
# The Development and Application of a Hybrid Metaheuristic Clustering Algorithm to the Capacitated Vehicle Routing Problem

***Author: Andrea Vaz de Sousa (721768)***

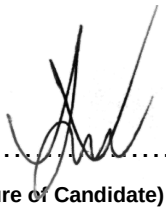
***Supervisor: Dr. Joke Buhrmann***

A research report submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, Johannesburg, in partial fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, November 2023

# Declaration

I declare that this research report is my own unaided work. It is being submitted to the Degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other University.



……..

**(Signature of Candidate)**

…22…. day of ……November……, ……2023……

# Abstract

The Vehicle Routing Problem (VRP) is an important combinatorial optimization problem in the field of operations research that remains a significant challenge for distribution and logistics operations globally. This research is concerned with a relatively simple variation of the VRP referred to as the Capacitated Vehicle Routing Problem (CVRP), and it focuses on the integration of metaheuristics into its solution. A Genetic Algorithm (GA) was selected and integrated into several CVRP solutions with various configurations of the algorithm. Additionally, a hybrid implementation was proposed, which augments the GA by incorporating conventional heuristics to seed the initial population with “good” solutions. The proposed hybrid solution was the best performing solution evaluated and yielded results comparable to the best-known solution for the smaller datasets. However, the solution quality with respect to the best-known solutions decreased with an increase in the size of the problem. This may be attributed to premature termination of the algorithm. Overall, the solutions evaluated were not able to match the best-known solution for each dataset, however successive improvements in the results suggest that GAs are effective at solving the CVRP. Moreover, combining metaheuristics with other methods is also an effective strategy for improving the efficiency of the solution space exploration.

# Acknowledgments

I would like to take this opportunity to acknowledge those who have supported me in the completion of this research.

Firstly, I would like to thank Dr. Buhrmann for her guidance and support throughout this project. Her expertise and feedback have been invaluable in shaping my research report.

Secondly, I am thankful to the School of Mechanical, Industrial and Aeronautical Engineering at University of Witwatersrand for the facilities and resources provided.

Finally, I am grateful for the support structures that my family, friends and partner have provided. Their ongoing support and encouragement are immensely appreciated.

# Table of Content

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **VRP** | Vehicle Routing Problem |
| **CVRP** | Capacitated Vehicle Routing Problem |
| **GA** | Genetic Algorithm |
| **TSP** | Travelling Salesman Problem |
| **VRPTW** | Vehicle Routing Problem with Time Windows |
| **MDVRP** | Multi-Depot Vehicle Routing Problem |
| **VRPB** | Vehicle Routing Problem with Backhauls |
| **VRPSDP** | Vehicle Routing Problem with Simultaneous Delivery and Pick-up |
| **DVRP** | Distance-Constrained Vehicle Routing Problem |

# 1. Introduction

The Vehicle Routing Problem (VRP) is a well-known optimization problem in operations research that consists of solving for the optimal set of routes to service customers given a set of constraints. This chapter serves as an introduction to this research report and provides an overview of its content and purpose. The first section explores the class of problems referred to as combinatorial optimization problems, and specifically discusses a subset of this class referred to as the VRP. The next section expands on the VRP, indicating its relevance and ongoing significance, as well as its practical application. In addition, some popular variations of the basic VRP that are widely defined in literature are also explored. Finally, the motivation and purpose of this research are provided, and the format for the remainder of this report is outlined.

## 1.1 Background

Combinatorial optimization problems are a subset of problems in mathematics that consist of optimizing the result of an objective function within a finite solution space. In many cases, the solution space is too large to be enumerated within a reasonable amount of time, so the goal is to develop more efficient algorithms that explore the feasible solution space to find the optimal solution in a reasonable amount of time (Papadimitriou & Steiglitz, 1982).

