intuitive heuristic that works by taking in an initial route, and iteratively swapping two edges in the tour with two others. This action is known as the 2-opt swap move. This is continued until it no longer yields a shorter route distance. This shortest obtainable route is 2-optimal [11].

Two-Opt is deemed to be a rather unique heuristic, mainly due to its nature of improving

Algorithm 5: Two-Opt

Data: A symmetric, complete graph G
Result: A list of vertices representing the route to take

```
1  Set the initial route to be route;
2  Set improvement to true;
3  while improvement is true do
4      for i in 1 → len(route) − 2 do
5          for j in i + 1 → len(route) do
6              if Swapping edges (i, i + 1) and (j, j + 1) results in route length decreasing
                then
7                  Set improvement to true;
8                  Swap the edges;
9              else
10                 Set improvement to false;
11 return route;
```

upon existing solutions[6]. The algorithm is renowned for its speed and relatively low usage of computational resources. Two-opt is generally considered to exhibit a runtime complexity of $O(n^2)$, where n is the number of vertices. This stems from the algorithm having to evaluate every possible pair of edges. However, more strategic implementations can reduce the runtime complexity down to $O(mn)$[7] [11]. However, just like the other heuristics, it often struggles to generate a global optimum solution. This is made worse by the fact that the quality of the solution produced by two-opt is very dependent on the quality of the initial solution passed to it; the worse this initial solution is, the harder it makes it for two-opt to get us closer to the global optimum. In fact, this makes two-opt quite vulnerable to converging to local optima, especially as n gets larger. This is the reason why two-opt is not considered desirable on its own, but is very enticing when used in tandem with other algorithms.

Unlike Nearest Neighbour and Insertion Heuristic w/ Convex Hull, two-opt doesn't generate its own initial solutions, but instead refines existing ones. Therefore, it shouldn't been deemed to be competing with these heuristics, but rather seen as having potential to complement them. If initial solutions of high enough solution quality are passed to it, it is capable of being a formidable algorithmic tool.

[6] This is interesting; it suggests that two-opt may have a vital role to play as part of a hybrid approach in our novel algorithm(s)

[7] We can prune the search space to only look at the 'm' nearest neighbours when considering which edges to swap, which can significantly better runtimes.

3.2.5 Large Neighbourhood Search

Large Neighbourhood Search (LNS) is a metaheuristic approach to solving the TSP. It takes in an initial solution and proceeds to iteratively destroy and repair parts of the solution until no further improvement in solution quality is registered. Unlike the heuristics that we have seen thus far, LNS operates on a more global scope, and is therefore far less likely to prematurely converge to local optima.

Pisinger and Ropke (2010) [13] provide the following explanation and example for **destroy** and **repair** in the context of LNS:

- The 'destroy and repair' mechanism of LNS allows us to explore a wider range of possible solutions beyond local ones
- Initially, a portion of the current solution is 'destroyed'. Assuming 15% of customers are removed from a current solution which has 100 customers in total, 85 customers would be remaining. A basic implementation of LNS selects these 15 nodes at random[8]. However, we're now left with an incomplete solution with shortcut edges and gaps.
- Following destruction, the repair phase is carried out, in which the removed nodes from the destroy phase are reintroduced into the solution to reconstruct a different solution. Heuristics are often strategically employed in this reinsertion in order to guide the nodes into more optimal positions.
- Through this iterative destroy and repair process, LNS is capable of exploring a 'large neighbourhood', and is therefore far more likely to diverge away from local optima and converge toward global optima.

Large Neighbourhood Search is considered by many to be the algorithm which manages the trade-off between runtime and solution quality the best. This is especially true for large values of n, where its solutions often don't lie too far away from the global optimum despite its runtime is far superior to that of exact methods[9]. The primary cause for this is that, unlike the heuristics that we've examined thus far, LNS explores a 'large neighbourhood', hence being able to escape local optima that regular heuristics may have otherwise been trapped in. Due to this tendency to diverge away from local optima, LNS is deemed a much more flexible and optimal

[8]Some say that this random selection allows for diversity to be injected into the problem, which is an opportunity for divergence away from local optima. Others say that this technique lacks structure and is therefore shouldn't be an avenue to pursue.

[9]This should be observable when implementing and running the algorithms in section 7.

Algorithm 6: Large Neighbourhood Search

Data: A symmetric, complete graph G
Result: A list of vertices representing the route to take

```
Set the initial route to be route;
Set improvement to true;
while improvement is true do
    Set destroyed to Destroy(route);
    Set repaired to Repair(route);
    if len(route) < len(route) then
        Set route to repaired;
        Set improvement to true;
    else
        Set improvement to false;
return route;
```

approach than regular heuristics, while also being much more practical than exact methods, hence earning its status as one of the industry-standard algorithms.

3.2.6 Genetic Algorithm

Genetic Algorithm (GA) is a meta-heuristic derived from the biological process of natural selection. Natural selection is Charles Darwin's theory of 'survival of the fittest', a theory which states that the 'fittest' individuals of each generation are the most likely to survive to breed, hence passing their genes down to the proceeding generation. In the context of GAs, individuals are analagous to potential solutions, and the 'fitness' of an individual describes how optimal the solution is [14]. The 'fittest' individual is the most optimal solution found.

Key GA Concepts:

- **Individual/Chromosome**[10] - One potential solution to the problem, typically displayed as an encoding of the problem. For example, in the context of TSP/VRP, one potential encoding of a route as a solution could be [1 4 5 3 2 1], which indicates a route which starts at city 1, goes to city 4, then 5, then 3, then 2, then back to 1. Each of these nodes represent a **gene**. Specifically, gene 2 would be city 4, gene 4 would be city 3, etc.
- **Initial Population** - GA begins with an initial population of individuals, which are often generated randomly. The number of individuals in this initial population is determined by the **population size**, which is a GA parameter that determines the number of individuals in each **generation**.

[10] We will use the terms 'Individual' and 'Chromosome' interchangeably from now on.

- **Generation** - A population of individuals at a particular iteration of the GA's execution.
- **GA Parameters** - A set of settings that can be configured to influence the performance and efficacy of the GA's execution. Includes parameters like **population size**, **mutation rate**, **crossover rate**, and **maximum number of generations**.
- **Fitness** - Each individual has an associated fitness, which determines how 'fit' said individual is. In VRP, this could be the route's distance for example.
- **Selection** - The process by which individuals are selected to breed and produce offspring for the next generation. Individuals are often selected on the basis of best fitness, a concept known as **elitism**. The selected individuals are called **parents**.
- **Crossover** - The process by which the genes of two parents are combined to produce offspring for the next generation. The likelihood that two parents will crossover to produce offspring is called **crossover rate**.
- **Mutation** - Introduces change on a random basis to the genes of individuals in the population. The likelihood that an individual will undergo mutation is called **mutation rate**.
- **Termination Condition** - A condition that, when fulfilled, invokes termination of the GA's execution. This could be hitting the maximum number of generations, not seeing improvement in the best fitness value over x generations, etc.

Algorithm 7: Genetic Algorithm Process [14]

1: **Initialise** the population by randomly generating n individuals, where $n = POPULATION\ SIZE$. This will constitute our first generation.
2: **Evaluate** the fitness of all individuals in the generation.
3: **Select** parents to breed, often on the basis of best fitness.
4: Undergo **crossovers** and **mutation** in the population, hence generating new individuals for the next generation.
5: **Replace** individuals from the old generation with some of the new offspring, hence forming a new generation.
6: **Repeat** this process until a termination condition has been fulfilled.

GAs usage of individuals and generations allows it to parallelise its execution; unlike the other heuristics and metaheuristics that we have explored (which start with a single solution and aim to iteratively improve that), GA's develop and evaluate numerous possible solutions simultaneously. This allows GAs to explore a vast area of the possible solution space, which is a desirable property in TSP algorithms. However, this is all heavily reliant on the parameters set

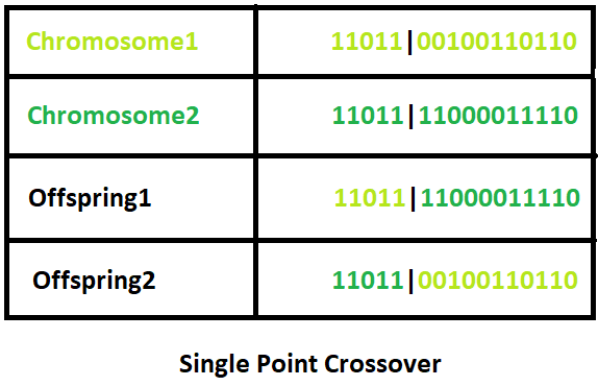

Figure 3.2: An image depicting the crossover between 2 parents (their chromosomes shown as *Chromosome 1* & *Chromosome 2*), and the offspring (*Offspring 1* & *Offspring 2*) produced by them. Image: https://www.geeksforgeeks.org/

by the user of the GA, which can heavily affect not just the quality of the solution generated but also the overall execution time. Configuring parameters correctly can result in poor exploration of the solution space (hence premature convergence to local optima), or excessive exploration of the solution space (hence unnecessarily high runtimes). GA parameters therefore often require ample trial and error in order to tune perfectly. The stochastic nature of GA could also cut both ways. On the one hand, GAs stochasticity in features like initial population generation, crossover, mutation, etc. can introduce more genetic diversity into the population, which may increase the chances of convergence to global optima. However, it often results in different runs of the algorithm producing different results, some very different to others due to premature convergence.

It is challenging to pin a specific computational complexity on Genetic Algorithms given the numerous factors that influence them. For example, larger populations and number of generations, while offering wider exploration of the solution space and hence more chance of obtaining the optimal solution, heavily increase computation time and resources utilised. Additionally, the computation of the fitness of each individual is an inherently variable process. This variability stems not only from the nature of the problem and data being processed, but also the efficiency of the implemented function to compute the fitness. This introduces further inconsistency in computational complexity.

While the regular heuristics that we have explored follow a deterministic and sequential process, GAs follow a much more stochastic and parallelised path. While this allows for a broader

exploration of the search space, it significantly increases computational expenses. When stacked up against MILP, GAs are much more scalable and flexible with larger problem instances, albeit being unable to guarantee optimality in the way that MILP can. GA shares a lot in common with LNS in terms of its supreme scalability and comprehensive exploration of the solution space. However, they differ in terms of the way they provide this. While LNS begins with a single viable solution and attempts to iteratively enhance this, GAs attempt to exercise a bit more innovation through the evolution of several generations of solutions. Nonetheless, LNS could be seen as the more consistent of the two. Given that it starts with one initial solution and repeatedly enhances that through the use of destruction and reparation, the trajectory with which it will execute is far more predictable than that of GA. A GA exhibits little consistency in its execution, but can produce consistent solutions if the parameters are configured in a similar way each run. All in all, GA is a robust algorithm that, if applied judiciously and configured correctly, have the ability to produce high quality solutions to complex problems.

3.3 Feasibility in Real-World Application

Given the comprehensive look at and comparisons between the six algorithms, we are now well-positioned to discuss the feasibility of the algorithms in the real-world.[11]. These comparisons have been categorised into three distinct tables, corresponding to problem instances of low, medium and high number of nodes. This decision is logical, given the observed variation in algorithm performance in comparison to problem size (a concept that we have previously established).

Algorithm Name	**Real App Feasibility**	**Usable alone?**	**Solution Quality**	**Runtime Efficiency**	**Hybrid Potential**	**Ease of Implementation**
NN	High	✓	Medium	High	✓	High
IHCV	High	✓	High	High	✓	Medium
MILP	High	✓	Optimal	Medium	×	Low
Two-Opt	Medium	×	N/A	High	✓	High
LNS	Medium	×	High	Medium	✓	Medium
GA	Medium	✓	High	Medium	✓	Low

Table 3.1: Algorithms' Feasibility for **Low** Number of Nodes

Algorithm Name	**Real App Feasibility**	**Usable alone?**	**Solution Quality**	**Runtime Efficiency**	**Hybrid Potential**	**Ease of Implementation**
NN	Medium	✓	Low	High	✓	High
IHCV	High	✓	Medium	Medium	✓	Medium
MILP	Medium	✓	Optimal	Low	×	Low
Two-Opt	High	×	N/A	High	✓	High
LNS	High	×	High	Medium	✓	Medium
GA	High	✓	High	Medium	✓	Low

Table 3.2: Algorithms' Feasibility for **Medium** Number of Nodes

Algorithm Name	**Real App Feasibility**	**Usable alone?**	**Solution Quality**	**Runtime Efficiency**	**Hybrid Potential**	**Ease of Implementation**
NN	Low	✓	Low	High	✓	High
IHCV	Medium	✓	Low	Medium	✓	Medium
MILP	Low	✓	Optimal	Very Low	×	Low
Two-Opt	High	×	N/A	Medium	✓	High
LNS	High	×	High	Low	✓	Medium
GA	High	✓	High	Low	✓	Low

Table 3.3: Algorithms' Feasibility for **High** Number of Nodes

[11]The solution quality of Two-Opt is marked as *N/A* because it highly depends on the quality of the initial solution fed to it

This comparative analysis is insightful and sets the foundation for the development of our novel algorithm[12]. In particular, the potential for algorithm hybridisation is strikingly evident. It seems optimal to combine algorithms that provide rapid solutions with algorithms that provide high quality solutions. For instance, a GA used in tandem with Two-Opt could potentially leverage GA's ability for vast solution exploration, while allowing Two-Opt to locally optimise individual routes at each generation. This would allow individuals to generally be of better fitness before the selection process, hence hopefully producing offspring of higher quality. However, applying Two-Opt to all individuals at every generation would not only be very computationally taxing, but may also decrease genetic diversity[13]. Another potential hybridisation opportunity is LNS and IHCV. IHCV's ability to generate a decent solution in decent time can be leveraged by passing it as an initial solution to LNS. This would provide a strong basis from which LNS's destroy and repair mechanisms could work. Combining more than two algorithms (e.g. NN to generate an initial solution, then GA and Two-Opt from that point) could also be in order. However, this could significantly affect computational power required, so caution must be exercised.

The insight does not stop at algorithm hybridisation. Given the novel algorithm will form the basis of a prospective vehicle routing application, the need for algorithm scalability is evident. The feasibility of an algorithm in the real-world is directly linked to this. For example, MILP is very practical for smaller instances of TSP due to its decent runtime and high quality of solutions. However, this very quickly changes as problem size due to its rapidly diminishing runtime efficiency. Our novel algorithm should be designed and developed with this in mind, ensuring it remains practical and adaptive to a range of problem sizes. When considering our development of the basis of the prospective application, there is also potential for developing multiple algorithms and using different ones for different situations. For example, a smaller problem instance inputted by the user could use the algorithm that slightly favours solution quality, while a larger problem instance would require the algorithm that slightly favours runtime.

[12]The suggestions for the novel algorithm in this section (4.3) are provisional. In section 7, we will implement the base algorithms, before assessing the empirical data to see which of their strengths we may wish to leverage in the novel algorithm.

[13]The population often benefits from having some better solutions and some worse solutions, because this increases genetic diversity. Increased genetic diversity gives more opportunity for unique outputs from crossover and mutation, hence more chance to escape local optima.

Chapter 4

Requirements & Specifications

This project aims to implement, test and benchmark the 6 previously explored algorithms. Insights gained from analysing the benchmarked results will then be leveraged to develop novel vehicle routing algorithm(s). This will form part of groundwork laid for a prospective vehicle routing application.

4.1 Requirements

4.1.1 General Requirements

- **G1** - the most appropriate programming language for optimisation algorithm development should be used.
- **G2** - the codebase should be modular. It should be maintainable, updatable and scalable.
- **G3** - all algorithms should support the same type of input data.
- **G4** - all algorithms must be able to integrate seamlessly with the other base algorithms into the prospective application.
- **G5** - all algorithms must return solutions in a form that can be processed and outputted as a graph.
- **G6** - all deliverables should be sufficient to begin the development of the prospective vehicle routing application.

4.1.2 Algorithm Development Requirements

- **A1** - each algorithm must aim for optimisation in solution quality & runtime.
- **A2** - each algorithm must be implemented in separate modules to each other.
- **A3** - where applicable, algorithms should have logging mechanisms.
- **A4** - where applicable, algorithms should have tuning parameters.
- **A5** - where applicable, algorithms should be developed in a way that allows them to be hybridised with other algorithms.

4.1.3 Novel Algorithm(s) Development Requirements

- **N1** - algorithm(s) must leverage insights gained from the six base algorithms.
- **N2** - algorithm(s) must demonstrate novelty.
- **N3** - algorithm(s) must provide more effective solutions than base algorithms in several dimensions.

4.1.4 Data Input & Processing Requirements

- **D1** - there must be an intuitive yet structured way for users to input data.
- **D2** - robust validation for all inputs must be present.
- **D3** - mechanisms should be in place to normalise input data.

4.1.5 Visualisation Requirements

- **V1** - dynamic visualisation of routes through graphs should be present.
- **V2** - graphs must be interactive, allowing users to zoom, pan, etc..
- **V3** - a library with extensive community support must be used to aid in displaying the graphs.
- **V4** - the library used to display the graphs must ensure compatibility with Web Apps
- **V5** - the library used to display the graphs must ensure compatibility with Swift[1] (iOs) and Android platforms.

[1] https://developer.apple.com/swift/

4.1.6 Testing & Benchmarking Requirements

- **T1** - perfom comprehensive unit testing of all modules.
- **T2** - ensure system can gracefully recover from errors.
- **T3** - all algorithms should be benchmarked for solution quality & runtime.
- **T4** - all algorithms should be benchmarked for factors like memory usage, CPU Time, etc.
- **T5** - all benchmarks should be run in a controlled environment, where fair external conditions for all algorithms are ensured.
- **T6** - benchmarked results comparing the algorithms should be visualised.
- **T7** - solution quality should be benchmarked against a standardised dataset. It should provide access to benchmarking instances and fully optimised solutions.

4.2 Specifications

The project may only be considered complete if, at a minimum, all specifications of *High* importance have been implemented.

Table 4.1: **General Specifications:**

Requirement Code	**Requirement**	**Specification**	**Importance**
G1	The most appropriate programming language for optimisation algorithm development should be used	Python will be the language used, due to its extensive community support and array of libraries capable of performing mathematical, graphical and optimisation operations	High
Continued on next page			

Table 4.1 – continued from previous page

Requirement Code	Requirement	Specification	Importance
G2	The codebase should be modular. It should be maintainable, updatable and scalable	All algorithms's implementations and corresponding unit tests will be housed within their own directory. The same applies for the code which handles user input, outputting graphs, etc.	High
G3	All algorithms should support the same type of input data	User will input data with a '.xlsx' file; all algorithms will have a way of reading data from the '.xlsx' files in a standardised way.	High
G4	All algorithms must be able to integrate seamlessly with the other base algorithms into the prospective application	All algorithms will return their solutions as a 2-tuple; the first element will be the route, the second element will be the length of the route. Algorithms which require an initial solution (like LNS) can read this 2-tuple and work from there.	High

Continued on next page

Table 4.1 – continued from previous page

Requirement Code	Requirement	Specification	Importance
G5	All algorithms must return solutions in a form that can be processed and outputted as a graph	All algorithms will return their solutions as a 2-tuple; the first element will be the route, the second element will be the length of the route. This will be passed to a function which can read this and output the corresponding graph.	High
G6	All deliverables should be sufficient to begin the development of the prospective vehicle routing application	The deliverables required to achieve this are the 6 base algorithms, the novel algorithm(s), a standard format for the user to pass in the '.xlsx' file, a way to handle and read the data from this file, and a way to output the solution as a graph.	Medium

Table 4.2: **Algorithm Development Specifications:**

Requirement Code	**Requirement**	**Specification**	**Importance**
A1	Each algorithm must aim for optimisation in solution quality and runtime	Each algorithm will make use of the Numpy library, a powerful library that leverages multi-dimensional arrays and vectorised operations to aid computational power and runtimes	High
A2	Each algorithm must be implemented in separate modules to each other	All algorithms's implementations and corresponding unit tests will be housed within their own directory.	High
A3	Where applicable, algorithms should have logging mechanisms	There should be 'print' statements at appropriate points. This will allow for easier debugging, spotting where optimisation could be applied, etc.	Low
A4	Where applicable, algorithms should have tuning parameters	Algorithms which can make use of tuning parameters should do so via global parameters. The algorithms which could do so are GA, LNS, Two-Opt and MILP.	Medium

Continued on next page

Table 4.2 – continued from previous page

Requirement Code	Requirement	Specification	Importance
A5	Where applicable, algorithms should be developed in a way that allows them to be hybridised with other algorithms	All algorithms will return their solutions as a 2-tuple; the first element will be the route, the second element will be the length of the route. Algorithms which require an initial solution (like LNS) can read this 2-tuple and work from there, hence allowing for hybridisation.	High

Table 4.3: **Novel Algorithm(s) Specifications:**

Requirement Code	Requirement	Specification	Importance
N1	Algorithm(s) must leverage knowledge gained from the six base algorithms	After the 6 base algorithms have been benchmarked and analysed, all positives from the algorithms should be analysed and considered in the development of the novel algorithm(s)	High
N2	Algorithm(s) must demonstrate novelty	The algorithm(s) should work in a way that the base algorithms do not	High

Continued on next page

Table 4.3 – continued from previous page

Requirement Code	Requirement	Specification	Importance
N3	Algorithm(s) must provide more effective solutions than base algorithms in several dimensions	The algorithm(s) should be benchmarked after implementation and demonstrate better performance in several dimensions to be considered complete	High

Table 4.4: **Data Input & Processing Specifications:**

Requirement Code	Requirement	Specification	Importance
D1	There must be an intuitive yet structured way for users to input data	The standard columns for the '.xlsx' file will be 'Node' (for inputting the node number), 'X' (for x-coordinate), 'Y' (for y-coordinate), 'Type' (either 'Start' or 'Waypoint') and 'Name' (for any descriptions of the node in natural language).	High
D2	Robust validation for all inputs must be present	When user inputs an invalid '.xlsx' file, the code should either auto-correct it or throw an error and fail gracefully based on the situation.	High

Continued on next page

Table 4.4 – continued from previous page

Requirement Code	Requirement	Specification	Importance
D3	Mechanisms must be in place to normalise input data	If user formats the '.xlsx' file unexpectedly but not in such a way that an error should be thrown, the code will auto-correct this to a normalised form, hence passing data to all algorithms in the same way.	Medium

Table 4.5: **Visualisation Specifications:**

Requirement Code	Requirement	Specification	Importance
V1	Dynamic visualisation of routes through graphs should be present	The Plotly library will be used for this purpose. A function called 'display_graph' will take in the route and the route distance and use them both to display the route and data about the route.	High
V2	Graphs must be interactive, allowing users to zoom, pan, etc.	The Plotly library is superior to other libraries like Matplotlib for the purpose of interactivity, with modernised panning, zooming, tooltips, visual customisation, etc.	High

Continued on next page

Table 4.5 – continued from previous page

Requirement Code	Requirement	Specification	Importance
V3	A library with extensive community support must be used to aid in displaying the graphs	The Plotly library has extensive community support.	High
V4	The library used to display the graphs must ensure compatibility with Web Apps	The Plotly library can be integrated directly into web browsers. Its visualisations can also be converted to JSON using 'plotly.io.to_json()', hence allowing it to be integrated with backends like Django.	High
V5	The library used to display the graphs must ensure compatibility with Swift and Android	The Plotly library can generate the visualisations as HTML, which can be converted to the relevant formats for both iOS and Android	Low

Table 4.6: **Testing & Benchmarking Specifications:**

Requirement Code	Requirement	Specification	Importance
T1	Perform comprehensive unit testing of all modules	Unit testing will be conducted using the PyTest framework. All modules will be covered, and an emphasis will be made on invalid inputs and edge cases	High
T2	Ensure system can gracefully recover from errors	Comprehensive exception handling and error logging will be key	Medium
T3	All algorithms should be benchmarked for solution quality and runtime	Benchmarks will all be run in one file. They will be run by simply running the algorithms individually. The results will be recorded, before Plotly is used to display the results graphically.	High
T4	All algorithms should be benchmarked for factors like memory usage, CPU Time, etc.	Benchmarks will all be run in one file. They will be run by simply running the algorithms individually. The results will be recorded, before Plotly is used to display the results graphically.	Medium

Continued on next page

Table 4.6 – continued from previous page

Requirement Code	Requirement	Specification	Importance
T5	All benchmarks should be run in a controlled environment, where fair external conditions for all algorithms are ensured	All benchmark runs will take place on the same Macbook Air 2020. Algorithms will be run once the computer is fresh from a restart. All runs will occur consecutively on the same day. All apps barring the 'Terminal' app will be open	High
T6	Benchmarked results comparing the algorithms should be visualised	Benchmarks will all be run in one file. They will be run by simply running the algorithms individually. The results will be recorded, before Plotly is used to display the results graphically.	High
T7	Solution quality should be benchmarked against a standardised dataset. It should provide access to benchmarking instances and fully optimised solutions	TSPLIB is the standardised dataset that will be used. They provide a comprehensive set of TSP benchmarking/test instances, and also provide what they claim to be the optimal solutions to each of these instances. This dataset is provided by Heidelberg University, one of the most prestigious in Germany, and therefore should be reliable. This will be very useful for benchmarking purposes.	High

4.3 Limitations

This section outlines the potential limitations and constraints that may influence the performance and credibility of the project, as per the aforementioned requirements and specifications.

1. The accuracy of benchmarked results may be affected by the degrees of optimisation in each implemented algorithm. In other words, the performance of an algorithm is not only dependent on the nature of the algorithm itself, but the skill exhibited by myself as the developer on the day of implementation.

2. Benchmarking results obtained by my Macbook Air 2020 may not be true universally. Results may vary on different machines.

3. Benchmarking on my personal computer introduces the possibility of external influences which are difficult to manage, such as silent background processes.

4. The algorithms are heavily dependent on third-party libraries, such as Numpy. If libraries are ever discontinued, substantial modifications to the codebase would need to be made.

5. The algorithms are only being benchmarked against one standardised regular TSP dataset. Their adaptability to real-world situations is unforeseen.

6. The system only works with *.xlsx* files. This may limit its usability in areas where data of other file formats are available.

Chapter 5

Design

In this section, we will aim to provide a modular architecture design for the overall system, as well as details of the algorithmic designs that support it. We are committed to central software engineering principles (such as modularity, maintainability, scalability, usability, etc.) and aim to demonstrate this throughout the section.

Our process follows an iterative development approach, where we will design, develop, test, refine and benchmark our base algorithms before considering the novel algorithm(s) in more detail (in chapter 6). This allows for a more systematic approach to developing our novel algorithm(s), while remaining aligned with the best software engineering practices.

5.1 System Architecture

The architecture of our system is designed such that each module can be developed and tested independently while retaining the ability to integrate seamlessly with other modules. This should allow for modularity, flexibility, and scalability. For example, if we alter an existing algorithm in the future, or even integrate a new one, this should be possible with relative ease without disrupting the other modules of the system.

The system is comprised of 4 main components. These are:

1. **Base Algorithms** - Each of the 6 base algorithms will be housed within their own directory. These directories will contain a Python script implementing the algorithm, as well as a test file. For example, the *Nearest Neighbour* directory would contain *nearest_neighbour.py* and *test_nearest_neighbour.py*.

2. **Novel Algorithm(s)** - The novel algorithm(s) will also be stored within its/their own directory. This will work the same as the base algorithms.

3. **TSP Utilities** - A utility module that contains elements that are regularly used and reused by all other modules. This includes TSP test instances, common functions for reuse, etc. Also stored within its own directory.

4. **Benchmarking** - Responsible for benchmarking all algorithms that're developed. Runs the algorithms, records the data obtained, and stores it as graphs to be outputted. Also stored within its own directory.

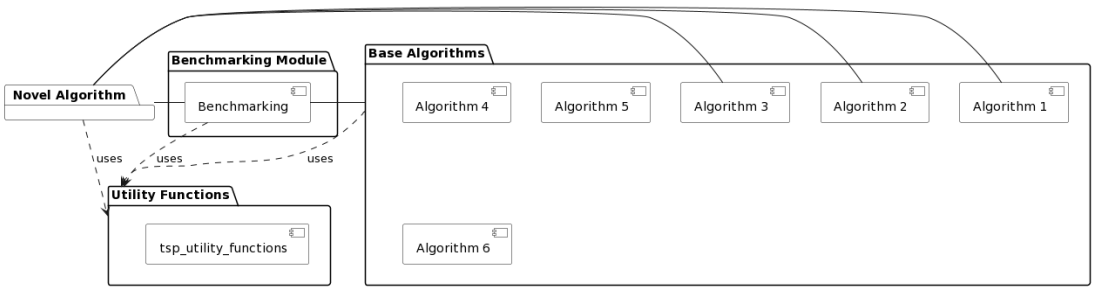

Figure 5.1: Component Diagram Illustrating Components' Interactions

The above component diagram depicts a high-level yet comprehensive view of the interactions between the different components of the system. The core of the system is the *Utility Functions* directory, which provides essential utilities and functions to support the operations of all modules in the system. This shows the system's reliance on reusable code. The base algorithms leverage the utility functions for each of their individual purposes. The novel algorithm(s) not only make use of the utility functions but also make use of insights gained/directly make use of some or all of the base algorithms. Finally the benchmarking module interfaces with all modules, evaluating the performance of all algorithms. All in all, the diagram encapsulates the modularity of the system, with each component having unique functionalities while engaging in interdependencies when required. This ensures loose coupling, which in turn allows for system maintainability and scalability.

5.2 Design Details

5.2.1 Languages & Libraries

The choice of programming language in a project with this magnitude of nuances is critical. Python emerges as the standout language for several reasons. The language's rich library support, prominently including *Numpy*[1] and *Scipy*[2] for example, is one of the biggest. The specific libraries available are often ideal for fields like data science and optimisation. This is in stark contrast to languages like Java, whose libraries are often geared towards the development of large-scale enterprise applications[3].

In addition, Python's syntax is considered one of the simplest, clear and concise available. This facilitates rapid prototyping, a key component of our iterative development process, which is often an invaluable part of algorithm development and refinement. Again, this is a sharp contradiction to Java and C++ for instance, which require a far more thorough, verbose approach that may hamper development time while producing rapid prototypes.

Python's mathematical superiority also must not be overlooked. Python's mathematical capability is solidified by libraries like *Numpy* for array and vectorised operations and *Pandas*[4] for data manipulation. The abundance of community support surrounding mathematics and mathematical libraries is also evident.

With the prospective integration of the system into a vehicle routing application in mind, the superiority of Python's *Django*[5] backend framework for this purpose becomes evident. Django boasts an array of web development tools straight out of the box, such as user authentication and databases. API endpoints can even be created from *Django* if the developers wish to work with a more modern frontend framework that supports single-page applications, state, etc. like *React*[6]. This is a possible advantage over languages like Javascript which, while having supreme web development capabilities, may lack the facilities to develop optimisation algorithms in the same way that Python can.

In summary, Python's extensive selection of libraries, active community support, mathematical focus and direct compatibility with a modern backend framework make it the prime

[1] https://numpy.org/
[2] https://scipy.org/
[3] https://radixweb.com/blog/python-vs-java
[4] https://pandas.pydata.org/
[5] https://www.djangoproject.com/
[6] https://react.dev/

option for our project.

5.2.2 Code Modularity

The design of our system exhibits several displays of modularity; none more so than the *TSP Utilities* module, which centralises a range of common utilities used by other modules.

The module begins with the file *tsp_utility_functions.py*, which houses several functionalities that are useful across the entire system. These include:

- **Reading User Input from *.xlsx* file** - Read information about the user's inputted TSP problem instance. Also provides meticulous error-handling capabilities in the event that user supplies an invalid *.xlsx* file. The file will leverage *Pandas* for this purpose.
- **Compute Distance Matrix** - Compute the euclidean distances between each node in the graph. The file will leverage both *Numpy* and *Scipy* for this purpose.
- **Compute Route Distance** - Compute the total distance of a route.
- **Display Route as Graph** - Take a route and display it as a graph. The file will leverage *Plotly*[7] for this purpose.

This file is essential, because it abstracts complex, frequent operations, hence allowing myself as the developer to purely focus on the core logic of each individual algorithm that I'm implementing. Further functionalities that could be added to the file as development progress and issues arise will be documented in section 6.

The TSP Utilities directory also serves as a place to hold benchmarking instances (both custom and TSPLIB), as well as a suite of invalid test inputs to aid rigorous testing practices. This underscores our commitment to thorough benchmarking and testing.

5.2.3 Data Input

The choice of *.xlsx* (Microsoft Excel Spreadsheet) over other popular file formats like *.csv* (CSV) is strategic. While *.csv* files are much simpler, quicker and more intuitive, which could be considered a positive, they lack numerous properties that *.xlsx* files don't. To begin with, most people seem to be more comfortable working with *.xlsx* files. This likely arises due to Excel's more tabular layout in comparison to CSV's reliance on comma-separated values. This

[7] https://plotly.com/

familiarity will reduce the likelihood of user frustration and invalid data formatting during the data preparation process.

Moreover, *.xlsx* files boast some advanced features that CSV's lack. The most notable of these is probably the ability to store multiple sheets in a single file. While it won't be relevant now, any future iterations of the system that demand expansions to required input data may benefit from the extra modularisation provided by this feature.

The *.xlsx* file comprises of the following columns, each of which must be unique:

- **Node** - The unique numerical identifier of each node
- **X** - The x-coordinate (latitude) of each node
- **Y** - The y-coordinate (longitude) of each node
- **Type** - Distinguishes between two types of nodes: *Start* and *Waypoint* nodes. There must be exactly one *Start* node.
- **Name** *(Optional)* - Provides descriptions in natural language of the node

To ensure user clarity, the prospective application could provide a template of this *.xlsx* file for users to download and edit before uploading.

Of course, there must also be robust error handling methods for dealing with invalid file inputs. As it stands, the following validations and responses are planned:

Check for:	**Throw Error?**	**Response in event of Invalidity**
Duplicate Column Names	✓	Throw error, provide appropriate error message
Columns hold Valid Datatypes	✓	Throw error, provide appropriate error message
Empty Rows	×	Delete empty rows
Nodes w/ Duplicate Coordinates	×	Keep the first node, drop the second one
Misspelled/Incorrect 'Type' Values	-	If slightly misspelled, auto-correct it. If heavily misspelled, throw error and provide appropriate error message
More than 1 Start Node	✓	Throw error, provide appropriate error message
No Start Nodes	✓	Throw error, provide appropriate error message
Less than 3 Nodes	✓	Throw error, provide appropriate error message

Node	X	Y	Type
1	1	2	Start
2	2	5	Waypoint
3	Two	One	Waypoint
4	6	2	Waypoint
Five	10	5	Waypoint
6	4	3	6
7	5	7	Waypoint

Node	X	Y	Type	Type
1	1	2	Start	Start
2	2	5	Waypoint	Waypoint
3	2	1	Waypoint	Waypoint
4	6	2	Waypoint	Waypoint
5	10	5	Waypoint	Waypoint
6	4	3	Waypoint	Waypoint
7	5	7	Waypoint	Waypoint

Node	X	Y	Type	Name
1	1	2	Start	
2	2	5	Waypoint	
3	2	1	Waypoint	
4	6	2	Waypoint	Corner Shop
5	10	5	Waypoint	
6	4	3	Waypoint	
7	5	7	Waypoint	

Figure 5.2: The first TSP instance (top left) depicts an invalid input. It is invalid because incorrect datatypes have been put into some of the cells. This would throw an error. The second TSP instance (top right) depicts another invalid input. It's invalid because it has two *Type* columns. This would be auto-corrected by the system by simply removing the second *Type* column. The final TSP instance (bottom center) is an example of a valid TSP instance. Images: Self

Pandas has been selected as the means through which we will read from the spreadsheets. Pandas is a library that greatly simplifies data reading and manipulation, which will help us detect invalid inputs, correct invalid inputs, extract node data, etc. Pandas is popular amongst developers, with over 1.6 billion installs as of 2024[8].

5.2.4 Data Visualisation

The end goal of this project is to lay the foundation for a prospective vehicle routing application. Because the application will be user-facing, there is a need to display the route visually. A graph is the most suitable way of doing so. Developing our own functionality to display graphs manually is theoretically possible, but we have opted out of that for three reasons:

- It is very sophisticated and potentially not accomplishable within the project's timespan.
- There are several libraries that are capable of performing this for us.
- We are simply forming the basis of an application. If future developers wish to alter the way in which graphs are displayed, the modularity of our code should easily facilitate this.

Instead, we have opted to make use of a library. There are several Python libraries that are capable of this. The decision of which library to use largely hinges on the levels of interactivity, customisation and ease of integration, all of which would go a long way to providing the optimal

[8]https://www.datacamp.com/blog/top-python-libraries-for-data-science

user experience. ***Plotly***, a low-code visualisation library renowned for its interactivity, versatile app integration capabilities and dynamic tooltip system, stands out in this respect. It is highly customisable and boasts the ability to pan, zoom, scroll and hover over nodes/edges to reveal tooltips. These are features which are assumed in routing applications that make use of graphs, and will therefore prove paramount in our system. *Plotly's* compatibility with various app technologies and responsive design is also key.

Other libraries were considered and rejected for various reasons. For instance, *Matplotlib*, while capable of presenting large datasets as static graphs, lacks the levels of interactivity desired in practical applications. Even worse, it seems to be native to Python and therefore not suitable for app integration. One area where *Matplotlib* undoubtedly usurps *Plotly* is in its ability to display vast volumes of data visually. *Plotly* sometimes experiences performance issues in this area, but this shouldn't be an issue for our use case. More specifically, *Plotly* reportedly experiences performance drops at around 15,000 nodes[9], which is an unrealistically high number in the context of vehicle routing. For context, the average number of nodes for an Amazon courier's journey would be around 200[10]. All in all, *Plotly's* abilities align more closely with what we desire in this project.

The actual visual design of the graphs does not matter at this stage, given that the colour palette, node styling, mood, etc. is highly dependent on the app it is a part of. Therefore, we will not explore that aspect of the graphs' design.

5.3 Algorithms Design

This section (5.3) will contain key design choices and considerations made before implementing the base algorithms. Not all algorithms will be covered here, because, due to our iterative development approach, a lot of them will have a design that will not become apparent until the time of development.

5.3.1 Distance Matrices

Distance Matrices will play a pivotal role in the algorithms implemented. In our codebase, distance matrices will be 2D Numpy arrays that store the distance between each node. It is such that the value at position $[i, j]$ in the matrix will correspond to the distance between node

[9] https://dash.plotly.com/performance
[10] https://www.aboutamazon.com/news/transportation/photos-day-in-the-life-amazon-delivery-driver

i and node j.

This approach offers a clever and nuanced way of maximising efficiency in solving the TSP. When a distance matrix is formulated, it has pre-calculated and stored the distances between all nodes in the problem, an operation which only needs to happen once. Instead of recalculating distances each time they're required, computing and storing it only once significantly reduces computational burden. All algorithms will make use of this, hence making its effect even more noticeable when it comes to hybridising them.

Node	X	Y	Type
1	1	2	Start
2	2	5	Waypoint
3	2	1	Waypoint
4	6	2	Waypoint
5	10	5	Waypoint
6	4	3	Waypoint
7	5	7	Waypoint

```
[[0.         3.16227766 1.41421356 5.         9.48683298 3.16227766
  6.40312424]
 [3.16227766 0.         4.         5.         8.         2.82842712
  3.60555128]
 [1.41421356 4.         0.         4.12310563 8.94427191 2.82842712
  6.70820393]
 [5.         5.         4.12310563 0.         5.         2.23606798
  5.09901951]
 [9.48683298 8.         8.94427191 5.         0.         6.32455532
  5.38516481]
 [3.16227766 2.82842712 2.82842712 2.23606798 6.32455532 0.
  4.12310563]
 [6.40312424 3.60555128 6.70820393 5.09901951 5.38516481 4.12310563
  0.        ]]
```

Figure 5.3: The distance matrix (right) for the TSP instance (left). For example, the 6th element of the 7th row (4.12310563) in the distance matrix represents the euclidean distance between node 7 and node 6. Because we're dealing with Symmetric TSP, the same value is held in the 7th position of the 6th row too. Images: Self

5.3.2 Insertion Heuristic w/ Convex Hull

Recall that IHCV begins with an initial Convex Hull. It is theoretically possible to implement our own code to compute the Convex Hull of a set of nodes. However, this implementation would not only take time and lines of code, but could also lack in efficiency. Therefore, it seems reasonable to make use of the *scipy.spatial* package's *ConvexHull* function, which takes in a Numpy array of coordinates and returns the corresponding *Convex Hull*. Their implementations have likely been optimised and tested across a much wider set of test cases than I could ever achieve in the timespan of this project as a lone developer.

5.3.3 Mixed-Integer Linear Programming

When it comes to MILP, there are 2 key design choices to make: the choice of **TSP formulation** and the choice of **framework/solver**.

TSP Formulation

A TSP Formulation is essentially a model that describes the TSP through an objective function and a set of constraints. They're written in a way that allows the programmer to easily

translate into code through the use of a MILP framework. The 3 most notable TSP formulations are *Miller-Tucker-Zemlin* (MTZ), *Dantzig-Fulkerson-Johnson* (DFJ) and *Gavish-Graves* (GG). Bazrafshan et al (2021) [2] pitted them against each other on five criteria, including number of constraints, number of variables, runtime and gap between the optimum and relaxed value. The results obtained showed that DFJ's relaxations led solutions closest to the optimal value. However, its practical applicability is doubtful due to its exponential growth in number of constraints as problem size increases. On the flip side, MTZ results in less efficient searches due to its looser relaxations. However, its ease of implementation could be deemed to compensate for that. GG seems to find a medium between the two.

One thing that should be noted about the aforementioned study is that it is in the context of Asymmetrical TSP (ATSP), which differs from the Symmetrical TSP (STSP) that we're working with. The structural differences between ATSP and STPS could influence the performance of each of the aforementioned formulations, thus potentially (although unlikely) rendering the conclusions reached irrelevant for our project.

I am opting for the MTZ formulation. Despite producing less accurate LP relaxations than both DFJ and GG, its simplicity and accessibility should allow for more focus on the application of the MILP approach without getting too caught up in the complexities of the model formulation.

$$x_{ij} = \begin{cases} 1 & \text{if arc } (i,j) \text{ is in the tour,} \\ 0 & \text{otherwise.} \end{cases}$$

$$\begin{array}{ll} \min & \sum_{i,j} c_{ij} x_{ij} \\ \text{s.t.} & \sum_i x_{ij} = 1 \quad \forall i, \\ & \sum_j x_{ji} = 1 \quad \forall i, \\ & 0 \le x_{ij} \le 1, \quad x_{ij} \text{ integer,} \end{array}$$

$$\begin{array}{l} u_1 = 1, \\ 2 \le u_i \le n \quad \forall i \ne 1, \\ u_i - u_j + 1 \le (n-1)(1 - x_{ij}) \quad \forall i \ne 1, \forall j \ne 1. \end{array}$$

Figure 5.4: *The MTZ formulation of TSP, linearly representing TSP mathematically. The decision variable x_{ij} = 1 if the edge from node i to node j is in the route, = 0 otherwise. The 'cost' (distance) between each node i and j is c_{ij}. It is ensured that each node has one outgoing edge ($\sum_i c_{ij} = 1$) and one incoming edge($\sum_j c_{ji} = 1$). The aim is to minimse travel cost (distance) by minimising the objective function $\sum_{i,j} c_{ij} x_{ij}$. Images: Pataki (2003) [12]*

Framework/Solver

Pyomo[11] was chosen as the framework to use for the MILP approach on the basis of its exceptional flexibility. *Pyomo* is capable of interfacing for numerous solvers, something which is not true of all frameworks. This list comprises both commercial and open-source solvers, including *BARON*, *CBC*, CPLEX and *Gurobi*[12]. Naturally, this is a great set of options to have, allowing us to select a solver based on optimising for runtime, optimising for solution quality, licencing needs, etc. Its high levels of abstraction were also an influential factor in its selection. The code written when modelling a problem using *Pyomo* is often readable by humans who have no background in modelling problems through code. This makes it easier to express formulations such as our *Miller-Tucker-Zemlin* formulation through code with clarity, hence making life easier for the developers of future iterations of the system.

Regarding the choice of solver, the primary contenders were *Gurobi* and *CPLEX*. Bernhard Meindl and Matthias Templ (2013) [10] indicates that commercial solvers generally outperform open-source solvers in both runtime and solution quality. In the benchmarks, *Gurobi* and *CPLEX* were the best performers, producing a solution in 89% of cases, far more than their open-source counterparts. *CPLEX* also ran 205x faster than the fastest open-source solver (*GLPK*). These results were in correlation with the results obtained by Hans D Mittelman[13] (5). It therefore seems safe to say that either one of *Gurobi* or *CPLEX* can be used in our case. We have opted for *Gurobi*.