This research report focuses on a specific class of combinatorial optimization problem referred to as the VRP, which was first introduced by Dantzig & Ramser (1959) as “The Truck Dispatching Problem”. In this seminal paper, the VRP is concerned with optimizing routes for a fleet of gasoline delivery trucks between a bulk terminal and a set of customer service stations. Broadly, the problem may simply be defined as solving for the least-cost delivery routes from a single depot to a set of geographically dispersed customers subject to a set of constraints (Laporte, 2009).

Due to its significance and ongoing relevance to the daily challenges encountered in the field of distribution management and logistics, many variations of the basic VRP model have been proposed that incorporate a variety of constraints encountered in real life. However, even in its most basic form, the VRP is still an extension of the well-known and well-researched Travelling Salesman Problem (TSP), and as a result, it is also considered NP-hard (Kumar & Panneerselvam, 2012). That is because the solution space grows exponentially with the size of the problem and so it becomes increasingly difficult to solve. As a result, there is no known solution that is solvable in polynomial time (Haque & Magld, 2012).

## 1.2 Application

The VRP continues to be a significant area of research within supply chain management and logistics due to its widespread application. The basic VRP model is a useful starting point and has been extensively researched. However, it is not sufficient to capture the complexities of real-world problems without the inclusion of additional constraints (Jozefowiez, et al., 2008).

Some of these additional constraints include:

- Capacity, which imposes a load restriction on the fleet of delivery vehicles that may not be exceeded on each tour.
- Time windows, which incorporates a time period that a customer may be serviced within.
- Workload balancing, which aims to equally distribute the workload among the available fleet of vehicles.
- Multiple depots, which allows a fleet of vehicles to service customers from multiple locations.

## 1.3 Variations of the VRP

This section lists some popular variations of the VRP defined in literature for a homogeneous fleet. Each variation is an extension of the basic VRP model that incorporates additional constraints. Real-world implementations of the VRP may consider a combination of these constraints and be extended to a heterogeneous fleet of vehicles as well (Jozefowiez, et al., 2008). However, these variations will not be considered further due to their increased complexity.

### *a) Capacitated Vehicle Routing Problem*

The Capacitated Vehicle Routing Problem (CVRP) extends the basic VRP model by including a constraint for the load capacity of the homogeneous fleet of delivery vehicles. The fleet is deployed from a common depot and requires vehicles to service all customers with consideration for the capacity constraints and the distance travelled. The objective is to minimize the total distance travelled without exceeding the load capacity of the vehicles on each tour (Uchoa, et al., 2017).

### *b) Vehicle Routing Problem with Time Windows*

Thangiah (1995) described a variation of the basic VRP model referred to as the Vehicle Routing Problem with Time Windows (VRPTW), which imposes a time constraint on the fleet of delivery vehicles. For this variation, each customer is assigned a time window in which it may be serviced by a vehicle. The vehicle must arrive at the customer within the stipulated time window defined by the start and end times. VRPTW is a challenging problem to solve. It requires servicing customers with consideration for both the capacity constraints of each vehicle and the time constraints of the customers. The objective is to minimize the total distance travelled while ensuring that all customers are served within their respective time windows and without exceeding the load capacity of the vehicles on each tour.

### *c) Multi-Depot Vehicle Routing Problem*

The Multi-Depot Vehicle Routing Problem (MDVRP) was described by Escobar, et al. (2014) as an extension of the basic VRP model. It includes multiple depots among the

customer nodes from which the fleet may be dispersed. MDVRP is a challenging problem to solve and requires servicing all customers with consideration for the distance travelled by each vehicle, the capacity constraints, and the location of each depot. Additionally, it is also a requirement of the problem that each vehicle returns to the same depot from which it was dispatched. The objective of MDVRP is to minimize the total distance travelled without exceeding the load capacity of the vehicles on each tour.

### *d) Vehicle Routing Problem with Backhauls*

The Vehicle Routing Problem with Backhauls (VRPB) is an extension of the basic VRP model which integrates two distinct sets of customers. The first set of customers, the consumers, are serviced through the delivery of a specified quantity of products from the depot. The second set of customers, the suppliers, necessitate the collection and subsequent delivery of a specified quantity of products to the depot. In order to mitigate the need for rearranging loads on the vehicle, it is generally a requirement that all deliveries are completed prior to commencing any collections on a given route. The objective of VRPB is to minimize the total distance travelled without exceeding the load capacity of the vehicles on each tour (Palhazi Cuervo, et al., 2014).

### *e) Vehicle Routing Problem with Simultaneous Delivery and Pick-up*

Dethloff (2001) describes the Vehicle Routing Problem with Simultaneous Delivery and Pick-up (VRPSDP) as an extension of the basic VRP model that integrates both pick-up and delivery demand constraints simultaneously for each customer. This differentiation sets the problem apart from VRPB, as it necessitates that both the pick-up and delivery constraints need to be satisfied when a vehicle services a customer. To ensure that VRPSDP remains consistent with the basic VRP model, each customer node may only be visited once. The objective of VRPSDP is to minimize the total distance travelled without exceeding the load capacity of the vehicles on each tour.

## 1.4 Motivation

The VRP and its variations remain one of the most extensively studied topics in combinatorial optimization due to the challenging nature of the problem, which is considered NP-hard. As variations of the basic VRP model have become increasingly constrained, it has been observed that exact methods, while capable of identifying the optimal solution, are only useful for small datasets. For this reason, VRP solution methods that are based on heuristic or metaheuristic approaches have gained widespread interest. In particular, algorithms such as the Genetic Algorithm (GA) have already proven to be highly effective at solving CVRP to near optimality (Baker & Ayechew, 2003).

This research focuses on the variation of the classical VRP referred to as the CVRP. The objective is to review and compare the impact of integrating a GA at various stages of a CVRP solution. The results were interpreted in terms of the computational requirements and published benchmark solutions available for each of the datasets used.

### 1.4.1 Problem Statement

The challenging nature of the CVRP necessitates that an innovative approach is used to improve solution quality, particularly for larger datasets where conventional methods have had limited success. The exploration of metaheuristic methods, such as the GA, have emerged as a promising technique to solve the CVRP and its variants.

### 1.4.2 Research Questions

The primary research questions are as follows:

- To what extent does incorporating metaheuristic algorithms, such as the GA, at different stages of the CVRP influence both the solution quality and computational effort?
- Additionally, what is the impact of integrating conventional techniques within the GA framework?

### 1.4.3 Research Objectives

The objectives of this research project are as follows:

- Evaluate the effectiveness of integrating a GA at different stages of the CVRP solution.
- Compare the performance of two well-known CVRP solution frameworks, namely direct routing and cluster-first route-second.
- Develop a hybrid approach that integrates conventional clustering techniques within the GA.
- Benchmark the conventional and hybrid approaches against the best-known solutions for each dataset.
- Analyse and interpret the results of each experiment in terms of solution quality and computational effort.

## 1.5 Report Structure

The remainder of this report is structured as follows. Chapter 2 provides an overview of the CVRP and explores some of the solution methods that have been proposed in literature. A detailed review of some clustering techniques that are relevant to the VRP and routing problems in general is provided in Chapter 3. Chapter 4 explores the GA in detail and describes its key features as well as the algorithmic implementation thereof. The research method provided in Chapter 5 describes the model configuration for each experiment evaluated that was benchmarked on the datasets identified from literature. The results of this research are presented and discussed in Chapter 6. Chapter 7 concludes the research project and provides an overview of the potential for future work.

# 2. Capacitated Vehicle Routing Problem

This chapter provides an overview of the CVRP. The first section introduces the class of problem referred to as the VRP, explains its significance, and then expands to the CVRP. Next, the definition of the CVRP is provided followed by the mathematical formulation as an integer linear programming model. Finally, a review and exploration of the existing solution methods that have been proposed in literature for solving the basic VRP and its variations, such as the CVRP, is completed.

## 2.1 Background

The VRP is a well-known and well-researched topic in the field of operations research that has a rich history dating back to the 1950's. Since its introduction, the VRP and its variations have continued to attract the attention of many researchers as a result of its complexity and significant economic importance (Laporte, 2009). Solving the VRP and its variations remains a challenge in the field of distribution management and logistics. While transportation entities and operations worldwide routinely solve these problems on a daily basis, the solutions obtained are usually not optimal.