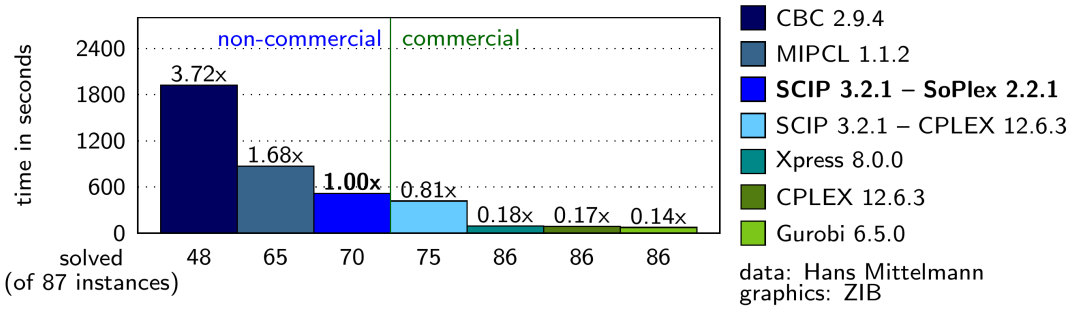

Figure 5.5: The results obtained by Hans Mittelman are congruent with that of Bernhard Meindl & Matthias Templ, hence indicating their credibility. Images: Hans D Mittelman

The only obvious drawback to commercial solvers is the need to obtain a commercial/academic licence to use them locally. However, even in the absence of a valid licence, *Pyomo's*

[11] https://www.pyomo.org/
[12] https://pyomo.readthedocs.io/en/stable/solving_pyomo_models.html
[13] Hans Mittelman is a prominent man in the field of operations research and data science, renowned for his comprehensive benchmarks of optimisation algorithms and software. He is/was affiliated with Arizona State University's School of Mathematical & Statistical Sciences.

compatibility with so many different solvers makes it relatively simple to switch a commercial solver out for an open-source one if needed.

Overall, the decision to use *Pyomo* and *Gurobi* is grounded in the pursuit of maximum computational efficiency, solution quality and flexibility in model construction.

5.4 Conceptual Integration of UI

This project aims to build the foundation of a prospective vehicle routing application. This means that no user interface will be developed, but the elements required to begin development of the application will be. This section (5.4) aims to detail what the workflow of the system may look like given a user interface.

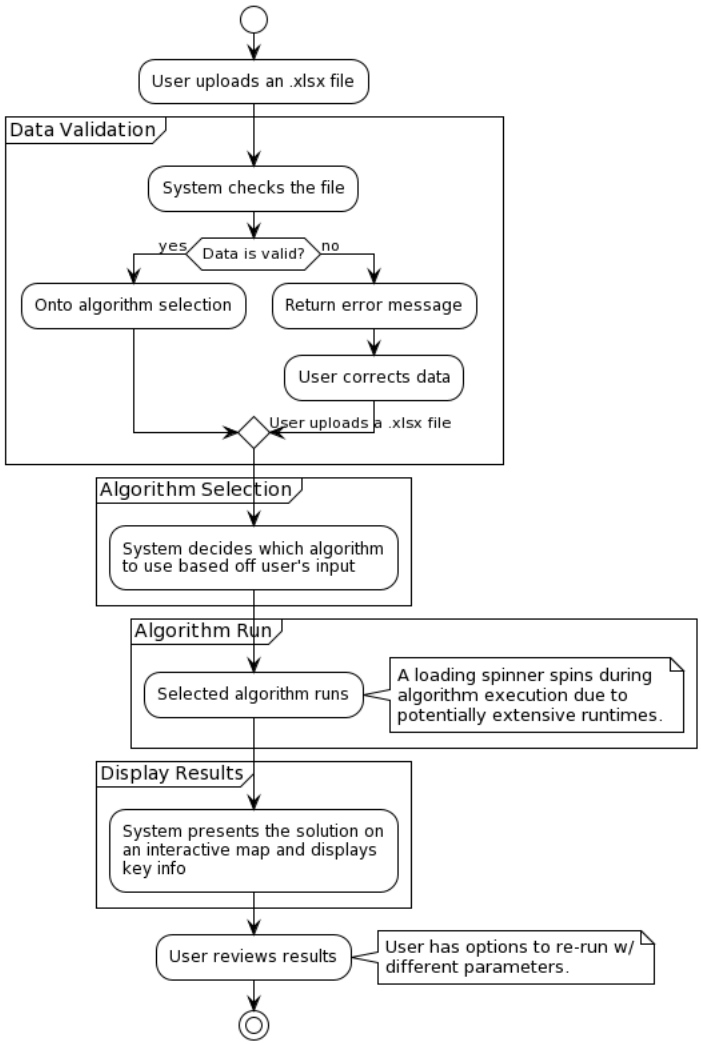

Figure 5.6: Activity Diagram details the workflow of the system given the presence of a user interface

5.5 Benchmarking

Overall, our benchmarks aim to help us gauge how well our algorithms perform over various dimensions like runtime, solution quality, CPU time, memory usage, etc. TSPLIB's benchmarking suite, which is credible (as mentioned in section 4.2), will be used given the comprehensive nature of it. We will benchmark against 6 of their TSP instances: *ulysses16*, *berlin52*, *pr76*, *pr107*, *pr136* and *tsp255*[14]. These instances were chosen due to the wide range in the number of nodes. This allows us to test the algorithms across a range of problem sizes, hence giving us valuable insights into the scalability of them individually. There would likely be benefit in testing instances beyond 225 nodes, but given what we learned about Amazon's daily delivery volume (approx. 200 customers per driver per day)[15] we will stick with a maximum of 225 for now. TSPLIB also claim to have solved each instance to optimality, and provide each of these solutions. We can therefore assess the solutions obtained by our implementations against TSPLIB's optimal solutions to rigorously assess the solution quality of the algorithms.

Here is some more key information relating to our benchmarks:

- **Runtime** - Measure the time taken by each algorithm to solve each instance of TSP. This includes the whole process, from reading the '.xlsx' file to displaying the graph at the end.
- **Solution Quality** - Assess the quality of solutions obtained by each algorithm by comparing the solution returned to the true optimal solution provided by TSPLIB. Percentage difference can be used for this.
- **Computational Resources** - Evaluate factors like CPU Time, memory usage, etc. when running each algorithm.
- **Scalability** - Analyse the difference in performance in each algorithm when dealing with problem instances of different sizes.
- **Integration** - These benchmarks will ensure that algorithms integrate well with other algorithms by ensuring to call the hybridised versions of the algorithms too.
- **Visualise Benchmarks** - The results of the benchmarks will be visualised as graphs.

[14]The trailing numbers represent the number of nodes in the TSP instance. E.g. *pr107* has 107 nodes.
[15]https://www.aboutamazon.com/news/transportation/photos-day-in-the-life-amazon-delivery-driver

5.6 Conclusion

In conclusion, this design chapter (chapter 5) describes a modular and robust way of going about the TSP/VRP. Thought was clearly put towards the choice of programming language, methods of data input, algorithmic details, visualisation techniques, benchmarking techniques, etc. While we've not only outlined the planned core logic and functionality of the system, the fact that we've also considered alternative designs and future iterations broadens our perspective and opens the door to potential innovations.

Chapter 6

Algorithms Implementation, Testing, Benchmarking and Results

This section (chapter 6) aims to detail the actual implementation (also benchmarking and testing if applicable) of all modules in practice. Relevant source code can be found in appendix C.

6.1 Development Approach

The implementation phase of this project followed an iterative development approach, which allowed for the progressive refinement of algorithms' implementations, testing and benchmarking over the course of time. As can be seen in previous chapters of this paper, rigorous research and design had been carried out in order to set the stage for implementation. However, by adopting an iterative development approach, we've allowed ourselves the flexibility to incrementally plan, implement and test. This strategy gave room to readily adapt our algorithms to fit their particular requirements should need be.

The benefits of the approach were most noticeable in the benchmarking phase (detailed in section 6.4) where algorithms were ran against a range of TSP instances of varying complexities. The iterative approach proved invaluable in this phase. For example, if through benchmarking we record that the algorithms seem to run as expected on smaller problem instances, but

experience unexpected difficulties on larger instances, an iterative approach endorses performing another iteration of designing, implementing and testing in an attempt to refine our previous attempt.

6.2 TSP Utility Functions

In the design section (chapter 5), we detailed how central the *TSP Utilities* module is to our system architecture. We also saw that the module contains a file named ***tsp_utility_functions.py*** (appendix C1), which facilitates modularity in our system. It does this by essentially abstracting away the most common (yet potentially intricate) tasks that several modules regularly undertake, hence allowing myself as the developer to simply focus on implementing the core logic of each individual algorithm. Each 'utility function' in this module is designed to represent one single, cohesive task.

6.2.1 Data Input & Validation

1. ***convert_tsp_lib_instance_to_spreadsheet*** - This function makes use of the *Pandas* library to heavily manipulate and convert *.tsp* files (the file format of TSP instances provided by TSPLIB) to *Pandas DataFrames*. This is a critical step to take before further converting these files into *.xlsx* files which, as discussed in chapter 6, are preferred in our system. The primary motivation behind converting .tsp files to *.xlsx* files before passing them into our algorithms, instead of passing them into our algorithms directly, is for modularity. By converting them into *.xlsx* files, we're capable of using the same algorithms for computing solutions that are applied for conventional *.xlsx* files, a strategy that further enhances the maintainability and scalability of our code.

2. ***import_node_data_tsp_lib*** - This function utilises Pandas to convert the output of *convert_tsp_lib_instance_to_spreadsheet* into a .xlsx file with the appropriate columns that our system expects.

3. ***import_node_data*** - This function is a prime example of our commitment to robustness. It uses Pandas to read the actual file and obtain it in a form Python can understand, before going through extreme lengths to ensure data validity and integrity. All validations which were present in our design made it through to the final implementation:

   (a) Check for Duplicate Columns. Throw an error if encountered.

(b) Check Columns hold Valid Datatypes. Throw an error if encountered.

(c) Check for Empty Rows. Delete any empty rows encountered.

(d) Check for Nodes w/ Duplicate Coordinates. If encountered, keep the first node and drop the second one.

(e) Check for Misspelled/Incorrect *Type* Values. If any slightly misspelled types are encountered, auto-correct them. If any heavily misspelled types are encountered, throw an error.

(f) Check for more or less than 1 *Start* node. If encountered, throw an error.

(g) Check for less than 3 nodes overall. If encountered, throw an error.

The following validations were not present in our initial design, but were incorporated by virtue of our iterative development approach:

(a) Check for singular cells that're empty (except in the *Name* column, where that's OK). If encountered, throw an error.

(b) Check that all required column names (*Node*, *X*, *Y*, *Columns*) are present. If encountered, throw an error.

(c) Check for nodes not in order of node ID[1]. Auto-correct this if encountered.

(d) Check for start node doesn't have node ID 1. If encountered, auto-correct this.

(e) Check for nodes don't have sequential IDs from 1 –¿ *n*. If encountered, auto-correct this.

The function makes use of Numpy for numerical validations, as well as the *Levenshtein* library for fuzzy string matching. This use of *Levenshtein* is a novel and innovative approach to auto-correcting minor misspellings. Levenshtein distance is essentially a metric that tells us how 'different' two strings are from each other by computing how many single character edits are required to make them equal[2]. This concept was leveraged to validate and potentially auto-correct data inputted into the 'Type' column, a potentially crucial feature if we wish to maximise user experience in any prospective application this system is a part of. The function essentially uses the *Levenshtein* library to compute the Levenshtein distance between each entry and the two expected entries: *Start* and *Waypoint*. If the user's entry has a Levenshtein distance ¡= 2 from either of these words,

[1] having nodes in order of node ID, as well as sequentially from 1 –¿ *n*, makes life a lot easier for our algorithms.

[2] https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

then the word is auto-corrected. Else, an error is thrown. This represents a clever step in our system's preprocessing, enhacing user experience by reducing the need for manual data re-entry.

Another element realised as a result of our iterative development approach was the importance of sequencing validations logically. More specifically, it was revealed that validating nicher properties before broader ones often resulted in the data becoming unusable downstream. Details on the ideas behind the ordering of validations are as follows:

(a) The initial validations were broader ones that check the general structural integrity of input data. This included things like checking for duplicate columns, checking that all necessary columns are present, etc. Checking these first was mandatory because, if not rectified early, more complex problems could be caused in the latter validation phases.

(b) Following this, checks for general content validity are in place. This includes removing entirely empty rows, throwing errors in the event of singular empty cells and removing any nodes with duplicate coordinates.

(c) A more detailed look at content validity then occurs. Things like checking the datatypes of all values in each column, auto-correcting values in the *Type* column if required, etc..

(d) Lastly, niche logical validations occur. This includes things like ensuring only 1 *Start* node exists, invalidating inputs with less than 3 nodes overall, etc.

This structured approach not only ensures that we don't encounter unexpected errors in data validation, but also guarantees more efficiency, efficacy and robustness in our data preprocessing. Every validation is paired with a corresponding reaction (whether that is throwing an appropriate error message or auto-correcting the user's mistake), which is crucial given that we're forming the foundation of a real-world application where user experience is top priority.

All validations are met with appropriate actions in the event of the user inputting unexpected input, with an appropriate error message being returned or auto-corrections happening. These validations are crucial given that we're forming the basis of a real-world application, where user experience is top priority.

6.2.2 Computational Functionalities

1. ***compute_distance_matrix*** - As discussed in section 5.3.1, distance matrices are central to the operations of all algorithms in our system. The *compute_distance_matrix* function makes use of both *Numpy* (for array conversion) and *scipy.spatial.distance.cdist* (for distance computation) to efficiently compute the Euclidean distance between every pair of nodes in the given problem.

   The function begins with a transformation of the nodes *Dataframe* into a list of coordinates. This list of coordinates is then transformed into a *Numpy* array, an intentional step which aims to leverage *Numpy*'s rapid computational speeds. This array is then passed to the *cdist* function. This function produces the distance matrix for us by computing the pairwise Euclidean distance between the *Numpy* array and itself. *cdist* was selected not only on the basis of its supreme computational efficiency, but also its compatibility with *Numpy* arrays.

2. ***compute_route_distance*** - This function demonstrates one instance of the precomputed distance matrices coming to use. This function displays a vectorised approach for computing the entire distance of a route, ensuring rapid computation.

3. ***display_route_as_graph*** - This function highlights our dedication to displaying insightful, interactive graphs for given routes through the use of the *Plotly* library. As can be seen empirically and through the implementation and empirically, the graphs are highly interactive and insightful:

   - User can pan, zoom, scroll
   - Route distance is on display
   - Start node is marked clearly with a special symbol and labelled in a legend
   - Node ID is displayed below each node
   - Nodes can be hovered over to reveal their coordinates, which node is the next node in the route and how long that distance is

   All the above functionalities are observable in figure 1. As mentioned in chapter 5, *Plotly* also offers high levels of visual customisation in their graphs. Therefore, graphs can be tweaked to suit the aesthetics of any prospective app they're a part of if desired.

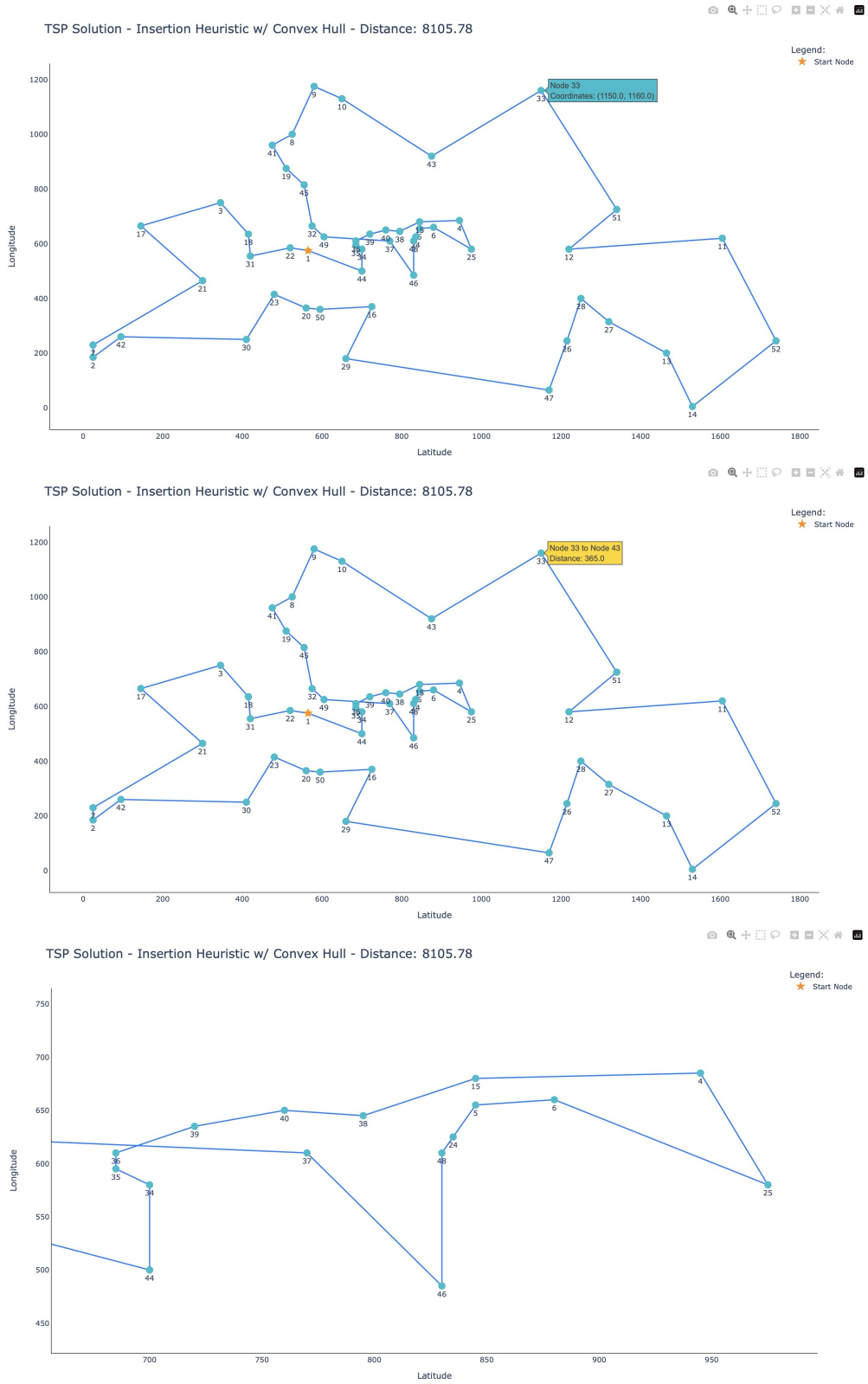

Figure 6.1: The above images demonstrate all the aforementioned functionalities of using *Plotly* to map graphs. The first image in particular shows the extra information detailed when hovering over a node. The second image shows further information displayed when hovering. The final image shows the ability to zoom, pan, etc. Further interactive options can be seen in the toolbars at the top right of all images. Images: Self

6.3 Base Algorithms

This section (6.3) aims to detail the implementation of each individual base algorithm. Section 6.4 will discuss the implications of the benchmarked results obtained.

One thing which should be noted is the presence of *run* functions in the code. Each algorithm primarily has two associated *run* functions: *run* (for running the algorithm with an *.xlsx* file as input) and *run_tsp_lib* (for running the algorithm with a *.tsp* file as input). This provides a layer of abstraction, allowing other modules to invoke algorithms and obtain their corresponding solutions with just one line of code.

6.3.1 Nearest Neighbour

All the algorithmic logic for NN was encapsulated within one function (as can be seen in appendix C2). Despite arguments that this may reduce modularity and extensibility, I believe that this approach is suitable for our use case due to the simplicity and clarity that comes with it.The NN algorithm is simple, even when considering the extreme lengths that we've gone to to optimise it. It only has one task: iteratively find the unvisited nearest neighbour until exhaustion. This is very cohesive, so encapsulating the entire logic of the algorithm in just one function should suffice.

The implementation displays a series of carefully considered decisions. Here are the most notable ones

- The computation of the distance matrix. This is done using *tsp_utility_functions*'s *compute_distance_matrix*, a clear display of modularity.
- The algorithm dynamically retrieves the start node by looking up the node with $Type == Start$ in the *DataFrame*.
- The algorithm makes use of the boolean *Numpy* array *visited* to keep track of all nodes that have been visited. *Numpy*'s memory efficient approach is evident here, as we have set all values in the array *False* by using *np.full*.
- Innovative distance masking has been utilised to find the nearest unvisited neighbour of each node, as opposed to methods that use loops or involve creating further data structures (which would be far more computationally taxing). It works by using *np.where* to conditionally replace distances to all visited nodes to infinity. This way, the algorithm

essentially disregards visited nodes when comparing which node is the closest node to visit, because finite numbers are always smaller than infinite ones. All in all, this is a smart use of *Numpy*'s capabilities.

- The act of iteratively appending the next nearest neighbour to the end of the route while simultaneously computing the total route distance aids code clarity.

6.3.2 Insertion Heuristic w/ Convex Hull

The implementation of IHCV's logic comprises of two functions:

1. ***compute_convex_hull*** - As we know, the initial stage of IHCV is computing the Convex Hull. The *compute_convex_hull* function transforms the list of node coordinates into a *Numpy* array, before passing this to *Scipy*'s *ConvexHull* function, which computes the Convex Hull from that. The decision to use *scipy.spatial.ConvexHull* was detailed in section 5.3.2.

2. ***cheapest_insertion*** - Once the Convex Hull has been computed, it is passed to the *cheapest_insertion* function, which is the core of the algorithm. Recall the primary goal of cheapest insertion is to insert the nodes not in the Convex Hull into subtour at their least costly positions. At each iteration of the loop, *cheapest_insertion* evaluates the cost of inserting each new node between each pair of nodes currently in the subtour, inserting it at the position that incurs the lowest cost increase. The importance of the pre-computed distance matrix becomes evident; we're constantly computing distance here, so having to constantly recompute distance would be a pain. The use of *Numpy* for numerous mathematical operations is also prevelant. For instance, *Numpy* is leveraged to compute total distance of the route from a *Numpy* array in a very minimal and intuitive way. Finally, the route is reordered at the end to ensure that it starts and ends at the requested *Start* node. This shows an understanding of VRP's real-world applications, where journeys start and end at a specific depot.

One thing to note is that IHCV is the only algorithm in the whole system that uses its own *compute_distance_matrix* function. This was done to avoid redundantly recomputing the coordinate array (which is done in both the *compute_convex_hull* function and *tsp_utility_functions*'s *compute_distance_matrix* function).

6.3.3 Mixed-Integer Linear Programming

While there isn't much to discuss with regards to the MILP implementation (given that the majority was essentially covered in section 5.3.3), one key point which our iterative development process shed light on was the need for the incorporation of the ***Big-M*** method. *Big-M* can be used in MILP to handle constraints involving logical conditions. M is a variable that essentially allows nonlinear constraints to be linearised.

$$u_i - u_j + 1 \leq (n-1)(1 - x_{ij}) \tag{6.1}$$
$$u_i - u_j + 1 - M_{ij} \leq 0 \tag{6.2}$$
$$M_{ij} \leq (n-1)(1 - x_{ij}) \tag{6.3}$$

Figure 6.2: The original MTZ subtour elimination constraint (6.1) had to be split into two new Big-M constraints (6.2) & (6.3). This was done to linearise the constraints, which is more compatible with *Gurobi*.

Before utilising *Big-M*, *Gurobi*'s Branch-and-bound mechanism seemed to get stuck locally very early on for more complex TSP instances. Since introducing Big-M however, some of this pressure seems to have been alleviated. This may be because *Gurobi* favours constraints that have been linearised[3].

6.3.4 Two Opt

Recall that Two Opt cannot work as a standalone algorithm, and therefore requires an initial solution to be passed to it. Therefore, we have no choice but to hybridise it with other algorithms. As a starting point, two candidates for this are NN and IHCV. The primary reason for this is their ability to produce reasonable solutions relatively quickly, hence providing a solid basis for the enhancement that Two Opt seeks to achieve. Of course, once benchmarking has taken place, analysis of those results will reveal potential opportunities for further hybridisations involving Two Opt.

The philosophy behind the functions in Two Opt is as follows:

1. ***two_opt_swap_move*** - This function implements the two opt swap move directly. Recall that the two opt swap move essential reverses a segment of the route in an attempt to generate a route shorter than the current shortest. The decision to use *Numpy* for this case, particularly to perform the inline reverse of all elements between i and j, demonstrates great attention to computational efficiency. A more naïve approach may involve,

[3] https://www.gurobi.com/documentation/current/refman/dealing_with_big_m_constra.html

for example, slicing the list at i and j, reversing that segment, and then concatenating these lists. However, this introduces unnecessary computational overhead that our implementation successfully avoids.

2. ***compute_nearest_neighbours*** - Yet again, our iterative development approach gave rise to invaluable insights. According to Nilsson (2003) [11], Steiglitz and Weiner observed that the efficiency of Two Opt could be increased by pruning the search space to only consider a list of nearest neighbours as the viable candidates for two opt swaps. Introducing this alone can reportedly bring computational complexity down from $O(n^2)$ to $O(mn)$. When using this approach, they mention that dynamically adjusting how many nearest neighbours to consider based on the size of the problem is key to ensuring computational efficiency. Our *compute_nearest_neighbours* function has a ***PROPORTION*** and a ***MIN_NEIGHBOURS*** argument to account for this, which both also act as global variables. The *PROPORTION* argument essentially allows us to dynamically adjust the number of nearest neighbours, while *MIN_NEIGHBOURS* sets a minimum threshold for this number. This ensures a minimum level of exploration, regardless of size of inputted dataset. Adjusting the *PROPORTION* argument allows us to decide on the trade-off between runtime efficiency and solution quality; a higher *PROPORTION* generally results in a slower runtime, and vice versa. Again, heavy use is made of *Numpy* in the actual obtainment of the nearest neighbours list.

3. ***two_opt*** - The *two_opt* function holds the core logic of the Two Opt algorithm, essentially bringing everything together. A *while* loop is at the heart of the *two_opt* function, which iterates until its unable to better the current best solution that it has obtained. The *no_improvement_found* variable tracks whether or not improvements have been found. The variable equalling *true* at the end of a *while* loop iteration indicates that no improvement was found, hence signalling the end of the search as we terminate the loop. The reliance on nearest neighbours (which is pre-invoked and passed to the *two_opt* function by the *run_two_opt_generic* function) and strategic integrations of the *two_opt_swap_moves* (both of which we have explained the benefits of above) are also central to the function's operations.

6.3.5 Large Neighbourhood Search

As with Two Opt, LNS is largely incapable of producing a solution on its own. Instead, it requires an initial solution to be passed to it, which it will then proceed to enhance. Therefore, we will again perform algorithm hybridisation earlier than intended. Given that Two Opt has now been implemented, we can experiment with various combinations of NN, IHCV and Two Opt producing initial solutions.

The 3 functions of the LNS approach are:

1. ***destroy*** - This function removes nodes from the current route. This process begins with converting the route list to a *Numpy* array, before using *np.random.choice* (chosen for its computational efficiency and ability to introduce unbiased stochasticity) to randomly select nodes for removal. The number of nodes to be removed is in accordance with the ***PROPORTION*** variable, which determines the percentage of nodes to be removed. The exact number of nodes to be removed is computed by *n_nodes_to_remove = np.maximum(1, int(len(route_array) * proportion))*, a dynamic yet robust way to ensure that at least 1 node is selected for removal. *Numpy*'s abilities and optimised performance are further leveraged in the masking process, where a boolean mask is used to isolate the segments of the route that are not to be removed. This is a far superior method to manually iterating over the array by virtue of significantly reducing computational overhead. All in all, this function efficiently manages the complexity associated with randomised node removal.

2. ***repair*** - While the *destroy* function was in charge of removing nodes, the *repair* function is responsible for reinserting them. In particular, it is responsible for inserting nodes in positions that minimise total distance increase. This does mean that calculations of the cost increase for all potential points of insertion must be undertaken, which is one of most computationally expensive parts of LNS. However, this overhead is slightly relieved by the use of the pre-computed distance matrix. The use of *Numpy* for these operations again highlights its capabilities in handling complex numerical calculations efficiently.

3. ***lns*** - This function executes the core logic of the LNS algorithm, orchestrating its operations by iteratively invoking the *destroy* and *repair* mechanisms. The function is heavily goverened by ***MAX_ITERATIONS*** and ***IMPROVEMENT_THRESHOLD***. The global variable *MAX_ITERATIONS* sets an upper bound on the number of iterations that

algorithm can undergo, which in turn ensures that the algorithm doesn't consume computational resources unnecessarily and/or run forever. *IMPROVEMENT_THRESHOLD* determines a lower bound for what is considered to be an acceptable improvement in solution quality from the best value of the previous iteration. If we go more than *(max_iterations * 0.1)* iterations without seeing an improvement[4] that meets the *IMPROVEMENT_THRESHOLD*, then the algorithm terminates early. This number could have been made more dynamic instead of setting it to a static 10%, and is therefore something that needs to be noted.

The extensive usage of *tsp_utility_functions* as a means of abstracting several common functions in this module should also be noted.

6.3.6 Genetic Algorithm

As implementation of the Genetic Algorithm (as a base algorithm) began, its potential for hybridisation as part of the novel algorithm became evident. However, which exact algorithms would best integrate with it was less clear. Therefore, the decision was made to postpone the development of the GA until the benchmarked results of all other algorithms were obtained and analysed, potentially making it a crucial part of our novel algorithm. It was preferable to not waste time implementing the algorithm solitarily when its potential for hybridisation was clear. This is in accordance with agile methodologies, which emphasise project flexibility, promote work quality over work volume and heavily advocate for prioritisation of tasks. All in all, this is a strategic decision that aligns with modern software development strategies.

6.4 Benchmarking Methodology

The various base algorithms have been benchmarked against each other in order to understand the advantages and disadvantages of each. This information will be considered in the development of the novel algorithm. Selecting metrics that accurately reflect the algorithms' real-world performance is therefore imperative. The metrics selected were runtime, solution quality and CPU time because of the holistic view they give of the performance of the algorithms.

Runtime denotes the speed of the algorithm, which is paramount in a world where time is so often limited. Fast algorithms are not just desired but often required in real-world situations.

[4]The number of iterations that we go through without experiencing sufficient improvement is tracked by the *n_iterations_without_improvement* variable.

Solution quality, gauging the proximity of solutions produced by an algorithm to the true optimal, is also crucial. The ultimate aim of a VRP algorithm, after all, is to provide a solution of high quality. ***CPU time*** (not to be confused with runtime) provides a better look at the computational efficiency of an algorithm. Despite initially considering memory usage as a benchmarking metric in the requirements/design, we have decided to exclude it. This is because, after initially attempting to benchmark it, the additional overhead introduced as a result of extra tracking mechanisms like *tracemalloc* slowed benchmarking down far too much. We therefore decided to drop it in favour of carrying out extensive, comprehensive benchmarking within the constrained timeline of the project. This again is a decision motivated by agile methodologies.

The *benchmarking_implementations.py* script houses the entire benchmarking suite. Its implementation can be found in appendix C. The suite is well commented and modular, allowing users to easily add, modify or run the code by simply uncommenting the part they desire. The suite contains a function called *compute_average_runtime*, which computes and returns the average runtime and average cpu time of an algorithm's run after n iterations. We passed $n = 10$ for our runs, a number which is enough to stablise the results obtained but also not so high that the benchmarking process becomes laborious. There are also functions at the end of the script to output the benchmarking graphs. The algorithms were run against 7 different TSP instances, one (*test_city_1*) being user-generated, and the other 6 being standardised instances from TSPLIB. The rationale behind selecting the specific instances used is detailed in section 6.5.

6.5 Base Benchmarking, Analysis & Implications

The benchmarks were run on the following base algorithms/base algorithm hybrids:

- Nearest Neighbour
- Insertion Heuristic w/ Convex Hull
- Mixed-Integer Linear Programming
- Nearest Neighbour → Two Opt
- Insertion Heuristic w/ Convex Hull → Two Opt
- Nearest Neighbour → Large Neighbourhood Search

- Insertion Heuristic w/ Convex Hull → Large Neighbourhood Search
- Nearest Neighbour → Two Opt → Large Neighbourhood Search
- Insertion Heuristic w/ Convex Hull → Two Opt → Large Neighbourhood Search

6.5.1 Results & Analysis

All raw and graphical results obtained from base algorithm benchmarking can be found in appendix A1. One thing to note is that the MILP implementation failed to reach a conclusion after a substantial amount of time[5] for *pr107*, *pr136* and *tsp225*. Therefore, for these instances, we've just assumed the solution obtained after 1 hour of execution.

Runtime

NN and IHCV produced consistently low runtimes, running the quickest of all algorithms over every TSP instance. For TSP instances up to 136 nodes, NN slightly edges out IHCV in terms of runtime, running faster than it by between 0.007-0.06 seconds. However, the difference was a more pronounced 0.5s for *tsp225*. This seems to indicate that NN is consistently the most reliable algorithm for producing a solution as quick as possible, which aligns with expectations, and its reliability only increases as the problem size scales.

Enhancing the solutions produced by NN and IHCV with Two Opt for smaller instances of TSP (107 nodes and below) barely introduces any runtime overhead. In the case of *test_city_1*, it actually performed better! However, for *pr136* and *tsp225*, a gulf begins to form between the runtimes of NN and IHCV and their Two-Opt-enhanced counterparts. Two Opt appears to produce solutions up to 3x as slow in these more complex instances, despite still keeping solutions around or below 1 second.

On the contrary, MILP and LNS show significantly higher runtimes (as expected), with this becoming more apparent as problem size scales. MILP in particular faces significant scalability issues, jumping from a 2.19s runtime for a problem size of 52 nodes to a whopping 238s for 76 nodes! While still facing scalability issues itself, LNS experienced a much slower rate of slowdown, albeit somewhat exponential. All of its variants remained at under 1s for instances less than or equal to 16 nodes, around or below 5s for 76 nodes and under and below 9s for 107

[5]Reasons for why this may be will be discussed in the Evaluation chapter

nodes. The jump in runtime from 136 nodes to 225 nodes is worth taking note of, as it jumped from 15-20s to 35-45s.

CPU Time

The results obtained by the CPU time metric indicate that runtime and computational efficiency aren't necessarily correlated, a finding which was relatively surprising. This is best demonstrated by looking at the performance of MILP. In the context if *ulysses16*, MILP had a runtime of more than double any other algorithm. However, its CPU time was more than 4x quicker than all LNS algorithms. This suggests that the implementation of LNS is computationally efficient, with most of its runtime actually being taken up by non-computational overheads rather than actual solving.

NN, IHCV and Two Opt exhibit relatively low CPU times up until 136 nodes, consistently staying below half the runtime. This remains true for NN and Two Opt at 225 nodes, while IHCV seems to require more computational resources to reach a conclusion. This suggests NN is slightly more scalable than IHCV in terms of computational efficiency too. Optimising NN and IHCV with Two Opt seemed to add very little, if any, additional computational overhead.

The LNS implementation seems to consistently have the poorest computational efficiency amongst all algorithms, with CPU time elapsing for longer than half the runtime on numerous occasions. The IHCV implementations (both with and without Two Opt) are the worst performers overall. This is especially evident in *pr107*, where the CPU time is 98.3% that of the overall runtime.

Solution Quality

The solution qualities obtained closely aligned with expectations for the most part. At *test_city_1*, NN was the only algorithm that failed to obtain an optimal solution, underscoring its bias to producing solutions rapidly instead of optimally. NN proceeded to produce solutions worse than every other algorithm for every other instance[6].

However, adding Two Opt into the equation completely changes things. A clear example of this is *pr76*, where *NN* alone produced a solution of 153461.92, only to be bettered by its

[6]Barring one of the instances where MILP failed to complete execution. Even in the other two instances where MILP failed to complete execution, NN still produced a worse result!

integration with Two Opt by 25.95% (113635.27). This highlights Two Opt's superiority when it comes to enhancing initial solutions.

Apart from its instances of non-completion, MILP produced solutions of absolute optimality as expected. LNS closely followed it, regularly being less than 1% away from the optimal solution, hence proving to be a potent option too.

All in all, solution quality produced by the algorithms seemed to be relatively consistent, hence raising little concern about its scalability. The only algorithm which produced results of inconsistent solution quality[7] was NN. This is likely due to the stochastic nature of NN, whose solution quality is highly influenced by which node was chosen as the starting point.

6.5.2 Conclusions & Implications

The extensive benchmarking and analysis that we have just conducted has revealed critical insights into their relative performances. This leads us into a discussion about their potential for integration with a GA for the novel algorithm. As we know through previous research (section 4.2.6), the GA is a robust technique that can search a vast space through a mechanisms akin to natural selection. The two elements central to GA is the initial population obtained and the progression through generations.

In the GA pseudocode, we said that the initial population is generated randomly. However, generating an initial population in a less stochastic way (so as to guide us towards global optima) seems to be a more methodical approach. This opens the door to algorithms that can quickly generate an initial solution of not necessarily high quality that can be iterated upon. The obvious candidates for this position are NN and IHCV for two reasons:

- Both exhibited incredible runtimes in our benchmarks
- Both proved the ability to provide great initial solutions because, after being enhanced by Two Opt and/or LNS, solutions of high quality were produced.

Because our benchmarks show no significant difference in solution quality when providing either NN or IHCV as an initial solution before enhancement, it makes sense to go with the quicker, less computationally taxing one. NN had consistently lower runtimes and CPU times, so it makes sense to choose it as the heuristic to aid in the generation of an initial population in our GA.

[7] Barring MILP's occurrences of incomplete executions.

Another opportunity for hybridisation is at each individual generation. Despite their broad search of the solution space, GAs remain prone to converging to local optima [4] at certain generations. In order to escape these convergences, GAs adopt the 'mutation' approach, where some individuals of the generation will randomly flip their genes to introduce more genetic diversity in the population. However, if this still doesn't do the trick, the Two Opt heuristic offers a viable method to enhance solutions, pushing them closer to the true optimal. Our benchmarks highlight Two Opt's ability to significantly improve solution quality. By applying it with GAs, we can enhance individual solutions, hence increasing the overall quality of the population. This refinement can, in turn, boost the probability of producing superior offspring through crossover while maintaining genetic diversity (since improvements are made on an individual basis).

Nevertheless, it must be noted that, while Two Opt incurs minimal extra wall-clock time and CPU time (as seen in our benchmarks), its impact on runtime and computational efficiency becomes more pronounced as problem size scales. Given that GAs typically involve managing hundreds to thousands of individuals per generation, applying Two Opt on such a large scale could repeat the challenges Two Opt experienced on larger problem sizes in our benchmarks, which could strain computational resources further.

It remains important to consider the other base algorithms for integration in our GA too, because each algorithm does bring its own strengths and weaknesses to the table. For example, while MILP offers the best possible solution quality, its issue with scalability makes it completely inviable for integration with our GA. It is a similar story for LNS, albeit on a smaller scale. LNS also exhibits poor computational efficiency, a problem not experienced by MILP but a problem that will hamper our GA's potential performance regardless.

In summary, our novel algorithm will be a ***Genetic Algorithm*** that uses ***Nearest Neighbour*** to help produce initial solutions and ***Two Opt*** to enhance the quality of certain individuals in each generation. Again, due to the iterative development approach that we are taken, the details of how this integration will happen will not be clear until the actual implementation. All implementation choices which are made will be documented in the next section (6.6) however.

6.6 Novel Algorithm

The following details the technical intricacies behind the implementation of the novel GA:

1. **Modularity** - The decision to highly modularise this module (to a degree that may appear more than necessary) was taken due to the highly complex nature of GAs. Isolating functionalities into distinct modules makes otherwise complex code easier to read, maintain, update, test, debug, etc. without the need for extensive rewriting of any of the core logic.

2. **Parameters** - The script makes use of several parameters in the form of global variables. The *POPULATION_SIZE*, *GENERATIONS*, *MUTATION_RATE* and *CROSSOVER_RATE* parameters can be tweaked to hone in on the perfect balance between exploration and exploitation of the solution space. The parameters *STAGNATION_TOLERANCE* and *TWO_OPT_APPLY_THRESHOLD* allow the GA to adapt to different situations based on its current status, making the algorithm dynamic and more resistant to premature local optima convergence. *NN_FRACTION* affects the generation of the initial population, while *ELITISM_THRESHOLD* dynamically determines how many 'elite' individuals are preserved between generations.

3. **Initial Population** - As we know, a diverse initial population setup is crucial in GAs. Our implementation follows a hybrid approach. *NN_FRACTION* of the initial population will be generated using NN, with the rest being generated randomly. This allows the GA to begin with a population containing individuals that are somewhat optimised (and likely to superior to the randomly generated ones), hence providing a solid base to build from.

4. **Crossover & Mutation** - Crossover and Mutation are the core of all GAs. Crossover[8] ensures attempts to ensure that offspring inherit genes from the strongest parents. In terms of mutation (a GA operator designed to introduce diversity into the population stochastically), my GA specifically adopts an *adaptive* mutation approach. This is done through the line *adaptive_rate = base_rate * (1 - generation / GENERATIONS)*, where mutation rate is dropped as the number of generations increases. This promotes more exploration in earlier generations, but less in later generations in the hope that we are converging to global optima.

[8]Specifically, Order Crossover (OX). Abdoun and Abouchabaka [1] conclude that OX typically produces better solutions than other GA crossover operators for TSP.

5. **Stagnation** - The concept of stagnation detection and its cooperation with Two Opt is pivotal in ensuring the GA avoids the trap of premature convergence. Stagnation is basically the state of little to no improvement in the population's best fitness value over several generations; this may indicate that the code is stuck locally. The *is_stagnant* function performs stagnation detection by examining whether the improvement over *stagnation_check_span* generations is more or less than *STAGNATION_TOLERANCE*. If it is less than *STAGNATION_TOLERANCE*, this indicates stagnation. Two opt is applied either when stagnation is detected, or after *TWO_OPT_APPLY_THRESHOLD* generations. This is checked by *should_apply_two_opt*. This way, optimisations are applied only when required, hence reducing the chances of redundant increases in computational overhead.

6. **Elitism** - Elitism is paramaterised by *ELITISM_THRESHOLDS*, a Python dictionary that maps different sizes of problem instance to the percentage of elite individuals that should be carried over to the next generation. For example, a TSP instance of between 50 and 99 nodes should carry over the top 25% fittest individuals to the next generation. All in all, this ensures that valuable, high quality genes are retained upon transitions between generations, increasing the chances of converging to global optima.

As in every other algorithm implemented thus far, *Numpy* had a huge part to play in boosting efficiency. For example, *Numpy*'s use of vectorised operations in *generate_route*, such as through the use of *np.arange* and *np.random.shuffle*, both helped to initialise the population randomly yet efficiently.

Evidently, the process of implementing the GA was very thorough. The iterative development process that we took had a large part to play in this, as every iteration of design, implement, test gave rise to new opportunities and insights. These insights included dynamic parameter adjustments, a more structured approach to elitism, etc.

6.7 Novel Benchmarking & Analysis

One thing that should be noted is that the performance of the GA is heavily dependent on the parameters used. Using different parameters can produce very different solutions with varying computational speeds and efficiency. Although there was extensive experimentation with different parameters in our benchmarks, it is very possible that the results obtained could be improved upon with further parameter tuning. The exact parameters used and corresponding results obtained for each TSP instance can be found in appendix A2. Given these results, we can deduce the following when stacking up the novel GA against the previously benchmarked algorithms:

- **Solution Quality** - The novel GA appears to be the most scalable in terms of solution quality. It matched the solution produced by MILP (which is an exact method) for each of the 4 smallest TSP instances, of course either matching or surpassing the results produced by all other algorithms in the process. When considering the 3 biggest TSP instances, however, GA's performance was unmatched. The solution produced by the algorithm for *tsp225* is 0.52% ***better*** than the solution obtained by TSPLIB, which is an incredible result and testament to the strength in accuracy that the algorithm demonstrates. All in all, the GA produced a result less than 0.1% over the true optimal for all TSP instances (barring *pr107*, with a still very respectable 0.31), which indicates very strong performance on the solution quality front that can only be challenged by MILP.

- **Runtime** - The benchmarks indicate that the GA requires plenty of time to execute. For the 3 smallest TSP instances, it was by far the slowest algorithm. This was especially true for *berlin52*, where the algorithm recorded a runtime of 11.6s, 9.5s more than the notoriously slow MILP. However, as problem sizes scaled from this point, the GA produced runtimes significantly quicker than that of MILP, while still lagging far behind the other algorithms. From 107 nodes onwards, runtime seems to increase linearly with problem size. Further benchmarking against instances with more nodes than 225 would reveal more on this front.

- **CPU Time** - In general, the algorithm requires much more time from the CPU in its execution than any other algorithm. For instance, the CPU time (2332s) elapsed in solving *tsp225* was approximately 80% that of the runtime, which was already a very high 2904s. This is an indication of poor computational efficiency.

In summary, it is evident that the novel GA is an algorithm that heavily favours solution quality

above all else. This is clear in all benchmarks, but is most evident against *tsp225*. The solution obtained was better than that obtained by TSPLIB, which is an unbelievable result given the reputability of TSPLIB and Heidelberg University[9]. Nonetheless, this was at the expense of significant runtimes and computational overheads. The algorithm took a lengthy 48 minutes to execute, and ate up significant CPU time in the process. Again, these metrics are variable and are dictated by the GA parameters set before the GA's run.

6.8 Testing

Rigorous testing is crucial in ensuring the correctness of our implementations. This is especially critical given the real-world implications of VRP discussed in chapter 1. The tests written were designed to cover a broad range of testing methodologies using Python's *unittest*. *Unittest* was the testing framework of choice due to its versatitlity in handling both simple unit test and complex test suites involving integration tests. The scope of testing encompassed not only the validation of the utility functions and the negative testing of user inputs, but also the stress testing of the various algorithms implemented against test inputs of various complexities. This is all with the end goal of a robust suite of TSP solutions in mind. All test scripts can be found in appendix C.