The VRP and its variations are considered extremely complex combinatorial optimization problems that are computationally difficult to solve. Even the most rudimentary forms of the problem, such as the CVRP, have been extensively researched since the 1960's with limited success in solving larger problems to optimality (Toth & Vigo, 2003).

## 2.2 Definition & Mathematical Formulation

Laporte et al. (2013) provides the following definition for the CVRP:

- Let $G = (V, A)$ be a complete graph where $V = \{0, \ldots, n\}$ is the set of vertices, and $A = \{(i,\ j), (i,\ j) \in V, (i \neq j)\}$ is the set of arcs.
- Vertex $j = 0$ corresponds to the depot, while vertices $j = \{1, \ldots, n\}$ correspond to the set of customers, each with a known non-negative demand $d_j$.
- The associated non-negative travel cost to go from vertex $i$ to vertex $j$ is $c_{ij}$ for each arc $A \in (i,\ j)$. Where $i = j$, it is assumed that the travel cost $c_{ij} = 0$.
- Where the cost matrix is symmetric,$c_{ij} = c_{ji}$ for all $(i,\ j) \in V$ and the triangular inequality, $c_{ij} + c_{jk} \geq c_{jk}$ for all $(i,\ j,\ k) \in V$, is satisfied.
- All customers are served by the fleet of vehicles $P$ which have equal load carrying capacity $Q$.
- Given a subset of all the customers $S \subseteq V$ that make up the mutually exclusive routes, let $d(S) = \sum d_j,\ j \in S$ be the total demand for each route.
- Each customer $j \in V$ must belong to one and only one subset $S$.
- The minimum number of vehicles needed to serve all customers is $\sum d_j / Q$ where $j \in V$.

The VRP is essentially an extension of the classic TSP and may be formulated using a variety of methods. One of the most popular methods is the vehicle flow formulation that presents the CVRP as either a two-index or a three-index integer linear programming model. The aim of the objective function is then to reduce the total cost of the arcs crossed. It is defined in terms of the cost to traverse an arc and a decision variable. The main difference between the two-index and three-index models is that

the three-index model provides for a distinction of which vehicle travelled across a particular arc while the two-index model does not (Laporte, 1992).

This research will consider the simpler two-index model provided by Laporte (1992) which may be represented as follows:

Minimize:

$$\sum_{i \neq j} c_{ij} x_{ij}, \ (i, \ j) \in V \tag{1}$$

Subject to:

$$\sum_{i \in V} x_{ij} = 1, \ \forall \, j \in V, \ j > 0 \tag{2}$$

$$\sum_{j \in V} x_{ij} = 1, \ \forall \, i \in V, \ i > 0 \tag{3}$$

$$\sum_{i \in V} x_{ik} + \sum_{j \in V} x_{kj} = 2, \ (k = 1, \ldots, n) \tag{4}$$

$$\sum_{i \in S} x_{i0} + \sum_{j \in S} x_{0j} = 2, \ \forall \, S \subseteq V \tag{5}$$

$$\sum_{j \in S} d_j \leq Q, \ \forall \, S \subseteq V \tag{6}$$

$$x_{ij} \in \{0, \ 1\}, \ \forall \, i \in V, \ \forall \, j \in V \tag{7}$$

The process of solving the CVRP consists of finding a collection of elementary routes that minimize the overall cost calculated using equation (1) subject to the following constraints:

- Equations (2) and (3) ensures that a vehicle arrives and leaves each customer exactly once.
- Equation (4) ensures that all routes are continuous. This, in combination with equations (2) and (3), necessitates that a customer is serviced by only one vehicle.
- Equation (5) ensures that each route starts and ends at the depot, preventing the formation of sub-tours as a consequence.
- Equation (6) ensures that the sum of the demand for all customers on a particular route does not exceed the load capacity of one vehicle.