6.8.1 Testing TSP Utility Functions

Thorough validation testing was carried out in *test_tsp_utility_functions.py* to verify accurate processing of valid input data. This is evident in the testing of the *compute_distance_matrix* function for example, where the correctness of distance matrix computation was validated by predefining the expected result, and asserting that the computed route was almost equal to that value. In addition, the *convert_tsp_lib_instance_to_spreadsheet* was validated by verifying that the *Pandas DataFrame* produced by the conversion from TSPLIB files by *tsp_utility_functions* indeed maintains its integrity of data.

While verifying that valid inputs are treated as expected (as we did with validation testing) is important, it is equally important to verify the robustness of our system through negative testing. Negative testing exercises our system's ability to deal with invalid user inputs, behaviour and data. In the prospective system architecture including a UI that we laid out in section 6.4, it is made clear that the only way a user interacts with the system at this stage

[9]Heidelberg University is the university associated with TSPLIB

would be through the submission of an *Excel* spreadsheet. This therefore is the main focus point of our negative testing. There are 20 different TSP instances that are invalid on various grounds located inside *TSP_Utilities/Test_Inputs*. These encapsulate a broad range of invalid inputs, all of which were outlined in section 6.2.1. Each of these invalid inputs, as well as some valid ones, have been negatively tested within *test_tsp_utility_functions.py*. If the invalidity is expected to invoke an error, *pytest* was used to verify that error. This is an effective way to test our system's ability to fail gracefully. If an auto-correct was otherwise expected, *pd.testing* was used to check if the auto-corrected *DataFrame* produced meets expectations.

6.8.2 Testing Algorithms

The basis of testing each individual algorithm was unit testing (using *unittest*), which was used to verfiy the correctness of virtually every function in each algorithm. Integration tests followed, which ensured that each unit of each algorithm integrated seamlessly with their corresponding units. This was particularly important for algorithms like MILP, where numerous contraints and variables had to be setup and interpreted with 100% accuracy in order to avoid producing an unexpected result. Integration testing was also used to verify that all hybrid algorithms (such as GA, NN x Two Opt, IHCV x LNS, etc.) functioned as expected.

Stress testing was also conducted at the same level as integration testing. This was essentially to evaluate the correctness of algorithms when faced with inputs of varying sizes and complexity. Each algorithm was tested against 3 different sizes of datasets: 'small' (to ensure basic functionality and catch trivial bugs), 'medium' (for more extensive analysis), 'large' (assess integrity when factoring in large-scale scalability).

There was not an opportunity to carry out negative testing for the individual algorithms. This was because the only opportunity for invalid inputs in the system as it stands is in the inputted *.xlsx* files, yet it is the *test_tsp_utility_functions* module that is responsible for the reading and processing of this.

6.8.3 Code Coverage

Code coverage is a form of structural testing that aims to report the extent to which a test suite covers the codebase. A higher code coverage means that your code 'touches' a higher percentage of statements in the code. It is important to recognise that high code coverage is not an indicator of high quality tests, but is conversely always a by product of good testing.

The general consensus is that coverage of 80% and above is a robust target for most projects[10].

Here are the code coverage numbers of our test suite:

	Code Coverage
nearest_neighbour.py	81%
ihcv.py	88%
mtz.py	96%
two_opt.py	88%
lns.py	89%
genetic_algorithm.py	93%
tsp_utility_functions.py	80%*

Table 6.1: *Writing unit tests for the *display_route_as_graph* function is not straightforward due its integration with *Plotly*. This brings the code coverage numbers down significantly. However, the function passes empirical and visual testing given that the graphs produced by it have no visible defects.

The fact that all numbers meet or exceed the 80% mark acts as statistical evidence that our tests cover a high enough proportion of statements in the codebase. However, it must be reiterated that this is not a direct indication of high quality tests.

[10]https://testing.googleblog.com/2020/08/code-coverage-best-practices.html

Chapter 7

Evaluation, Conclusion and Future Work

7.1 Requirement-based Evaluation

This section aims to compare the deliverables of this project with the requirements laid out in section 5. Recall that all requirements of *High* priority had to be met in order to consider this project fully complete.

A table outlining whether or not each requirement was met can be found in appendix D. As the table outlines, all requirements of high importance were met. In fact, only two requirements were not met. The first one was *"Where applicable, algorithms should have logging mechanisms"* (A3). While some algorithms (such as GA) exhibited logging mechanisms eliciting the flow of the algorithm's execution, not all did. The other requirement which was not met was *"All algorithms should be benchmarked for factors like memory usage, CPU time, etc."* (T4). This requirement wasn't met due to the failure to benchmark memory usage; the reason for this is outlined in section 6.4). Despite this, given that the two requirements were of low and medium importance respectively, not meeting them does not have a detrimental impact on the perceived success of our project.

7.2 Algorithms Evaluation

7.2.1 Base Algorithms

In general, all implementations were efficient and served their expected purposes. This is evident by our benchmarked results analysis (section 6.5) correlating with the research set out earlier in the paper. Regardless, it is important to remain judicious and understand that no implementation is the best implementation out there. In particular, we experience non-completion in the execution of MILP at 107 nodes and above. While we were not able to diagnose exactly why that was, careful thought has gone into thinking why the issue may have arisen:

- It may have completed execution on a more powerful machine
- It may have completed execution with a different solver[1].
- It may have completed execution with the use of a different TSP formulation[2].
- Potentially the way in which Big-M is used was misunderstood, hence causing MILP to not complete execution
- It may have simply required more time to execute

Any permutation of varying degrees of any of the above may have resulted in the incomplete execution of MILP.

Another potential area for improvement in the base algorithms was in the implementation of IHCV. In the current implementation, a brute-force approach is used to compute the cheapest insertion for each node. This does ensure that the most cost-effective insertion is found, but it comes at the cost of significant computational overhead. Our implementation may have benefitted from the development of an advanced heuristic. One such advanced heuristic could be the use of a nearest neighbour approach of pruning the search space before evaluating possible positions to insert, just like how we did in our Two Opt implementation.

Otherwise, our algorithms' performances align with expectations. *Numpy* was used extensively throughout; the use of its vectorised approach and array manipulation proved invaluable. Benchmarked results obtained were not wildly out of the ordinary (barring MILP for more complex TSP instances) and aligned with expectations developed during research.

[1] Bernhard Meindel and Matthias Templ (2013) [10] indicate that CPLEX slightly edges out Gurobi in terms of solution quality. Maybe using CPLEX would have helped us avert the problem.

[2] e.g. GG or DFJ, as previously discussed in section 6.3.3

7.2.2 Novel Algorithm

Through the prior analysis of the benchmarked results obtained, it is evident that our novel GA excels in producing solutions of high quality for small, medium and large datasets. However, this came at the cost of extremely poor runtimes and computational efficiencies. While this is to be expected with GAs, there is always room for improvement. Here are some potential reasons as to why the process may have been so costly and how we could reduce their impact:

- **Frequent Stagnation Checks & Two Opt Application** - The frequent checks for stagnation and the mass application of two opt was one of the key drivers of the algorithms success on the solution quality front. However, it is one of the main drawbacks in terms of time and resources. Altering stagnation thresholds and reducing the rate of two opt applications could mitigate this, but potentially at the cost of solution quality.
- **Fitness Function Evaluation** - Evaluating the fitness of so many routes is computationally expensive. There is not really a way to avoid this, but potential optimisations to mitigate its impact could be considered. For example, it appears that redundant computations of computations occurred for individuals with exactly the same genes. We therefore maybe could have tried to cache the fitness values of certain individuals.
- **Parameters** - The GA parameters used of course have a huge impact on efficiency. The GA parameters we used in our benchmarks were quite aggressive; runtime and computational efficiency may have benefitted from these being toned down.

Another thing to consider is the potential for further experimentation with the GA. We planned, designed and implemented the GA with a static idea of integrating it with NN and Two Opt. However, we may have benefitted from trying different approaches, such as IHCV as part of the initial population generation.

Overall, it is clear that our novel GA is geared towards providing high quality solutions over providing rapid and efficient solutions, which is OK! There are multiple scenarios where this would be desired, which we will explore in the next section (7.3).

7.3 Potential for Real-World Application Integration

Recall that the end-goal of this project was to produce deliverables that would suffice as the basis of a prospective vehicle routing application. We have successfully produced a system

which can read user input, process that, run an algorithm which uses the input, and produce a graph. So, we have successfully in laying the groundwork for a potential real-world application, as we initially planned to do. All that needs to be done by prospective developers willing to integrate the produced system with an application is put mechanisms in place to actualise the system architecture detailed in section 5.4.

The question to ask now is: 'in what real-world scenarios would our novel algorithm be desired?' Clearly, it would be desired in situations where solution quality is highly prioritised over all else. Here are some relevant example scenarios:

- **Public Transport** - Routes are typically only mapped once every few years. Public convenience is of utmost importance, so spending extra time to get a great solution is a worthy trade-off.
- **Interplanetary Route Planning** - As mentioned before (section 2.4), Interplanetary Expeditions are undertaken infrequently. They have high stakes and potentially dire consequences. Therefore, sacrificing runtime in favour of higher solution quality is desired. Our algorithm would prove invaluable here.

However, there are again limitations to our approach and things that need to be considered. The glaring issue is our algorithm's abstraction of what the VRP really entails. Specifically, through the assumption of VRP as simply TSP, it completely disregards real-world factors such as traffic. Traffic on a shorter route can cause the route to be longer in time than a longer route with less traffic. Any future work on our novel GA should take this complex factor into account.

7.4 Conclusion

Embarking on this project has exposed me to numerous areas of data science and operations research that I had never been exposed to before, an experience which I have found to be invaluable. As we have mentioned, the fact that the requirements have been sufficiently met is indication of the project's success.

Of course, the produced algorithms were not 100% perfect, as we have acknowledged. The next stage is to refine the system, including the *.xlsx* file template and novel algorithm, to deal with more realistic VRP inputs. Once this is done, we can consider its wide-scale integration for realistic vehicle routing applications that desire premium solution quality.

Chapter 8

Legal, Social, Ethical and Professional Issues

We ensured compliance with the Code of Conduct and Code of Good Practice (issued by the BCS) throughout the duration of this project. Great care was taken in aligning with their guidelines.

Legally, GDPR regulations set out that data must hold integrity, must be protected and remain confidential. Because our project does not work with sensitive data directly by virtue of lack of a UI and linked user data, this is not a rule that we have infringed. However, it is something that must be kept in mind in future applications. Socially, the fact that our algorithms aim to minimise route distance can have an indirect influence on reducing carbon emissions, which can contributes to environment sustainability. Our deliverables also uphold ethical considerations. For example, in reducing route distances, our algorithms help to reduce road congestion and often time spent on the road, promoting a more convenient environment for society. This is especially true of our novel GA, which reduces route distance extremely optimally. In addition, all code used is entirely my own. Any external libraries used have been mentioned. Our system also holds up to BCS's professional standards. This is demonstrated, for instances, through rigorous testing, modularity of code, etc. This doesn't only show the professionalism demonstrated throughout the project, but by following these standards, it increases the levels of trust that people can have in this paper and any future work produced by myself.

References

[1] Otman Abdoun and Jaafar Abouchabaka. A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem. *arXiv preprint arXiv:1203.3097*, 2012.

[2] Ramin Bazrafshan, Sarfaraz Hashemkhani Zolfani, and S Mohammad J Mirzapour Al-e hashem. Comparison of the sub-tour elimination methods for the asymmetric traveling salesman problem applying the seca method. *Axioms*, 10(1):19, 2021.

[3] Giovanni Calabrò, Vincenza Torrisi, Giuseppe Inturri, and Matteo Ignaccolo. Improving inbound logistic planning for large-scale real-world routing problems: a novel ant-colony simulation-based optimization. *European Transport Research Review*, 12:1–11, 2020.

[4] Noe Casas. Genetic algorithms for multimodal optimization: a review. *arXiv preprint arXiv:1508.05342*, 2015.

[5] Geoff Clarke and John W Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations research*, 12(4):568–581, 1964.

[6] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[7] Yuan Gao and Jiaxing Zhu. Characteristics, impacts and trends of urban transportation. *Encyclopedia*, 2(2):1168–1182, 2022.

[8] Lingying Huang, Xiaomeng Chen, Wei Huo, Jiazheng Wang, Fan Zhang, Bo Bai, and Ling Shi. Branch and bound in mixed integer linear programming problems: A survey of techniques and trends. *arXiv preprint arXiv:2111.06257*, 2021.

[9] Vijendra Kumar and SM Yadav. A state-of-the-art review of heuristic and metaheuristic optimization techniques for the management of water resources. *Water supply*, 22(4):3702–3728, 2022.

[10] Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for the cell suppression problem. *Trans. Data Priv.*, 6(2):147–159, 2013.

[11] Christian Nilsson. Heuristics for the traveling salesman problem. *Linkoping University*, 38:00085–9, 2003.

[12] Gábor Pataki. Teaching integer programming formulations using the traveling salesman problem. *SIAM review*, 45(1):116–123, 2003.

[13] David Pisinger and Stefan Ropke. *Large Neighborhood Search*, pages 399–419. 09 2010.

[14] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the world congress on engineering*, volume 2, pages 1–6. International Association of Engineers Hong Kong, China, 2011.

[15] Muhammad Sharif, Syeda Zilley Zahra Naqvi, Mudassar Raza, and Waqas Haider. A new approach to compute convex hull. *Innovative Systems Design and Engineering*, 2(3):186–192, 2011.

Appendix A

Benchmarked Results

A.1 Base Algorithms

This section displays the computational results obtained after benchmarking our implementations of the base algorithms.

A.1.1 Raw Results

test_city_1

	NN	**IHCV**	**MILP**	**NN, TO**	**IHCV, TO**	**NN, LNS**	**IHCV, LNS**	**NN, TO, LNS**	**IHCV, TO, LNS**
My Result	29.96	23.63	23.63	23.63	23.63	23.63	23.63	23.63	23.63
True Optimal	23.63	23.63	23.63	23.63	23.63	23.63	23.63	23.63	23.63
Percentage Increase	26.79	OPT	OPT	OPT	OPT	OPT	OPT	OPT	OPT

Table A.1: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.2912s	0.3192s	0.4854s	0.3369s	0.3982s	0.5936s	0.6017s	0.6001s	0.6019s
CPU Time	0.0643s	0.0586s	0.0814s	0.0589s	0.0585s	0.2524s	0.2479s	0.2650s	0.2553s

Table A.2: Runtime/CPU Time

ulysses16

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
My Result	104.73	75.11	73.99	75.97	73.99	73.99	73.99	73.99	73.99
TSPLIB's Optimal	74	74	74	74	74	74	74	74	74
Percentage Increase	41.5	1.5	OPT	2.67	OPT	OPT	OPT	OPT	OPT

Table A.3: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.3927s	0.4028s	1.9439s	0.4849s	0.4879s	0.7752s	0.7552s	0.8133s	0.8137s
CPU Time	0.0685s	0.0698	0.0933s	0.0727s	0.0658s	0.4259s	0.4167s	0.4447s	0.4334s

Table A.4: Runtime/CPU Time

berlin52

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
My Result	8980.92	8105.78	7544.37	8056.83	8074.56	7911.33	7911.33	7887.23	7544.37
TSPLIB's Optimal	7542	7542	7542	7542	7542	7542	7542	7542	7542
Percentage Increase	19.08	7.48	0.03	6.83	7.06	4.90	4.90	4.58	0.03

Table A.5: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.4872s	0.5403s	2.1881s	0.5839s	0.5570s	2.4629s	2.0279s	2.0701s	2.3267s
CPU Time	0.1028s	0.1162s	0.3090s	0.1202s	0.1256s	2.3432s	1.7926s	1.8235s	1.7669s

Table A.6: Runtime/CPU Time

pr76

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IH TO LI***
My Result	153461.92	114808.11	108159.44	113635.27	113113.89	110666.14	109044.07	109067.89	10
TSPLIB's Optimal	108159	108159	108159	108159	108159	108159	108159	108159	10
Percentage Increase	41.89	6.15	OPT	5.06	4.58	2.31	0.82	0.84	0.8

Table A.7: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.5912s	0.6079s	238.9238s	0.6298s	0.6108s	4.3152s	5.9477s	4.3653s	5.3695s
CPU Time	0.1356s	0.1563s	0.6113s	0.1651s	0.1528s	3.6238s	4.7678	3.8795s	4.6750s

Table A.8: Runtime/CPU Time

pr107

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
My Result	46678.15	45730.01	54606.7 (1 hour)	44767.07	45533.19	44436.24	44481.17	44436.24	44301.68
TSPLIB's Optimal	44301	44301	44301	44301	44301	44301	44301	44301	44301
Percentage Increase	5.36	3.22	23.26	1.05	2.78	0.31	0.41	0.31	OPT

Table A.9: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.5175s	0.6871s	*1 hour*	0.8510s	0.7930s	6.8574s	8.8580s	7.0712s	8.8959s
CPU Time	0.1707s	0.2842s	*1 hour*	0.2458s	0.2809s	6.4553s	7.6825s	6.4950s	8.7437s

Table A.10: Runtime/CPU Time

pr136

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
My Result	120777.8	102695.6	97389.7 (1 hour)	105114.4	99777.2	100906.8	98357.6	100787.6	97319.9
TSPLIB's Optimal	96772	96772	96772	96772	96772	96772	96772	96772	96772
Percentage Increase	24.81	6.12	0.64	8.62	3.11	4.27	1.64	4.15	0.57

Table A.11: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.5670s	0.6169s	*1 hour*	1.1399s	0.8790s	20.4485s	15.4880s	19.9038s	18.6097s
CPU Time	0.2090s	0.3438s	*1 hour*	0.3806s	0.4541s	18.9200s	12.0378s	11.2766s	13.6971s

Table A.12: Runtime/CPU Time

tsp225

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
My Result	4829	4442.27	4248.42 (1 hour)	4123.82	4329.24	4166.27	3989.86	3946.93	4000.3
TSPLIB's Optimal	3916	3916	3916	3916	3916	3916	3916	3916	3916
Percentage Increase	23.31	14.44	8.49	5.31	10.55	6.39	1.89	0.79	2.15

Table A.13: Solution Quality

	NN	***IHCV***	***MILP***	***NN, TO***	***IHCV, TO***	***NN, LNS***	***IHCV, LNS***	***NN, TO, LNS***	***IHCV, TO, LNS***
Runtime	0.6898s	1.1035s	*1 hour*	1.5792s	1.6633s	46.4287s	36.3342s	39.0564s	48.6119s
CPU Time	0.3299s	0.8435s	*1 hour*	0.8629s	1.1295s	36.5277s	34.3786s	29.1612s	37.3684s

Table A.14: Runtime/Solution Quality

A.1.2 Scalability

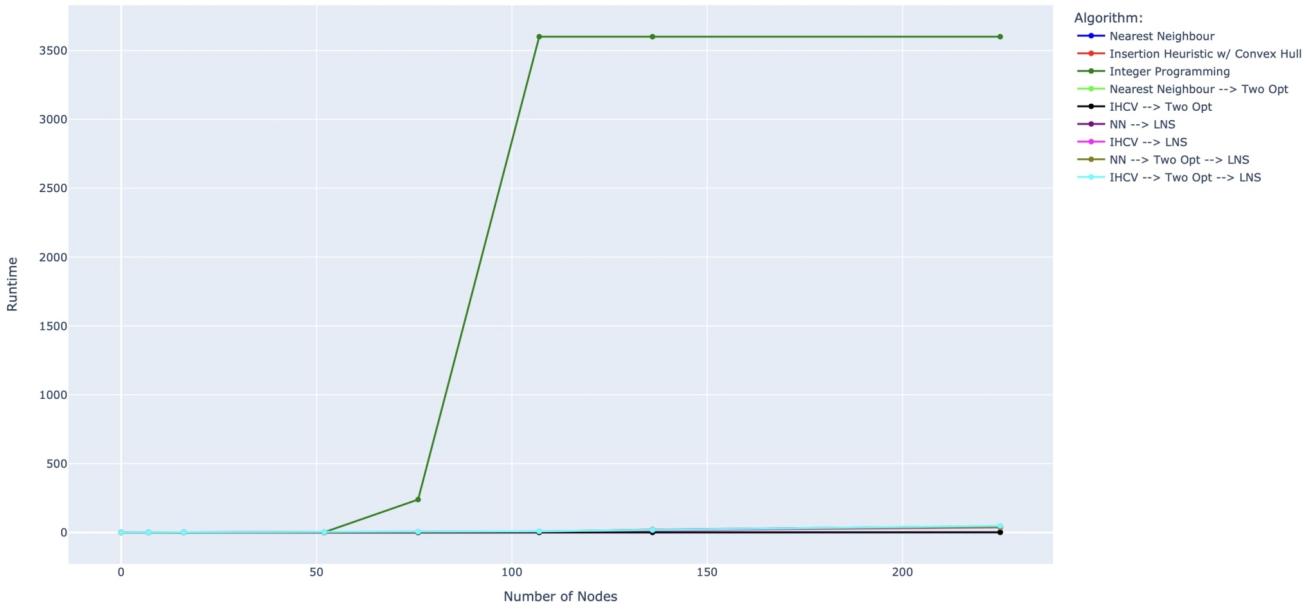

Figure A.1: Runtime, all Algorithms.

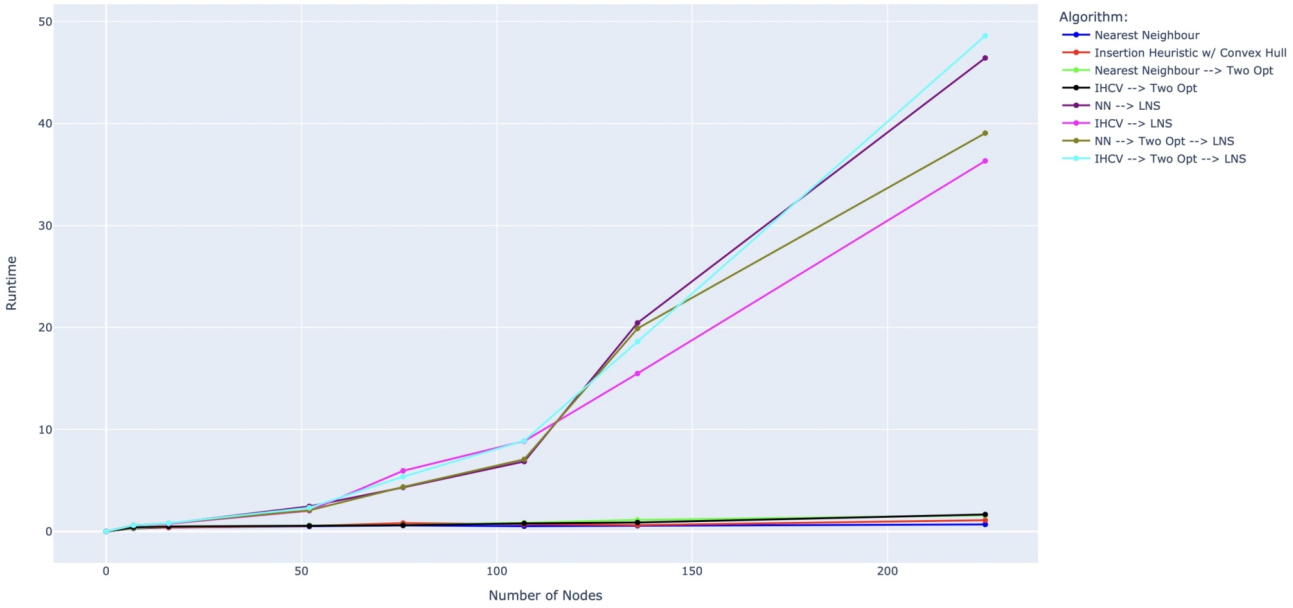

Figure A.2: Runtime, no Integer Programming.

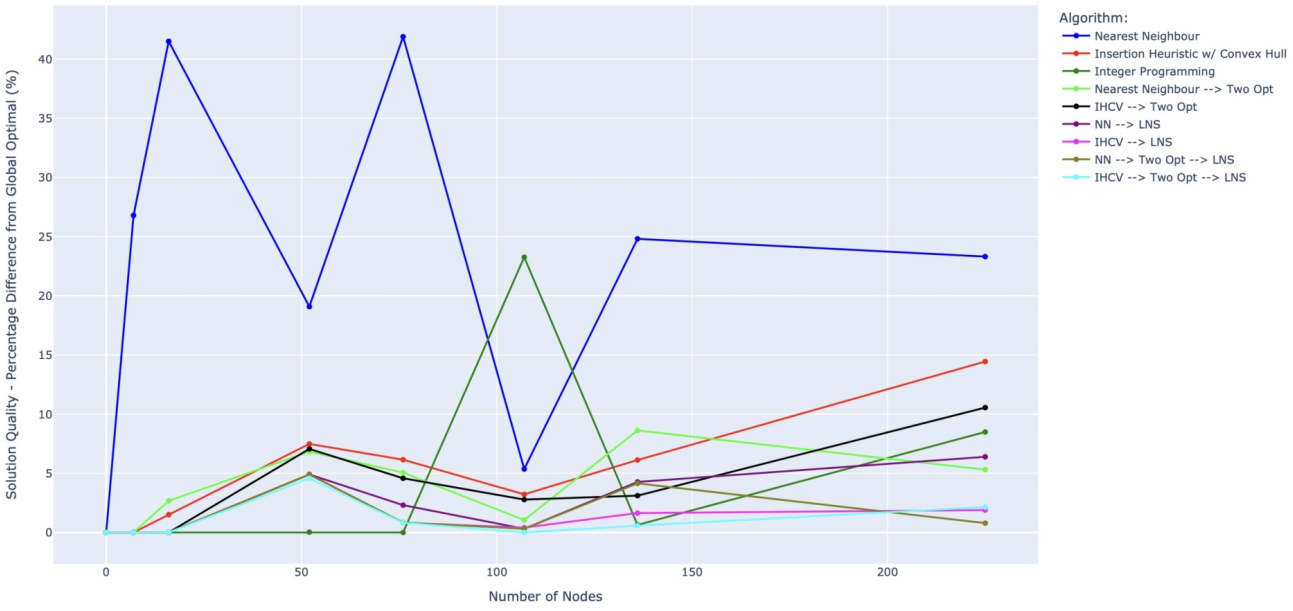

Figure A.3: Solution Quality, all Algorithms.

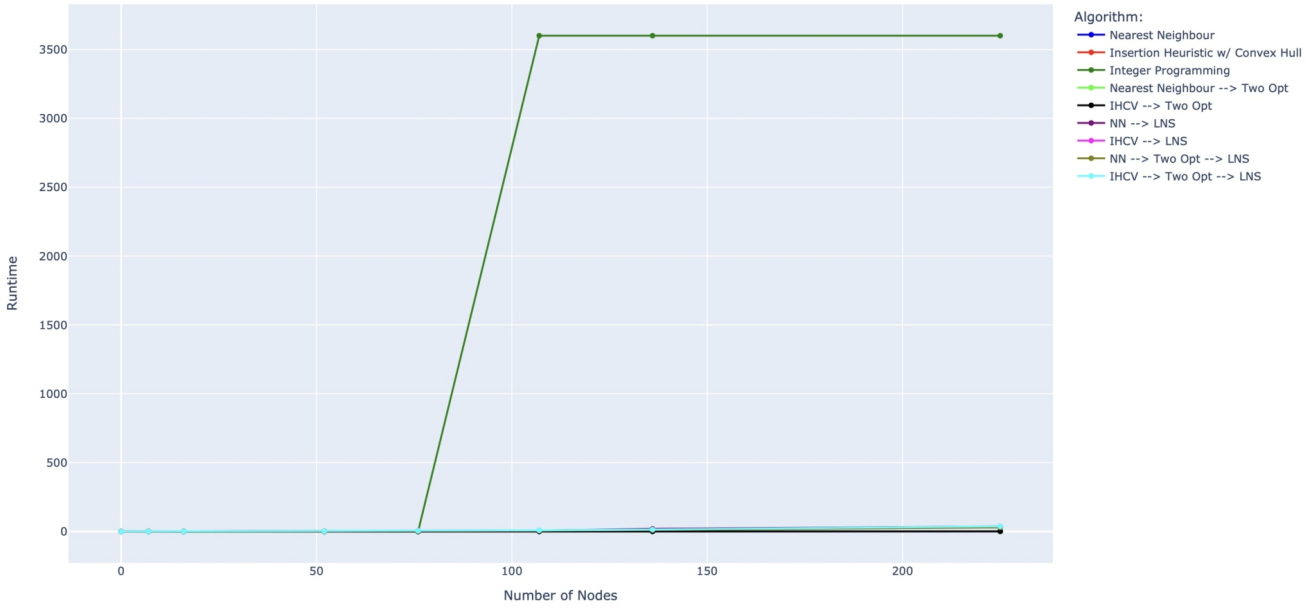

Figure A.4: CPU Time, all Algorithms.

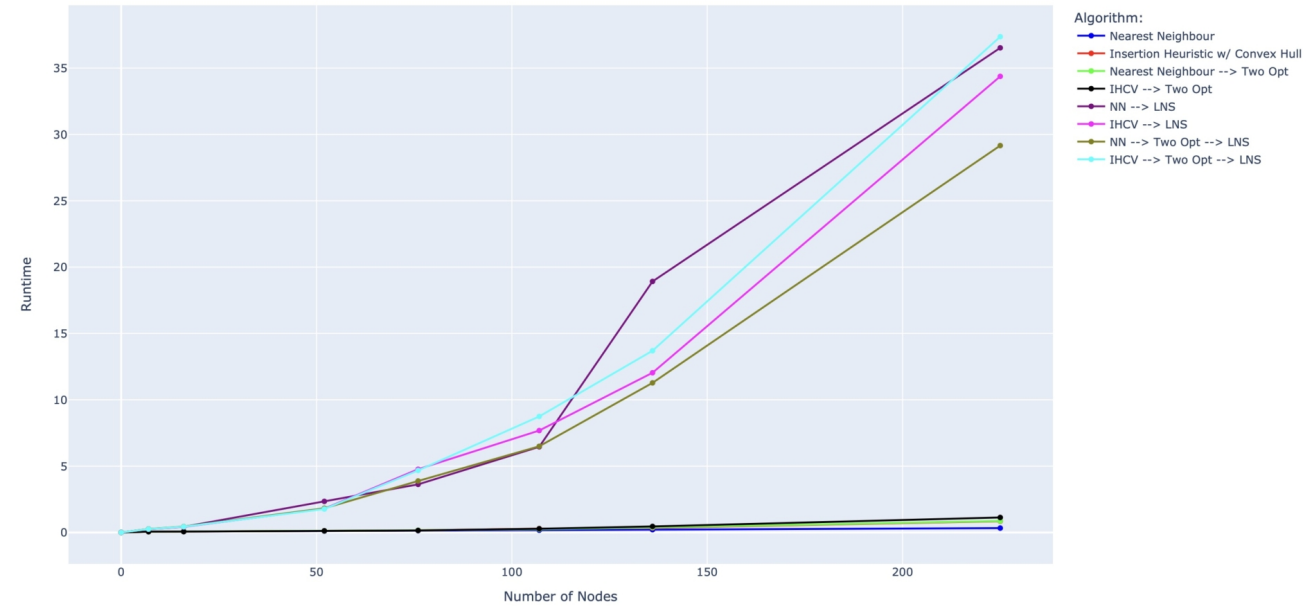

Figure A.5: CPU Time, no Integer Programming.

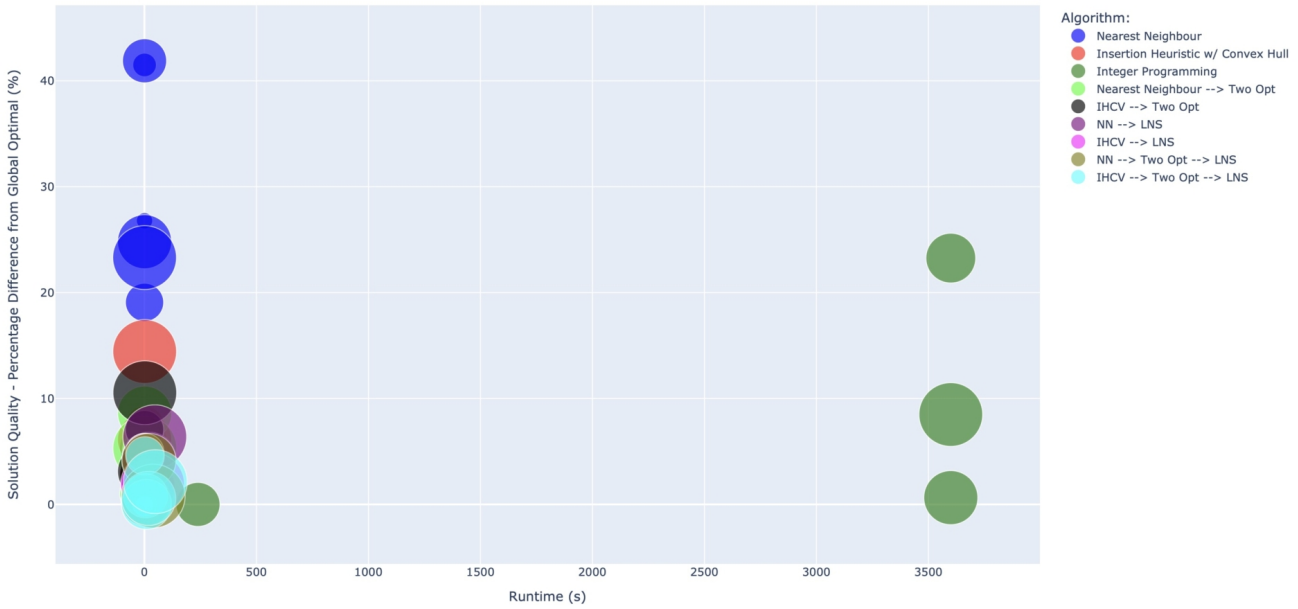

Figure A.6: Holistic Performance Comparison, Runtime vs Solution Quality, all Algorithms.

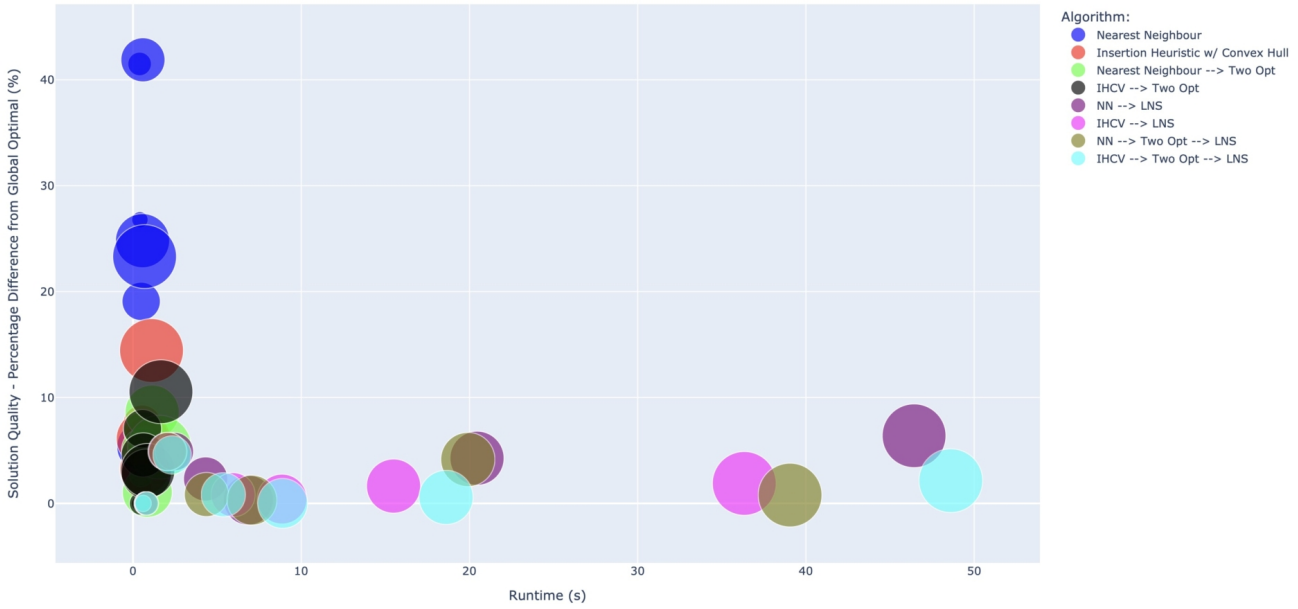

Figure A.7: Holistic Performance Comparison, Runtime vs Solution Quality, no Integer Programming.

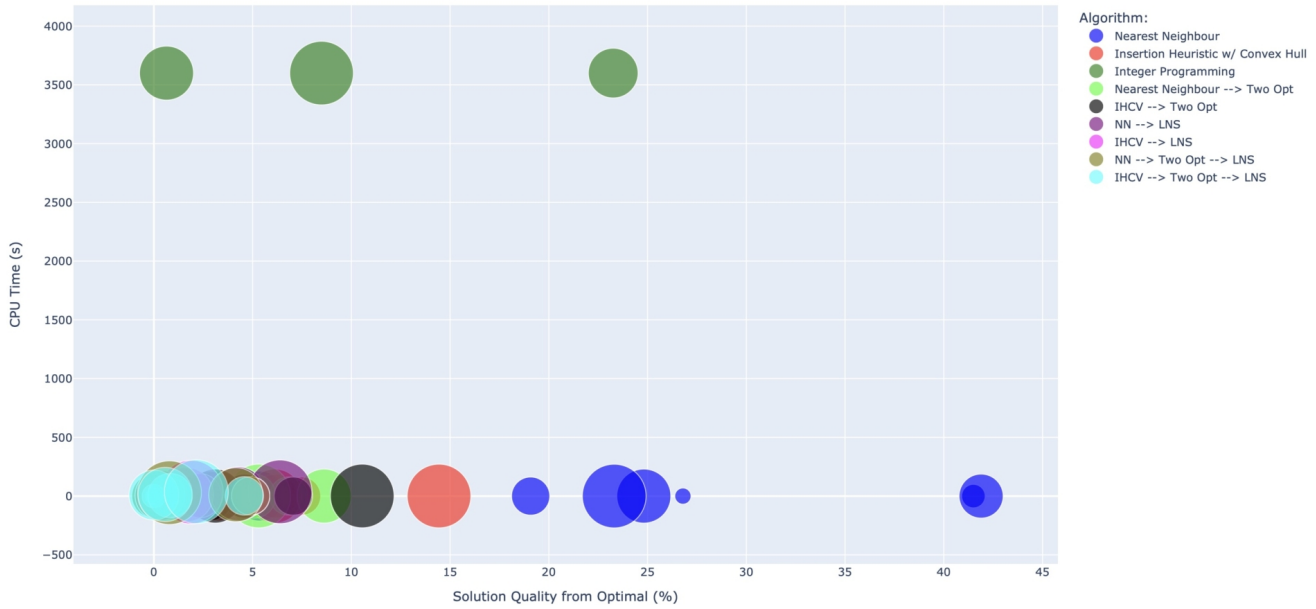

Figure A.8: Holistic Performance Comparison, Solution Quality vs CPU Time, all Algorithms.

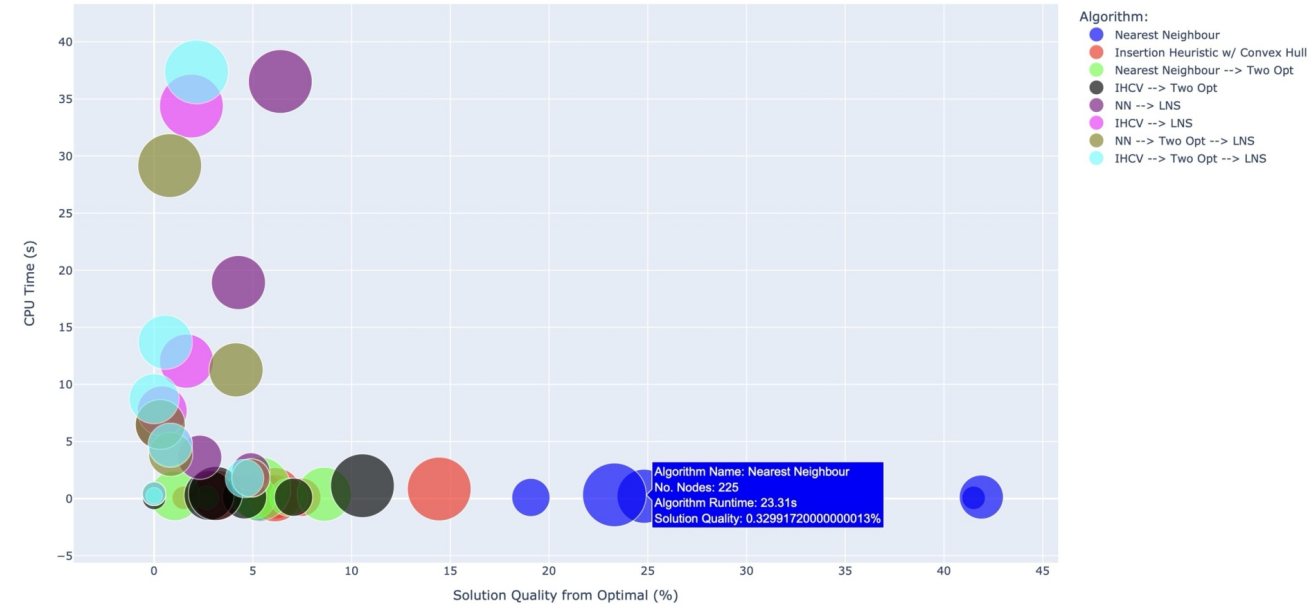

Figure A.9: Holistic Performance Comparison, Solution Quality vs CPU Time, no Integer Programming.

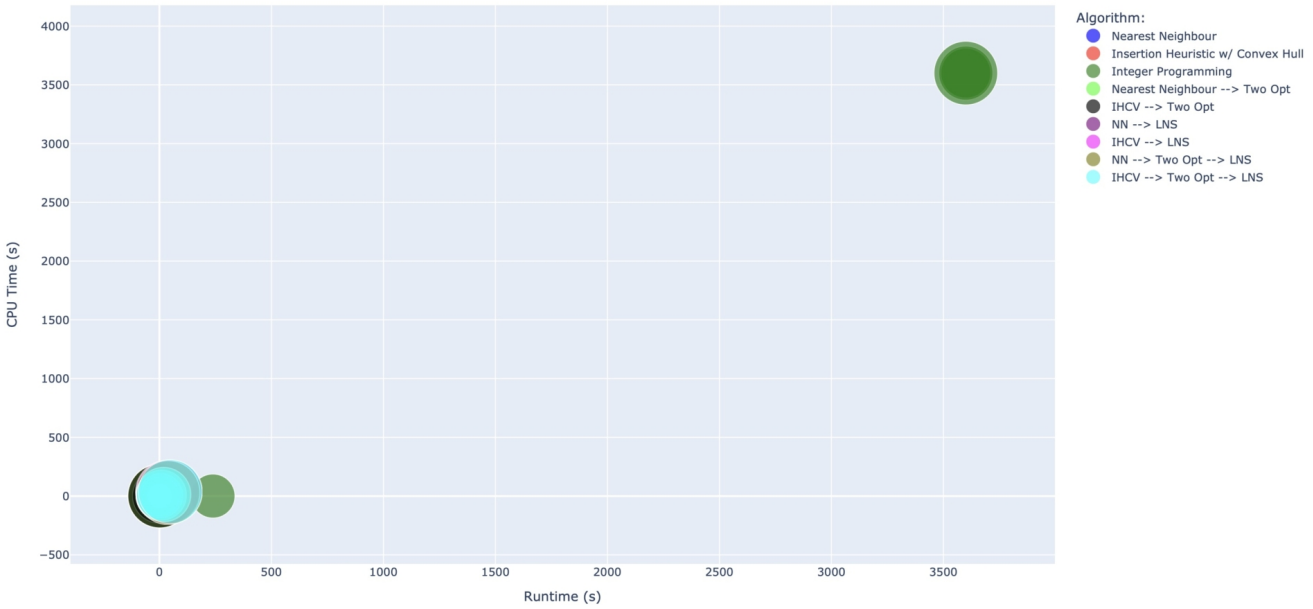

Figure A.10: Holistic Performance Comparison, Runtime vs CPU Time, all Algorithms.

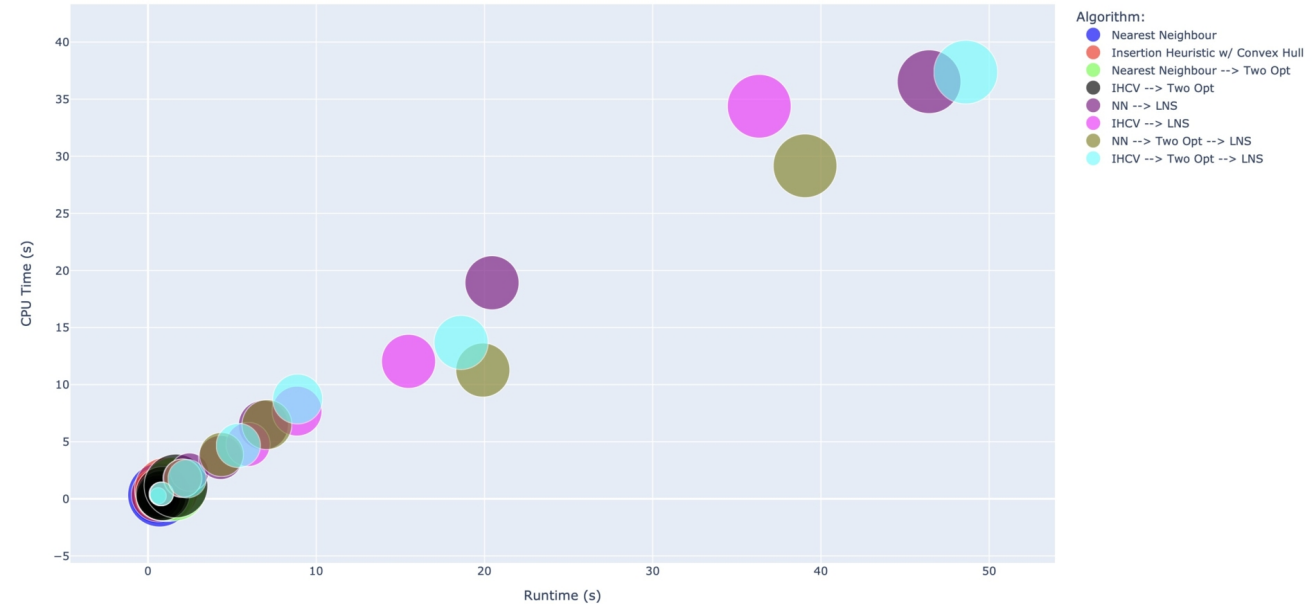

Figure A.11: Holistic Performance Comparison, Runtime vs CPU Time, no Integer Programming.

A.2 Novel Algorithm

The GA parameters used were as follows:

	test_city_1	***ulysses16***	***berlin52***	***pr76***	***pr107***	***pr136***	***tsp225***
Population Size	100	250	250	450	400	750	750
Generations	20	80	80	250	200	500	500
Mutation Rate	0.35	0.35	0.35	0.35	0.35	0.35	0.35
Crossover Rate	0.7	0.7	0.7	0.75	0.85	0.65	0.65

Table A.15: GA Parameters used when benchmarking against each TSP instance

A.2.1 Raw Results

	test_city_1	***ulysses16***	***berlin52***	***pr76***	***pr107***	***pr136***	***tsp225***
Obtained Result	23.63	73.99	7544.37	108159.44	44436.24	96860.88	3895.66
Optimal	23.63	74	7542	108159	44301	96772	3916
Percentage Increase	OPT	OPT	0.03	OPT	0.31	0.09	-0.52

Table A.16: Solution Quality for Novel Genetic Algorithm against all TSP Instances

	test_city_1	***ulysses16***	***berlin52***	***pr76***	***pr107***	***pr136***	***tsp225***
Runtime	0.4611s	2.0774s	11.6509s	51.3955s	94.0561s	749.3404s	2904.7282s
CPU Time	0.1689s	1.8290s	6.7966s	45.3103s	74.5496	721.6387s	2332.1639s

Table A.17: Runtime/CPU Time for Novel Genetic Algorithm against all TSP Instances

A.2.2 Scalability

This section displays the holistic computational results obtained after benchmarking our implementations of the base algorithms **(excluding MILP)** vs the novel GA.

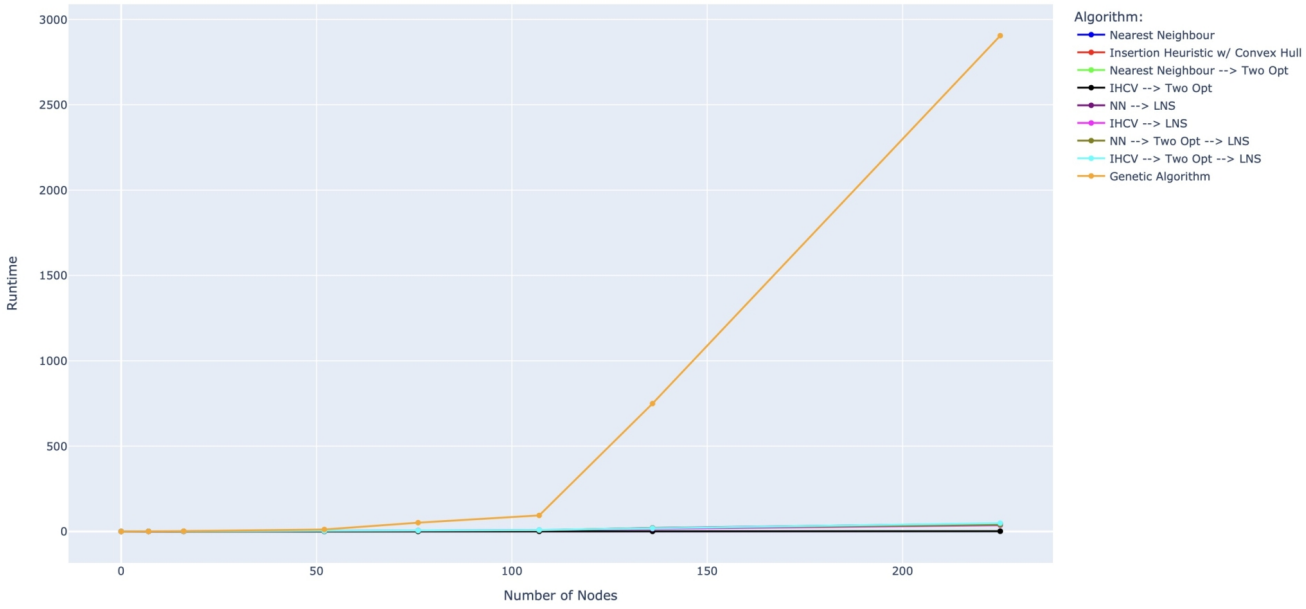

Figure A.12: Runtime

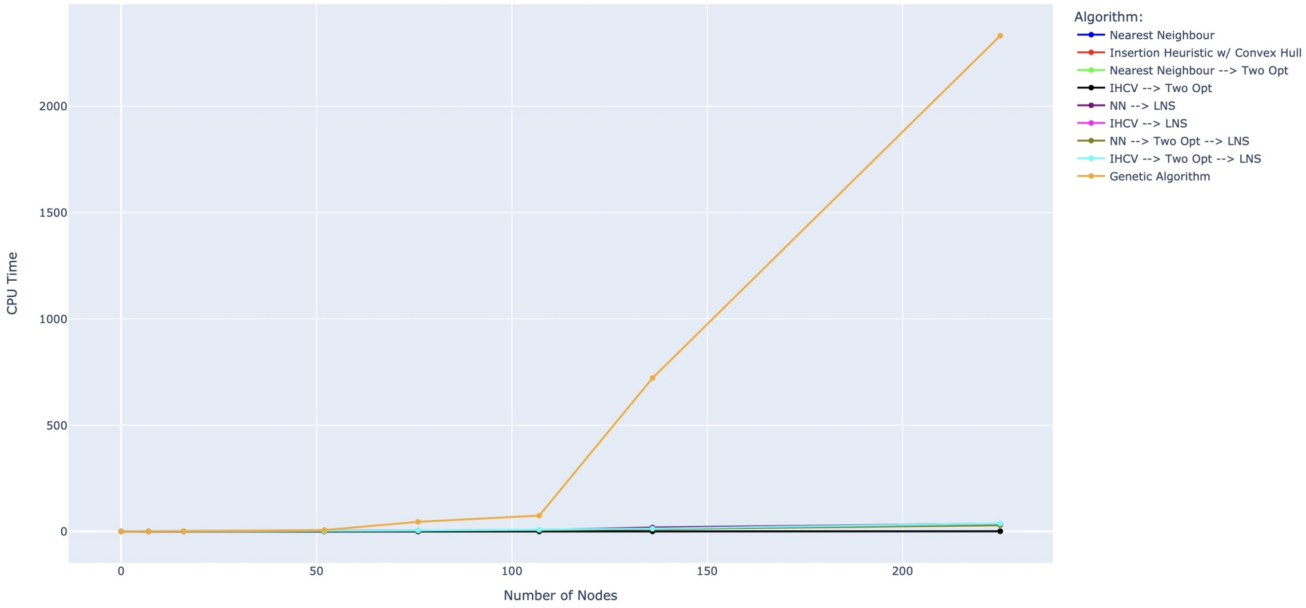

Figure A.13: CPU Time

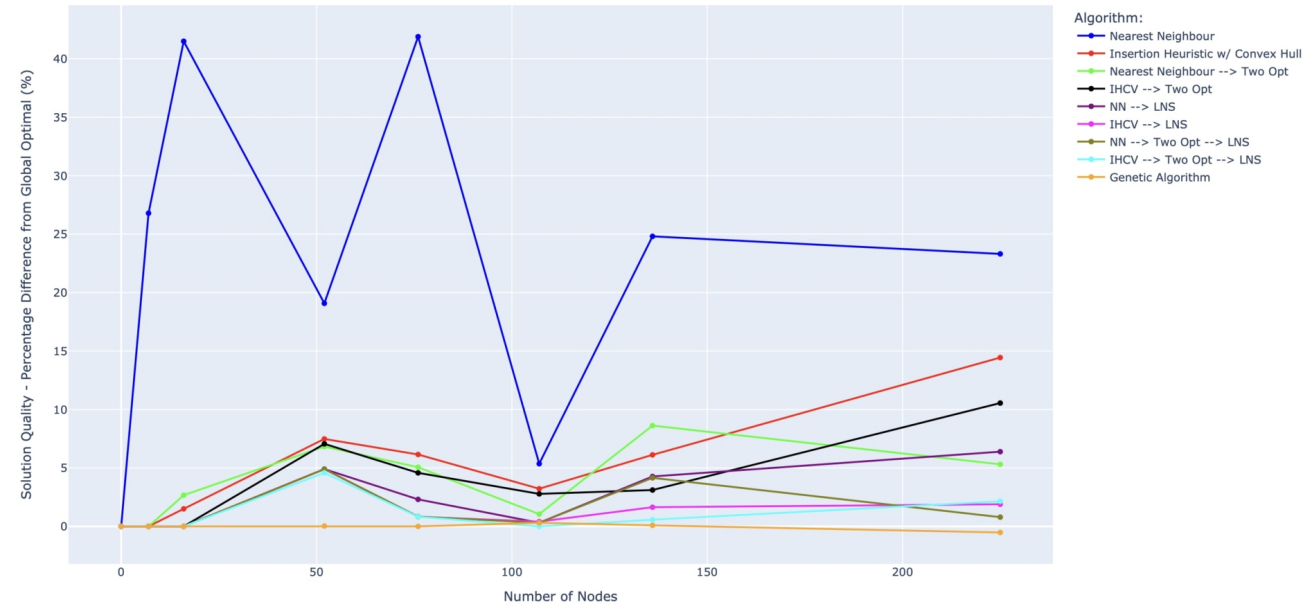

Figure A.14: Solution Quality

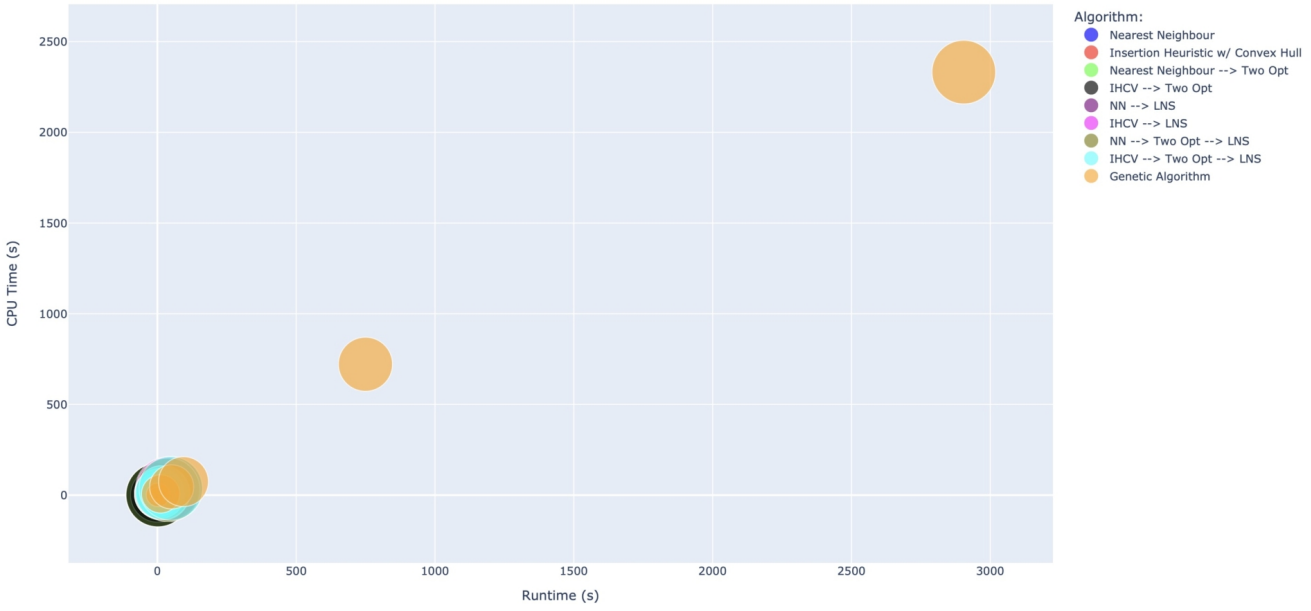

Figure A.15: Holistic Performance Comparison, Runtime v CPU Time

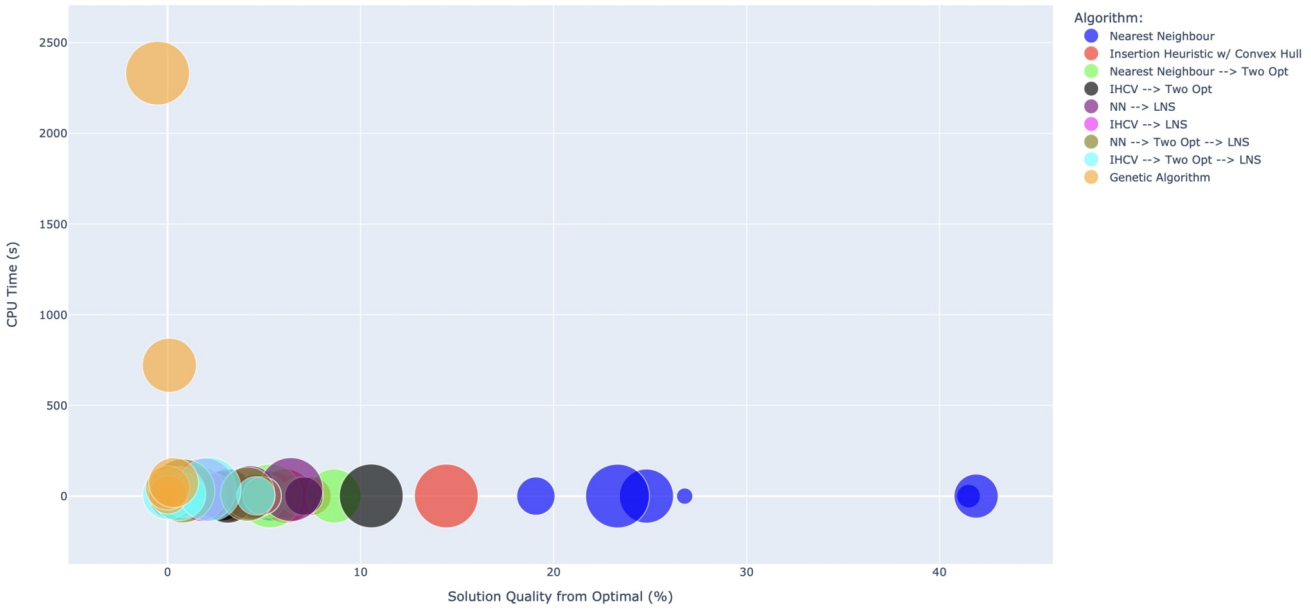

Figure A.16: Holistic Performance Comparison, CPU Time v Solution Quality

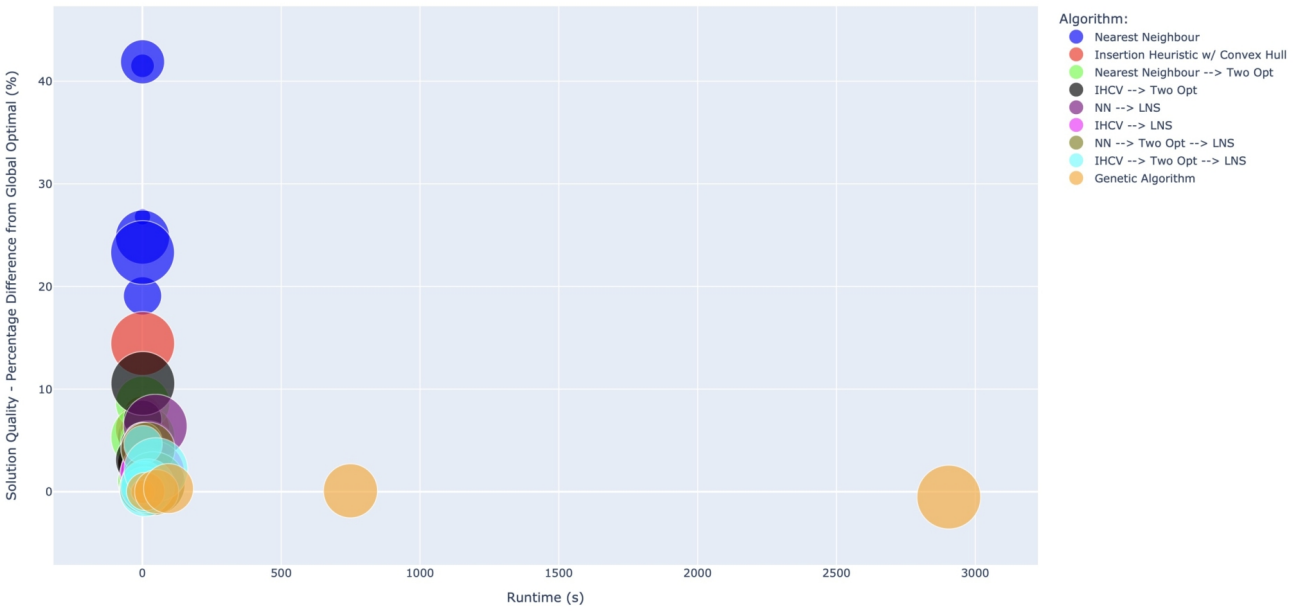

Figure A.17: Holistic Performance Comparison, Solution Quality v Runtime

Appendix B

User Guide

B.1 Instructions

Ensure that all dependencies are installed by navigating to the *Codebase* directory and running *pip install requirements.txt*. Then, to run a specific algorithm, follow this process:

1. If you wish to add your own TSP Instance to run with one of our algorithm's, ensure it is an *.xlsx* file that corresponds with our valid structure and add it to the directory called *TSP_Utilities/Test_Inputs/TSP_Instances*. The valid structure is detailed in the Design section of this report (specifically section 5.2.3).

2. Navigate into that algorithm's directory.

3. Make a call at the bottom of the script to the algorithm's relevant run functions. Many examples can be found commented out at the bottom of each script.

4. Run *python _____.py*, replacing the gap with the algorithm's filename. For example, to run NN, run *python nearest_neighbour.py*.

5. If you wish graph to be displayed on running the algorithm, ensure that the *display_route* parameter of the algorithm's *run* function is set to *True*.

Appendix C

Source Code

C.1 TSP Utility Functions

```
import numpy as np, plotly.graph_objects as go, pandas as pd, Levenshtein
from scipy.spatial.distance import cdist

# Import .tsp files from TSPLIB and convert them into Spreadsheets to be processed for extracting Node Data
def convert_tsp_lib_instance_to_spreadsheet(file_name):
    lines = []
    with open(f"../TSP_Utilities/Test_Inputs/TSPLIB_Instances/{file_name}", "r") as my_file:
        lines = my_file.readlines()

    name = [line for line in lines if line.startswith("NAME")][0]
    spreadsheet_file_name = name.split(":")[1].strip() + ".xlsx"

    node, x, y = [], [], []
    coordinate_section = False

    for line in lines:
        if coordinate_section:
            if line.strip() == "EOF":
                break
            else:
                node_columns = line.split()
                node.append(node_columns[0])
                x.append(node_columns[1])
```

```
24                  y.append(node_columns[2])
25          elif line.startswith("NODE_COORD_SECTION"):
26              coordinate_section = True
27
28      nodes = pd.DataFrame({"Node": node, "X": x, "Y": y})
29      nodes["Type"] = "Waypoint"
30      nodes.loc[0, "Type"] = "Start"
31
32      file_path = f"../TSP_Utilities/Test_Inputs/TSPLIB_Instances/{
        spreadsheet_file_name}"
33      nodes.to_excel(file_path, sheet_name = "Nodes")
34
35      return spreadsheet_file_name
36
37  #=====================Import Node Data from Spreadsheet (Includes Robust
        Validation) ======================
38  def import_node_data(file_name, for_testing_purposes=False):
39      if for_testing_purposes:
40          nodes = pd.read_excel(f"../TSP_Utilities/Test_Inputs/
        Invalid_TSP_Instances_Testing/{file_name}", sheet_name = "Nodes")
41      else:
42          nodes = pd.read_excel(f"../TSP_Utilities/Test_Inputs/TSP_Instances/{
        file_name}", sheet_name = "Nodes")
43
44      # Check for Duplicate Columns
45      actual_column_names = [column.split('.')[0] for column in nodes.columns]
46      if len(actual_column_names) != len(set(actual_column_names)):
47          raise ValueError("Duplicate column names found. Please ensure that each
        column name is unique & valid. 'Node', 'X', 'Y', 'Type' are valid and
        required. 'Name' is an optional column.")
48
49      # Check that all columns are valid
50      required_columns = ['Node', 'X', 'Y', 'Type']
51      optional_columns = ['Name']
52      valid_columns = required_columns + optional_columns
53
54      columns_set = set(nodes.columns)
55
56      missing_required_columns = set(required_columns) - columns_set # Check for
        missing columns that're required
57      if len(missing_required_columns) > 0:
58          raise ValueError(f"Missing required column(s) found: {', '.join(
```

```
        missing_required_columns)}. Please ensure that each column name is unique &
        valid. 'Node', 'X', 'Y', 'Type' are valid and required. 'Name' is an
        optional column.")
59
60      surplus_columns = columns_set - set(valid_columns) # Check for additional
        surplus columns that aren't valid
61      if len(surplus_columns) > 0:
62          raise ValueError(f"Surplus column(s) found: {', '.join(surplus_columns)
        }. Please ensure that each column name is unique & valid. 'Node', 'X', 'Y',
        'Type' are valid and required. 'Name' is an optional column.")
63
64      # Remove rows where all cells are empty (NaN)
65      nodes_cleaned = nodes.dropna(how='all')
66
67      # Check for singular cells that're empty (NaN)... except in the 'Name'
        column, where it's OK to have empty Cells
68      columns_to_check = nodes_cleaned.columns.difference(['Name'])
69      if nodes_cleaned[columns_to_check].isnull().any(axis=1).any():
70          raise ValueError("Please ensure that all cells have been filled out
        entirely, excluding cells in the 'Name' column")
71
72      # Validate Column Data types ("Node", "X", "Y" Columns)
73      if not np.issubdtype(nodes_cleaned['Node'].dtype, np.number) or (
        nodes_cleaned['Node'] < 0).any():
74          raise ValueError("Please ensure that all values in the 'Node' column are
         non-negative integers.")
75
76      if not np.issubdtype(nodes_cleaned['X'].dtype, np.number) or not np.
        issubdtype(nodes_cleaned['Y'].dtype, np.number):
77          raise ValueError("Please ensure that all values in the 'X' and 'Y'
        columns are numeric values.")
78
79      # Remove nodes w/ duplicate coordinates... keep the first one
80      nodes_cleaned = nodes_cleaned.drop_duplicates(subset=['X', 'Y'], keep='first
        ')
81
82      # Validate (and potentially auto-correct) "Type" Column
83      valid_types = ['Start', 'Waypoint']
84      errors = []
85      for (index, value) in nodes_cleaned.iterrows():
86          if value['Type'] not in valid_types:
87              distance_from_start = Levenshtein.distance(str(value['Type']), '
```

```
Start')
88              distance_from_waypoint = Levenshtein.distance(str(value['Type']), '
    Waypoint')
89
90              # Auto-correct if Levenshtein distance is <= 2
91              if distance_from_start <= 2:
92                  nodes_cleaned.at[index, 'Type'] = 'Start'
93              elif distance_from_waypoint <= 2:
94                  nodes_cleaned.at[index, 'Type'] = 'Waypoint'
95              else:
96                  errors.append(f" - '{value['Type']}'")
97
98      if len(errors) > 0:
99          raise ValueError("Misspelling error(s) found in 'Type' column. Only '
    Start' and 'Waypoint' are valid. Instead, the following were found: \n" + "\
    n".join(errors))
100
101     # More than 1 Start Node, or no Start Nodes
102     start_node_count = len(nodes_cleaned[nodes_cleaned['Type'] == 'Start'])
103     if start_node_count != 1:
104         raise ValueError(f"Please ensure that there's exactly one 'Start' node.
    You entered {start_node_count}.")
105
106     # Invalidate Inputs w/ < than 3 Nodes
107     n_nodes = len(nodes_cleaned)
108     if n_nodes < 3:
109         raise ValueError(f"You've only entered {n_nodes}. Please ensure you
    enter at least 3.")
110
111     # Order Nodes by Node Number
112     nodes_cleaned = nodes_cleaned.sort_values(by='Node').reset_index(drop=True)
113
114     # Start Node is not Node Number 1
115     start_node_index = nodes_cleaned[nodes_cleaned['Type'] == 'Start'].index[0]
116     start_node_number = nodes_cleaned.loc[start_node_index, 'Node']
117
118     if start_node_number != 1:
119         nodes_cleaned.loc[:start_node_index - 1, "Node"] += 1
120         nodes_cleaned.loc[start_node_index, 'Node'] = 1
121         nodes_cleaned = nodes_cleaned.sort_values(by='Node').reset_index(drop=
    True)
122
```

```
123     # Ensure all Node Numbers are from 1 --> n
124     nodes_cleaned['Node'] = range(1, len(nodes_cleaned) + 1)
125 
126     # print(nodes_cleaned)
127 
128     return nodes_cleaned
129 
130 # Import TSPLIB Node Data from Spreadsheet
131 def import_node_data_tsp_lib(file_name):
132     nodes = pd.read_excel(f"../TSP_Utilities/Test_Inputs/TSPLIB_Instances/{
        file_name}", sheet_name = "Nodes")
133 
134     return nodes
135 
136 def map_nodes_to_index(nodes):
137     return {node: index for (index, node) in enumerate(nodes['Node'])}
138 
139 # Compute Distance Matrix - Euclidean Distances between each Node
140 def compute_distance_matrix(nodes):
141     coordinate_list = list(zip(nodes['X'], nodes['Y']))
142     coordinate_array = np.array(coordinate_list)
143 
144     distance_matrix = cdist(coordinate_array, coordinate_array, metric='
        euclidean')
145 
146     return distance_matrix
147 
148 def compute_route_distance(route, distance_matrix, node_index_mapping):
149     return sum([distance_matrix[node_index_mapping[route[i]], node_index_mapping
        [route[i + 1]]] for i in range(len(route) - 1)])
150 
151 # Display Route as Graph
152 def display_route_as_graph(nodes, solution, name):
153     route, distance_travelled = solution
154     n_nodes = len(route)
155 
156     fig = go.Figure()
157 
158     # Plot Edges
159     for i in range(n_nodes - 1):
160         start_node_number = nodes[nodes["Node"] == route[i]]
161         end_node_number = nodes[nodes["Node"] == route[i+1]]
```

```
        coordinates_start = start_node_number[["X", "Y"]].values[0]
        coordinates_end = end_node_number[["X", "Y"]].values[0]

        line_distance = np.linalg.norm(coordinates_start - coordinates_end)

        fig.add_trace(go.Scatter(x=[coordinates_start[0], coordinates_end[0]],
                                 y=[coordinates_start[1], coordinates_end[1]],
                                 mode="lines, markers", line=dict(color="#007bff"),
                                 marker=dict(size=10, color="#ffd700"),
                                 text=[f"Node {route[i]} to Node {route[i+1]}<br>Distance: {np.round(line_distance, 2)}"], hoverinfo="text",
                                 showlegend=False))

    # Plot Nodes
    for _, node in nodes.iterrows():
        x_coordinate = node["X"]
        y_coordinate = node["Y"]
        node_number = node["Node"]

        fig.add_trace(go.Scatter(x=[x_coordinate],
                                 y=[y_coordinate],
                                 mode="text, markers",
                                 marker=dict(size=12, symbol="star" if node["Type"] == "Start" else "circle", color="DarkOrange" if node["Type"] == "Start" else "#17becf"),
                                 text=[f"{node_number}"],
                                 textposition="bottom center", hoverinfo="text", hovertext=f"Node {node_number}<br>Coordinates: ({x_coordinate}, {y_coordinate})",
                                 showlegend=False))

    # Legend
    fig.add_trace(go.Scatter(x=[None],
                             y=[None],
                             mode='markers',
                             marker=dict(size=12, symbol='star', color='DarkOrange'),
                             showlegend=True,
                             name='Start Node'))

    # Configure Graph
```

```
197     fig.update_layout(title=f"TSP Solution - {name} - Distance: {np.round(
        distance_travelled, 2)}",
198                       title_font_size=20,
199                       xaxis=dict(showline=True, showgrid=False, linecolor='#333'
        ),
200                       yaxis=dict(showline=True, showgrid=False, linecolor='#333'
        ),
201                       xaxis_title="Latitude",
202                       yaxis_title="Longitude",
203                       plot_bgcolor="rgba(0,0,0,0)", # Transparent background for
         later on when we integrate w/ HTML
204                       legend_title_text="Legend: ",
205                       legend=dict(orientation= "v", yanchor="middle", y=1.02,
        xanchor="right", x=1.05))
206
207     fig.show()
```

Listing C.1: Source Code for *tsp_utility_functions.py*

C.2 Nearest Neighbour

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities import tsp_utility_functions as tsp
4
5  import numpy as np, time
6
7  #=============================================================Nearest Neighbour Algorithm
       ============================================================
8  def nearest_neighbour(nodes, start_node_index=None):
9      n_nodes = len(nodes)
10     distance_matrix = tsp.compute_distance_matrix(nodes)
11     print(f"\n{distance_matrix}\n")
12
13     start_node_index = nodes[nodes["Type"] == "Start"].index[0]
14
15     visited = np.full(n_nodes, False)
16     route_indexed = [start_node_index]
17     current_node = start_node_index
18     distance_travelled = 0
19
```

```
20      while len(route_indexed) < n_nodes:
21          visited[current_node] = True
22          distances = distance_matrix[current_node]
23
24          mask_distances = np.where(visited, np.inf, distances) # If (visited) ==
        true, set corresponding value in distances to np.inf. If (visited) == false,
         keep corresponding value in distances. We're doing this so that we can
        select the nearest distance from the current node that hasn't been visited
        yet. This is an efficient way to allow us to update the array/matrix values
        without modifying/making copies of the original, because it is temporary and
         will be set back later. This is the process of 'masking'.
25
26          nearest_node = np.argmin(mask_distances)
27          nearest_distance = mask_distances[nearest_node]
28
29          distance_travelled += nearest_distance
30          route_indexed.append(nearest_node)
31          current_node = nearest_node
32
33      distance_travelled += distance_matrix[current_node][start_node_index]
34      route_indexed.append(start_node_index)
35
36      route = [nodes.iloc[n]["Node"] for n in route_indexed]
37
38      return route, np.round(distance_travelled, 2)
39
40  #=============================================Run Algorithm
        ==============================================
41  def run_nearest_neighbour_generic(file_name, import_node_data_func,
        display_route=True, name="Nearest Neighbour"):
42      nodes = import_node_data_func(file_name)
43      solution = nearest_neighbour(nodes)
44      if display_route:
45          tsp.display_route_as_graph(nodes, solution, name)
46
47      return solution, nodes
48
49  def run_nearest_neighbour(file_name, display_route=True):
50      return run_nearest_neighbour_generic(file_name, tsp.import_node_data,
        display_route)[0]
51
52  def run_nearest_neighbour_tsp_lib(file_name, display_route=True):
```

```
53      spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
54      return run_nearest_neighbour_generic(spreadsheet_name, tsp.
        import_node_data_tsp_lib, display_route)[0]
55
56  def run_nearest_neighbour_initial_solution(file_name, display_route=True):
57      return run_nearest_neighbour_generic(file_name, tsp.import_node_data,
        display_route)
58
59  def run_nearest_neighbour_tsp_lib_initial_solution(file_name, display_route=True
        ):
60      spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
61      return run_nearest_neighbour_generic(spreadsheet_name, tsp.
        import_node_data_tsp_lib, display_route)
62
63
64  # print(run_nearest_neighbour_tsp_lib("tsp225.tsp"))
65  # print(run_nearest_neighbour("test_city_1.xlsx"))
```

Listing C.2: Source Code for *nearest_neighbour.py*

C.3 Insertion Heuristic w/ Convex Hull

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities import tsp_utility_functions as tsp
4
5  import numpy as np, time, pandas as pd
6  from scipy.spatial import ConvexHull
7  from scipy.spatial.distance import cdist
8  np.set_printoptions(threshold=np.inf, linewidth=np.get_printoptions()['linewidth
       '])
9
10 #=============================================================IHCV Helper
       Functions=============================================================
11 # Compute Distance Matrix - Euclidean Distances between each Node
12 def compute_distance_matrix(coordinate_array):
13     distance_matrix = cdist(coordinate_array, coordinate_array, metric='
       euclidean')
14
15     return distance_matrix
16
```

```
17 #=============================================================IHCV Algorithm
       ============================================================
18 # Compute the initial Convex Hull to be built upon
19 def compute_convex_hull(nodes):
20     coordinate_list = list(zip(nodes['X'], nodes['Y']))
21     coordinate_array = np.array(coordinate_list) # Coordinates as Numpy Array
22     convex_hull = ConvexHull(coordinate_array)
23 
24     return coordinate_array, convex_hull
25 
26 # Take the Convex Hull, and iteratively find the "cheapest" insertion of each
       remaining node
27 def cheapest_insertion(coordinate_array, convex_hull, distance_matrix, nodes):
28     subtour_indices = list(convex_hull.vertices)
29     not_in_subtour = []
30     for n in range(len(nodes)):
31         if n not in subtour_indices:
32             not_in_subtour.append(n)
33 
34     while len(not_in_subtour) > 0:
35         lowest_cost = np.inf
36         for node in not_in_subtour:
37             # Go through the current subtour and try to insert nodes from
       not_in_subtour in all postions to see where the "cheapest" insertion for
       that node is
38             for i in range(len(subtour_indices)):
39                 previous_node = subtour_indices[i]
40                 next_node = subtour_indices[(i + 1) % len(subtour_indices)]
41                 cost = distance_matrix[previous_node, node] + distance_matrix[
       node, next_node] - distance_matrix[previous_node, next_node]
42 
43                 if cost < lowest_cost:
44                     lowest_cost = cost
45                     optimal_insertion = (node, i + 1)
46 
47         new_node, position = optimal_insertion
48         subtour_indices.insert(position, new_node)
49         not_in_subtour.remove(new_node)
50 
51     # Calculate total distance travelled in route
52     subtour_indices.append(subtour_indices[0])
53     route = coordinate_array[subtour_indices]
```

```
54      difference = np.diff(route, axis=0)
55      distances = np.sqrt(np.sum(np.square(difference), axis= 1))
56      distance_travelled = np.sum(distances)
57
58      # Return route w/ node numbers instead of indices
59      numbered_route = list(nodes['Node'].iloc[subtour_indices])
60
61      # Reorder route to begin with start node
62      start_node_row = nodes[nodes["Type"] == "Start"]
63      start_node = start_node_row["Node"].iloc[0]
64
65      start_node_index = numbered_route.index(start_node)
66
67      numbered_route.pop()
68      reordered_route = numbered_route[start_node_index:] + numbered_route[:
        start_node_index] + [numbered_route[start_node_index]]
69
70
71      # print(len(reordered_route))
72
73      return reordered_route, np.round(distance_travelled, 2)
74
75  #=============================================================Run Algorithm
        ============================================================
76  def run_ihcv_generic(file_name, import_node_data_func, display_route=True, name=
        "Insertion Heuristic w/ Convex Hull"):
77      nodes = import_node_data_func(file_name)
78      coordinate_array, hull = compute_convex_hull(nodes)
79      distance_matrix = compute_distance_matrix(coordinate_array)
80      # print(distance_matrix.tolist())
81      solution = cheapest_insertion(coordinate_array, hull, distance_matrix, nodes
        )
82      if display_route:
83          tsp.display_route_as_graph(nodes, solution, name)
84
85      return solution, nodes
86
87  def run_ihcv(file_name, display_route=True):
88      return run_ihcv_generic(file_name, tsp.import_node_data, display_route)[0]
89
90  def run_ihcv_tsp_lib(file_name, display_route=True):
91      spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
```

```
92      return run_ihcv_generic(spreadsheet_name, tsp.import_node_data_tsp_lib,
        display_route)[0]
93
94  def run_ihcv_initial_solution(file_name, display_route=True):
95      return run_ihcv_generic(file_name, tsp.import_node_data, display_route)
96
97  def run_ihcv_tsp_lib_initial_solution(file_name, display_route=True):
98      spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
99      return run_ihcv_generic(spreadsheet_name, tsp.import_node_data_tsp_lib,
        display_route)
100
101
102 # print(run_ihcv_tsp_lib("tsp225.tsp"))
103 # print(run_ihcv_tsp_lib("berlin52.tsp"))
104 # print(run_ihcv("test_city_1.xlsx"))
```

Listing C.3: Source Code for *ihcv.py*

C.4 Mixed-Integer Linear Programming

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities import tsp_utility_functions as tsp
4
5  import pyomo.environ as pyo, numpy as np, time
6  from pyomo.opt import SolverFactory
7
8  #=============================================================Miller-Tucker-
       Zemlin Implementation
       ============================================================
9  # Optional Parameter "gap": Configure Solver to stop once value within this "gap
       " is found
10 # Optional Parameter "time_limit": Configure Solver to have a Maximum Execution
       Time to Solve Problem
11 def formulate_and_solve_mtz(nodes, distance_matrix):
12     #====================Create Model====================
13     model = pyo.ConcreteModel()
14     n_nodes = len(nodes)
15
16     #====================Variables====================
17     model.x = pyo.Var(range(n_nodes), range(n_nodes), within=pyo.Binary)
```

```
18      x = model.x
19
20      model.u = pyo.Var(range(n_nodes), within=pyo.Integers)
21      u = model.u
22
23      model.M = pyo.Var(range(n_nodes), range(n_nodes), within=pyo.
        NonNegativeReals) # Big-M
24      M = model.M
25
26      #====================Objective Function====================
27      obj_expr = sum(sum(distance_matrix[i, j] * x[i, j] for i in range(n_nodes))
        for j in range(n_nodes))
28      model.obj = pyo.Objective(expr = obj_expr)
29
30      #====================Constraints====================
31      model.C1 = pyo.ConstraintList()
32      for i in range(n_nodes):
33          model.C1.add(sum([x[i, j] for j in range(n_nodes) if i != j]) == 1)
34
35      model.C2 = pyo.ConstraintList()
36      for i in range(n_nodes):
37          model.C2.add(sum([x[j, i] for j in range(n_nodes) if i != j]) == 1)
38
39      model.C3 = pyo.Constraint(expr = u[0] == 1)
40
41      model.C4 = pyo.ConstraintList()
42      for i in range(1, n_nodes):
43          model.C4.add(pyo.inequality(2, u[i], n_nodes)) # model.C4.add(pyo.
        inequality(1, u[i], n_nodes - 1))
44      # model.C4 = pyo.ConstraintList()
45      # for i in range(1, n_nodes):
46      #     model.C4.add(pyo.inequality(1, u[i], n_nodes - 1)) # model.C4.add(pyo.
        inequality(1, u[i], n_nodes - 1))
47
48
49      #====================vvThe 5th Constraint has been split up into 2
        Constraints in order to Linearise it according to Big-M====================
50      model.C5 = pyo.ConstraintList()
51      for i in range(1, n_nodes):
52          for j in range(1, n_nodes):
53              if i != j:
54                  model.C5.add(u[i] - u[j] + 1 - M[i, j] <= 0)
```

```
55
56      model.C6 = pyo.ConstraintList()
57      for i in range(1, n_nodes):
58          for j in range(1, n_nodes):
59              if i != j:
60                  model.C6.add(M[i, j] <= (n_nodes - 1)*(1 - x[i, j]))
61      #=====================^^The 5th Constraint has been split up into 2
        Constraints in order to Linearise it according to Big-M=====================
62      # model.pprint()
63
64      #=====================Solve=====================
65      opt = SolverFactory('gurobi')
66      # opt.options['TimeLimit'] = 400
67      # opt.options['MIPGap'] = 0.05
68      result = opt.solve(model, tee=True)
69      # result = opt.solve(model)
70
71      #=====================Process/Return Results=====================
72      # Form Solution Path Matrix
73      solution_path_matrix = np.zeros((n_nodes, n_nodes))
74
75      for i in range(n_nodes):
76          for j in range(n_nodes):
77              if i != j:
78                  solution_path_matrix[i, j] = pyo.value(x[i, j])
79              else:
80                  solution_path_matrix[i, j] = 0
81
82      # print(solution_path_matrix)
83
84      # Calculate Total Route Distance
85      distance_travelled = np.sum(solution_path_matrix * distance_matrix)
86
87      # Obtain Route as List of Indexed Nodes
88      start_node_index = nodes.index[nodes["Type"] == "Start"][0]
89      route = [start_node_index]
90      current_node = start_node_index
91
92      while True:
93          next_node = np.argmax(solution_path_matrix[current_node])
94          if next_node == start_node_index and len(route) == n_nodes:
95              route.append(next_node)
```

```
96              break
97          route.append(next_node)
98          current_node = next_node
99
100     # Convert Route of Indexed Nodes to Numbered Nodes
101     numbered_route = list(nodes.loc[route, "Node"])
102
103     return numbered_route, np.round(distance_travelled, 2), model
104
105 #=============================================================Run Algorithm
    ==========================================================
106 def run_mtz_generic(file_name, import_node_data_func, display_route=True, name="
    Integer Programming (MTZ)"):
107     nodes = import_node_data_func(file_name)
108     distance_matrix = tsp.compute_distance_matrix(nodes)
109     solution_with_model = formulate_and_solve_mtz(nodes, distance_matrix)
110     solution = solution_with_model[:2]
111     if display_route:
112         tsp.display_route_as_graph(nodes, solution, name)
113
114     return solution_with_model
115
116 def run_mtz(file_name, display_route=True):
117     return run_mtz_generic(file_name, tsp.import_node_data, display_route)
118
119 def run_mtz_tsp_lib(file_name, display_route=True):
120     spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
121     return run_mtz_generic(spreadsheet_name, tsp.import_node_data_tsp_lib,
    display_route)
122
123
124 # print(run_mtz_tsp_lib("berlin52.tsp")[:2])
125 # print(run_mtz_tsp_lib("ulysses16.tsp")[:2])
126 # print(run_mtz_tsp_lib("tsp225.tsp")[:2])
127 # print(run_mtz("test_city_1.xlsx")[:2])
```

Listing C.4: Source Code for *mtz.py*

C.5 Two Opt

```
1 import sys, numpy as np
```

```
sys.path.append("../")
from TSP_Utilities import tsp_utility_functions as tsp

from NearestNeighbour import nearest_neighbour as nn
from IHCV import ihcv as ih

PROPORTION = 1
MIN_NEIGHBOURS = 5

#=============================================================Two-Opt Algorithm
    ============================================================
# Remember to come back to this after lunch and optimise using numpy if possible
    ...
def two_opt_swap_move(route, i, j):
    route_array = np.array(route)
    route_array[i:j + 1] = route_array[i:j + 1][::-1]  # inline swap
    swapped_route = route_array.tolist()

    return swapped_route

# Pruning search space through use of nearest neighbours allows us to speed up k
    -opt, according to Steiglitz & Weiner in the paper "TSP Heuristics - Nillson
     2003"
"""Proportion argument allows us to dynamically adjust the number of nearest
    neighbours. Min_Neighbours sets a minimum threshold for this number.
This ensures a minimum level of exploration, regardless of size of inputted
    dataset. Adjusting the Proportion Argument allows us to decide on
the trade-off between runtime efficiency & solution quality. Higher proportion
    generally = more accurate solution + slower runtime, and vice versa.
"""
def compute_nearest_neighbours(distance_matrix, proportion=PROPORTION,
    min_neighbours=MIN_NEIGHBOURS):
    n_nodes = len(distance_matrix)
    n = max(int(proportion * n_nodes), min_neighbours)
    nearest_neighbours = []

    for i in range(n_nodes):
        distances = distance_matrix[i, :] # Get all elements in the i'th row
        neighbours_indexed = np.argsort(distances)
        nearest_neighbours.append(neighbours_indexed[1:n + 1].tolist()) # Get
    the "n" nearest neighbours to the node

```

```
35      return nearest_neighbours
36
37  def two_opt(initial_solution, distance_matrix, nearest_neighbours,
        node_index_mapping):
38      initial_route = initial_solution[0]
39      initial_distance = initial_solution[1]
40      route = initial_route
41      shortest_distance = initial_distance
42
43      while True:
44          no_improvement_found = True   # Assume we won't find an improvement in
        this 2-opt iteration
45
46          for i in range(1, len(route) - 2):
47              for j in nearest_neighbours[node_index_mapping[route[i]]]:  # We
        only consider the nearest neighbours for swapping!
48                  i_index = node_index_mapping[route[i - 1]]
49                  j_index = node_index_mapping[route[j]]
50                  next_node_index = node_index_mapping[route[(j + 1) % len(route)
        ]]
51
52                  current_distance = distance_matrix[i_index, node_index_mapping[
        route[i]]] + distance_matrix[j_index, next_node_index]
53                  swapped_distance = distance_matrix[i_index, j_index] +
        distance_matrix[node_index_mapping[route[i]], next_node_index]
54
55                  if current_distance > swapped_distance:  # We then prune the
        search space by only proceeding to swapping if there's potential for the
        swap to lead to improvement in decreasing total distance!
56                      swapped_route = two_opt_swap_move(route, i, j % len(route))
57                      swapped_route_distance = sum([distance_matrix[
        node_index_mapping[swapped_route[k]], node_index_mapping[swapped_route[k +
        1]]] for k in range(len(swapped_route) - 1)])
58
59                      if swapped_route_distance < shortest_distance:
60                          route = swapped_route
61                          shortest_distance = swapped_route_distance
62                          no_improvement_found = False  # Improvement found!...
63                          break  # ...therefore, we'll break out this loop.
64
65              if no_improvement_found == False:  # Resume outer loop because we
        found improvement
```

```
66                  break
67
68          if no_improvement_found:  # Didn't find an improvement, so we stop
69              break
70
71      return route, np.round(shortest_distance, 2)
72
73
74  #=============================================================Run Algorithm
        ============================================================
75  def run_two_opt_generic(initial_solution_func, file_name, name="Insertion
        Heuristic w/ Convex Hull --> Two-Opt", display_route=True):
76      initial_solution, nodes = initial_solution_func(file_name, False)
77      distance_matrix = tsp.compute_distance_matrix(nodes)
78      nearest_neighbours = compute_nearest_neighbours(distance_matrix)
79      node_index_mapping = tsp.map_nodes_to_index(nodes)
80      solution = two_opt(initial_solution, distance_matrix, nearest_neighbours,
        node_index_mapping)
81      if display_route:
82          tsp.display_route_as_graph(nodes, solution, name)
83
84      return solution, nodes
85
86  def run_two_opt_nearest_neighbour(file_name, display_route=True):
87      return run_two_opt_generic(nn.run_nearest_neighbour_initial_solution,
        file_name, "Nearest Neighbour --> Two-Opt", display_route)[0]
88
89  def run_two_opt_ihcv(file_name, display_route=True):
90      return run_two_opt_generic(ih.run_ihcv_initial_solution, file_name, "
        Insertion Heuristic w/ Convex Hull --> Two-Opt", display_route)[0]
91
92  def run_two_opt_nearest_neighbour_tsp_lib(file_name, display_route=True):
93      return run_two_opt_generic(nn.run_nearest_neighbour_tsp_lib_initial_solution
        , file_name, "Nearest Neighbour --> Two-Opt", display_route)[0]
94
95  def run_two_opt_ihcv_tsp_lib(file_name, display_route=True):
96      return run_two_opt_generic(ih.run_ihcv_tsp_lib_initial_solution, file_name,
        "Insertion Heuristic w/ Convex Hull --> Two-Opt", display_route)[0]
97
98
99  def run_two_opt_nearest_neighbour_initial_solution(file_name, display_route=True
        ):
```

```
100     return run_two_opt_generic(nn.run_nearest_neighbour_initial_solution,
        file_name, "Nearest Neighbour --> Two-Opt", display_route)
101
102 def run_two_opt_ihcv_initial_solution(file_name, display_route=True):
103     return run_two_opt_generic(ih.run_ihcv_initial_solution, file_name, "
        Insertion Heuristic w/ Convex Hull --> Two-Opt", display_route)
104
105 def run_two_opt_nearest_neighbour_tsp_lib_initial_solution(file_name,
        display_route=True):
106     return run_two_opt_generic(nn.run_nearest_neighbour_tsp_lib_initial_solution
        , file_name, "Nearest Neighbour --> Two-Opt", display_route)
107
108 def run_two_opt_ihcv_tsp_lib_initial_solution(file_name, display_route=True):
109     return run_two_opt_generic(ih.run_ihcv_tsp_lib_initial_solution, file_name,
        "Insertion Heuristic w/ Convex Hull --> Two-Opt", display_route)
110
111
112
113 # print(run_two_opt_ihcv_tsp_lib("tsp225.tsp")) #4329
114 # print(run_two_opt_ihcv_tsp_lib("ulysses16.tsp"))
115 # print(run_two_opt_ihcv_tsp_lib("berlin52.tsp"))
116 # print(run_two_opt_nearest_neighbour_tsp_lib("tsp225.tsp"))
117 # print(run_two_opt_ihcv_tsp_lib("pr76.tsp"))
118 # print(run_two_opt_nearest_neighbour_tsp_lib("pr76.tsp"))
119 # print(run_two_opt_nearest_neighbour("test_city_1.xlsx"))
```

Listing C.5: Source Code for *two_opt.py*

C.6 Large Neighbourhood Search

```
1 import sys, numpy as np
2 sys.path.append("../")
3 from TSP_Utilities import tsp_utility_functions as tsp
4 from NearestNeighbour import nearest_neighbour as nn
5 from IHCV import ihcv as ih
6 from TwoOpt import two_opt as to
7
8 PROPORTION = 0.15
9 MAX_ITERATIONS = 100000
10 IMPROVEMENT_THRESHOLD = 0.000001
11
```

```
#=================================================Large Neighbourhood Search Algorithm=================================================
def destroy(route, proportion=PROPORTION):
    route_array = np.array(route)
    n_nodes_to_remove = np.maximum(1, int(len(route_array) * proportion))

    removable_indices = np.arange(1, len(route_array) - 1) # All nodes except the start node can be a destroyed node
    indices_to_remove = np.random.choice(removable_indices, size = n_nodes_to_remove, replace = False)
    mask_removed = np.full(len(route_array), True, dtype=bool)
    mask_removed[indices_to_remove] = False

    remaining_route = (route_array[mask_removed]).tolist()
    removed_nodes = (route_array[indices_to_remove]).tolist()

    return remaining_route, removed_nodes

def repair(remaining_route, removed_nodes, distance_matrix, node_index_mapping):
    remaining_route_array = np.array(remaining_route)

    for node in removed_nodes:
        optimal_position = None
        min_cost_increase = np.inf

        node_index = node_index_mapping[node]
        costs_to_node = distance_matrix[:, node_index] # Retrieve the column corresponding to node_index inside the dm - all distances to the node at node_index
        costs_from_node = distance_matrix[node_index, :] # Retrieve the row, hence gets all distances from the node

        for i in range(1, len(remaining_route_array)):
            cost_increase = ((costs_to_node[node_index_mapping[remaining_route_array[i - 1]]]) + (costs_from_node[node_index_mapping[remaining_route_array[i]]]) -
                             (distance_matrix[node_index_mapping[remaining_route_array[i - 1]], node_index_mapping[remaining_route_array[i]]]))

            if cost_increase < min_cost_increase:
                min_cost_increase = cost_increase
```

```
44                  optimal_position = i
45
46          remaining_route_array = (np.insert(remaining_route_array,
    optimal_position, node)).tolist()
47
48      return remaining_route_array
49
50  def lns(initial_route, distance_matrix, node_index_mapping, max_iterations=
    MAX_ITERATIONS, improvement_threshold=IMPROVEMENT_THRESHOLD):
51      if improvement_threshold >= 1:
52          raise ValueError("improvement_threshold must be < 1")
53
54      shortest_route = initial_route
55      shortest_distance = tsp.compute_route_distance(shortest_route,
    distance_matrix, node_index_mapping)
56      n_iterations_without_improvement = 0 # Tracks the number of iterations w/out
     improvement in solution
57
58      for _ in range(max_iterations):
59          remaining_route, removed_nodes = destroy(shortest_route)
60          possible_shortest_route = repair(remaining_route, removed_nodes,
    distance_matrix, node_index_mapping)
61          possible_shortest_distance = tsp.compute_route_distance(
    possible_shortest_route, distance_matrix, node_index_mapping)
62
63          if possible_shortest_distance < (shortest_distance * (1 -
    improvement_threshold)):
64              shortest_route = possible_shortest_route
65              shortest_distance = possible_shortest_distance
66              n_iterations_without_improvement = 0
67
68          else:
69              n_iterations_without_improvement += 1
70
71          if n_iterations_without_improvement > (max_iterations * 0.1):  # If the
    number of iterations w/out improvement exceeds 10% of the maximum no. of
    iterations, then algorithm terminates early
72              break
73
74      return shortest_route, np.round(shortest_distance, 2)
75
76  #=============================================================Run Algorithm
```

```
77 def run_lns_generic(initial_solution_func, file_name, name, display_route=True):
78     initial_solution, nodes = initial_solution_func(file_name, False)
79     initial_route = initial_solution[0]
80     distance_matrix = tsp.compute_distance_matrix(nodes)
81     node_index_mapping = tsp.map_nodes_to_index(nodes)
82     solution = lns(initial_route, distance_matrix, node_index_mapping)
83     if display_route:
84         tsp.display_route_as_graph(nodes, solution, name)
85 
86     return solution
87 
88 # NN --> LNS
89 def run_lns_nearest_neighbour(file_name, display_route=True):
90     return run_lns_generic(nn.run_nearest_neighbour_initial_solution, file_name,
       "Nearest Neighbour --> Large Neighbourhood Search", display_route)
91 
92 # IHCV --> LNS
93 def run_lns_ihcv(file_name, display_route=True):
94     return run_lns_generic(ih.run_ihcv_initial_solution, file_name, "Insertion
      Heuristic w/ Convex Hull --> Large Neighbourhood Search", display_route)
95 
96 # NN --> LNS
97 def run_lns_nearest_neighbour_tsp_lib(file_name, display_route=True):
98     return run_lns_generic(nn.run_nearest_neighbour_tsp_lib_initial_solution,
      file_name, "Nearest Neighbour --> Large Neighbourhood Search", display_route
      )
99 
100 # IHCV --> LNS
101 def run_lns_ihcv_tsp_lib(file_name, display_route=True):
102     return run_lns_generic(ih.run_ihcv_tsp_lib_initial_solution, file_name, "
      Insertion Heuristic w/ Convex Hull --> Large Neighbourhood Search",
      display_route)
103 
104 # NN --> 2-Opt --> LNS
105 def run_lns_two_opt_nearest_neighbour(file_name, display_route=True):
106     return run_lns_generic(to.run_two_opt_nearest_neighbour_initial_solution,
      file_name, "Nearest Neighbour --> Two-Opt --> Large Neighbourhood Search",
      display_route)
107 
108 # IHCV --> 2-Opt --> LNS
109 def run_lns_two_opt_ihcv(file_name, display_route=True):
```

```
110     return run_lns_generic(to.run_two_opt_ihcv_initial_solution, file_name, "
        Insertion Heuristic w/ Convex Hull --> Two-Opt --> Large Neighbourhood
        Search", display_route)
111
112 # NN --> 2-Opt --> LNS
113 def run_lns_two_opt_nearest_neighbour_tsp_lib(file_name, display_route=True):
114     return run_lns_generic(to.
        run_two_opt_nearest_neighbour_tsp_lib_initial_solution, file_name, "Nearest
        Neighbour --> Two-Opt --> Large Neighbourhood Search", display_route)
115
116 # IHCV --> 2-Opt --> LNS
117 def run_lns_two_opt_ihcv_tsp_lib(file_name, display_route=True):
118     return run_lns_generic(to.run_two_opt_ihcv_tsp_lib_initial_solution,
        file_name, "Insertion Heuristic w/ Convex Hull --> Two-Opt --> Large
        Neighbourhood Search", display_route)
119
120
121 # print(run_lns_ihcv_tsp_lib("pr76.tsp"))
122 # print(run_lns_ihcv_tsp_lib("berlin52.tsp"))
123 # run_lns_two_opt_ihcv_tsp_lib("berlin52.tsp")
124 # run_lns_two_opt_nearest_neighbour_tsp_lib("berlin52.tsp")
125 # run_lns_ihcv_tsp_lib("tsp225.tsp")
126 # nodes = tsp.import_node_data("test_city_1.xlsx")
127 # print(nodes)
128 # node_to_retrieve = 1
129 # node_index = nodes.index[nodes["Node"] == node_to_retrieve][0]
130 # print(node_index)
131 # print(run_lns_nearest_neighbour("test_city_1.xlsx"))
132 # print(run_lns_nearest_neighbour_tsp_lib("tsp225.tsp"))
133 # print(run_lns_two_opt_ihcv_tsp_lib("berlin52.tsp", False))
```

Listing C.6: Source Code for *lns.py*

C.7 Genetic Algorithm

```
1 import sys
2 sys.path.append("../")
3 from TSP_Utilities import tsp_utility_functions as tsp
4 from TwoOpt import two_opt as to
5 from NearestNeighbour import nearest_neighbour as nn
6
```

```
7  import numpy as np, matplotlib.pyplot as plt
8  import time
9
10 #==========================GA Params==========================
11 POPULATION_SIZE = 60
12 GENERATIONS = 30
13 MUTATION_RATE = 0.35
14 CROSSOVER_RATE = 0.65
15 # POPULATION_SIZE = 750
16 # GENERATIONS = 500
17 # MUTATION_RATE = 0.35
18 # CROSSOVER_RATE = 0.65
19
20 #=========Further Params=========
21 STAGNATION_TOLERANCE = 0.01
22 TWO_OPT_APPLY_THRESHOLD = 15
23 NN_FRACTION = 0.2
24 ELITISM_THRESHOLDS = {
25     10: 0.5,  # For TSP instance of 10 nodes = preserve top 50%. This top 50%
       will be considered the "Elite" Individuals
26     30: 0.35,  # For 30 nodes = preserve top 35%
27     50: 0.3,  # For 50 nodes = preserve top 25%
28     100: 0.25,  # For 100 nodes = preserve top 15%
29     250: 0.08,  # For 250 nodes = preserve top 8%
30     500: 0.05,  # For 500 nodes = preserve top 5%
31     np.inf: 0.01  # For 500+ nodes = preserve top 1%
32 }
33
34 #==========================GA Algorithm Design==========================
35 #=========Setup Individuals=========
36 def generate_route(n_nodes):
37     # A nice way of randomising route using Numpy...
38     unvisited = np.arange(2, n_nodes + 1)
39     np.random.shuffle(unvisited)
40     route = np.concatenate(([1], unvisited, [1]))
41
42     return route
43
44 def encode_individual(route, n_nodes):
45     individual = np.zeros((n_nodes, n_nodes))
46
47     for i in range(len(route) - 1):
```

```
48          start_index = route[i] - 1   # Adjust index because we're now working w/
        Numpy
49          end_index = route[i + 1] - 1
50          individual[start_index, end_index] = 1  # Link strat node to end node
51
52      # print(individual)
53
54      return individual
55
56  def generate_individual(n_nodes):
57      route = generate_route(n_nodes)
58      individual_encoded = encode_individual(route, n_nodes)
59
60      return (individual_encoded, route)
61
62  def spawn_new_individuals(population, n_new_individuals, n_nodes):
63      replace_indices = np.random.choice(len(population), n_new_individuals,
        replace=False)
64
65      for i in replace_indices:
66          population[i] = generate_individual(n_nodes)
67
68      return population
69
70  #==========Generate Population==========
71  def initialise_population(nodes, n_nodes):
72      nn_population_size = int(POPULATION_SIZE * NN_FRACTION)
73      nn_population = []
74
75      for _ in range(nn_population_size):
76          start_node_index = np.random.choice(np.arange(1, n_nodes))
77          route, _ = nn.nearest_neighbour(nodes, start_node_index)
78          individual_encoded = encode_individual(route, n_nodes)
79          nn_population.append((individual_encoded, np.array(route)))
80
81      random_population = [generate_individual(n_nodes) for _ in range(
        POPULATION_SIZE - nn_population_size)]
82      random_population = [(np.array(individual[0]), np.array(individual[1])) for
        individual in random_population]
83
84      population = nn_population + random_population
85
```

```
86      return population
87
88  def compute_fitness(individual_route, distance_matrix):
89      individual_route = np.array(individual_route)
90      individual_route_adjusted = individual_route - 1 # for Numpy
91      distances = distance_matrix[individual_route_adjusted[:-1],
        individual_route_adjusted[1:]]
92
93      travelled_distance = np.sum(distances)
94
95      return travelled_distance
96
97  def compute_population_fitness(population, distance_matrix):
98      fitness_values = []
99      fitness_values = [compute_fitness(route, distance_matrix) for _, route in
        population]
100
101     return fitness_values
102
103 def sort_population(population, fitness_values):
104     population_fitness_values = list(zip(population, fitness_values))
105     population_fitness_values.sort(key = lambda x: x[1])
106
107     sorted_population = [i[0] for i in population_fitness_values]
108     sorted_fitness_values = [i[1] for i in population_fitness_values]
109
110     return sorted_population, sorted_fitness_values
111
112 #=================================GA Operators=================================
113 #==========Select Parents==========
114 def select_parent_binary_tournament(population, distance_matrix):
115     population_size = len(population)
116
117     candidate_indices = np.random.choice(population_size, 2, replace=False)
118     candidate_parent_1 = population[candidate_indices[0]]
119     candidate_parent_2 = population[candidate_indices[1]]
120
121     parent_1_fitness = compute_fitness(candidate_parent_1[1], distance_matrix)
122     parent_2_fitness = compute_fitness(candidate_parent_2[1], distance_matrix)
123
124     if parent_1_fitness < parent_2_fitness:
125         return candidate_parent_1
```

```
126     else:
127         return candidate_parent_2
128 
129 #=========Crossover=========
130 def fill_chromosome(offspring, parent, crossover_point_1, crossover_point_2):
131     offspring_to_fill = np.where(offspring == -1)[0]
132     parent_genes_not_in_offspring = parent[~np.isin(parent, offspring[
    crossover_point_1:crossover_point_2])]
133 
134     offspring[offspring_to_fill] = parent_genes_not_in_offspring[:
    offspring_to_fill.size]
135 
136 # Specifically, this is an "Order Crossover"
137 def crossover(parent_1, parent_2):
138     parent_1 = np.array(parent_1)
139     parent_2 = np.array(parent_2)
140     length = parent_1.size
141 
142     crossover_point_1, crossover_point_2 = np.sort(np.random.choice(np.arange(1,
     length - 1), 2, replace=False))
143 
144     # Placeholders for offspring_1 & 2's Genes before Parents Crossover
145     offspring_1 = np.full(length, -1)
146     offspring_2 = np.full(length, -1)
147 
148     # Copy corresponding Crossover Segment from Parents to Offspring
149     offspring_1[crossover_point_1:crossover_point_2] = parent_1[
    crossover_point_1:crossover_point_2]
150     offspring_2[crossover_point_1:crossover_point_2] = parent_2[
    crossover_point_1:crossover_point_2]
151 
152     # Fill out empty Genes in offspring_1's Chromosome w/ Genes from parent_2
153     fill_chromosome(offspring_1, parent_2, crossover_point_1, crossover_point_2)
154     # Fill out empty Genes in offspring_2's Chromosome w/ Genes from parent_1
155     fill_chromosome(offspring_2, parent_1, crossover_point_1, crossover_point_2)
156 
157     offspring_1[-1] = offspring_1[0]
158     offspring_2[-1] = offspring_2[0]
159 
160     return offspring_1.tolist(), offspring_2.tolist()
161 
162 #=========Mutation=========
```

```
163 def is_stagnant(generations_best_fitness_values, generation):
164     stagnation_check_percentage = 0
165     if GENERATIONS <= 100:
166         stagnation_check_percentage = 0.5  # Check the last 50% of generations
    for small generation counts
167     elif GENERATIONS <= 500:
168         stagnation_check_percentage = 0.3  # 30% for medium
169     else:
170         stagnation_check_percentage = 0.2  # 20% for larger generation counts
171 
172     stagnation_check_span = int(GENERATIONS * stagnation_check_percentage)
173     if generation < stagnation_check_span or len(generations_best_fitness_values
    ) < stagnation_check_span:
174         return False
175 
176     recent_fitness_values = generations_best_fitness_values[-
    stagnation_check_span:]
177     if recent_fitness_values[0] != 0:
178         improvement = (recent_fitness_values[0] - recent_fitness_values[-1])/(
    recent_fitness_values[0])
179     else:
180         improvement = 0
181 
182     return improvement < STAGNATION_TOLERANCE
183 
184 # An "Adaptive Mutation" Approach has been taken here; mutation rate may change
    over time. The idea of this is to mitigate the possibility of converging to
    local optima
185 def mutation(individual_route, generation, generations_best_fitness_values,
    n_nodes):
186     individual_route = np.array(individual_route)
187     stagnant = is_stagnant(generations_best_fitness_values, generation)
188 
189     if stagnant:
190         base_rate = MUTATION_RATE * 2
191     else:
192         base_rate = MUTATION_RATE
193 
194     adaptive_rate = base_rate * (1 - generation /GENERATIONS)
195 
196     if np.random.rand() < adaptive_rate:
197         mutation_point1, mutation_point2 = np.random.choice(np.arange(1, n_nodes
```

```
    ), 2, replace=False)
198         individual_route[mutation_point1], individual_route[mutation_point2] =
    individual_route[mutation_point2], individual_route[mutation_point1]
199
200     return individual_route.tolist()
201
202 #==========Two Opt for further Optimisation==========
203 def should_apply_two_opt(generations_best_fitness_values):
204     if len(generations_best_fitness_values) < TWO_OPT_APPLY_THRESHOLD:
205         return False
206
207     recent_best_value = generations_best_fitness_values[-TWO_OPT_APPLY_THRESHOLD
    :]
208     if recent_best_value[-1] == recent_best_value[0]:
209         return True  # no improvement - apply two opt
210
211     return False
212
213 def apply_two_opt(route, distance_matrix, nearest_neighbours, nodes):
214     optimised_route, distance_travelled = to.two_opt((route, compute_fitness(
    route, distance_matrix)), distance_matrix, nearest_neighbours, tsp.
    map_nodes_to_index(nodes))
215
216     return optimised_route
217
218 #==========Generate Next Generation==========
219 def compute_elitism_count(n_nodes, population_size):
220     for threshold, percentage in ELITISM_THRESHOLDS.items():
221         if n_nodes <= threshold:
222             return int(population_size * percentage)
223
224 def generate_next_generation(sorted_population, generation,
    generations_best_fitness_values, n_nodes, distance_matrix,
    nearest_neighbours, nodes):
225     new_population = []
226
227     # Elitism
228     elitism_count = compute_elitism_count(n_nodes, POPULATION_SIZE)
229     elite_individuals = sorted_population[:elitism_count]
230     new_population.extend(elite_individuals)
231
232     while len(new_population) < POPULATION_SIZE:
```

```
233         parent_1, parent_1_route = select_parent_binary_tournament(
    sorted_population, distance_matrix)
234         parent_2, parent_2_route = select_parent_binary_tournament(
    sorted_population, distance_matrix)
235 
236         while np.array_equal(parent_1_route, parent_2_route): # If they happen
    to be equal, then change parent_2
237             parent_2, parent_2_route = select_parent_binary_tournament(
    sorted_population, distance_matrix)
238 
239         if np.random.rand() < CROSSOVER_RATE:
240             offspring_1_route, offspring_2_route = crossover(parent_1_route,
    parent_2_route)
241         else:
242             offspring_1_route, offspring_2_route = parent_1_route,
    parent_2_route
243 
244         if should_apply_two_opt(generations_best_fitness_values):
245             offspring_1_route = apply_two_opt(offspring_1_route, distance_matrix
    , nearest_neighbours, nodes)
246             offspring_2_route = apply_two_opt(offspring_2_route, distance_matrix
    , nearest_neighbours, nodes)
247 
248         offspring_1_route = mutation(offspring_1_route, generation,
    generations_best_fitness_values, n_nodes)
249         offspring_2_route = mutation(offspring_2_route, generation,
    generations_best_fitness_values, n_nodes)
250 
251         offspring_1 = encode_individual(offspring_1_route, n_nodes)
252         offspring_2 = encode_individual(offspring_2_route, n_nodes)
253 
254         new_population.append((offspring_1, offspring_1_route))
255 
256         if len(new_population) < POPULATION_SIZE:
257             new_population.append((offspring_2, offspring_2_route))
258 
259     return new_population
260 
261 #============================GA Run/Performance Tracking
    ============================
262 generations_best_fitness_value = np.inf
263 generations_best_route = None
```

```
264 generations_best_fitness_values = []
265 generations_average_fitness_values = []
266
267 #==========Run GA - Functions==========
268 def run_ga_generic(file_name, import_node_data_func,population_size=
    POPULATION_SIZE, generations=GENERATIONS, mutation_rate=MUTATION_RATE,
    crossover_rate=CROSSOVER_RATE, display_route=True):
269     global POPULATION_SIZE, GENERATIONS, MUTATION_RATE, CROSSOVER_RATE
270     # Update Global Variable values
271     POPULATION_SIZE = population_size
272     GENERATIONS = generations
273     MUTATION_RATE = mutation_rate
274     CROSSOVER_RATE = crossover_rate
275
276     nodes = import_node_data_func(file_name)
277
278     n_nodes = len(nodes)
279     distance_matrix = tsp.compute_distance_matrix(nodes)
280     nearest_neighbours = to.compute_nearest_neighbours(distance_matrix)
281
282     start_time = time.time()
283     population = initialise_population(nodes, n_nodes)
284     population_fitness = compute_population_fitness(population, distance_matrix)
285     sorted_population, sorted_fitness_values = sort_population(population,
    population_fitness)
286
287     if sorted_fitness_values[0] < generations_best_fitness_value:
288         generations_best_fitness = sorted_fitness_values[0]
289         generations_best_route = sorted_population[0][1]
290
291     generations_best_fitness_values.append(generations_best_fitness)
292     generations_average_fitness_values.append(sum(sorted_fitness_values) / len(
    sorted_fitness_values))
293
294     stagnation_counter = 0
295
296     for generation in range(1, GENERATIONS):
297         if generation % 50 == 0:  # Every 50 generations, introduce some new
    individuals into the population (and remove the POPULATION_SIZE/50 least fit
     individuals) in order to introduce some genetic diversity and hopefully
    mitigate convergence towards local optima, hence pushing towards global
    optimum
```

```
298             sorted_population = spawn_new_individuals(sorted_population, int(
    POPULATION_SIZE/25), n_nodes)
299         new_population = generate_next_generation(sorted_population, generation,
     generations_best_fitness_values, n_nodes, distance_matrix,
    nearest_neighbours, nodes)
300         new_population_fitness = compute_population_fitness(new_population,
    distance_matrix)
301         sorted_new_population, sorted_new_fitness_values = sort_population(
    new_population, new_population_fitness)
302 
303         if sorted_new_fitness_values[0] < generations_best_fitness:
304             generations_best_fitness = sorted_new_fitness_values[0]
305             generations_best_route = sorted_new_population[0][1]
306             stagnation_counter = 0  # Improvement - reset stagnant counter
307         else:
308             stagnation_counter += 1
309 
310         generations_best_fitness_values.append(generations_best_fitness)
311         generations_average_fitness_values.append(sum(sorted_new_fitness_values)
     / len(sorted_new_fitness_values))
312 
313         # Check if we've stagnated and therefore need to terminate prematurely
314         if is_stagnant(generations_best_fitness_values, generation):
315             print(f"Early termination (generation {generation}/{GENERATIONS})
    due to stagnation")
316             break
317 
318         sorted_population = sorted_new_population
319 
320         print(np.round(generations_best_fitness, 2))
321         print(f"^{np.round((generation/GENERATIONS) * 100, 2)}% - Generation {
    generation}")
322 
323     end_time = time.time()
324     time_elapsed = end_time - start_time
325 
326     print(f"Time Elapsed: {np.round(time_elapsed, 6)}s")
327 
328     if display_route:
329         tsp.display_route_as_graph(nodes, (generations_best_route,
    generations_best_fitness), "Genetic Algorithm")
330 
```

```
331     return list(generations_best_route), np.round(generations_best_fitness, 2)
332
333 def run_ga(file_name, display_route=True):
334     return run_ga_generic(file_name, tsp.import_node_data, display_route=
        display_route)
335
336 def run_ga_tsp_lib(file_name, display_route=True):
337     spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet(file_name)
338     return run_ga_generic(spreadsheet_name, tsp.import_node_data_tsp_lib,
        display_route=display_route)
339
340 #=========Run GA - Execution=========
341 print(run_ga("test_city_1.xlsx"))
342 # print(run_ga_tsp_lib("pr76.tsp"))
343 # print(run_ga_tsp_lib("berlin52.tsp", False))
344 # print(run_ga_tsp_lib("ulysses16.tsp"))
345 # print(run_ga_tsp_lib("pr107.tsp"))
346 # print(run_ga_tsp_lib("tsp225.tsp"))
347 # print(run_ga_tsp_lib("pr136.tsp"))
348
349 #=========Display GA Performance Graphs=========
350 # plt.plot(generations_best_fitness_values)
351 # plt.xlabel('Generation Number')
352 # plt.ylabel('Best Fitness Value')
353 # plt.title('Best Fitness Value per Generation')
354 # plt.show()
355
356 # plt.plot(generations_average_fitness_values)
357 # plt.xlabel('Generation Number')
358 # plt.ylabel('Average Fitness Value')
359 # plt.title('Average Fitness Value per Generation')
360 # plt.show()
```

Listing C.7: Source Code for *genetic_algorithm.py*

C.8 Benchmarking Implementations

```
1 """This file aims to run all implemented algorithms for the standard VRP against
2 each other, comparing them for things like runtime, solution quality, etc."""
3
4 """     One key thing to note is that these runtimes include the fully
```

```
    integrated process,
        from reading the files to displaying the calculated route as a graph
        """

import sys, time, numpy as np, plotly.graph_objects as go
sys.path.append("../")
from NearestNeighbour import nearest_neighbour
from IHCV import ihcv
from IntegerProgramming import mtz
from TwoOpt import two_opt
from LargeNeighbourhood import lns
from GeneticAlgorithm import genetic_algorithm as ga

# Compute Average Runtime & CPU Time after "n" Iterations
def compute_average_times(algorithm, dataset, n):
    times = []
    cpu_times = []

    for _ in range(n):
        start_time = time.time()
        start_cpu_time = time.process_time()

        algorithm(*dataset)

        end_time = time.time()
        end_cpu_time = time.process_time()

        time_elapsed = end_time - start_time
        cpu_time_elapsed = end_cpu_time - start_cpu_time

        times.append(time_elapsed)
        cpu_times.append(cpu_time_elapsed)

    average_runtime = np.mean(times)
    average_cpu_times = np.mean(cpu_times)

    return f"{average_runtime}s", f"{average_cpu_times}s",

def compute_percentage_difference(generated_solution, optimal_solution):
    percentage_difference = (abs(generated_solution - optimal_solution) /
    optimal_solution) * 100

```

```
44      return percentage_difference
45
46
47
48  ##==========================Dataset 1 - Test City 1 - 7 Nodes - Runtime,
        CPU Time==========================
49  ## Nearest Neighbour - 0.2912076759338379s, 0.06434819999999997s (10 runs)
50  # nn_tc1_runtime = compute_average_times(nearest_neighbour.run_nearest_neighbour
        , ("test_city_1.xlsx",), 10)
51  # print(nn_tc1_runtime)
52
53  ## Insertion Heuristic w/ Convex Hull - 0.31925201416015625s,
        0.05864439999999984s (10 runs)
54  # ihcv_tc1_runtime = compute_average_times(ihcv.run_ihcv, ("test_city_1.xlsx",),
         10)
55  # print(ihcv_tc1_runtime)
56
57  ## Integer Programming - 0.48547115325927737s, 0.08146700000000004s (10 runs)
58  # ip_tc1_runtime = compute_average_times(mtz.run_mtz, ("test_city_1.xlsx",), 10)
59  # print(ip_tc1_runtime)
60
61  ## Nearest Neighbour --> Two Opt - 0.3369459629058838s, 0.058959400000000176s
        (10 runs)
62  # nn_two_opt_tc1_runtime = compute_average_times(two_opt.
        run_two_opt_nearest_neighbour, ("test_city_1.xlsx",), 10)
63  # print(nn_two_opt_tc1_runtime)
64
65  ## IHCV --> Two Opt - 0.398201584815979s, 0.05850529999999994s (10 runs)
66  # ihcv_two_opt_tc1_runtime = compute_average_times(two_opt.run_two_opt_ihcv, ("
        test_city_1.xlsx",), 10)
67  # print(ihcv_two_opt_tc1_runtime)
68
69  ## NN --> LNS - 0.593600082397461s, 0.25243659999999996s (10 runs)
70  # nn_lns_tc1_runtime = compute_average_times(lns.run_lns_nearest_neighbour, ("
        test_city_1.xlsx",), 10)
71  # print(nn_lns_tc1_runtime)
72
73  ## IHCV --> LNS - 0.6017665624618531s, 0.24799320000000016s (10 runs)
74  # ihcv_lns_tc1_runtime = compute_average_times(lns.run_lns_ihcv, ("test_city_1.
        xlsx",), 10)
75  # print(ihcv_lns_tc1_runtime)
76
```

```
77 # # NN --> Two Opt --> LNS - 0.6001407957077026s, 0.2650253s (10 runs)
78 # nn_two_opt_lns_tc1_runtime = compute_average_times(lns.
       run_lns_two_opt_nearest_neighbour, ("test_city_1.xlsx",), 10)
79 # print(nn_two_opt_lns_tc1_runtime)
80
81 # # IHCV --> Two Opt --> LNS - 0.6019640445709229s, 0.2553482999999999s (10 runs
       )
82 # ihcv_two_opt_lns_tc1_runtime = compute_average_times(lns.run_lns_two_opt_ihcv,
        ("test_city_1.xlsx",), 10)
83 # print(ihcv_two_opt_lns_tc1_runtime)
84
85 # # Genetic Algorithm - 0.46117939949035647s, 0.16895220000000002s (10 runs)
86 # ga_tc1_runtime = compute_average_times(ga.run_ga, ("test_city_1.xlsx",), 10)
87 # print(ga_tc1_runtime)
88
89 """
90 GA PARAMS USED:
91     POPULATION_SIZE = 100
92     GENERATIONS = 20
93     MUTATION_RATE = 0.35
94     CROSSOVER_RATE = 0.7
95 """
96
97 # #==========================Solution Quality==========================
       - Optimal = 23.63
98 # # Nearest Neighbour - 29.96
99 # nn_tc1_solution = nearest_neighbour.run_nearest_neighbour("test_city_1.xlsx")
       [1]
100 # print(nn_tc1_solution)
101
102 # # Insertion Heuristic w/ Convex Hull - 23.63
103 # ihcv_tc1_solution = ihcv.run_ihcv("test_city_1.xlsx")[1]
104 # print(ihcv_tc1_solution)
105
106 # # Integer Programming - 23.63
107 # ip_tc1_solution = mtz.run_mtz("test_city_1.xlsx")[1]
108 # print(ip_tc1_solution)
109
110 # # Nearest Neighbour --> Two Opt - 23.63
111 # nn_two_opt_tc1_solution = two_opt.run_two_opt_nearest_neighbour("test_city_1.
       xlsx")[1]
112 # print(nn_two_opt_tc1_solution)
```

```
113
114 # # IHCV --> Two Opt - 23.63
115 # ihcv_two_opt_tc1_solution = two_opt.run_two_opt_ihcv("test_city_1.xlsx")[1]
116 # print(ihcv_two_opt_tc1_solution)
117
118 # # NN --> LNS - 23.63
119 # nn_lns_tc1_solution = lns.run_lns_nearest_neighbour("test_city_1.xlsx")[1]
120 # print(nn_lns_tc1_solution)
121
122 # # IHCV --> LNS - 23.63
123 # ihcv_lns_tc1_solution = lns.run_lns_ihcv("test_city_1.xlsx")[1]
124 # print(ihcv_lns_tc1_solution)
125
126 # # NN --> Two Opt --> LNS - 23.63
127 # nn_two_opt_lns_tc1_solution = lns.run_lns_two_opt_nearest_neighbour("
        test_city_1.xlsx")[1]
128 # print(nn_two_opt_lns_tc1_solution)
129
130 # # IHCV --> Two Opt --> LNS - 23.63
131 # ihcv_two_opt_lns_tc1_solution = lns.run_lns_two_opt_ihcv("test_city_1.xlsx")
        [1]
132 # print(ihcv_two_opt_lns_tc1_solution)
133
134 # # Genetic Algorithm - 23.63
135 # ga_tc1_solution = ga.run_ga("test_city_1.xlsx")[1]
136 # print(ga_tc1_solution)
137
138 """
139 GA PARAMS USED:
140     POPULATION_SIZE = 100
141     GENERATIONS = 20
142     MUTATION_RATE = 0.35
143     CROSSOVER_RATE = 0.7
144 """
145
146
147
148 # #============================Dataset 2 - ulysses16 - 16 Nodes - Runtime
        ============================
149 # # Nearest Neighbour - 0.3927781581878662s, 0.06856299999999979s (10 runs)
150 # nn_ulysses16_runtime = compute_average_times(nearest_neighbour.
        run_nearest_neighbour_tsp_lib, ("ulysses16.tsp",), 10)
```

```
151 # print(nn_ulysses16_runtime)
152
153 # # Insertion Heuristic w/ Convex Hull - 0.4028491497039795s,
      0.06983930000000002s (10 runs)
154 # ihcv_ulysses16_runtime = compute_average_times(ihcv.run_ihcv_tsp_lib, ("
      ulysses16.tsp",), 10)
155 # print(ihcv_ulysses16_runtime)
156
157 # # Integer Programming - 1.9439756274223328s, 0.0933790000000001s (4 runs)
158 # ip_ulysses16_runtime = compute_average_times(mtz.run_mtz_tsp_lib, ("ulysses16.
      tsp",), 4)
159 # print(ip_ulysses16_runtime)
160
161 # # Nearest Neighbour --> Two Opt - 0.4849355220794678s, 0.07274639999999995s
      (10 runs)
162 # nn_two_opt_ulysses16_runtime = compute_average_times(two_opt.
      run_two_opt_nearest_neighbour_tsp_lib, ("ulysses16.tsp",), 10)
163 # print(nn_two_opt_ulysses16_runtime)
164
165 # # IHCV --> Two Opt - 0.4879587411880493s, 0.06585739999999998s (10 runs)
166 # ihcv_two_opt_ulysses16_runtime = compute_average_times(two_opt.
      run_two_opt_ihcv_tsp_lib, ("ulysses16.tsp",), 10)
167 # print(ihcv_two_opt_ulysses16_runtime)
168
169 # # NN --> LNS - 0.7752989768981934s, 0.42592280000000005s (10 runs)
170 # nn_lns_ulysses16_runtime = compute_average_times(lns.
      run_lns_nearest_neighbour_tsp_lib, ("ulysses16.tsp",), 10)
171 # print(nn_lns_ulysses16_runtime)
172
173 # # IHCV --> LNS - 0.7552976131439209s, 0.41674609999999995s (10 runs)
174 # ihcv_lns_ulysses16_runtime = compute_average_times(lns.run_lns_ihcv_tsp_lib,
      ("ulysses16.tsp",), 10)
175 # print(ihcv_lns_ulysses16_runtime)
176
177 # # NN --> Two Opt --> LNS - 0.8133480072021484s, 0.4447246999999999s (10 runs)
178 # nn_two_opt_lns_ulysses16_runtime = compute_average_times(lns.
      run_lns_two_opt_nearest_neighbour_tsp_lib, ("ulysses16.tsp",), 10)
179 # print(nn_two_opt_lns_ulysses16_runtime)
180
181 # # IHCV --> Two Opt --> LNS - 0.8137904930114746s, 0.4334836999999997s (10 runs
      )
182 # ihcv_two_opt_lns_ulysses16_runtime = compute_average_times(lns.
```

```
        run_lns_two_opt_ihcv_tsp_lib, ("ulysses16.tsp",), 10)
# print(ihcv_two_opt_lns_ulysses16_runtime)

## Genetic Algorithm - 2.077482557296753s, 1.8290260999999997s (10 runs)
# ga_ulysses16_runtime = compute_average_times(ga.run_ga_tsp_lib, ("ulysses16.
        tsp",), 10)
# print(ga_ulysses16_runtime)

"""
GA PARAMS USED:
        POPULATION_SIZE = 250
        GENERATIONS = 80
        MUTATION_RATE = 0.35
        CROSSOVER_RATE = 0.7
"""

##==========================Solution Quality==========================
        - TSPLIB's Optimal = 74
## Nearest Neighbour - 104.73
# nn_ulysses16_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("
        ulysses16.tsp")[1]
# print(nn_ulysses16_solution)

## Insertion Heuristic w/ Convex Hull - 75.11
# ihcv_ulysses16_solution = ihcv.run_ihcv_tsp_lib("ulysses16.tsp")[1]
# print(ihcv_ulysses16_solution)

## Integer Programming - 73.99
# ip_ulysses16_solution = mtz.run_mtz_tsp_lib("ulysses16.tsp")[1]
# print(ip_ulysses16_solution)

## Nearest Neighbour --> Two Opt - 75.97
# nn_two_opt_ulysses16_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib
        ("ulysses16.tsp")[1]
# print(nn_two_opt_ulysses16_solution)

## IHCV --> Two Opt - 73.99
# ihcv_two_opt_ulysses16_solution = two_opt.run_two_opt_ihcv_tsp_lib("ulysses16.
        tsp")[1]
# print(ihcv_two_opt_ulysses16_solution)

## NN --> LNS - 73.99
```

```
# nn_lns_ulysses16_solution = lns.run_lns_nearest_neighbour_tsp_lib("ulysses16.tsp")[1]
# print(nn_lns_ulysses16_solution)

# # IHCV --> LNS - 73.99
# ihcv_lns_ulysses16_solution = lns.run_lns_ihcv_tsp_lib("ulysses16.tsp")[1]
# print(ihcv_lns_ulysses16_solution)

# # NN --> Two Opt --> LNS - 73.99
# nn_two_opt_lns_ulysses16_solution = lns.run_lns_two_opt_nearest_neighbour_tsp_lib("ulysses16.tsp")[1]
# print(nn_two_opt_lns_ulysses16_solution)

# # IHCV --> Two Opt --> LNS - 73.99
# ihcv_two_opt_lns_ulysses16_solution = lns.run_lns_two_opt_ihcv_tsp_lib("ulysses16.tsp")[1]
# print(ihcv_two_opt_lns_ulysses16_solution)

# # Genetic Algorithm - 73.99
# ga_ulysses16_solution = ga.run_ga_tsp_lib("ulysses16.tsp")[1]
# print(ga_ulysses16_solution)

"""
GA PARAMS USED:
    POPULATION_SIZE = 250
    GENERATIONS = 80
    MUTATION_RATE = 0.35
    CROSSOVER_RATE = 0.7
"""


# #==========================Dataset 3 - berlin52 - 52 Nodes - Runtime, CPU Time==========================
# # Nearest Neighbour - 0.4872169017791748s, 0.1028905s (10 runs)
# nn_berlin52_runtime = compute_average_times(nearest_neighbour.run_nearest_neighbour_tsp_lib, ("berlin52.tsp",), 10)
# print(nn_berlin52_runtime)

# # Insertion Heuristic w/ Convex Hull - 0.5403737020492554s, 0.11625139999999998s (10 runs)
# ihcv_berlin52_runtime = compute_average_times(ihcv.run_ihcv_tsp_lib, ("berlin52.tsp",), 10)
```

```
254 # print(ihcv_berlin52_runtime)
255
256 # # Integer Programming - 2.188145935535431s, 0.3090054999999998s (4 runs)
257 # ip_berlin52_runtime = compute_average_times(mtz.run_mtz_tsp_lib, ("berlin52.
        tsp",), 4)
258 # print(ip_berlin52_runtime)
259
260 # # Nearest Neighbour --> Two Opt - 0.5839934539794921s, 0.12024610000000022s
        (10 runs)
261 # nn_two_opt_berlin52_runtime = compute_average_times(two_opt.
        run_two_opt_nearest_neighbour_tsp_lib, ("berlin52.tsp",), 10)
262 # print(nn_two_opt_berlin52_runtime)
263
264 # # IHCV --> Two Opt - 0.5570610189437866s, 0.1256982999999999s (10 runs)
265 # ihcv_two_opt_berlin52_runtime = compute_average_times(two_opt.
        run_two_opt_ihcv_tsp_lib, ("berlin52.tsp",), 10)
266 # print(ihcv_two_opt_berlin52_runtime)
267
268 # # NN --> LNS - 2.462945890426636s, 2.3432852s (10 runs)
269 # nn_lns_berlin52_runtime = compute_average_times(lns.
        run_lns_nearest_neighbour_tsp_lib, ("berlin52.tsp",), 10)
270 # print(nn_lns_berlin52_runtime)
271
272 # # IHCV --> LNS - 2.0279680490493774s, 1.7926807999999999s (10 runs)
273 # ihcv_lns_berlin52_runtime = compute_average_times(lns.run_lns_ihcv_tsp_lib, ("
        berlin52.tsp",), 10)
274 # print(ihcv_lns_berlin52_runtime)
275
276 # # NN --> Two Opt --> LNS - 2.0701633214950563s, 1.8235629999999996s (10 runs)
277 # nn_two_opt_lns_berlin52_runtime = compute_average_times(lns.
        run_lns_two_opt_nearest_neighbour_tsp_lib, ("berlin52.tsp",), 10)
278 # print(nn_two_opt_lns_berlin52_runtime)
279
280 # # IHCV --> Two Opt --> LNS - 2.3267541885375977s, 1.7669904000000003s (10 runs
        )
281 # ihcv_two_opt_lns_berlin52_runtime = compute_average_times(lns.
        run_lns_two_opt_ihcv_tsp_lib, ("berlin52.tsp",), 10)
282 # print(ihcv_two_opt_lns_berlin52_runtime)
283
284 # # Genetic Algorithm - 11.650958061218262s, 6.796694400000002s (10 runs)
285 # ga_berlin52_runtime = compute_average_times(ga.run_ga_tsp_lib, ("berlin52.tsp
        ",), 10)
```

```
# print(ga_berlin52_runtime)

"""
GA PARAMS USED:
    POPULATION_SIZE = 250
    GENERATIONS = 80
    MUTATION_RATE = 0.35
    CROSSOVER_RATE = 0.7
"""

##==================================Solution Quality==================================
    - TSPLIB's Optimal = 7542
## Nearest Neighbour - 8980.92
# nn_berlin52_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("
    berlin52.tsp")[1]
# print(nn_berlin52_solution)

## Insertion Heuristic w/ Convex Hull - 8105.78
# ihcv_berlin52_solution = ihcv.run_ihcv_tsp_lib("berlin52.tsp")[1]
# print(ihcv_berlin52_solution)

## Integer Programming - 7544.37
# ip_berlin52_solution = mtz.run_mtz_tsp_lib("berlin52.tsp")[1]
# print(ip_berlin52_solution)

## Nearest Neighbour --> Two Opt - 8056.83
# nn_two_opt_berlin52_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib("
    berlin52.tsp")[1]
# print(nn_two_opt_berlin52_solution)

## IHCV --> Two Opt - 8074.56
# ihcv_two_opt_berlin52_solution = two_opt.run_two_opt_ihcv_tsp_lib("berlin52.
    tsp")[1]
# print(ihcv_two_opt_berlin52_solution)

## NN --> LNS - 7911.33
# nn_lns_berlin52_solution = lns.run_lns_nearest_neighbour_tsp_lib("berlin52.tsp
    ")[1]
# print(nn_lns_berlin52_solution)

## IHCV --> LNS - 7911.33
# ihcv_lns_berlin52_solution = lns.run_lns_ihcv_tsp_lib("berlin52.tsp")[1]
```

```
# print(ihcv_lns_berlin52_solution)

# # NN --> Two Opt --> LNS - 7911.33
# nn_two_opt_lns_berlin52_solution = lns.
    run_lns_two_opt_nearest_neighbour_tsp_lib("berlin52.tsp")[1]
# print(nn_two_opt_lns_berlin52_solution)

# # IHCV --> Two Opt --> LNS - 7887.23
# ihcv_two_opt_lns_berlin52_solution = lns.run_lns_two_opt_ihcv_tsp_lib("
    berlin52.tsp")[1]
# print(ihcv_two_opt_lns_berlin52_solution)

# # Genetic Algorithm - 7544.37
# ga_berlin52_solution = ga.run_ga_tsp_lib("berlin52.tsp")[1]
# print(ga_berlin52_solution)

"""
GA PARAMS USED:
    POPULATION_SIZE = 250
    GENERATIONS = 80
    MUTATION_RATE = 0.35
    CROSSOVER_RATE = 0.7
"""



# #==========================Dataset 4 - pr76 - 76 Nodes - Runtime, CPU Time
    ==========================
# # Nearest Neighbour - 0.5912003755569458s, 0.13564180000000023s (10 runs)
# nn_pr76_runtime = compute_average_times(nearest_neighbour.
    run_nearest_neighbour_tsp_lib, ("pr76.tsp",), 10)
# print(nn_pr76_runtime)

# # Insertion Heuristic w/ Convex Hull - 0.6079960489273072s,
    0.15631350000000008s (10 runs)
# ihcv_pr76_runtime = compute_average_times(ihcv.run_ihcv_tsp_lib, ("pr76.tsp",)
    , 10)
# print(ihcv_pr76_runtime)

# # Integer Programming - 238.9238269329071s, 0.6113710000000001s (2 runs)
# ip_pr76_runtime = compute_average_times(mtz.run_mtz_tsp_lib, ("pr76.tsp",), 2)
# print(ip_pr76_runtime)
```

```
359
360 # # Nearest Neighbour --> Two Opt - 0.6298728704452514s, 0.16514700000000007s
        (10 runs)
361 # nn_two_opt_pr76_runtime = compute_average_times(two_opt.
        run_two_opt_nearest_neighbour_tsp_lib, ("pr76.tsp",), 10)
362 # print(nn_two_opt_pr76_runtime)
363
364 # # IHCV --> Two Opt - 0.6108887052536011s, 0.15282080000000012s (10 runs)
365 # ihcv_two_opt_pr76_runtime = compute_average_times(two_opt.
        run_two_opt_ihcv_tsp_lib, ("pr76.tsp",), 10)
366 # print(ihcv_two_opt_pr76_runtime)
367
368 # # NN --> LNS - 4.3152546882629395s, 3.6238146s (10 runs)
369 # nn_lns_pr76_runtime = compute_average_times(lns.
        run_lns_nearest_neighbour_tsp_lib, ("pr76.tsp",), 10)
370 # print(nn_lns_pr76_runtime)
371
372 # # IHCV --> LNS - 5.947743972142537s, 4.7678232000000005s (10 runs)
373 # ihcv_lns_pr76_runtime = compute_average_times(lns.run_lns_ihcv_tsp_lib, ("pr76
        .tsp",), 10)
374 # print(ihcv_lns_pr76_runtime)
375
376 # # NN --> Two Opt --> LNS - 4.365378888448079s, 3.8795434s (10 runs)
377 # nn_two_opt_lns_pr76_runtime = compute_average_times(lns.
        run_lns_two_opt_nearest_neighbour_tsp_lib, ("pr76.tsp",), 10)
378 # print(nn_two_opt_lns_pr76_runtime)
379
380 # # IHCV --> Two Opt --> LNS - 5.3695143063863116s, 4.6750652s (10 runs)
381 # ihcv_two_opt_lns_pr76_runtime = compute_average_times(lns.
        run_lns_two_opt_ihcv_tsp_lib, ("pr76.tsp",), 10)
382 # print(ihcv_two_opt_lns_pr76_runtime)
383
384 # # Genetic Algorithm - 51.395593881607056s, 45.3103905s (10 runs)
385 # ga_pr76_runtime = compute_average_times(ga.run_ga_tsp_lib, ("pr76.tsp",), 10)
386 # print(ga_pr76_runtime)
387
388 """
389 GA PARAMS USED:
390     POPULATION_SIZE = 450
391     GENERATIONS = 250
392     MUTATION_RATE = 0.35
393     CROSSOVER_RATE = 0.75
```

```
394 """
395
396 ##==============================Solution Quality==============================
      - TSPLIB's Optimal = 108159
397 ## Nearest Neighbour - 153461.92
398 # nn_pr76_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("pr76.tsp")
      [1]
399 # print(nn_pr76_solution)
400
401 ## Insertion Heuristic w/ Convex Hull - 114808.11
402 # ihcv_pr76_solution = ihcv.run_ihcv_tsp_lib("pr76.tsp")[1]
403 # print(ihcv_pr76_solution)
404
405 ## Integer Programming - 108159.44
406 # ip_pr76_solution = mtz.run_mtz_tsp_lib("pr76.tsp")[1]
407 # print(ip_pr76_solution)
408
409 ## Nearest Neighbour --> Two Opt - 113635.27
410 # nn_two_opt_pr76_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib("pr76
      .tsp")[1]
411 # print(nn_two_opt_pr76_solution)
412
413 ## IHCV --> Two Opt - 113113.89
414 # ihcv_two_opt_pr76_solution = two_opt.run_two_opt_ihcv_tsp_lib("pr76.tsp")[1]
415 # print(ihcv_two_opt_pr76_solution)
416
417 ## NN --> LNS - 110666.14
418 # nn_lns_pr76_solution = lns.run_lns_nearest_neighbour_tsp_lib("pr76.tsp")[1]
419 # print(nn_lns_pr76_solution)
420
421 ## IHCV --> LNS - 109044.07
422 # ihcv_lns_pr76_solution = lns.run_lns_ihcv_tsp_lib("pr76.tsp")[1]
423 # print(ihcv_lns_pr76_solution)
424
425 ## NN --> Two Opt --> LNS - 109067.89
426 # nn_two_opt_lns_pr76_solution = lns.run_lns_two_opt_nearest_neighbour_tsp_lib("
      pr76.tsp")[1]
427 # print(nn_two_opt_lns_pr76_solution)
428
429 ## IHCV --> Two Opt --> LNS - 109044.07
430 # ihcv_two_opt_lns_pr76_solution = lns.run_lns_two_opt_ihcv_tsp_lib("pr76.tsp")
      [1]
```

```
# print(ihcv_two_opt_lns_pr76_solution)

## Genetic Algorithm - 108159.44
# ga_pr76_solution = ga.run_ga_tsp_lib("pr76.tsp")[1]
# print(ga_pr76_solution)

"""
GA PARAMS USED:
    POPULATION_SIZE = 450
    GENERATIONS = 250
    MUTATION_RATE = 0.35
    CROSSOVER_RATE = 0.75
"""



## ==========Dataset 5 - pr107 - 107 Nodes - Runtime, CPU Time==========
## Nearest Neighbour - 0.5175444841384887s, 0.17077940000000016s (10 runs)
# nn_pr107_runtime = compute_average_cpu_time(nearest_neighbour.run_nearest_neighbour_tsp_lib, ("pr107.tsp",), 10)
# print(nn_pr107_runtime)

## Insertion Heuristic w/ Convex Hull - 0.6871454000473023s, 0.2842870999999999s (10 runs)
# ihcv_pr107_runtime = compute_average_cpu_time(ihcv.run_ihcv_tsp_lib, ("pr107.tsp",), 10)
# print(ihcv_pr107_runtime)

## Integer Programming - 1 hour execution

## Nearest Neighbour --> Two Opt - 0.8510305881500244s, 0.24584100000000006s (10 runs)
# nn_two_opt_pr107_runtime = compute_average_cpu_time(two_opt.run_two_opt_nearest_neighbour_tsp_lib, ("pr107.tsp",), 10)
# print(nn_two_opt_pr107_runtime)

## IHCV --> Two Opt - 0.7930902640024821s, 0.2809012s (10 runs)
# ihcv_two_opt_pr107_runtime = compute_average_cpu_time(two_opt.run_two_opt_ihcv_tsp_lib, ("pr107.tsp",), 10)
# print(ihcv_two_opt_pr107_runtime)
```

```
466
467 # # NN --> LNS - 6.857412974039714s, 6.4553855s (10 runs)
468 # nn_lns_pr107_runtime = compute_average_cpu_time(lns.
        run_lns_nearest_neighbour_tsp_lib, ("pr107.tsp",), 10)
469 # print(nn_lns_pr107_runtime)
470
471 # # IHCV --> LNS - 8.85807736714681s, 7.682524599999998s (10 runs)
472 # ihcv_lns_pr107_runtime = compute_average_cpu_time(lns.run_lns_ihcv_tsp_lib, ("
        pr107.tsp",), 10)
473 # print(ihcv_lns_pr107_runtime)
474
475 # # NN --> Two Opt --> LNS - 7.071200450261434s, 6.4950489000000005s (10 runs)
476 # nn_two_opt_lns_pr107_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_nearest_neighbour_tsp_lib, ("pr107.tsp",), 10)
477 # print(nn_two_opt_lns_pr107_runtime)
478
479 # # IHCV --> Two Opt --> LNS - 8.895966529846191s, 8.743738600000002s (10 runs)
480 # ihcv_two_opt_lns_pr107_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_ihcv_tsp_lib, ("pr107.tsp",), 10)
481 # print(ihcv_two_opt_lns_pr107_runtime)
482
483 # # Genetic Algorithm - 94.05615496635437s, 74.549643s (10 runs)
484 # ga_pr107_runtime = compute_average_cpu_time(ga.run_ga_tsp_lib, ("pr107.tsp",),
        10)
485 # print(ga_pr107_runtime)
486
487 """
488 GA PARAMS USED:
489     POPULATION_SIZE = 400
490     GENERATIONS = 200
491     MUTATION_RATE = 0.35
492     CROSSOVER_RATE = 0.85
493 """
494
495 # #============================Solution Quality============================
      - TSPLIB's Optimal = 44301
496 # # Nearest Neighbour - 46678.15
497 # nn_pr107_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("pr107.tsp
        ")[1]
498 # print(nn_pr107_solution)
499
500 # # Insertion Heuristic w/ Convex Hull - 45730.01
```

```
# ihcv_pr107_solution = ihcv.run_ihcv_tsp_lib("pr107.tsp")[1]
# print(ihcv_pr107_solution)

## Integer Programming (1 hour) - 54606.7574
# ip_pr107_solution = mtz.run_mtz_tsp_lib("pr107.tsp")[1]
# print(ip_pr107_solution)

## Nearest Neighbour --> Two Opt - 44767.07
# nn_two_opt_pr107_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib("
    pr107.tsp")[1]
# print(nn_two_opt_pr107_solution)

## IHCV --> Two Opt - 45533.19
# ihcv_two_opt_pr107_solution = two_opt.run_two_opt_ihcv_tsp_lib("pr107.tsp")[1]
# print(ihcv_two_opt_pr107_solution)

## NN --> LNS - 44436.24
# nn_lns_pr107_solution = lns.run_lns_nearest_neighbour_tsp_lib("pr107.tsp")[1]
# print(nn_lns_pr107_solution)

## IHCV --> LNS - 44481.17
# ihcv_lns_pr107_solution = lns.run_lns_ihcv_tsp_lib("pr107.tsp")[1]
# print(ihcv_lns_pr107_solution)

## NN --> Two Opt --> LNS - 44436.24
# nn_two_opt_lns_pr107_solution = lns.run_lns_two_opt_nearest_neighbour_tsp_lib
    ("pr107.tsp")[1]
# print(nn_two_opt_lns_pr107_solution)

## IHCV --> Two Opt --> LNS - 44301.68
# ihcv_two_opt_lns_pr107_solution = lns.run_lns_two_opt_ihcv_tsp_lib("pr107.tsp
    ")[1]
# print(ihcv_two_opt_lns_pr107_solution)

## Genetic Algorithm - 44436.24
# ga_pr107_solution = ga.run_ga_tsp_lib("pr107.tsp")[1]
# print(ga_pr107_solution)

"""
GA PARAMS USED:
    POPULATION_SIZE = 400
    GENERATIONS = 200
```

```
540     MUTATION_RATE = 0.35
541     CROSSOVER_RATE = 0.85
542 """
543
544
545 ##==============================Dataset 6 - pr136 - 136 Nodes - Runtime, CPU
        Time==============================
546 ## Nearest Neighbour - 0.5670787572860718s, 0.2090824999999999s (10 runs)
547 # nn_pr136_runtime = compute_average_cpu_time(nearest_neighbour.
        run_nearest_neighbour_tsp_lib, ("pr136.tsp",), 10)
548 # print(nn_pr136_runtime)
549
550 ## Insertion Heuristic w/ Convex Hull - 0.6169691801071167s,
        0.34389060000000005s (10 runs)
551 # ihcv_pr136_runtime = compute_average_cpu_time(ihcv.run_ihcv_tsp_lib, ("pr136.
        tsp",), 10)
552 # print(ihcv_pr136_runtime)
553
554 ## Integer Programming - 1 hour execution
555
556 ## Nearest Neighbour --> Two Opt - 1.1399709383646648s, 0.38065879999999996s
        (10 runs)
557 # nn_two_opt_pr136_runtime = compute_average_cpu_time(two_opt.
        run_two_opt_nearest_neighbour_tsp_lib, ("pr136.tsp",), 10)
558 # print(nn_two_opt_pr136_runtime)
559
560 ## IHCV --> Two Opt - 0.8790188789367676s, 0.45419820000000016s (10 runs)
561 # ihcv_two_opt_pr136_runtime = compute_average_cpu_time(two_opt.
        run_two_opt_ihcv_tsp_lib, ("pr136.tsp",), 10)
562 # print(ihcv_two_opt_pr136_runtime)
563
564 ## NN --> LNS - 20.448569536209106s, 18.92006533333333s (10 runs)
565 # nn_lns_pr136_runtime = compute_average_cpu_time(lns.
        run_lns_nearest_neighbour_tsp_lib, ("pr136.tsp",), 10)
566 # print(nn_lns_pr136_runtime)
567
568 ## IHCV --> LNS - 15.488086581230164s, 12.037884333333333s (10 runs)
569 # ihcv_lns_pr136_runtime = compute_average_cpu_time(lns.run_lns_ihcv_tsp_lib, ("
        pr136.tsp",), 10)
570 # print(ihcv_lns_pr136_runtime)
571
572 ## NN --> Two Opt --> LNS - 19.903863668441772s, 11.2766025s (10 runs)
```

```
573 # nn_two_opt_lns_pr136_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_nearest_neighbour_tsp_lib, ("pr136.tsp",), 10)
574 # print(nn_two_opt_lns_pr136_runtime)
575
576 # # IHCV --> Two Opt --> LNS - 18.60973048210144s, 13.697148s (10 runs)
577 # ihcv_two_opt_lns_pr136_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_ihcv_tsp_lib, ("pr136.tsp",), 10)
578 # print(ihcv_two_opt_lns_pr136_runtime)
579
580 # # Genetic Algorithm - 749.3404741287231s, 721.638777333333333s (10 runs)
581 # ga_pr136_runtime = compute_average_cpu_time(ga.run_ga_tsp_lib, ("pr136.tsp",),
        10)
582 # print(ga_pr136_runtime)
583
584 """
585 GA PARAMS USED:
586     POPULATION_SIZE = 750
587     GENERATIONS = 500
588     MUTATION_RATE = 0.35
589     CROSSOVER_RATE = 0.65
590 """
591
592 # #================================Solution Quality================================
      - TSPLIB's Optimal = 96772
593 # # Nearest Neighbour - 120777.86
594 # nn_pr136_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("pr136.tsp
        ")[1]
595 # print(nn_pr136_solution)
596
597 # # Insertion Heuristic w/ Convex Hull - 102695.68
598 # ihcv_pr136_solution = ihcv.run_ihcv_tsp_lib("pr136.tsp")[1]
599 # print(ihcv_pr136_solution)
600
601 # # Integer Programming (1 hour) - 97389.78
602 # ip_pr136_solution = mtz.run_mtz_tsp_lib("pr136.tsp")[1]
603 # print(ip_pr136_solution)
604
605 # # Nearest Neighbour --> Two Opt - 105114.41
606 # nn_two_opt_pr136_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib("
      pr136.tsp")[1]
607 # print(nn_two_opt_pr136_solution)
608
```

```
609 # # IHCV --> Two Opt - 99777.24
610 # ihcv_two_opt_pr136_solution = two_opt.run_two_opt_ihcv_tsp_lib("pr136.tsp")[1]
611 # print(ihcv_two_opt_pr136_solution)
612
613 # # NN --> LNS - 100906.82
614 # nn_lns_pr136_solution = lns.run_lns_nearest_neighbour_tsp_lib("pr136.tsp")[1]
615 # print(nn_lns_pr136_solution)
616
617 # # IHCV --> LNS - 98357.67
618 # ihcv_lns_pr136_solution = lns.run_lns_ihcv_tsp_lib("pr136.tsp")[1]
619 # print(ihcv_lns_pr136_solution)
620
621 # # NN --> Two Opt --> LNS - 100787.66
622 # nn_two_opt_lns_pr136_solution = lns.run_lns_two_opt_nearest_neighbour_tsp_lib
        ("pr136.tsp")[1]
623 # print(nn_two_opt_lns_pr136_solution)
624
625 # # IHCV --> Two Opt --> LNS - 97319.9
626 # ihcv_two_opt_lns_pr136_solution = lns.run_lns_two_opt_ihcv_tsp_lib("pr136.tsp
        ")[1]
627 # print(ihcv_two_opt_lns_pr136_solution)
628
629 # # Genetic Algorithm - 96860.88
630 # ga_pr136_solution = ga.run_ga_tsp_lib("pr136.tsp")[1]
631 # print(ga_pr136_solution)
632
633 """
634 GA PARAMS USED:
635     POPULATION_SIZE = 750
636     GENERATIONS = 500
637     MUTATION_RATE = 0.35
638     CROSSOVER_RATE = 0.65
639 """
640
641
642 # #==============================Dataset 7 - tsp225 - 225 Nodes - Runtime, CPU
        Time==============================
643 # # Nearest Neighbour - 0.6898097515106201s, 0.32991720000000013s (10 runs)
644 # nn_tsp225_runtime = compute_average_cpu_time(nearest_neighbour.
        run_nearest_neighbour_tsp_lib, ("tsp225.tsp",), 10)
645 # print(nn_tsp225_runtime)
646
```

```
647 # # Insertion Heuristic w/ Convex Hull - 1.1035214662551885, 0.8435991999999999s
        (10 runs)
648 # ihcv_tsp225_runtime = compute_average_cpu_time(ihcv.run_ihcv_tsp_lib, ("tsp225
        .tsp",), 10)
649 # print(ihcv_tsp225_runtime)
650
651 # # Integer Programming - 1 hour execution
652
653 # # Nearest Neighbour --> Two Opt - 1.5792440176010132s, 0.8629096s (10 runs)
654 # nn_two_opt_tsp225_runtime = compute_average_cpu_time(two_opt.
        run_two_opt_nearest_neighbour_tsp_lib, ("tsp225.tsp",), 10)
655 # print(nn_two_opt_tsp225_runtime)
656
657 # # IHCV --> Two Opt - 1.6633020639419556s, 1.1295614999999999s (10 runs)
658 # ihcv_two_opt_tsp225_runtime = compute_average_cpu_time(two_opt.
        run_two_opt_ihcv_tsp_lib, ("tsp225.tsp",), 10)
659 # print(ihcv_two_opt_tsp225_runtime)
660
661 # # NN --> LNS - 46.42879557609558s, 36.527747s (10 runs)
662 # nn_lns_tsp225_runtime = compute_average_cpu_time(lns.
        run_lns_nearest_neighbour_tsp_lib, ("tsp225.tsp",), 10)
663 # print(nn_lns_tsp225_runtime)
664
665 # # IHCV --> LNS - 36.334240078926086s, 34.378658s (10 runs)
666 # ihcv_lns_tsp225_runtime = compute_average_cpu_time(lns.run_lns_ihcv_tsp_lib,
        ("tsp225.tsp",), 10)
667 # print(ihcv_lns_tsp225_runtime)
668
669 # # NN --> Two Opt --> LNS - 39.056431531906128s, 29.161218999999996s (10 runs)
670 # nn_two_opt_lns_tsp225_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_nearest_neighbour_tsp_lib, ("tsp225.tsp",), 10)
671 # print(nn_two_opt_lns_tsp225_runtime)
672
673 # # IHCV --> Two Opt --> LNS - 48.611959338188175, 37.368492s  (10 runs)
674 # ihcv_two_opt_lns_tsp225_runtime = compute_average_cpu_time(lns.
        run_lns_two_opt_ihcv_tsp_lib, ("tsp225.tsp",), 10)
675 # print(ihcv_two_opt_lns_tsp225_runtime)
676
677 # # Genetic Algorithm - 2904.7282733239018s, 2332.163935s (10 runs)
678 # ga_tsp225_runtime = compute_average_cpu_time(ga.run_ga_tsp_lib, ("tsp225.tsp
        ",), 10)
679 # print(ga_tsp225_runtime)
```

```
680
681 """
682 GA PARAMS USED:
683     POPULATION_SIZE = 750
684     GENERATIONS = 500
685     MUTATION_RATE = 0.35
686     CROSSOVER_RATE = 0.65
687 """
688
689 ##==========================Solution Quality==========================
      - TSPLIB's Optimal = 3916
690 ## Nearest Neighbour - 4829.0
691 # nn_tsp225_solution = nearest_neighbour.run_nearest_neighbour_tsp_lib("tsp225.
      tsp")[1]
692 # print(nn_tsp225_solution)
693
694 ## Insertion Heuristic w/ Convex Hull - 4442.27
695 # ihcv_tsp225_solution = ihcv.run_ihcv_tsp_lib("tsp225.tsp")[1]
696 # print(ihcv_tsp225_solution)
697
698 ## Integer Programming (1 hour) - 4248.42
699 # ip_tsp225_solution = mtz.run_mtz_tsp_lib("tsp225.tsp")[1]
700 # print(ip_tsp225_solution)
701
702 ## Nearest Neighbour --> Two Opt - 4123.82
703 # nn_two_opt_tsp225_solution = two_opt.run_two_opt_nearest_neighbour_tsp_lib("
      tsp225.tsp")[1]
704 # print(nn_two_opt_tsp225_solution)
705
706 ## IHCV --> Two Opt - 4329.24
707 # ihcv_two_opt_tsp225_solution = two_opt.run_two_opt_ihcv_tsp_lib("tsp225.tsp")
      [1]
708 # print(ihcv_two_opt_tsp225_solution)
709
710 ## NN --> LNS - 4166.27
711 # nn_lns_tsp225_solution = lns.run_lns_nearest_neighbour_tsp_lib("tsp225.tsp")
      [1]
712 # print(nn_lns_tsp225_solution)
713
714 ## IHCV --> LNS - 3989.86
715 # ihcv_lns_tsp225_solution = lns.run_lns_ihcv_tsp_lib("tsp225.tsp")[1]
716 # print(ihcv_lns_tsp225_solution)
```

```
717
718 # # NN --> Two Opt --> LNS - 3946.93
719 # nn_two_opt_lns_tsp225_solution = lns.run_lns_two_opt_nearest_neighbour_tsp_lib
      ("tsp225.tsp")[1]
720 # print(nn_two_opt_lns_tsp225_solution)
721
722 # # IHCV --> Two Opt --> LNS - 4000.3
723 # ihcv_two_opt_lns_tsp225_solution = lns.run_lns_two_opt_ihcv_tsp_lib("tsp225.
      tsp")[1]
724 # print(ihcv_two_opt_lns_tsp225_solution)
725
726 # # Genetic Algorithm - 3895.66
727 # ga_tsp225_solution = ga.run_ga_tsp_lib("tsp225.tsp")[1]
728 # print(ga_tsp225_solution)
729
730 """
731 GA PARAMS USED:
732     POPULATION_SIZE = 750
733     GENERATIONS = 500
734     MUTATION_RATE = 0.35
735     CROSSOVER_RATE = 0.65
736 """
737
738 # #==========Plot Runtime Graph - all Algorithms
      ==========
739 # nodes = [0, 7, 16, 52, 76, 107, 136, 225]
740 # algorithms = {
741 #     'Nearest Neighbour': {'data': [0, 0.2912076759338379, 0.3927781581878662,
      0.4872169017791748, 0.5912003755569458, 0.5175444841384887,
      0.5670787572860718, 0.6898097515106201], 'color': 'blue'},
742 #     'Insertion Heuristic w/ Convex Hull': {'data': [0, 0.31925201416015625,
      0.4028491497039795, 0.5403737020492554, 0.8179960489273072,
      0.6871454000473023, 0.6169691801071167, 1.103521466255188], 'color': 'red'},
743 #     'Integer Programming': {'data': [0, 0.48547115325927737,
      1.9439756274223328, 2.188145935535431, 238.9238269329071, 3600, 3600, 3600],
       'color': 'green'},
744 #     'Nearest Neighbour --> Two Opt': {'data': [0, 0.3369459629058838,
      0.4849355220794678, 0.5839934539794921, 0.6298728704452514,
      0.8510305881500244, 1.1399709383646648, 1.5792440176010132], 'color': 'lime
      '},
745 #     'IHCV --> Two Opt': {'data': [0, 0.398201584815979, 0.4879587411880493,
      0.5570610189437866, 0.6108887052536011, 0.7930902640024821,
```

```
      0.8790188789367676, 1.6633020639419556], 'color': 'black'},
746 #     'NN —> LNS': {'data': [0, 0.593600082397461, 0.7752989768981934,
      2.462945890426636, 4.3152546882629395, 6.857412974039714,
      20.448569536209106, 46.42879557609558], 'color': 'purple'},
747 #     'IHCV —> LNS': {'data': [0, 0.6017665624618531, 0.7552976131439209,
      2.0279680490493774, 5.947743972142537, 8.85807736714681, 15.488086581230164,
       36.334240078926086], 'color': 'magenta'},
748 #     'NN —> Two Opt —> LNS': {'data': [0, 0.6001407957077026,
      0.8133480072021484, 2.0701633214950563, 4.365378888448079,
      7.071200450261434, 19.903863668441772, 39.056431531906128], 'color': 'olive
      '},
749 #     'IHCV —> Two Opt —> LNS': {'data': [0, 0.6019640445709229,
      0.8137904930114746, 2.3267541885375977, 5.3695143063863116,
      8.895966529846191, 18.60973048210144, 48.61195933818817], 'color': 'aqua'},
750 #     # 'Genetic Algorithm': {'data': [0, 0.46117939949035647,
      2.077482557296753, 11.650958061218262, 51.395593881607056,
      94.05615496635437, 749.3404741287231, 2904.7282733239018], 'color': 'orange
      '},
751 # }
752
753 # def plot_runtime_comparison(nodes, algorithms_data, title, all_algorithms):
754 #     fig = go.Figure()
755 #     for algorithm_name, details in algorithms_data.items():
756 #         if algorithm_name in all_algorithms:
757 #             fig.add_trace(go.Scatter(x=nodes,
758 #                                      y=details['data'],
759 #                                      mode='lines, markers',
760 #                                      name=algorithm_name,
761 #                                      line=dict(color=details['color'])))
762
763 #     fig.update_layout(title=title,
764 #                       xaxis_title='Number of Nodes',
765 #                       yaxis_title='Runtime',
766 #                       legend_title='Algorithm: ')
767
768 #     fig.show()
769
770
771 # plot_runtime_comparison(nodes, algorithms, 'Algorithm Runtime Comparison - All
      Algorithms', algorithms.keys())
772 # plot_runtime_comparison(nodes, algorithms, 'Algorithm Runtime Comparison -
      Excluding Integer Programming', ['Nearest Neighbour', 'Insertion Heuristic w
```

```
    / Convex Hull', 'Nearest Neighbour --> Two Opt', 'IHCV --> Two Opt', 'NN -->
     LNS', 'IHCV --> LNS', 'NN --> Two Opt --> LNS', 'IHCV --> Two Opt', '
    Genetic Algorithm'])
# plot_runtime_comparison(nodes, algorithms, 'Algorithm Runtime Comparison -
    Excluding Integer Programming', ['Nearest Neighbour', 'Insertion Heuristic w
    / Convex Hull', 'Nearest Neighbour --> Two Opt', 'IHCV --> Two Opt', 'NN -->
     LNS', 'IHCV --> LNS', 'NN --> Two Opt --> LNS', 'IHCV --> Two Opt --> LNS
    '])

### ==========Plot CPU Time Graph - all Algorithms ==========
nodes = [0, 7, 16, 52, 76, 107, 136, 225]
algorithms = {
    'Nearest Neighbour': {'data': [0, 0.06434819999999997, 0.06856299999999979,
    0.1028905, 0.13564180000000023, 0.17077940000000016, 0.2090824999999999,
    0.32991720000000013], 'color': 'blue'},
    'Insertion Heuristic w/ Convex Hull': {'data': [0, 0.05864439999999984,
    0.06983930000000002, 0.11625139999999998, 0.15631350000000008,
    0.2842870999999999, 0.34389060000000005, 0.8435991999999999], 'color': 'red'
    },
    'Integer Programming': {'data': [0, 0.08146700000000004, 0.0933790000000001,
     0.3090054999999998, 0.6113710000000001, 3600, 3600, 3600], 'color': 'green'
    },
    'Nearest Neighbour --> Two Opt': {'data': [0, 0.058959400000000176,
    0.07274639999999995, 0.12024610000000022, 0.16514700000000007,
    0.24584100000000006, 0.38065879999999996, 0.8629096], 'color': 'lime'},
    'IHCV --> Two Opt': {'data': [0, 0.05850529999999994, 0.06585739999999998,
    0.1256982999999999, 0.15282080000000012, 0.2809012, 0.45419820000000016,
    1.1295614999999999], 'color': 'black'},
    'NN --> LNS': {'data': [0, 0.25243659999999996, 0.42592280000000005,
    2.3432852, 3.6238146, 6.4553855, 18.92006533333333, 36.527747], 'color': '
    purple'},
    'IHCV --> LNS': {'data': [0, 0.24799320000000016, 0.41674609999999995,
    1.7926807999999999, 4.7678232000000005, 7.682524599999998,
    12.037884333333333, 34.378658], 'color': 'magenta'},
    'NN --> Two Opt --> LNS': {'data': [0, 0.2650253, 0.4447246999999999,
    1.8235629999999996, 3.8795434, 6.4950489000000005, 11.2766025,
    29.161218999999996], 'color': 'olive'},
    'IHCV --> Two Opt --> LNS': {'data': [0, 0.2553482999999999,
    0.4334836999999997, 1.7669904000000003, 4.6750652, 8.743738600000002,
    13.697148, 37.368492], 'color': 'aqua'},
    # 'Genetic Algorithm': {'data': [0, 0.16895220000000002, 1.8290260999999997,
```

```
        6.796694400000002, 45.3103905, 74.549643, 721.638777333333333,
        2332.163935], 'color': 'orange'},
}

def plot_cpu_time_comparison(nodes, algorithms_data, title, all_algorithms):
    fig = go.Figure()
    for algorithm_name, details in algorithms_data.items():
        if algorithm_name in all_algorithms:
            fig.add_trace(go.Scatter(x=nodes,
                                     y=details['data'],
                                     mode='lines, markers',
                                     name=algorithm_name,
                                     line=dict(color=details['color'])))

    fig.update_layout(title=title,
                      xaxis_title='Number of Nodes',
                      yaxis_title='CPU Time',
                      legend_title='Algorithm: ')

    fig.show()


plot_cpu_time_comparison(nodes, algorithms, 'Algorithm CPU Time Comparison - All
    Algorithms', algorithms.keys())
# plot_cpu_time_comparison(nodes, algorithms, 'Algorithm CPU Time Comparison -
    Excluding Integer Programming', ['Nearest Neighbour', 'Insertion Heuristic w
    / Convex Hull', 'Nearest Neighbour --> Two Opt', 'IHCV --> Two Opt', 'NN -->
    LNS', 'IHCV --> LNS', 'NN --> Two Opt --> LNS', 'IHCV --> Two Opt --> LNS
    '])



## ==========Plot Solution Quality Graph - all Algorithms
    ==========
# nodes = [0, 7, 16, 52, 76, 107, 136, 225]
# algorithms = {
#     'Nearest Neighbour': {'data': [0, 26.79, 41.5, 19.08, 41.89, 5.36, 24.81,
    23.31], 'color': 'blue'},
#     'Insertion Heuristic w/ Convex Hull': {'data': [0, 0, 1.5, 7.48, 6.15,
    3.22, 6.12, 14.44], 'color': 'red'},
#     'Integer Programming': {'data': [0, 0, 0, 0.03, 0, 23.26, 0.64, 8.49], '
```

```
    color': 'green'},
820 #     'Nearest Neighbour --> Two Opt': {'data': [0, 0, 2.67, 6.83, 5.06, 1.05,
    8.62, 5.31], 'color': 'lime'},
821 #     'IHCV --> Two Opt': {'data': [0, 0, 0, 7.06, 4.58, 2.78, 3.11, 10.55], '
    color': 'black'},
822 #     'NN --> LNS': {'data': [0, 0, 0, 4.90, 2.31, 0.31, 4.27, 6.39], 'color': '
    purple'},
823 #     'IHCV --> LNS': {'data': [0, 0, 0, 4.90, 0.82, 0.41, 1.64, 1.89], 'color':
     'magenta'},
824 #     'NN --> Two Opt --> LNS': {'data': [0, 0, 0, 4.90, 0.84, 0.31, 4.15,
    0.79], 'color': 'olive'},
825 #     'IHCV --> Two Opt --> LNS': {'data': [0, 0, 0, 4.58, 0.82, 0, 0.57, 2.15],
     'color': 'aqua'},
826 #     # 'Genetic Algorithm': {'data': [0, 0, 0, 0.03, 0, 0.31, 0.09, -0.52], '
    color': 'orange'},
827 # }
828
829 # def plot_runtime_comparison(nodes, algorithms_data, title, all_algorithms):
830 #     fig = go.Figure()
831 #     for algorithm_name, details in algorithms_data.items():
832 #         if algorithm_name in all_algorithms:
833 #             fig.add_trace(go.Scatter(x=nodes,
834 #                                      y=details['data'],
835 #                                      mode='lines, markers',
836 #                                      name=algorithm_name,
837 #                                      line=dict(color=details['color'])))
838
839 #     fig.update_layout(title=title,
840 #                       xaxis_title='Number of Nodes',
841 #                       yaxis_title='Solution Quality - Percentage Difference
    from Global Optimal (%)',
842 #                       legend_title='Algorithm: ')
843
844 #     fig.show()
845
846
847 # plot_runtime_comparison(nodes, algorithms, 'Algorithm Runtime Comparison - All
     Algorithms', algorithms.keys())
848 # plot_runtime_comparison(nodes, algorithms, 'Algorithm Solution Quality
    Comparison - Excluding Integer Programming', ['Nearest Neighbour', '
    Insertion Heuristic w/ Convex Hull', 'Nearest Neighbour --> Two Opt', 'IHCV
    --> Two Opt', 'NN --> LNS', 'IHCV --> LNS', 'NN --> Two Opt --> LNS', 'IHCV
```

```
    —> Two Opt —> LNS'])
849
850 # print(compute_percentage_difference(3895.66, 3916))
851
852
853
854
855 # #==========Plot Runtime vs CPU Time (Bubble Graph - bigger bubble = more nodes in the input dataset)==========
856 # nodes = [7, 16, 52, 76, 107, 136, 225]
857 # algorithms_runtime = {
858 #     'Nearest Neighbour': {'data': [0.4112076759338379, 0.3927781581878662,
    0.4872169017791748, 0.5912003755569458, 0.5175444841384887,
    0.5670787572860718, 0.6898097515106201], 'color': 'blue'},
859 #     'Insertion Heuristic w/ Convex Hull': {'data': [0.31925201416015625,
    0.4028491497039795, 0.5403737020492554, 0.8179960489273072,
    0.6871454000473023, 0.6169691801071167, 1.103521466255188], 'color': 'red'},
860 #     # 'Integer Programming': {'data': [0.48547115325927737,
    1.9439756274223328, 2.188145935535431, 238.9238269329071, 3600, 3600, 3600],
     'color': 'green'},
861 #     'Nearest Neighbour —> Two Opt': {'data': [0.3369459629058838,
    0.4849355220794678, 0.5839934539794921, 0.6298728704452514,
    0.8510305881500244, 1.1399709383646648, 1.5792440176010132], 'color': 'lime
    '},
862 #     'IHCV —> Two Opt': {'data': [0.398201584815979, 0.4879587411880493,
    0.5570610189437866, 0.6108887052536011, 0.7930902640024821,
    0.8790188789367676, 1.6633020639419556], 'color': 'black'},
863 #     'NN —> LNS': {'data': [0.593600082397461, 0.7752989768981934,
    2.462945890426636, 4.3152546882629395, 6.857412974039714,
    20.448569536209106, 46.42879557609558], 'color': 'purple'},
864 #     'IHCV —> LNS': {'data': [0.6017665624618531, 0.7552976131439209,
    2.0279680490493774, 5.947743972142537, 8.85807736714681, 15.488086581230164,
     36.334240078926086], 'color': 'magenta'},
865 #     'NN —> Two Opt —> LNS': {'data': [0.6001407957077026,
    0.8133480072021484, 2.0701633214950563, 4.365378888448079,
    7.071200450261434, 19.903863668441772, 39.056431531906128], 'color': 'olive
    '},
866 #     'IHCV —> Two Opt —> LNS': {'data': [0.6019640445709229,
    0.8137904930114746, 2.3267541885375977, 5.3695143063863116,
    8.895966529846191, 18.60973048210144, 48.61195933818817], 'color': 'aqua'},
867 #     # 'Genetic Algorithm': {'data': [0.46117939949035647, 2.077482557296753,
    11.650958061218262, 51.395593881607056, 94.05615496635437,
```

```
        749.3404741287231, 2904.7282733239018], 'color': 'orange'},
868 # }
869
870 # algorithms_cpu_time = {
871 #     'Nearest Neighbour': {'data': [0.06434819999999997, 0.06856299999999979,
        0.1028905, 0.13564180000000023, 0.17077940000000016, 0.2090824999999999,
        0.32991720000000013], 'color': 'blue'},
872 #     'Insertion Heuristic w/ Convex Hull': {'data': [0.05864439999999984,
        0.06983930000000002, 0.11625139999999998, 0.15631350000000008,
        0.2842870999999999, 0.34389060000000005, 0.8435991999999999], 'color': 'red
        '},
873 #     # 'Integer Programming': {'data': [0.08146700000000004,
        0.0933790000000001, 0.3090054999999998, 0.6113710000000001, 3600, 3600,
        3600], 'color': 'green'},
874 #     'Nearest Neighbour --> Two Opt': {'data': [0.058959400000000176,
        0.07274639999999995, 0.12024610000000022, 0.16514700000000007,
        0.24584100000000006, 0.38065879999999996, 0.8629096], 'color': 'lime'},
875 #     'IHCV --> Two Opt': {'data': [0.05850529999999994, 0.06585739999999998,
        0.1256982999999999, 0.15282080000000012, 0.2809012, 0.45419820000000016,
        1.1295614999999999], 'color': 'black'},
876 #     'NN --> LNS': {'data': [0.25243659999999996, 0.42592280000000005,
        2.3432852, 3.6238146, 6.4553855, 18.92006533333333, 36.527747], 'color': '
        purple'},
877 #     'IHCV --> LNS': {'data': [0.24799320000000016, 0.41674609999999995,
        1.7926807999999999, 4.7678232000000005, 7.682524599999998,
        12.037884333333333, 34.378658], 'color': 'magenta'},
878 #     'NN --> Two Opt --> LNS': {'data': [0.2650253, 0.4447246999999999,
        1.8235629999999996, 3.8795434, 6.4950489000000005, 11.2766025,
        29.161218999999996], 'color': 'olive'},
879 #     'IHCV --> Two Opt --> LNS': {'data': [0.2553482999999999,
        0.4334836999999997, 1.7669904000000003, 4.6750652, 8.743738600000002,
        13.697148, 37.368492], 'color': 'aqua'},
880 #     # 'Genetic Algorithm': {'data': [0.16895220000000002, 1.8290260999999997,
        6.796694400000002, 45.3103905, 74.549643, 721.638777333333333, 2332.163935],
         'color': 'orange'},
881 # }
882
883 # def plot_performance_comparison(nodes, algorithms_runtime, algorithms_cpu_time
        , title):
884 #     fig = go.Figure()
885 #     for algorithm_name, algorithm_runtime_details in algorithms_runtime.items
        ():
```

```
886 #         if algorithm_name in algorithms_cpu_time:
887 #             algorithm_quality_details = algorithms_cpu_time[algorithm_name]
888 #             sizes = [10 + 2*np.log(node)**2 for node in nodes]
889 
890 #             runtime_values = algorithm_runtime_details['data']
891 #             quality_values = algorithm_quality_details['data']
892 
893 #             hover_text = [ f"Algorithm Name: {algorithm_name}<br>No. Nodes: {
    node}<br>Algorithm Runtime: {np.round(runtime, 2)}s<br>Solution Quality: {
    quality}%" for (node, runtime, quality) in zip(nodes, runtime_values,
    quality_values) ]
894 
895 #             fig.add_trace(go.Scatter(x=runtime_values,
896 #                                      y=quality_values,
897 #                                      mode='markers',
898 #                                      marker=dict(size=sizes,
899 #                                                  color=
    algorithm_quality_details['color'],
900 #                                                  opacity=0.65),
901 #                                      text=hover_text, hoverinfo="text",
902 #                                      name=algorithm_name))
903 
904 #             fig.update_layout(
905 #                 title=title,
906 #                 xaxis_title='Runtime (s)',
907 #                 yaxis_title='CPU Time (s)',
908 #                 legend_title='Algorithm: '
909 #             )
910 
911 #     fig.show()
912 
913 # plot_performance_comparison(nodes, algorithms_runtime, algorithms_cpu_time, '
    Holistic Algorithm Performance Comparison: Runtime vs CPU Time')
914 # #==========================Plot Runtime vs CPU Time (Bubble Graph - bigger
     bubble = more nodes in the input dataset)==========================
915 # nodes = [7, 16, 52, 76, 107, 136, 225]
916 # algorithms_quality = {
917 #     'Nearest Neighbour': {'data': [26.79, 41.5, 19.08, 41.89, 5.36, 24.81,
    23.31], 'color': 'blue'},
918 #     'Insertion Heuristic w/ Convex Hull': {'data': [0, 1.5, 7.48, 6.15, 3.22,
    6.12, 14.44], 'color': 'red'},
919 #     # 'Integer Programming': {'data': [0, 0, 0.03, 0, 23.26, 0.64, 8.49], '
```

```
    color': 'green'},
#     'Nearest Neighbour --> Two Opt': {'data': [0, 2.67, 6.83, 5.06, 1.05,
    8.62, 5.31], 'color': 'lime'},
#     'IHCV --> Two Opt': {'data': [0, 0, 7.06, 4.58, 2.78, 3.11, 10.55], 'color
    ': 'black'},
#     'NN --> LNS': {'data': [0, 0, 4.90, 2.31, 0.31, 4.27, 6.39], 'color': '
    purple'},
#     'IHCV --> LNS': {'data': [0, 0, 4.90, 0.82, 0.41, 1.64, 1.89], 'color': '
    magenta'},
#     'NN --> Two Opt --> LNS': {'data': [0, 0, 4.90, 0.84, 0.31, 4.15, 0.79], '
    color': 'olive'},
#     'IHCV --> Two Opt --> LNS': {'data': [0, 0, 4.58, 0.82, 0, 0.57, 2.15], '
    color': 'aqua'},
#     # 'Genetic Algorithm': {'data': [0, 0, 0.03, 0, 0.31, 0.09, -0.52], 'color
    ': 'orange'},
# }

# algorithms_cpu_time = {
#     'Nearest Neighbour': {'data': [0.06434819999999997, 0.06856299999999979,
    0.1028905, 0.13564180000000023, 0.17077940000000016, 0.2090824999999999,
    0.32991720000000013], 'color': 'blue'},
#     'Insertion Heuristic w/ Convex Hull': {'data': [0.05864439999999984,
    0.06983930000000002, 0.11625139999999998, 0.15631350000000008,
    0.2842870999999999, 0.34389060000000005, 0.8435991999999999], 'color': 'red
    '},
#     # 'Integer Programming': {'data': [0.08146700000000004,
    0.0933790000000001, 0.3090054999999998, 0.6113710000000001, 3600, 3600,
    3600], 'color': 'green'},
#     'Nearest Neighbour --> Two Opt': {'data': [0.058959400000000176,
    0.07274639999999995, 0.12024610000000022, 0.16514700000000007,
    0.24584100000000006, 0.38065879999999996, 0.8629096], 'color': 'lime'},
#     'IHCV --> Two Opt': {'data': [0.05850529999999994, 0.06585739999999998,
    0.1256982999999999, 0.15282080000000012, 0.2809012, 0.45419820000000016,
    1.1295614999999999], 'color': 'black'},
#     'NN --> LNS': {'data': [0.25243659999999996, 0.42592280000000005,
    2.3432852, 3.6238146, 6.4553855, 18.92006533333333, 36.527747], 'color': '
    purple'},
#     'IHCV --> LNS': {'data': [0.24799320000000016, 0.41674609999999995,
    1.7926807999999999, 4.7678232000000005, 7.682524599999998,
    12.037884333333333, 34.378658], 'color': 'magenta'},
#     'NN --> Two Opt --> LNS': {'data': [0.2650253, 0.4447246999999999,
    1.8235629999999996, 3.8795434, 6.4950489000000005, 11.2766025,
```

```
    29.161218999999996], 'color': 'olive'},
#     'IHCV --> Two Opt --> LNS': {'data': [0.2553482999999999,
    0.4334836999999997, 1.7669904000000003, 4.6750652, 8.743738600000002,
    13.697148, 37.368492], 'color': 'aqua'},
#     # 'Genetic Algorithm': {'data': [0.16895220000000002, 1.8290260999999997,
    6.796694400000002, 45.3103905, 74.549643, 721.638777333333333, 2332.163935],
     'color': 'orange'},
# }

# def plot_performance_comparison(nodes, algorithms_quality, algorithms_cpu_time
    , title):
#     fig = go.Figure()
#     for algorithm_name, algorithm_runtime_details in algorithms_quality.items
    ():
#         if algorithm_name in algorithms_cpu_time:
#             algorithm_quality_details = algorithms_cpu_time[algorithm_name]
#             sizes = [10 + 2*np.log(node)**2 for node in nodes]

#             runtime_values = algorithm_runtime_details['data']
#             quality_values = algorithm_quality_details['data']

#             hover_text = [ f"Algorithm Name: {algorithm_name}<br>No. Nodes: {
    node}<br>Algorithm Runtime: {np.round(runtime, 2)}s<br>Solution Quality: {
    quality}%" for (node, runtime, quality) in zip(nodes, runtime_values,
    quality_values) ]

#             fig.add_trace(go.Scatter(x=runtime_values,
#                                      y=quality_values,
#                                      mode='markers',
#                                      marker=dict(size=sizes,
#                                                  color=
    algorithm_quality_details['color'],
#                                                  opacity=0.65),
#                                     text=hover_text, hoverinfo="text",
#                                     name=algorithm_name))

#             fig.update_layout(
#                 title=title,
#                 xaxis_title='Solution Quality from Optimal (%)',
#                 yaxis_title='CPU Time (s)',
#                 legend_title='Algorithm: '
#             )
```

```
969
970 #     fig.show()
971
972 # plot_performance_comparison(nodes, algorithms_quality, algorithms_cpu_time, '
      Holistic Algorithm Performance Comparison: Solution Quality vs CPU Time')
973
974
975
976 ##================================Plot Runtime vs Solution Quality Graph (Bubble
      Graph - bigger bubble = more nodes in the input dataset)
      ================================
977 # nodes = [7, 16, 52, 76, 107, 136, 225]
978 # algorithms_runtime = {
979 #     'Nearest Neighbour': {'data': [0.4112076759338379, 0.3927781581878662,
      0.4872169017791748, 0.5912003755569458, 0.5175444841384887,
      0.5670787572860718, 0.6898097515106201], 'color': 'blue'},
980 #     'Insertion Heuristic w/ Convex Hull': {'data': [0.31925201416015625,
      0.4028491497039795, 0.5403737020492554, 0.8179960489273072,
      0.6871454000473023, 0.6169691801071167, 1.103521466255188], 'color': 'red'},
981 #     # 'Integer Programming': {'data': [0.48547115325927737,
      1.9439756274223328, 2.188145935535431, 238.9238269329071, 3600, 3600, 3600],
       'color': 'green'},
982 #     'Nearest Neighbour --> Two Opt': {'data': [0.3369459629058838,
      0.4849355220794678, 0.5839934539794921, 0.6298728704452514,
      0.8510305881500244, 1.1399709383646648, 1.5792440176010132], 'color': 'lime
      '},
983 #     'IHCV --> Two Opt': {'data': [0.398201584815979, 0.4879587411880493,
      0.5570610189437866, 0.6108887052536011, 0.7930902640024821,
      0.8790188789367676, 1.6633020639419556], 'color': 'black'},
984 #     'NN --> LNS': {'data': [0.593600082397461, 0.7752989768981934,
      2.462945890426636, 4.3152546882629395, 6.857412974039714,
      20.448569536209106, 46.42879557609558], 'color': 'purple'},
985 #     'IHCV --> LNS': {'data': [0.6017665624618531, 0.7552976131439209,
      2.0279680490493774, 5.947743972142537, 8.85807736714681, 15.488086581230164,
       36.334240078926086], 'color': 'magenta'},
986 #     'NN --> Two Opt --> LNS': {'data': [0.6001407957077026,
      0.8133480072021484, 2.0701633214950563, 4.365378888448079,
      7.071200450261434, 19.903863668441772, 39.056431531906128], 'color': 'olive
      '},
987 #     'IHCV --> Two Opt --> LNS': {'data': [0.6019640445709229,
      0.8137904930114746, 2.3267541885375977, 5.3695143063863116,
      8.895966529846191, 18.60973048210144, 48.61195933818817], 'color': 'aqua'},
```

```
988 #     # 'Genetic Algorithm': {'data': [0.46117939949035647, 2.077482557296753,
      11.650958061218262, 51.395593881607056, 94.05615496635437,
      749.3404741287231, 2904.7282733239018], 'color': 'orange'},
989 # }
990
991 # algorithms_quality = {
992 #     'Nearest Neighbour': {'data': [26.79, 41.5, 19.08, 41.89, 5.36, 24.81,
      23.31], 'color': 'blue'},
993 #     'Insertion Heuristic w/ Convex Hull': {'data': [0, 1.5, 7.48, 6.15, 3.22,
      6.12, 14.44], 'color': 'red'},
994 #     # 'Integer Programming': {'data': [0, 0, 0.03, 0, 23.26, 0.64, 8.49], '
      color': 'green'},
995 #     'Nearest Neighbour --> Two Opt': {'data': [0, 2.67, 6.83, 5.06, 1.05,
      8.62, 5.31], 'color': 'lime'},
996 #     'IHCV --> Two Opt': {'data': [0, 0, 7.06, 4.58, 2.78, 3.11, 10.55], 'color
      ': 'black'},
997 #     'NN --> LNS': {'data': [0, 0, 4.90, 2.31, 0.31, 4.27, 6.39], 'color': '
      purple'},
998 #     'IHCV --> LNS': {'data': [0, 0, 4.90, 0.82, 0.41, 1.64, 1.89], 'color': '
      magenta'},
999 #     'NN --> Two Opt --> LNS': {'data': [0, 0, 4.90, 0.84, 0.31, 4.15, 0.79], '
      color': 'olive'},
1000 #     'IHCV --> Two Opt --> LNS': {'data': [0, 0, 4.58, 0.82, 0, 0.57, 2.15], '
      color': 'aqua'},
1001 #     # 'Genetic Algorithm': {'data': [0, 0, 0.03, 0, 0.31, 0.09, -0.52], 'color
      ': 'orange'},
1002 # }
1003
1004 # def plot_performance_comparison(nodes, algorithms_runtime, algorithms_quality,
       title):
1005 #     fig = go.Figure()
1006 #     for algorithm_name, algorithm_runtime_details in algorithms_runtime.items
      ():
1007 #         if algorithm_name in algorithms_quality:
1008 #             algorithm_quality_details = algorithms_quality[algorithm_name]
1009 #             sizes = [10 + 2*np.log(node)**2 for node in nodes]
1010
1011 #             runtime_values = algorithm_runtime_details['data']
1012 #             quality_values = algorithm_quality_details['data']
1013
1014 #             hover_text = [ f"Algorithm Name: {algorithm_name}<br>No. Nodes: {
      node}<br>Algorithm Runtime: {np.round(runtime, 2)}s<br>Solution Quality: {
```

```
        quality}%" for (node, runtime, quality) in zip(nodes, runtime_values,
        quality_values) ]
1015
1016 #             fig.add_trace(go.Scatter(x=runtime_values,
1017 #                                      y=quality_values,
1018 #                                      mode='markers',
1019 #                                      marker=dict(size=sizes,
1020 #                                                  color=
        algorithm_quality_details['color'],
1021 #                                                  opacity=0.65),
1022 #                                     text=hover_text, hoverinfo="text",
1023 #                                     name=algorithm_name))
1024
1025 #             fig.update_layout(
1026 #                 title=title,
1027 #                 xaxis_title='Runtime (s)',
1028 #                 yaxis_title='Solution Quality - Percentage Difference from
        Global Optimal (%)',
1029 #                 legend_title='Algorithm: '
1030 #             )
1031
1032 #     fig.show()
1033
1034 # plot_performance_comparison(nodes, algorithms_runtime, algorithms_quality, '
        Holistic Algorithm Performance Comparison: Runtime vs Solution Quality -
        Excluding Integer Programming')
```

Listing C.8: Source Code for *benchmarking_implementations.py*

C.9 TEST - TSP Utility Functions

```
1 import unittest, pytest, os, pandas as pd, numpy as np
2 from tsp_utility_functions import convert_tsp_lib_instance_to_spreadsheet,
      import_node_data, compute_distance_matrix , compute_route_distance
3 from Test_Inputs.mock_route_data import get_test_city_1_distance_matrix,
      get_test_city_1_node_index_mapping
4
5 # Helper Function - Converts decimal numbers like 1.0, 2.0, etc. to their
      regular integer form (1, 2, etc.), while leaving other decimals (e.g. 3.2)
      untouched
6 def downcast_to_int64(table):
```

```
    for column in table.columns:
        if table[column].dtype == float:
            table[column] = table[column].astype('int64')

    return table

class TestTSPUtilityFunctions(unittest.TestCase):
    def setUp(self):
        self.att48_name = "att48.tsp"

        self.test_city_1_file = "test_city_1.xlsx"

        self.nodes = pd.DataFrame({'Node': [1, 2, 3, 4, 5], 'X': [0, 4, 8, 7,
    3], 'Y': [0, 3, 0, 5, 8], 'Type': ['Start', 'Waypoint', 'Waypoint', '
    Waypoint', 'Waypoint']})
        self.n_nodes = len(self.nodes)

        file_path = "Test_Inputs/TSPLIB_Instances/att48.xlsx"
        if os.path.exists(file_path):
            os.remove(file_path)


        # Invalid TSP Instances Files:
        self.duplicate_columns_name = "DuplicateColumns.xlsx"
        self.empty_cells_name = "EmptyCells.xlsx"
        self.empty_row_name = "EmptyRow.xlsx"
        self.input_wrong_data_type_into_columns_name = "
    InputWrongDataTypeIntoColumns.xlsx"
        self.invalid_node_numbers_name = "InvalidNodeNumbers.xlsx"
        self.invalid_type_text_name = "InvalidTypeText.xlsx"
        self.misspelled_type_name = "MisspelledType.xlsx"
        self.multiple_issues_1_name = "MultipleIssues_1.xlsx"
        self.multiple_issues_2_name = "MultipleIssues_2.xlsx"
        self.multiple_nodes_with_same_node_number_name = "
    MultipleNodesWithSameNodeNumber.xlsx"
        self.multiple_start_nodes_name = "MultipleStartNodes.xlsx"
        self.node_numbers_not_sequential_name = "NodeNumbersNotSequential.xlsx"
        self.no_start_nodes_name = "NoStartNodes.xlsx"
        self.not_enough_nodes_name = "NotEnoughNodes.xlsx"
        self.rows_not_in_order_of_nodes_name = "RowsNotInOrderOfNodes.xlsx"
        self.start_node_is_not_1_name = "StartNodeIsNot1.xlsx"
        self.two_nodes_with_same_coordinates_name = "TwoNodesWithSameCoordinates
```

```
.xlsx"
45          self.valid_name_name = "Valid_Name.xlsx"
46          self.valid_name = "Valid.xlsx"
47          self.wrong_column_names_name = "WrongColumnNames.xlsx"
48          self.wrong_number_of_columns_name = "WrongNumberOfColumns.xlsx"
49
50      def tearDown(self):
51          file_path = "Test_Inputs/TSPLIB_Instances/att48.xlsx"
52          if os.path.exists(file_path):
53              os.remove(file_path)
54
55      def test_compute_distance_matrix(self):
56          expected_distance_matrix = np.array([
57              [0, np.sqrt(np.square(4-0) + np.square(3-0)), np.sqrt(np.square(8-0)
 + np.square(0-0)), np.sqrt(np.square(7-0) + np.square(5-0)), np.sqrt(np.
square(3-0) + np.square(8-0))],
58              [np.sqrt(np.square(4-0) + np.square(3-0)), 0, np.sqrt(np.square(8-4)
 + np.square(0-3)), np.sqrt(np.square(7-4) + np.square(5-3)), np.sqrt(np.
square(3-4) + np.square(8-3))],
59              [np.sqrt(np.square(8-0) + np.square(0-0)), np.sqrt(np.square(8-4) +
np.square(0-3)), 0, np.sqrt(np.square(7-8) + np.square(5-0)), np.sqrt(np.
square(3-8) + np.square(8-0))],
60              [np.sqrt(np.square(7-0) + np.square(5-0)), np.sqrt(np.square(7-4) +
np.square(5-3)), np.sqrt(np.square(7-8) + np.square(5-0)), 0, np.sqrt(np.
square(3-7) + np.square(8-5))],
61              [np.sqrt(np.square(3-0) + np.square(8-0)), np.sqrt(np.square(3-4) +
np.square(8-3)), np.sqrt(np.square(3-8) + np.square(8-0)), np.sqrt(np.square
(3-7) + np.square(8-5)), 0]
62          ])
63
64          generated_distance_matrix = compute_distance_matrix(self.nodes)
65
66          np.testing.assert_array_almost_equal(expected_distance_matrix,
generated_distance_matrix, decimal = 1, err_msg = "The computed distance
matrix is not the same as the expected distance matrix", verbose = True)
67
68      def test_compute_route_distance(self):
69          generated_route_distance = compute_route_distance([1, 3, 6, 4, 5, 7, 2,
1], get_test_city_1_distance_matrix(), get_test_city_1_node_index_mapping())
70          expected_route_distance = 23.63
71
72          self.assertAlmostEqual(np.round(generated_route_distance, 2),
```

```
        expected_route_distance)
73
74      def check_table_row_values(self, table_row, expected_values):
75          for i, expected in expected_values.items():
76              self.assertEqual(table_row[i], expected, f"{i} should have been {
        expected}, but was instead {table_row[i]}")
77
78      # Test 'convert_tsp_lib_instance_to_spreadsheet' function
79      def test_convert_tsp_lib_instance_to_spreadsheet(self):
80          spreadsheet_name = convert_tsp_lib_instance_to_spreadsheet(self.
        att48_name)
81          self.assertTrue(os.path.exists(f"Test_Inputs/TSPLIB_Instances/{
        spreadsheet_name}")) # Check if new .xlsx file exists in correct place
82
83          # Verify contents of the generated .xlsx file
84          generated_table = pd.read_excel(f"Test_Inputs/TSPLIB_Instances/{
        spreadsheet_name}")
85
86          self.check_table_row_values(generated_table.iloc[0].to_dict(), {"Node":
        1, "X": 6734, "Y": 1453, "Type": "Start"})
87          self.check_table_row_values(generated_table.iloc[47].to_dict(), {"Node":
         48, "X": 3023, "Y": 1942, "Type": "Waypoint"})
88          self.check_table_row_values(generated_table.iloc[12].to_dict(), {"Node":
         13, "X": 4706, "Y": 2674, "Type": "Waypoint"})
89
90          self.assertEqual(len(generated_table), 48, f"The generated table should
        have 48 rows, but instead has {len(generated_table)}")
91
92      # Test 'import_node_data' function
93      def test_import_node_data(self):
94          nodes = import_node_data(self.test_city_1_file)
95          expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
        2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
96                                         "Type": ["Start", "Waypoint", "Waypoint",
         "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
97          pd.testing.assert_frame_equal(nodes, expected_table)
98
99      # Invalidate Duplicate Columns in Input Data
100     def test_duplicate_columns(self):
101         with pytest.raises(ValueError) as e:
102             import_node_data(self.duplicate_columns_name, for_testing_purposes=
        True)
```

```
103
104     # Invalidate Empty Cells in Input Data
105     def test_empty_cells(self):
106         with pytest.raises(ValueError) as e:
107             import_node_data(self.empty_cells_name, for_testing_purposes=True)
108
109     # Invalidate/Correct Empty Row(s) in Input Data
110     def test_empty_rows(self):
111         nodes = import_node_data(self.empty_row_name, for_testing_purposes=True)
112         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6], "X": [1, 2,
    2, 10, 4, 5], "Y": [2, 5, 1, 5, 3, 7],
113                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint"]})
114
115         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
116
117     # Invalidate Wrong Data Type in Columns in Input Data
118     def test_input_wrong_data_type_into_columns(self):
119         with pytest.raises(ValueError) as e:
120             import_node_data(self.input_wrong_data_type_into_columns_name,
    for_testing_purposes=True)
121
122     # Invalidate Invalid Node Numbers in Input Data
123     def test_invalid_node_numbers(self):
124         with pytest.raises(ValueError) as e:
125             import_node_data(self.invalid_node_numbers_name,
    for_testing_purposes=True)
126
127     # Invalidate Invalid Type Text in Input Data
128     def test_invalid_type_text(self):
129         with pytest.raises(ValueError) as e:
130             import_node_data(self.invalid_type_text_name, for_testing_purposes=
    True)
131
132     # Invalidate/Correct Misspelled Type in Input Data
133     def test_misspelled_type(self):
134         nodes = import_node_data(self.misspelled_type_name, for_testing_purposes
    =True)
135         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
136                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
```

```
137         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
138 
139     # Invalidate/Correct TSP Instance (1) w/ Multiple Issues
140     def test_multiple_issues_1(self):
141         nodes = import_node_data(self.multiple_issues_1_name,
    for_testing_purposes=True)
142         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6], "X": [1, 2,
    2, 6, 10, 4], "Y": [2, 5, 1, 2, 5, 3],
143                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint"]})
144 
145         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
146 
147     # Invalidate/Correct TSP Instance (2) w/ Multiple Issues
148     def test_multiple_issues_2(self):
149         nodes = import_node_data(self.multiple_issues_2_name,
    for_testing_purposes=True)
150         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6], "X": [6, 1,
    2, 2, 10, 4], "Y": [2, 2, 5, 1, 5, 3],
151                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint"]})
152 
153         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
154 
155     # Invalidate/Correct Multiple Nodes w/ Same Node Number in Input Data
156     def test_multiple_nodes_with_same_node_number(self):
157         nodes = import_node_data(self.multiple_nodes_with_same_node_number_name,
     for_testing_purposes=True)
158         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 10, 6, 4, 5], "Y": [2, 5, 1, 5, 2, 3, 7],
159                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
160 
161         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
162 
163     # Invalidate Multiple Start Nodes in Input Data
164     def test_multiple_start_nodes(self):
165         with pytest.raises(ValueError) as e:
166             import_node_data(self.multiple_start_nodes_name,
    for_testing_purposes=True)
167 
168     # Invalidate no start nodes in Input Data
```

```
169     def test_no_start_nodes(self):
170         with pytest.raises(ValueError) as e:
171             import_node_data(self.no_start_nodes_name, for_testing_purposes=True
    )
172
173     # Invalidate/Correct Nodes don't range from 1 --> n in Input Data
174     def test_node_numbers_not_sequential(self):
175         nodes = import_node_data(self.node_numbers_not_sequential_name,
    for_testing_purposes=True)
176         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
177                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
178
179         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
180
181     # Invalidate no Start Nodes in Input Data
182     def test_not_enough_nodes(self):
183         with pytest.raises(ValueError) as e:
184             import_node_data(self.not_enough_nodes_name, for_testing_purposes=
    True)
185
186     # Invalidate/Correct rows not in order of Nodes in Input Data
187     def test_rows_not_in_order_of_nodes(self):
188         nodes = import_node_data(self.node_numbers_not_sequential_name,
    for_testing_purposes=True)
189         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
190                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
191
192         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
193
194     # Invalidate/Correct Start Node is not Node Number 1 in Input Data
195     def test_start_node_is_not_1(self):
196         nodes = import_node_data(self.start_node_is_not_1_name,
    for_testing_purposes=True)
197         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [6,
    1, 2, 2, 10, 4, 5], "Y": [2, 2, 5, 1, 5, 3, 7],
198                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
199
```

```
200         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
201
202     # Invalidate/Correct 2 Nodes have the Same Coordinates in Input Data
203     def test_two_nodes_with_same_coordinates(self):
204         nodes = import_node_data(self.two_nodes_with_same_coordinates_name,
    for_testing_purposes=True)
205         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6], "X": [1, 2,
    2, 6, 10, 4], "Y": [2, 5, 1, 2, 5, 3],
206                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint"]})
207
208         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
209
210     # Invalidate Column(s) having incorrect names
211     def test_wrong_column_names(self):
212         with pytest.raises(ValueError) as e:
213             import_node_data(self.wrong_column_names_name, for_testing_purposes=
    True)
214
215     # Invalidate incorrect number of columns
216     def test_wrong_number_of_columns(self):
217         with pytest.raises(ValueError) as e:
218             import_node_data(self.wrong_number_of_columns_name,
    for_testing_purposes=True)
219
220     # Validate that Valid Input Data is accepted
221     def test_valid_input_data(self):
222         nodes = import_node_data(self.valid_name, for_testing_purposes=True)
223         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
224                                        "Type": ["Start", "Waypoint", "Waypoint",
     "Waypoint", "Waypoint", "Waypoint", "Waypoint"]})
225
226         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
227
228     # Validate that Valid Input Data w/ a 'Name' Column is accepted
229     def test_valid_input_data_with_name_column(self):
230         nodes = import_node_data(self.valid_name_name, for_testing_purposes=True
    )
231         expected_table = pd.DataFrame({"Node": [1, 2, 3, 4, 5, 6, 7], "X": [1,
    2, 2, 6, 10, 4, 5], "Y": [2, 5, 1, 2, 5, 3, 7],
232                                        "Type": ["Start", "Waypoint", "Waypoint",
```

```
        "Waypoint", "Waypoint", "Waypoint", "Waypoint"],
233                                             "Name": [pd.NA, pd.NA, pd.NA, "Corner
    Shop", pd.NA, pd.NA, pd.NA]})
234
235         pd.testing.assert_frame_equal(downcast_to_int64(nodes), expected_table)
236
237 if __name__ == "__main__":
238     unittest.main(warnings = "ignore")
```

Listing C.9: Source Code for *test_tsp_utility_functions.py*

C.10 TEST - Nearest Neighbour

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities.Test_Inputs import mock_route_data as mock
4  from TSP_Utilities import tsp_utility_functions as tsp
5
6  import unittest
7  from NearestNeighbour import nearest_neighbour as nn
8
9  class TestNearestNeighbour(unittest.TestCase):
10     def setUp(self):
11         self.test_city_1_nodes = mock.get_test_city_1_nodes()
12         self.berlin52_nodes = mock.get_berlin52_nodes()
13         self.tsp225_nodes = mock.get_tsp225_nodes()
14
15     def test_nearest_neighbour_small(self):
16         expected_route = [1, 3, 6, 4, 2, 7, 5, 1]
17         expected_distance = 29.96
18
19         generated_route, generated_distance = nn.nearest_neighbour(self.
    test_city_1_nodes)
20
21         self.assertEqual(generated_route, expected_route)
22         self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
23
24     def test_nearest_neighbour_mid(self):
25         expected_route = [1, 22, 49, 32, 36, 35, 34, 39, 40, 38, 37, 48, 24, 5,
    15, 6, 4, 25, 46, 44, 16, 50, 20, 23, 31, 18, 3, 19, 45, 41, 8, 10, 9, 43,
    33, 51, 12, 28, 27, 26, 47, 13, 14, 52, 11, 29, 30, 21, 17, 42, 7, 2, 1]
```

```
26          expected_distance = 8980.92
27
28          generated_route, generated_distance = nn.nearest_neighbour(self.
        berlin52_nodes)
29
30          self.assertEqual(generated_route, expected_route)
31          self.assertAlmostEqual(generated_distance, expected_distance)
32
33      def test_nearest_neighbour_large(self):
34          expected_route = [1, 200, 3, 198, 4, 197, 5, 6, 7, 8, 9, 10, 11, 12, 13,
         14, 15, 16, 17, 20, 203, 19, 18, 22, 21, 23, 24, 208, 25, 26, 34, 33, 35,
        30, 202, 206, 31, 216, 219, 217, 77, 78, 79, 80, 81, 95, 209, 94, 93, 92,
        91, 90, 87, 210, 84, 83, 82, 85, 86, 131, 211, 130, 222, 129, 128, 127, 126,
         125, 124, 123, 122, 121, 175, 120, 185, 119, 118, 186, 187, 117, 116, 223,
        115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 220, 104, 103, 102,
        101, 100, 99, 98, 97, 96, 221, 28, 204, 29, 32, 38, 39, 40, 41, 42, 43, 44,
        46, 194, 218, 193, 196, 192, 191, 199, 224, 133, 190, 225, 47, 2, 207, 49,
        51, 57, 56, 55, 52, 53, 54, 70, 71, 72, 73, 74, 75, 76, 69, 68, 67, 66, 65,
        64, 63, 62, 61, 60, 59, 58, 50, 48, 45, 195, 205, 189, 27, 188, 184, 182,
        173, 181, 174, 180, 179, 176, 177, 178, 172, 171, 170, 169, 168, 212, 214,
        151, 150, 149, 152, 153, 154, 155, 156, 157, 144, 143, 201, 142, 141, 140,
        139, 138, 137, 136, 183, 135, 134, 215, 164, 165, 166, 167, 213, 158, 163,
        162, 161, 160, 159, 146, 147, 148, 145, 132, 88, 89, 37, 36, 1]
35          expected_distance = 4829.0
36
37          generated_route, generated_distance = nn.nearest_neighbour(self.
        tsp225_nodes)
38
39          self.assertEqual(generated_route, expected_route)
40          self.assertAlmostEqual(generated_distance, expected_distance)
41
42      def test_run_nearest_neighbour_generic(self):
43          generated_route, generated_distance = nn.run_nearest_neighbour_generic("
        berlin52.xlsx", tsp.import_node_data_tsp_lib, False)[0]
44
45          expected_route = [1, 22, 49, 32, 36, 35, 34, 39, 40, 38, 37, 48, 24, 5,
        15, 6, 4, 25, 46, 44, 16, 50, 20, 23, 31, 18, 3, 19, 45, 41, 8, 10, 9, 43,
        33, 51, 12, 28, 27, 26, 47, 13, 14, 52, 11, 29, 30, 21, 17, 42, 7, 2, 1]
46          expected_distance = 8980.92
47
48          self.assertEqual(generated_route, expected_route)
49          self.assertAlmostEqual(generated_distance, expected_distance)
```

```
50
51
52
53 if __name__ == "__main__":
54     unittest.main(warnings = "ignore")
```

Listing C.10: Source Code for *test_nearest_neighbour.py*

C.11 TEST - Insertion Heuristic w/ Convex Hull

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities.Test_Inputs import mock_route_data as mock
4  from TSP_Utilities import tsp_utility_functions as tsp
5
6  import unittest
7  from IHCV import ihcv as ih
8
9  class TestIHCV(unittest.TestCase):
10     def setUp(self):
11         self.test_city_1_nodes = mock.get_test_city_1_nodes()
12         self.berlin52_nodes = mock.get_berlin52_nodes()
13         self.tsp225_nodes = mock.get_tsp225_nodes()
14
15     def test_compute_convex_hull(self):
16         expected_hull = [32, 8, 16, 6, 1, 13, 51, 10]
17
18         generated_hull = ih.compute_convex_hull(self.berlin52_nodes)[1]
19         generated_hull_as_list = list(generated_hull.vertices)
20
21         self.assertEqual(expected_hull, generated_hull_as_list)
22
23     def test_ihcv_integrated_small(self):
24         expected_route = [1, 3, 6, 4, 5, 7, 2, 1]
25         expected_distance = 23.63
26
27         coordinate_array, hull = ih.compute_convex_hull(self.test_city_1_nodes)
28         distance_matrix = ih.compute_distance_matrix(coordinate_array)
29         generated_route, generated_distance = ih.cheapest_insertion(
       coordinate_array, hull, distance_matrix, self.test_city_1_nodes)
30
```

```
31          self.assertEqual(generated_route, expected_route)
32          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
33
34      def test_ihcv_integrated_mid(self):
35          expected_route = [1, 22, 31, 18, 3, 17, 21, 7, 2, 42, 30, 23, 20, 50,
        16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 37, 46, 48, 24, 5, 6, 25, 4, 15, 38, 40, 39, 36, 35, 34, 44, 1]
36          expected_distance = 8105.78
37
38          coordinate_array, hull = ih.compute_convex_hull(self.berlin52_nodes)
39          distance_matrix = ih.compute_distance_matrix(coordinate_array)
40          generated_route, generated_distance = ih.cheapest_insertion(
        coordinate_array, hull, distance_matrix, self.berlin52_nodes)
41
42          self.assertEqual(generated_route, expected_route)
43          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
44
45      def test_ihcv_integrated_large(self):
46          expected_route = [1, 200, 3, 198, 4, 197, 195, 46, 194, 218, 193, 45,
        48, 196, 192, 191, 224, 199, 133, 205, 189, 190, 225, 49, 50, 51, 57, 52,
        53, 54, 55, 56, 58, 59, 207, 2, 47, 27, 188, 117, 187, 118, 116, 223, 114,
        62, 60, 61, 63, 66, 68, 69, 67, 65, 64, 112, 110, 111, 113, 115, 119, 186,
        185, 120, 175, 121, 124, 123, 122, 184, 182, 171, 170, 172, 173, 181, 174,
        180, 179, 176, 178, 177, 148, 149, 150, 169, 168, 212, 214, 125, 126, 127,
        167, 151, 152, 153, 154, 156, 157, 155, 147, 146, 145, 144, 143, 201, 142,
        141, 140, 139, 138, 137, 136, 183, 135, 163, 161, 159, 160, 162, 158, 213,
        166, 165, 164, 134, 215, 132, 129, 128, 222, 130, 211, 131, 86, 85, 210, 87,
         84, 82, 83, 92, 90, 88, 89, 91, 94, 93, 96, 97, 209, 95, 221, 81, 80, 79,
        78, 77, 217, 219, 216, 98, 99, 100, 101, 103, 105, 107, 109, 108, 106, 220,
        104, 102, 73, 71, 70, 72, 75, 74, 76, 31, 32, 206, 202, 30, 29, 28, 204, 35,
         33, 34, 26, 25, 36, 208, 24, 23, 13, 12, 14, 16, 17, 15, 22, 21, 20, 203,
        19, 18, 11, 37, 38, 39, 10, 9, 40, 41, 42, 44, 43, 8, 7, 6, 5, 1]
47          expected_distance = 4442.27
48
49          coordinate_array, hull = ih.compute_convex_hull(self.tsp225_nodes)
50          distance_matrix = ih.compute_distance_matrix(coordinate_array)
51          generated_route, generated_distance = ih.cheapest_insertion(
        coordinate_array, hull, distance_matrix, self.tsp225_nodes)
52
53          self.assertEqual(generated_route, expected_route)
54          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
55
```

```
56      def test_run_ihcv_generic(self):
57          generated_route, generated_distance = ih.run_ihcv_generic("berlin52.xlsx
        ", tsp.import_node_data_tsp_lib, False)[0]
58
59          expected_route = [1, 22, 31, 18, 3, 17, 21, 7, 2, 42, 30, 23, 20, 50,
        16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 37, 46, 48, 24, 5, 6, 25, 4, 15, 38, 40, 39, 36, 35, 34, 44, 1]
60          expected_distance = 8105.78
61
62          self.assertEqual(generated_route, expected_route)
63          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
64
65
66
67  if __name__ == "__main__":
68      unittest.main(warnings = "ignore")
```

Listing C.11: Source Code for *test_ihcv.py*

C.12 TEST - Mixed-Integer Linear Programming

```
1   import sys
2   sys.path.append("../")
3   from TSP_Utilities import tsp_utility_functions
4
5   import unittest, pyomo.environ as pyo
6   from mtz import run_mtz
7
8   class TestMTZ(unittest.TestCase):
9       @classmethod
10      def setUpClass(cls):
11          cls.file_name = "test_city_1.xlsx"
12
13          cls.nodes = tsp_utility_functions.import_node_data(cls.file_name)
14          cls.n_nodes = len(cls.nodes)
15
16          cls.solution_with_model = run_mtz(cls.file_name)
17
18          cls.model = cls.solution_with_model[2]
19          cls.x = cls.model.x
20          cls.u = cls.model.u
```

```
        cls.M = cls.model.M

        cls.variables = cls.model.component_map(pyo.Var)

    """Unit Tests"""
    def test_x_variable(self):
        # Verify the number of "x"
        n_x = sum(len(TestMTZ.variables[var]) for var in TestMTZ.variables if "x
    " in var)
        self.assertEqual(n_x, 49)

        # Equal 1
        self.assertEqual(pyo.value(TestMTZ.x[0, 1]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[1, 6]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[2, 0]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[3, 5]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[4, 3]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[5, 2]), 1)
        self.assertEqual(pyo.value(TestMTZ.x[6, 4]), 1)

        # Selected a few to Test that Equal 0
        self.assertEqual(pyo.value(TestMTZ.x[0, 6]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[2, 3]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[3, 1]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[5, 3]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[4, 0]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[3, 4]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[5, 1]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[5, 4]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[6, 0]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[6, 2]), 0)
        self.assertEqual(pyo.value(TestMTZ.x[6, 5]), 0)

    def test_u_variable(self):
        # Assert values of "u" variable
        self.assertEqual(pyo.value(TestMTZ.u[0]), 1)
        self.assertEqual(pyo.value(TestMTZ.u[1]), 2)
        self.assertEqual(pyo.value(TestMTZ.u[2]), 7)
        self.assertEqual(pyo.value(TestMTZ.u[3]), 5)
        self.assertEqual(pyo.value(TestMTZ.u[4]), 4)
        self.assertEqual(pyo.value(TestMTZ.u[5]), 6)
        self.assertEqual(pyo.value(TestMTZ.u[6]), 3)
```

```
62
63      def test_M_variable(self):
64          # Verify the number of "M"
65          n_M = sum(len(TestMTZ.variables[var]) for var in TestMTZ.variables if "M
    " in var)
66          self.assertEqual(n_M, 49)
67
68          # Assert values of a few "M"
69          self.assertEqual(pyo.value(TestMTZ.M[1, 2]), 6)
70          self.assertEqual(pyo.value(TestMTZ.M[1, 4]), 6)
71          self.assertEqual(pyo.value(TestMTZ.M[2, 3]), 6)
72          self.assertEqual(pyo.value(TestMTZ.M[2, 5]), 6)
73          self.assertEqual(pyo.value(TestMTZ.M[3, 1]), 6)
74          self.assertEqual(pyo.value(TestMTZ.M[3, 2]), 6)
75          self.assertEqual(pyo.value(TestMTZ.M[4, 2]), 6)
76          self.assertEqual(pyo.value(TestMTZ.M[4, 6]), 6)
77          self.assertEqual(pyo.value(TestMTZ.M[5, 3]), 6)
78          self.assertEqual(pyo.value(TestMTZ.M[6, 5]), 6)
79
80          self.assertEqual(pyo.value(TestMTZ.M[1, 6]), 0)
81          self.assertEqual(pyo.value(TestMTZ.M[3, 5]), 0)
82          self.assertEqual(pyo.value(TestMTZ.M[4, 3]), 0)
83          self.assertEqual(pyo.value(TestMTZ.M[5, 2]), 0)
84          self.assertEqual(pyo.value(TestMTZ.M[6, 4]), 0)
85
86      def test_constraint_1(self):
87          self.assertEqual(str(TestMTZ.model.C1[1].expr), "x[0,1] + x[0,2] + x
    [0,3] + x[0,4] + x[0,5] + x[0,6]  ==  1")
88          self.assertEqual(str(TestMTZ.model.C1[2].expr), "x[1,0] + x[1,2] + x
    [1,3] + x[1,4] + x[1,5] + x[1,6]  ==  1")
89          self.assertEqual(str(TestMTZ.model.C1[3].expr), "x[2,0] + x[2,1] + x
    [2,3] + x[2,4] + x[2,5] + x[2,6]  ==  1")
90          self.assertEqual(str(TestMTZ.model.C1[4].expr), "x[3,0] + x[3,1] + x
    [3,2] + x[3,4] + x[3,5] + x[3,6]  ==  1")
91          self.assertEqual(str(TestMTZ.model.C1[5].expr), "x[4,0] + x[4,1] + x
    [4,2] + x[4,3] + x[4,5] + x[4,6]  ==  1")
92          self.assertEqual(str(TestMTZ.model.C1[6].expr), "x[5,0] + x[5,1] + x
    [5,2] + x[5,3] + x[5,4] + x[5,6]  ==  1")
93          self.assertEqual(str(TestMTZ.model.C1[7].expr), "x[6,0] + x[6,1] + x
    [6,2] + x[6,3] + x[6,4] + x[6,5]  ==  1")
94
95      def test_constraint_2(self):
```

```
96          self.assertEqual(str(TestMTZ.model.C2[1].expr), "x[1,0] + x[2,0] + x
    [3,0] + x[4,0] + x[5,0] + x[6,0]  ==  1")
97          self.assertEqual(str(TestMTZ.model.C2[2].expr), "x[0,1] + x[2,1] + x
    [3,1] + x[4,1] + x[5,1] + x[6,1]  ==  1")
98          self.assertEqual(str(TestMTZ.model.C2[3].expr), "x[0,2] + x[1,2] + x
    [3,2] + x[4,2] + x[5,2] + x[6,2]  ==  1")
99          self.assertEqual(str(TestMTZ.model.C2[4].expr), "x[0,3] + x[1,3] + x
    [2,3] + x[4,3] + x[5,3] + x[6,3]  ==  1")
100         self.assertEqual(str(TestMTZ.model.C2[5].expr), "x[0,4] + x[1,4] + x
    [2,4] + x[3,4] + x[5,4] + x[6,4]  ==  1")
101         self.assertEqual(str(TestMTZ.model.C2[6].expr), "x[0,5] + x[1,5] + x
    [2,5] + x[3,5] + x[4,5] + x[6,5]  ==  1")
102         self.assertEqual(str(TestMTZ.model.C2[7].expr), "x[0,6] + x[1,6] + x
    [2,6] + x[3,6] + x[4,6] + x[5,6]  ==  1")
103
104     def test_constraint_3(self):
105         self.assertEqual(str(TestMTZ.model.C3.expr), "u[0]  ==  1")
106
107     def test_constraint_4(self):
108         self.assertEqual(str(TestMTZ.model.C4[1].expr), "2  <=  u[1]  <=  7")
109         self.assertEqual(str(TestMTZ.model.C4[2].expr), "2  <=  u[2]  <=  7")
110         self.assertEqual(str(TestMTZ.model.C4[3].expr), "2  <=  u[3]  <=  7")
111         self.assertEqual(str(TestMTZ.model.C4[4].expr), "2  <=  u[4]  <=  7")
112         self.assertEqual(str(TestMTZ.model.C4[5].expr), "2  <=  u[5]  <=  7")
113         self.assertEqual(str(TestMTZ.model.C4[6].expr), "2  <=  u[6]  <=  7")
114
115     def test_constraint_5(self):
116         n_C5 = len(TestMTZ.model.component("C5"))
117         self.assertEqual(n_C5, 30)
118
119         self.assertEqual(str(TestMTZ.model.C5[1].expr), "u[1] - u[2] + 1 - M
    [1,2]  <=  0")
120         self.assertEqual(str(TestMTZ.model.C5[3].expr), "u[1] - u[4] + 1 - M
    [1,4]  <=  0")
121         self.assertEqual(str(TestMTZ.model.C5[9].expr), "u[2] - u[5] + 1 - M
    [2,5]  <=  0")
122         self.assertEqual(str(TestMTZ.model.C5[12].expr), "u[3] - u[2] + 1 - M
    [3,2]  <=  0")
123         self.assertEqual(str(TestMTZ.model.C5[15].expr), "u[3] - u[6] + 1 - M
    [3,6]  <=  0")
124         self.assertEqual(str(TestMTZ.model.C5[16].expr), "u[4] - u[1] + 1 - M
    [4,1]  <=  0")
```

```
125         self.assertEqual(str(TestMTZ.model.C5[22].expr), "u[5] - u[2] + 1 - M
    [5,2]  <=  0")
126         self.assertEqual(str(TestMTZ.model.C5[24].expr), "u[5] - u[4] + 1 - M
    [5,4]  <=  0")
127         self.assertEqual(str(TestMTZ.model.C5[28].expr), "u[6] - u[3] + 1 - M
    [6,3]  <=  0")
128         self.assertEqual(str(TestMTZ.model.C5[30].expr), "u[6] - u[5] + 1 - M
    [6,5]  <=  0")
129
130     # .replace(" ","")
131     def test_objective_function(self):
132         self.maxDiff = None
133         if 'unittest.util' in __import__('sys').modules:
134             __import__('sys').modules['unittest.util']._MAX_LENGTH = 999999999
135
136         extracted_objective_expression = str(TestMTZ.model.obj.expr).replace(" "
    ,"")
137         sense = "maximize" if TestMTZ.model.obj.sense == pyo.maximize else "
    minimize"
138
139         expected_objective_expression = "0.0*x[0,0] + 3.1622776601683795*x[1,0]
    + 1.4142135623730951*x[2,0] + 5.0*x[3,0] + 9.486832980505138*x[4,0] +
    3.1622776601683795*x[5,0] + 6.4031242374328485*x[6,0] + 3.1622776601683795*x
    [0,1] + 0.0*x[1,1] + 4.0*x[2,1] + 5.0*x[3,1] + 8.0*x[4,1] +
    2.8284271247461903*x[5,1] + 3.605551275463989*x[6,1] + 1.4142135623730951*x
    [0,2] + 4.0*x[1,2] + 0.0*x[2,2] + 4.123105625617661*x[3,2] +
    8.94427190999916*x[4,2] + 2.8284271247461903*x[5,2] + 6.708203932499369*x
    [6,2] + 5.0*x[0,3] + 5.0*x[1,3] + 4.123105625617661*x[2,3] + 0.0*x[3,3] +
    5.0*x[4,3] + 2.23606797749979*x[5,3] + 5.0990195135927845*x[6,3] +
    9.486832980505138*x[0,4] + 8.0*x[1,4] + 8.94427190999916*x[2,4] + 5.0*x[3,4]
     + 0.0*x[4,4] + 6.324555320336759*x[5,4] + 5.385164807134504*x[6,4] +
    3.1622776601683795*x[0,5] + 2.8284271247461903*x[1,5] + 2.8284271247461903*x
    [2,5] + 2.23606797749979*x[3,5] + 6.324555320336759*x[4,5] + 0.0*x[5,5] +
    4.123105625617661*x[6,5] + 6.4031242374328485*x[0,6] + 3.605551275463989*x
    [1,6] + 6.708203932499369*x[2,6] + 5.0990195135927845*x[3,6] +
    5.385164807134504*x[4,6] + 4.123105625617661*x[5,6] + 0.0*x[6,6]".replace("
    ","")
140
141         self.assertEqual(extracted_objective_expression,
    expected_objective_expression)
142         self.assertEqual(sense, "minimize")
143
```

```
144     """Integration Tests"""
145     def test_mtz(self):
146         route, distance_travelled = TestMTZ.solution_with_model[:2]
147 
148         self.assertEqual(route, [1, 2, 7, 5, 4, 6, 3, 1])
149         self.assertAlmostEqual(distance_travelled, 23.63)
150 
151 
152 if __name__ == "__main__":
153     unittest.main(warnings = "ignore")
```

Listing C.12: Source Code for *test_mtz.py*

C.13 TEST - Two Opt

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities.Test_Inputs import mock_route_data as mock
4  from TSP_Utilities import tsp_utility_functions as tsp
5  from IHCV import ihcv as ih
6  from TwoOpt import two_opt as to
7 
8  import unittest
9 
10 class TestTwoOpt(unittest.TestCase):
11     def setUp(self):
12         self.test_city_1_nodes = mock.get_test_city_1_nodes()
13         self.berlin52_nodes = mock.get_berlin52_nodes()
14         self.tsp225_nodes = mock.get_tsp225_nodes()
15 
16         self.test_city_1_distance_matrix = mock.get_test_city_1_distance_matrix
       ()
17         self.berlin52_distance_matrix = mock.get_berlin52_distance_matrix()
18 
19     def test_two_opt_swap_move(self):
20         route = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
21         swapped_route = to.two_opt_swap_move(route, 1, 10)
22 
23         expected_swapped_route = [1, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 12, 13, 14,
        15]
24 
```

```
25          self.assertEqual(expected_swapped_route, swapped_route)
26
27      def test_compute_nearest_neighbours(self):
28          nearest_neighbours = to.compute_nearest_neighbours(self.
        test_city_1_distance_matrix, 1, 5)
29
30          expected_nearest_neighbours = [[2, 1, 5, 3, 6, 4], [5, 0, 6, 2, 3, 4],
        [0, 5, 1, 3, 6, 4], [5, 2, 0, 1, 4, 6], [3, 6, 5, 1, 2, 0], [3, 1, 2, 0, 6,
        4], [1, 5, 3, 4, 0, 2]]
31
32          self.assertEqual(expected_nearest_neighbours, nearest_neighbours)
33
34      def test_nn_to_integrated_small(self):
35          initial_solution = ([1, 3, 6, 4, 2, 7, 5, 1], 29.96)
36          distance_matrix = self.test_city_1_distance_matrix
37          nearest_neighbours = [[2, 1, 5, 3, 6, 4], [5, 0, 6, 2, 3, 4], [0, 5, 1,
        3, 6, 4], [5, 2, 0, 1, 4, 6], [3, 6, 5, 1, 2, 0], [3, 1, 2, 0, 6, 4], [1, 5,
         3, 4, 0, 2]]
38          node_index_mapping = mock.get_test_city_1_node_index_mapping()
39          generated_route, generated_distance = to.two_opt(initial_solution,
        distance_matrix, nearest_neighbours, node_index_mapping)
40
41          expected_route = [1, 3, 6, 4, 5, 7, 2, 1]
42          expected_distance = 23.63
43
44          self.assertEqual(expected_route, generated_route)
45          self.assertAlmostEqual(expected_distance, generated_distance)
46
47      def test_ihcv_to_integrated_mid(self):
48          initial_solution = ([1, 22, 31, 18, 3, 17, 21, 7, 2, 42, 30, 23, 20, 50,
         16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 37, 46, 48, 24, 5, 6, 25, 4, 15, 38, 40, 39, 36, 35, 34, 44, 1],
         8105.78)
49          distance_matrix = self.berlin52_distance_matrix
50          nearest_neighbours = [[21, 48, 31, 34, 35, 33, 30, 43, 17, 38, 22, 36,
        39, 19, 49, 44, 37, 15, 47, 23, 45, 2, 20, 4, 14, 18, 5, 29, 40, 3, 28, 24,
        7, 16, 42, 9, 41, 8, 6, 11, 1, 27, 25, 50, 46, 26, 32, 12, 10, 13, 51], [6,
        41, 29, 20, 16, 22, 30, 19, 17, 49, 28, 21, 2, 0, 15, 48, 31, 43, 34, 33,
        35, 44, 38, 18, 36, 45, 39, 40, 37, 47, 23, 4, 7, 14, 5, 24, 3, 42, 9, 8,
        46, 25, 27, 11, 26, 50, 12, 32, 13, 10, 51], [17, 18, 30, 16, 44, 21, 31,
        40, 0, 48, 20, 7, 22, 35, 34, 38, 33, 39, 43, 19, 36, 37, 49, 8, 9, 29, 47,
        14, 23, 4, 15, 5, 41, 45, 42, 3, 6, 1, 28, 24, 11, 32, 27, 50, 25, 26, 46,
```

12, 10, 13, 51], [5, 14, 4, 24, 23, 47, 37, 39, 36, 38, 45, 42, 33, 35, 34,
11, 43, 48, 31, 15, 0, 50, 44, 27, 21, 18, 49, 19, 25, 32, 7, 26, 17, 9, 22,
30, 40, 28, 2, 8, 46, 10, 20, 29, 12, 16, 13, 51, 41, 6, 1], [14, 23, 5,
47, 37, 39, 36, 3, 38, 24, 33, 35, 45, 34, 43, 48, 42, 31, 0, 15, 44, 21,
11, 49, 18, 19, 17, 30, 22, 7, 27, 40, 50, 2, 28, 9, 25, 20, 8, 26, 32, 29,
46, 16, 10, 12, 41, 6, 13, 1, 51], [4, 14, 23, 3, 47, 37, 39, 36, 24, 38,
45, 33, 35, 34, 43, 42, 48, 31, 0, 15, 11, 44, 21, 49, 18, 19, 27, 50, 17,
22, 30, 7, 40, 9, 28, 25, 2, 26, 32, 8, 20, 29, 46, 10, 16, 12, 41, 13, 51,
6, 1], [1, 41, 20, 29, 16, 22, 30, 19, 17, 49, 21, 2, 28, 0, 31, 48, 15, 43,
34, 33, 35, 44, 38, 18, 36, 45, 39, 40, 37, 47, 23, 7, 4, 14, 5, 24, 3, 42,
9, 8, 46, 25, 27, 11, 26, 50, 12, 32, 13, 10, 51], [40, 18, 9, 8, 44, 2,
31, 42, 17, 48, 38, 21, 35, 39, 0, 34, 37, 14, 33, 30, 36, 4, 23, 5, 47, 16,
3, 43, 20, 22, 45, 24, 19, 49, 32, 15, 29, 11, 28, 41, 50, 6, 27, 1, 25,
26, 46, 10, 12, 13, 51], [9, 7, 40, 18, 44, 42, 2, 31, 48, 39, 38, 14, 17,
32, 37, 35, 4, 34, 21, 5, 36, 0, 23, 33, 3, 47, 30, 16, 43, 24, 45, 20, 22,
19, 49, 15, 11, 50, 29, 28, 27, 41, 6, 25, 26, 1, 10, 46, 12, 51, 13], [8,
7, 40, 18, 42, 44, 31, 2, 14, 39, 38, 32, 37, 48, 4, 35, 5, 36, 3, 34, 23,
17, 47, 33, 21, 0, 30, 43, 24, 45, 16, 22, 20, 15, 19, 49, 11, 50, 29, 27,
28, 41, 25, 26, 10, 6, 1, 46, 12, 51, 13], [50, 11, 51, 26, 27, 12, 25, 13,
24, 3, 46, 32, 5, 4, 14, 23, 47, 45, 42, 37, 36, 39, 38, 33, 43, 15, 35, 34,
48, 31, 0, 28, 49, 44, 19, 9, 21, 18, 22, 7, 8, 40, 30, 17, 29, 2, 20, 16,
41, 6, 1], [27, 50, 24, 26, 3, 25, 5, 4, 10, 23, 14, 47, 45, 37, 36, 12, 39,
42, 38, 46, 33, 43, 34, 35, 15, 32, 48, 51, 31, 13, 0, 49, 28, 19, 21, 44,
22, 18, 9, 30, 17, 7, 40, 8, 29, 2, 20, 16, 41, 6, 1], [26, 13, 25, 51, 27,
46, 10, 11, 50, 24, 45, 3, 5, 47, 15, 23, 4, 14, 37, 28, 36, 43, 39, 33, 38,
34, 35, 49, 19, 42, 48, 0, 31, 22, 32, 21, 29, 44, 30, 17, 18, 20, 7, 9, 2,
40, 8, 41, 16, 1, 6], [12, 51, 46, 26, 25, 27, 10, 11, 50, 24, 45, 15, 28,
3, 5, 47, 23, 4, 14, 43, 36, 37, 49, 39, 33, 38, 34, 19, 35, 48, 0, 42, 22,
29, 31, 21, 32, 30, 44, 17, 20, 18, 2, 7, 40, 9, 41, 8, 1, 6, 16], [4, 5,
23, 37, 47, 39, 3, 36, 38, 24, 35, 33, 34, 45, 43, 42, 48, 31, 0, 44, 15,
21, 18, 11, 49, 19, 17, 30, 22, 7, 40, 9, 27, 50, 2, 28, 8, 32, 25, 20, 26,
29, 46, 16, 10, 12, 41, 6, 1, 13, 51], [49, 43, 45, 19, 28, 33, 34, 35, 36,
22, 0, 47, 38, 23, 48, 39, 37, 21, 4, 24, 5, 31, 14, 29, 30, 3, 17, 20, 44,
25, 27, 2, 11, 46, 18, 42, 26, 41, 40, 16, 7, 50, 6, 1, 12, 9, 8, 13, 32,
10, 51], [2, 20, 17, 30, 21, 41, 22, 18, 0, 31, 44, 40, 6, 48, 29, 1, 7, 19,
35, 49, 34, 33, 38, 43, 39, 36, 37, 15, 8, 9, 47, 23, 4, 14, 28, 45, 5, 42,
3, 24, 11, 32, 27, 25, 46, 50, 26, 12, 10, 13, 51], [30, 21, 2, 0, 31, 48,
20, 44, 22, 18, 35, 16, 34, 33, 38, 19, 43, 49, 40, 39, 36, 37, 7, 29, 15,
47, 23, 4, 14, 45, 5, 41, 28, 3, 42, 9, 6, 24, 8, 1, 11, 27, 25, 32, 50, 46,
26, 12, 10, 13, 51], [44, 40, 7, 2, 31, 17, 48, 21, 9, 0, 8, 35, 38, 34,
30, 39, 33, 37, 42, 36, 14, 4, 23, 47, 43, 16, 5, 20, 22, 3, 45, 19, 49, 15,
24, 29, 32, 28, 41, 11, 6, 1, 50, 27, 25, 26, 46, 10, 12, 13, 51], [49, 22,

15, 29, 43, 0, 28, 21, 30, 33, 34, 48, 35, 20, 45, 31, 17, 38, 36, 39, 47,
37, 23, 4, 14, 5, 2, 44, 24, 41, 3, 16, 18, 6, 1, 40, 7, 42, 25, 46, 27, 11,
26, 9, 8, 50, 12, 32, 13, 10, 51], [30, 22, 17, 29, 21, 16, 19, 0, 2, 41,
49, 31, 48, 6, 1, 43, 34, 35, 33, 44, 15, 38, 28, 18, 36, 39, 40, 37, 45,
47, 23, 4, 7, 14, 5, 3, 24, 42, 9, 8, 11, 25, 27, 46, 26, 50, 32, 12, 13,
10, 51], [0, 48, 31, 30, 17, 34, 35, 22, 33, 43, 38, 19, 44, 49, 2, 39, 20,
36, 37, 18, 15, 47, 23, 45, 4, 14, 29, 5, 40, 16, 7, 28, 3, 24, 42, 41, 9,
8, 6, 1, 11, 27, 25, 50, 46, 26, 32, 12, 10, 13, 51], [19, 49, 30, 21, 29,
0, 20, 17, 43, 48, 15, 31, 34, 33, 35, 28, 38, 36, 45, 2, 39, 37, 47, 44,
23, 41, 16, 4, 14, 18, 5, 6, 1, 24, 3, 40, 7, 42, 9, 25, 11, 8, 27, 46, 26,
50, 32, 12, 13, 10, 51], [47, 4, 37, 14, 5, 36, 39, 38, 3, 45, 33, 24, 35,
34, 43, 48, 31, 0, 15, 42, 21, 44, 49, 19, 11, 18, 22, 17, 30, 27, 28, 7,
40, 2, 50, 25, 9, 20, 29, 26, 8, 32, 46, 16, 12, 10, 41, 6, 1, 13, 51], [3,
5, 23, 47, 4, 14, 45, 37, 36, 39, 11, 38, 33, 43, 34, 35, 15, 27, 42, 48,
50, 31, 0, 25, 26, 49, 21, 19, 44, 28, 22, 46, 18, 30, 17, 32, 7, 12, 40,
10, 9, 2, 29, 20, 8, 13, 16, 51, 41, 6, 1], [26, 27, 46, 12, 11, 13, 24, 45,
50, 15, 3, 51, 47, 5, 23, 10, 4, 28, 14, 43, 36, 37, 39, 33, 38, 49, 34,
35, 19, 48, 0, 22, 42, 31, 21, 29, 30, 44, 17, 32, 20, 18, 2, 7, 40, 9, 41,
8, 16, 6, 1], [27, 25, 12, 11, 46, 13, 50, 10, 51, 24, 45, 3, 5, 47, 23, 4,
15, 14, 37, 36, 43, 39, 28, 33, 38, 34, 35, 49, 42, 19, 48, 0, 31, 21, 22,
32, 29, 44, 30, 17, 18, 20, 7, 9, 40, 2, 8, 16, 41, 6, 1], [26, 25, 11, 12,
24, 50, 46, 3, 10, 45, 5, 47, 23, 4, 13, 14, 51, 37, 36, 15, 39, 43, 33, 38,
34, 35, 28, 42, 49, 48, 19, 0, 31, 21, 32, 22, 44, 30, 29, 17, 18, 7, 9,
20, 40, 2, 8, 16, 41, 6, 1], [49, 15, 19, 29, 22, 43, 45, 33, 0, 34, 21, 35,
36, 30, 48, 38, 20, 47, 23, 39, 37, 31, 24, 4, 17, 46, 5, 14, 25, 41, 3,
27, 1, 6, 44, 2, 26, 11, 16, 18, 42, 40, 12, 7, 50, 13, 9, 8, 10, 51, 32],
[22, 19, 49, 20, 28, 30, 41, 15, 21, 0, 43, 17, 6, 1, 48, 33, 34, 31, 35,
45, 16, 38, 2, 36, 39, 37, 47, 23, 44, 4, 14, 5, 18, 24, 3, 40, 7, 46, 25,
42, 27, 11, 9, 26, 8, 50, 12, 13, 32, 10, 51], [17, 21, 0, 20, 22, 31, 48,
2, 19, 49, 34, 35, 33, 43, 44, 16, 29, 38, 18, 39, 36, 15, 37, 40, 47, 45,
23, 4, 41, 14, 28, 7, 5, 6, 3, 1, 24, 42, 9, 8, 11, 27, 25, 46, 26, 50, 32,
12, 10, 13, 51], [48, 0, 21, 35, 34, 38, 33, 44, 17, 39, 30, 36, 43, 18, 37,
2, 47, 23, 22, 4, 14, 19, 5, 49, 40, 45, 15, 7, 20, 3, 42, 24, 16, 29, 9,
28, 8, 41, 11, 6, 27, 1, 32, 25, 50, 26, 46, 12, 10, 13, 51], [42, 50, 9, 3,
5, 14, 8, 11, 4, 24, 23, 37, 47, 39, 7, 36, 38, 44, 18, 40, 10, 35, 34, 33,
45, 31, 48, 27, 43, 0, 21, 26, 15, 17, 2, 25, 30, 49, 19, 22, 12, 51, 46,
28, 20, 16, 29, 13, 41, 6, 1], [34, 35, 38, 36, 43, 39, 48, 37, 47, 0, 23,
31, 45, 4, 14, 21, 5, 15, 49, 19, 3, 24, 22, 44, 30, 17, 18, 42, 2, 28, 20,
29, 40, 7, 11, 9, 16, 27, 8, 25, 50, 26, 41, 46, 32, 6, 1, 12, 10, 13, 51],
[35, 33, 38, 48, 36, 39, 43, 37, 0, 31, 47, 23, 21, 4, 14, 45, 5, 15, 49,
44, 19, 30, 22, 17, 3, 24, 18, 2, 42, 20, 28, 40, 7, 29, 11, 9, 16, 8, 27,
25, 50, 41, 26, 46, 32, 6, 1, 12, 10, 13, 51], [34, 33, 38, 48, 39, 36, 43,

37, 31, 0, 47, 23, 4, 21, 14, 45, 5, 44, 15, 49, 3, 30, 17, 19, 22, 24, 18,
42, 2, 40, 20, 7, 28, 29, 9, 11, 16, 8, 27, 25, 50, 41, 26, 32, 46, 6, 1,
12, 10, 13, 51], [39, 37, 38, 47, 23, 33, 35, 34, 4, 14, 5, 43, 45, 48, 3,
31, 24, 0, 15, 21, 44, 49, 19, 42, 22, 30, 17, 18, 28, 2, 11, 40, 7, 20, 29,
27, 9, 25, 50, 8, 26, 16, 32, 46, 41, 12, 10, 6, 1, 13, 51], [39, 36, 23,
47, 4, 14, 38, 5, 33, 35, 34, 3, 45, 43, 48, 24, 31, 0, 21, 15, 42, 44, 49,
19, 18, 17, 30, 22, 11, 7, 40, 2, 28, 9, 27, 20, 50, 29, 8, 25, 26, 32, 16,
46, 41, 12, 10, 6, 1, 13, 51], [39, 35, 34, 36, 33, 37, 47, 23, 48, 4, 14,
43, 31, 5, 0, 45, 21, 3, 44, 24, 15, 49, 17, 30, 19, 18, 42, 22, 2, 40, 7,
20, 28, 29, 9, 11, 8, 16, 27, 50, 25, 32, 26, 46, 41, 6, 1, 12, 10, 13, 51],
[37, 36, 38, 23, 47, 35, 4, 14, 33, 34, 5, 48, 43, 45, 31, 3, 0, 24, 21,
44, 15, 42, 49, 18, 17, 19, 30, 22, 40, 7, 2, 11, 28, 9, 20, 29, 27, 8, 50,
25, 16, 32, 26, 46, 41, 12, 10, 6, 1, 13, 51], [7, 18, 44, 8, 9, 2, 31, 17,
48, 21, 0, 42, 38, 35, 30, 39, 34, 33, 16, 37, 36, 14, 4, 23, 47, 5, 43, 20,
3, 22, 45, 19, 49, 24, 15, 32, 29, 41, 28, 11, 6, 1, 50, 27, 25, 26, 46,
10, 12, 13, 51], [6, 1, 20, 29, 16, 22, 30, 19, 17, 49, 21, 2, 0, 28, 48,
31, 15, 43, 34, 33, 35, 44, 38, 18, 36, 45, 39, 40, 37, 47, 23, 4, 7, 14, 5,
24, 3, 42, 9, 8, 46, 25, 27, 11, 26, 50, 12, 32, 13, 10, 51], [14, 3, 5, 4,
37, 39, 23, 9, 47, 38, 36, 44, 24, 7, 35, 32, 18, 34, 33, 8, 31, 48, 40,
45, 43, 0, 11, 21, 50, 17, 2, 15, 30, 49, 19, 27, 22, 20, 26, 25, 28, 16,
10, 29, 46, 12, 41, 6, 51, 1, 13], [33, 34, 35, 36, 45, 15, 38, 0, 48, 39,
47, 37, 49, 23, 19, 21, 31, 4, 14, 22, 5, 30, 24, 3, 17, 28, 44, 29, 20, 18,
2, 42, 40, 11, 7, 27, 25, 16, 9, 46, 26, 41, 50, 8, 6, 1, 32, 12, 10, 13,
51], [18, 31, 40, 7, 48, 2, 17, 21, 0, 35, 38, 34, 39, 33, 30, 37, 36, 14,
9, 4, 42, 23, 47, 43, 5, 8, 22, 3, 45, 20, 16, 19, 49, 15, 24, 29, 28, 32,
11, 41, 6, 50, 27, 1, 25, 26, 46, 10, 12, 13, 51], [47, 43, 36, 23, 15, 33,
37, 4, 24, 39, 5, 34, 38, 35, 14, 3, 48, 49, 0, 19, 31, 21, 28, 22, 11, 30,
27, 44, 42, 17, 25, 29, 18, 26, 20, 46, 2, 50, 40, 7, 9, 12, 16, 8, 32, 41,
10, 6, 13, 1, 51], [25, 26, 12, 27, 13, 11, 28, 15, 45, 24, 51, 43, 47, 49,
23, 3, 5, 4, 36, 19, 50, 37, 14, 33, 10, 39, 34, 38, 35, 22, 29, 0, 48, 21,
31, 30, 42, 17, 20, 44, 18, 2, 41, 32, 40, 7, 1, 6, 9, 16, 8], [23, 4, 37,
36, 5, 14, 39, 38, 45, 33, 3, 35, 34, 24, 43, 48, 31, 15, 0, 21, 42, 44, 49,
19, 11, 22, 30, 18, 17, 28, 27, 7, 40, 2, 50, 25, 20, 9, 29, 26, 8, 32, 46,
16, 12, 10, 41, 6, 1, 13, 51], [31, 0, 35, 34, 21, 33, 38, 43, 39, 36, 17,
37, 44, 30, 47, 23, 4, 22, 14, 19, 45, 49, 18, 5, 15, 2, 20, 3, 40, 24, 7,
42, 29, 28, 16, 9, 8, 11, 41, 27, 6, 25, 1, 50, 32, 26, 46, 12, 10, 13, 51],
[19, 22, 15, 43, 28, 29, 0, 21, 33, 34, 30, 48, 35, 45, 38, 36, 31, 20, 17,
39, 47, 37, 23, 4, 14, 5, 24, 44, 2, 3, 41, 18, 16, 6, 1, 40, 42, 25, 7,
46, 27, 11, 26, 9, 8, 50, 12, 32, 13, 10, 51], [11, 10, 27, 24, 3, 26, 5,
32, 25, 14, 4, 42, 23, 47, 12, 37, 45, 36, 39, 51, 38, 33, 35, 34, 43, 46,
15, 48, 13, 31, 0, 44, 9, 49, 21, 18, 19, 7, 28, 8, 40, 22, 17, 30, 2, 29,
20, 16, 41, 6, 1], [12, 13, 10, 26, 27, 25, 46, 11, 50, 24, 3, 45, 5, 47,

```
        23, 4, 14, 15, 37, 36, 39, 43, 28, 32, 38, 33, 42, 34, 35, 49, 19, 48, 0,
        31, 21, 22, 44, 29, 30, 17, 18, 9, 7, 40, 20, 2, 8, 41, 16, 6, 1]]
51          node_index_mapping = mock.get_berlin52_node_index_mapping()
52          generated_route, generated_distance = to.two_opt(initial_solution,
        distance_matrix, nearest_neighbours, node_index_mapping)
53
54          expected_route = [1, 22, 31, 18, 3, 17, 21, 42, 7, 2, 30, 23, 20, 50,
        16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 39, 40, 38, 15, 4, 25, 6, 5, 24, 48, 46, 37, 36, 35, 34, 44, 1]
55          expected_distance = 8074.56
56
57          self.assertEqual(generated_route, expected_route)
58          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
59
60      def test_nn_to_integrated_large(self):
61          initial_solution = ([1, 200, 3, 198, 4, 197, 5, 6, 7, 8, 9, 10, 11, 12,
        13, 14, 15, 16, 17, 20, 203, 19, 18, 22, 21, 23, 24, 208, 25, 26, 34, 33,
        35, 30, 202, 206, 31, 216, 219, 217, 77, 78, 79, 80, 81, 95, 209, 94, 93,
        92, 91, 90, 87, 210, 84, 83, 82, 85, 86, 131, 211, 130, 222, 129, 128, 127,
        126, 125, 124, 123, 122, 121, 175, 120, 185, 119, 118, 186, 187, 117, 116,
        223, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 220, 104, 103,
        102, 101, 100, 99, 98, 97, 96, 221, 28, 204, 29, 32, 38, 39, 40, 41, 42, 43,
         44, 46, 194, 218, 193, 196, 192, 191, 199, 224, 133, 190, 225, 47, 2, 207,
        49, 51, 57, 56, 55, 52, 53, 54, 70, 71, 72, 73, 74, 75, 76, 69, 68, 67, 66,
        65, 64, 63, 62, 61, 60, 59, 58, 50, 48, 45, 195, 205, 189, 27, 188, 184,
        182, 173, 181, 174, 180, 179, 176, 177, 178, 172, 171, 170, 169, 168, 212,
        214, 151, 150, 149, 152, 153, 154, 155, 156, 157, 144, 143, 201, 142, 141,
        140, 139, 138, 137, 136, 183, 135, 134, 215, 164, 165, 166, 167, 213, 158,
        163, 162, 161, 160, 159, 146, 147, 148, 145, 132, 88, 89, 37, 36, 1],
        4829.0)
62          distance_matrix = tsp.compute_distance_matrix(self.tsp225_nodes)
63          nearest_neighbours = to.compute_nearest_neighbours(distance_matrix, 1,
        5)
64          node_index_mapping = mock.get_tsp225_node_index_mapping()
65          generated_route, generated_distance = to.two_opt(initial_solution,
        distance_matrix, nearest_neighbours, node_index_mapping)
66
67          expected_route = [1, 200, 198, 197, 195, 43, 42, 44, 46, 194, 218, 193,
        45, 48, 196, 192, 191, 205, 189, 27, 188, 117, 187, 186, 119, 118, 116, 223,
         115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 67, 66, 65, 64, 63, 62,
        61, 60, 58, 59, 2, 207, 47, 225, 190, 133, 199, 224, 50, 49, 51, 57, 56, 55,
         52, 53, 54, 70, 68, 69, 71, 72, 73, 74, 75, 76, 31, 216, 219, 217, 77, 78,
```

```
        79, 80, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 220, 105, 89, 88, 125,
        124, 123, 122, 121, 175, 120, 185, 184, 182, 173, 181, 180, 179, 178, 177,
        176, 174, 172, 171, 170, 169, 168, 150, 149, 148, 147, 146, 145, 143, 201,
        142, 141, 140, 139, 138, 137, 136, 183, 135, 134, 215, 164, 165, 166, 167,
        213, 158, 163, 162, 161, 160, 159, 144, 157, 156, 155, 154, 153, 152, 151,
        212, 214, 126, 127, 128, 129, 132, 222, 130, 211, 131, 86, 85, 82, 83, 84,
        210, 87, 90, 91, 92, 93, 94, 209, 221, 81, 28, 204, 34, 33, 35, 29, 30, 202,
         206, 32, 37, 36, 26, 25, 208, 24, 23, 22, 21, 20, 203, 19, 18, 17, 16, 15,
        14, 13, 12, 11, 38, 39, 40, 41, 10, 9, 8, 7, 6, 5, 4, 3, 1]
68          expected_distance = 4123.82
69
70          self.assertEqual(expected_route, generated_route)
71          self.assertEqual(expected_distance, generated_distance)
72
73      def test_run_ihcv_to(self):
74          generated_route, generated_distance = to.run_two_opt_generic(ih.
        run_ihcv_tsp_lib_initial_solution, "berlin52.tsp", display_route=False)[0]
75
76          expected_route = [1, 22, 31, 18, 3, 17, 21, 42, 7, 2, 30, 23, 20, 50,
        16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 39, 40, 38, 15, 4, 25, 6, 5, 24, 48, 46, 37, 36, 35, 34, 44, 1]
77          expected_distance = 8074.56
78
79          self.assertEqual(generated_route, expected_route)
80          self.assertAlmostEqual(expected_distance, generated_distance, delta=0.5)
81
82  if __name__ == "__main__":
83      unittest.main(warnings = "ignore")
```

Listing C.13: Source Code for *test_two_opt.py*

C.14 TEST - Large Neighbourhood Search

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities.Test_Inputs import mock_route_data as mock
4  from TSP_Utilities import tsp_utility_functions as tsp
5  from LargeNeighbourhood import lns
6
7  import unittest, numpy as np
8
```

```
class TestLNS(unittest.TestCase):
    def setUp(self):
        self.test_city_1_nodes = mock.get_test_city_1_nodes()
        self.berlin52_nodes = mock.get_berlin52_nodes()
        self.tsp225_nodes = mock.get_tsp225_nodes()

        self.test_city_1_distance_matrix = mock.get_test_city_1_distance_matrix()
        self.test_city_1_node_index_mapping = mock.get_test_city_1_node_index_mapping()

        self.berlin52_distance_matrix = mock.get_berlin52_distance_matrix()
        self.berlin52_node_index_mapping = mock.get_berlin52_node_index_mapping()

        self.test_city_1_sample_route = [1, 2, 3, 4, 5, 6, 7, 1]

    def test_destroy(self):
        destroyed_route, _ = lns.destroy(self.test_city_1_sample_route, proportion=0.15)
        self.assertTrue(len(destroyed_route) < len(self.test_city_1_sample_route), msg=f"Route has not been destroyed; there are {len(destroyed_route)} nodes in the supposedly destroyed route, while there were {len(self.test_city_1_sample_route)} nodes in the original route.")
        self.assertTrue(all(node in self.test_city_1_sample_route for node in destroyed_route), msg=f"The nodes in the destroyed route are {destroyed_route}, some of which are not inside the original route, which was {self.test_city_1_sample_route}.")

    def test_repair(self):
        reamaining_route, removed_nodes = lns.destroy(self.test_city_1_sample_route, proportion=0.15)
        repaired_route = lns.repair(reamaining_route, removed_nodes, self.test_city_1_distance_matrix, self.test_city_1_node_index_mapping)
        self.assertTrue(all(node in self.test_city_1_sample_route for node in repaired_route))

    def test_nn_lns_integrated_small(self):
        initial_solution = ([1, 3, 6, 4, 2, 7, 5, 1], 29.96)
        initial_route = initial_solution[0]
        generated_route, generated_distance = lns.lns(initial_route, self.test_city_1_distance_matrix, self.test_city_1_node_index_mapping)
```

```
37
38          expected_distance = 23.63
39
40          self.assertAlmostEqual(expected_distance, generated_distance)
41
42      def test_ihcv_lns_integrated_mid(self):
43          initial_solution = ([1, 22, 31, 18, 3, 17, 21, 7, 2, 42, 30, 23, 20, 50,
 16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
45, 32, 49, 37, 46, 48, 24, 5, 6, 25, 4, 15, 38, 40, 39, 36, 35, 34, 44, 1],
 8105.78)
44          initial_route = initial_solution[0]
45          initial_distance = initial_solution[1]
46          generated_route, generated_distance = lns.lns(initial_route, self.
    berlin52_distance_matrix, self.berlin52_node_index_mapping, max_iterations
    =100000, improvement_threshold=0.000001)
47
48          self.assertEqual(len(generated_route), len(initial_route))
49          self.assertEqual(generated_route[0], initial_route[0])
50          self.assertEqual(generated_route[-1], initial_route[-1])
51          self.assertTrue(generated_distance < initial_distance)
52
53      def test_nn_lns_integrated_large(self):
54          initial_route = [1, 200, 3, 198, 4, 197, 5, 6, 7, 8, 9, 10, 11, 12, 13,
    14, 15, 16, 17, 20, 203, 19, 18, 22, 21, 23, 24, 208, 25, 26, 34, 33, 35,
    30, 202, 206, 31, 216, 219, 217, 77, 78, 79, 80, 81, 95, 209, 94, 93, 92,
    91, 90, 87, 210, 84, 83, 82, 85, 86, 131, 211, 130, 222, 129, 128, 127, 126,
 125, 124, 123, 122, 121, 175, 120, 185, 119, 118, 186, 187, 117, 116, 223,
    115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 220, 104, 103, 102,
    101, 100, 99, 98, 97, 96, 221, 28, 204, 29, 32, 38, 39, 40, 41, 42, 43, 44,
    46, 194, 218, 193, 196, 192, 191, 199, 224, 133, 190, 225, 47, 2, 207, 49,
    51, 57, 56, 55, 52, 53, 54, 70, 71, 72, 73, 74, 75, 76, 69, 68, 67, 66, 65,
    64, 63, 62, 61, 60, 59, 58, 50, 48, 45, 195, 205, 189, 27, 188, 184, 182,
    173, 181, 174, 180, 179, 176, 177, 178, 172, 171, 170, 169, 168, 212, 214,
    151, 150, 149, 152, 153, 154, 155, 156, 157, 144, 143, 201, 142, 141, 140,
    139, 138, 137, 136, 183, 135, 134, 215, 164, 165, 166, 167, 213, 158, 163,
    162, 161, 160, 159, 146, 147, 148, 145, 132, 88, 89, 37, 36, 1]
55          initial_distance = 4829.0
56
57          generated_route, generated_distance = lns.lns(initial_route, tsp.
    compute_distance_matrix(self.tsp225_nodes), mock.
    get_tsp225_node_index_mapping())
58
```

```
59          self.assertEqual(len(generated_route), len(initial_route))
60          self.assertEqual(generated_route[0], initial_route[0])
61          self.assertEqual(generated_route[-1], initial_route[-1])
62          self.assertTrue(generated_distance < initial_distance)
63
64      def test_run_to_lns_integrated(self):
65          generated_route, generated_distance = lns.run_lns_two_opt_ihcv_tsp_lib("
        berlin52.tsp", False)
66
67          expected_route = [1, 22, 31, 18, 3, 17, 21, 42, 7, 2, 30, 23, 20, 50,
        16, 29, 47, 26, 28, 27, 13, 14, 52, 11, 12, 51, 33, 43, 10, 9, 8, 41, 19,
        45, 32, 49, 36, 35, 34, 39, 40, 37, 38, 48, 24, 5, 15, 6, 4, 25, 46, 44, 1]
68          expected_distance = 7887.23
69
70          self.assertEqual(generated_route, expected_route)
71          self.assertAlmostEqual(generated_distance, expected_distance)
72
73  if __name__ == "__main__":
74      unittest.main(warnings = "ignore")
```

Listing C.14: Source Code for *test_lns.py*

C.15 TEST - Genetic Algorithm

```
1  import sys
2  sys.path.append("../")
3  from TSP_Utilities.Test_Inputs import mock_route_data as mock
4  from TSP_Utilities import tsp_utility_functions as tsp
5  from GeneticAlgorithm import genetic_algorithm as ga
6  from TwoOpt import two_opt as to
7
8  import unittest, numpy as np, copy
9
10 class TestGeneticAlgorithm(unittest.TestCase):
11     def setUp(self):
12         self.test_city_1_nodes = mock.get_test_city_1_nodes()
13         self.berlin52_nodes = mock.get_berlin52_nodes()
14         self.tsp225_nodes = mock.get_tsp225_nodes()
15
16         self.original_population_size = ga.POPULATION_SIZE
17         self.original_generations = ga.GENERATIONS
```

```
18          self.original_mutation_rate = ga.MUTATION_RATE
19          self.original_crossover_rate = ga.CROSSOVER_RATE
20
21      def tearDown(self):
22          ga.POPULATION_SIZE = self.original_population_size
23          ga.GENERATIONS = self.original_generations
24          ga.MUTATION_RATE = self.original_mutation_rate
25          ga.CROSSOVER_RATE = self.original_crossover_rate
26
27      def test_generate_route(self):
28          n_nodes = 5
29          route = ga.generate_route(n_nodes)
30
31          self.assertEqual(len(route), n_nodes + 1)
32          self.assertTrue(np.array_equal(np.sort(route[1:-1]), np.arange(2,
    n_nodes + 1)))
33
34      def test_encode_individual(self):
35          route = [1, 2, 3, 4, 1]
36          n_nodes = 4
37
38          individual = ga.encode_individual(route, n_nodes)
39          expected_individual = np.array([[0, 1, 0, 0],
40                                          [0, 0, 1, 0],
41                                          [0, 0, 0, 1],
42                                          [1, 0, 0, 0]])
43
44          self.assertTrue(np.array_equal(individual, expected_individual))
45
46      def test_generate_individual(self):
47          n_nodes = 5
48          individual_encoded, route = ga.generate_individual(n_nodes)
49
50          self.assertEqual(len(route), n_nodes + 1)
51          self.assertEqual(individual_encoded.shape, (n_nodes, n_nodes))
52
53      def test_spawn_new_individuals(self):
54          n_nodes = 5
55          population = [ga.generate_individual(n_nodes) for i in range(10)]
56          n_new_individuals = 2
57
58          new_population = ga.spawn_new_individuals(population, n_new_individuals,
```

```
        n_nodes)

        self.assertEqual(len(new_population), 10)

    def test_initialise_population(self):
        nodes = self.test_city_1_nodes
        n_nodes = len(nodes)

        population = ga.initialise_population(nodes, n_nodes)

        self.assertEqual(len(population), ga.POPULATION_SIZE)

    def test_compute_fitness(self):
        individual_route = [1, 2, 3, 1]
        distance_matrix = np.array([[0, 1, 2],
                                    [1, 0, 3],
                                    [2, 3, 0]])

        fitness = ga.compute_fitness(individual_route, distance_matrix)

        self.assertEqual(6, fitness)

    def test_compute_population_fitness(self):
        population = [([1, 2, 3, 1], [1, 3, 2, 1])]
        distance_matrix = np.array([[0, 1, 2],
                                    [1, 0, 3],
                                    [2, 3, 0]])

        fitness_values = ga.compute_population_fitness(population,
    distance_matrix)

        self.assertEqual(len(fitness_values), 1)
        self.assertEqual(fitness_values[0], 6)

    def test_sort_population(self):
        population = [([1, 2, 3, 1], [1, 3, 2, 1])]
        fitness_values = [6]

        sorted_population, sorted_fitness_values = ga.sort_population(population
    , fitness_values)

        self.assertEqual(sorted_fitness_values[0], 6)
```

```
98
99      def test_select_parent_binary_tournament(self):
100         population = [ga.generate_individual(5) for _ in range(10)]
101         distance_matrix = np.random.rand(5, 5)
102
103         parent = ga.select_parent_binary_tournament(population, distance_matrix)
104         self.assertIsInstance(parent, tuple)
105         self.assertEqual(len(parent), 2)
106
107     def test_fill_chromosome(self):
108         parent = np.array([1, 3, 2, 5, 4, 1])
109         crossover_point_1 = 1
110         crossover_point_2 = 3
111         offspring = np.array([1, -1, -1, 2, -1, 1])
112
113         expected_filled_offspring = np.array([1, 1, 3, 2, 2, 1])
114         ga.fill_chromosome(offspring, parent, crossover_point_1,
    crossover_point_2)
115
116         self.assertTrue(np.array_equal(offspring, expected_filled_offspring))
117
118     def test_crossover(self):
119         parent_1_route = np.array([1, 2, 3, 4, 1])
120         parent_2_route = np.array([1, 4, 3, 2, 1])
121
122         (offspring_1_route, offspring_2_route) = ga.crossover(parent_1_route,
    parent_2_route)
123
124         self.assertEqual(len(offspring_1_route), len(parent_1_route))
125         self.assertEqual(len(offspring_2_route), len(parent_2_route))
126
127         for route in [offspring_1_route, offspring_2_route]:
128             self.assertEqual(route[0], route[-1], msg="Offspring Route doesn't
    start and end at start node.")
129
130             unique_nodes = set(route[:-1])
131             expected_unique_count = len(parent_1_route) - 1
132
133             self.assertEqual(len(unique_nodes), expected_unique_count, msg="
    Offspring Route doesn't contain unique nodes (barring start and end node)")
134
135
```

```
136     def test_mutation(self):
137         original_route = [1, 2, 3, 4, 1]
138         n_nodes = 4
139         generation = 1
140         generations_best_fitness_values = [1000]
141 
142         ga.MUTATION_RATE = 1.0  # Guarantee mutation for testing purposes;
    attempts to mitigate effect of the stochastic nature of the Algorithm
143 
144         has_mutated = False
145         for i in range(100): # Try several times
146             mutated_route = ga.mutation(copy.deepcopy(original_route),
    generation, generations_best_fitness_values, n_nodes)
147             if mutated_route != original_route:
148                 has_mutated = True
149                 break
150             else:
151                 continue
152 
153         self.assertTrue(has_mutated, msg="Did not Mutate, despite the fact that
    Mutation was expected")
154 
155     def test_should_apply_two_opt(self):
156         generations_best_fitness_values = [1000, 995, 995, 990, 990, 985]
157 
158         self.assertEqual(ga.should_apply_two_opt(generations_best_fitness_values
    ), False)
159 
160     def test_compute_elitism_count(self):
161         n_nodes = 100
162         population_size = 750
163         elitism_count = ga.compute_elitism_count(n_nodes, population_size)
164 
165         expected_elitism_count = int(population_size * ga.ELITISM_THRESHOLDS
    [100])
166 
167         self.assertEqual(elitism_count, expected_elitism_count)
168 
169     def test_generate_next_generation(self):
170         n_nodes = len(self.test_city_1_nodes)
171 
172         distance_matrix = tsp.compute_distance_matrix(self.test_city_1_nodes)
```

```
173         nearest_neighbours = to.compute_nearest_neighbours(distance_matrix)
174 
175         population = ga.initialise_population(self.test_city_1_nodes, n_nodes)
176         population_fitness = ga.compute_population_fitness(population,
    distance_matrix)
177         sorted_population, sorted_fitness_values = ga.sort_population(population
    , population_fitness)
178 
179         generations_best_fitness_values = [min(sorted_fitness_values)]
180         new_population = ga.generate_next_generation(sorted_population, 1,
    generations_best_fitness_values, n_nodes, distance_matrix,
    nearest_neighbours, self.test_city_1_nodes)
181 
182         # print(new_population)
183 
184         self.assertEqual(len(new_population), ga.POPULATION_SIZE)
185 
186         # Check that the n most elite individuals are the same in the new gen
187         elitism_count = ga.compute_elitism_count(n_nodes, len(population))
188         for i in range(elitism_count):
189             self.assertTrue(np.array_equal(sorted_population[i][1],
    new_population[i][1]))
190 
191         # Check diversity in new gen
192         unique_individuals_count = len(set([tuple(individual[1]) for individual
    in new_population])) # Obtains number of unique routes in the population
193         proportion_unique_individuals = unique_individuals_count  / len(
    new_population) * 100
194         self.assertTrue(proportion_unique_individuals >= 40, f"New Generation
    isn't adequately diverse; only {np.round(proportion_unique_individuals)}%
    are unique") # Checks that there at least 40% of the population are unique
    individuals
195 
196     def test_run_ga_small(self):
197         ga.POPULATION_SIZE = 300
198         ga.GENERATIONS = 40
199         ga.MUTATION_RATE = 0.35
200         ga.CROSSOVER_RATE = 0.65
201 
202         generated_route, generated_distance = ga.run_ga_generic(file_name="
    test_city_1.xlsx",
203 
```

```
        import_node_data_func=tsp.import_node_data,
204                                                     population_size=
        ga.POPULATION_SIZE,
205                                                     generations=ga.
        GENERATIONS,
206                                                     mutation_rate=ga
        .MUTATION_RATE,
207                                                     crossover_rate=
        ga.CROSSOVER_RATE,
208                                                     display_route=
        False)
209
210         expected_distance = 23.63
211
212         start_node_row = self.test_city_1_nodes[self.test_city_1_nodes['Type']
        == 'Start']
213         start_node_number = start_node_row['Node'].iloc[0]
214
215         self.assertAlmostEqual(expected_distance, generated_distance)
216         self.assertEqual(generated_route[0], start_node_number)
217         self.assertEqual(generated_route[-1], start_node_number)
218
219     def test_run_ga_mid(self):
220         ga.POPULATION_SIZE = 500
221         ga.GENERATIONS = 220
222         ga.MUTATION_RATE = 0.2
223         ga.CROSSOVER_RATE = 0.8
224
225         spreadsheet_name = tsp.convert_tsp_lib_instance_to_spreadsheet("berlin52
        .tsp")
226         generated_route, generated_distance = ga.run_ga_generic(file_name=
        spreadsheet_name,
227
        import_node_data_func=tsp.import_node_data_tsp_lib,
228                                                     population_size=
        ga.POPULATION_SIZE,
229                                                     generations=ga.
        GENERATIONS,
230                                                     mutation_rate=ga
        .MUTATION_RATE,
231                                                     crossover_rate=
        ga.CROSSOVER_RATE,
```

```
232                                                                 display_route=
    False)
233
234         expected_distance = 7544.37
235
236         start_node_row = self.test_city_1_nodes[self.test_city_1_nodes['Type']
    == 'Start']
237         start_node_number = start_node_row['Node'].iloc[0]
238
239         self.assertAlmostEqual(expected_distance, generated_distance)
240         self.assertEqual(generated_route[0], start_node_number)
241         self.assertEqual(generated_route[-1], start_node_number)
242
243
244 if __name__ == "__main__":
245     unittest.main(warnings = "ignore")
```

Listing C.15: Source Code for *test_genetic_algorithm.py*

Appendix D

Requirement-based Evaluation Results

Table D.1: **Requirement-based Evaluation Results:**

Requirement Code	Requirement	Specification
G1	High	✓
G2	High	✓
G3	High	✓
G4	High	✓
G5	High	✓
G6	Medium	✓
A1	High	✓
A2	High	✓
A3	Low	×
A4	Medium	✓
A5	High	✓
N1	High	✓
N2	High	✓
N3	High	✓

Continued on next page

Table D.1 – continued from previous page

Requirement Code	Requirement	Specification
D1	High	✓
D2	High	✓
D3	Medium	✓
V1	High	✓
V2	High	✓
V3	High	✓
V4	High	✓
V5	Low	✓
T1	High	✓
T2	Medium	✓
T3	High	✓
T4	Medium	×
T5	High	✓
T6	High	✓
T7	High	✓