## 2.3 Solution Methods

This section presents some of the existing solutions that have been proposed to solve variations of the VRP, which can broadly be classified into four categories: exact, heuristic, metaheuristic, and hybrid methods. These methods are well-researched and have a varied scope of application across different problem sizes and variations of the VRP in literature. Exact methods are known to provide optimal solutions but are limited to smaller datasets due to their computational complexity. Heuristic methods are simple and efficient problem specific methods of exploring the solution space that provide near-optimal solutions. Metaheuristic methods are complex problem independent methods of exploring the solution space that provide near-optimal solutions and are favoured for highly complex problems. Hybrid methods are a relatively new approach that combine the strengths of different techniques to produce high-quality solutions (Laporte, et al., 2013).

### *a) Exact Methods*

Christofides, et al. (1981) proposed one of the earliest exact methods for solving variations of the TSP and VRP in 1981. Their solution formulated the problem as a dynamic program and utilized a state space relaxation to obtain a lower bound for the solution from q-paths by inserting them into a branch and bound algorithm. This technique proved to be effective at solving both the TSP and VRP. It yielded high-quality solutions, but only for problem sizes of 10 to 25 nodes.

Hadjiconstantinou, et al. (1995) expanded on the works of Christofides, et al. (1981) and introduced another exact algorithm for solving the basic VRP. It benefited from a combination of two relaxations of the original problem. The paper presented a method for computing the higher bounds based on an iterative combination of q-paths and k-shortest paths that were embedded into a branch and bound algorithm. The results of this work showed that the basic VRP was solved to optimality for problem sizes up to 50 customers. High-quality results were also obtained for problem sizes of up to 150 customers, although the computation was halted after a 12-hour period.

Baldacci, et al. (2008) proposed an exact algorithm for solving the CVRP based on the set partitioning formulation. The algorithm incorporated a bounding procedure that solved for a near optimal dual solution of the linear programming relaxation by combining three dual ascent methods to reduce the size of the problem. A final dual solution was used to generate a problem that only contained routes that fell between the upper and lower bounds determined in the previous step. The resulting problem was then solved using an integer linear programming solver. The results of this work indicated that optimal solutions were obtained for problem sizes up to 100 nodes, although the larger problems evaluated required significant computational resources.

Exact methods of solution, despite their ability to provide optimal solutions, are not widely adopted due to their practical limitations in solving only small instances of the VRP within a reasonable timeframe, and with limited constraints. This is consequent to the nature of the VRP, which is considered NP-hard, where the solution space grows exponentially with the size of the problem. Given the size and complexity of real-world instances of the VRP, exact methods are often considered an infeasible approach.

### *b) Heuristic Methods*

Dantzig & Ramser (1959) first introduced the VRP to the research community and presented a relatively simple matching heuristic for solving the problem. The algorithm used a matching process to construct complete routes from nodes or partial routes. The matching process was formulated as a set of linear programs that aimed to minimize the total cost while ensuring that all the demands were met without exceeding the capacity of the fleet. The proposed solution was designed to be computationally efficient, however, it was only tested on a single problem that consisted of 12 customer nodes.

Clarke & Wright (1964) developed a simple savings heuristic for solving the basic VRP. Their method incorporated the concept of savings when combining two or more customer routes into one through an iterative process until a single route was obtained. The savings were calculated by comparing the cost of the combined route to the cost of the individual routes. The algorithm started by selecting the closest pair of customers as the first route, after which customers were added one by one based on the customers that generated the highest saving. The algorithm was halted when all

customers were associated with a route and no more savings were possible. The results indicated that the savings algorithm improved on known methods and was effective at solving problem sizes of up to 30 nodes.

Fisher & Jaikumar (1981) were the first to introduce a two-phased cluster-first route-second approach for solving the CVRP. Their approach first formulated the problem as a generalized assignment problem with an objective function that approximated the total cost of servicing customers. This problem was subsequently solved using a simple heuristic. The actual routes were then solved to optimality using exact methods afterwards. The results of this work indicated that the proposed solution outperformed the best-known heuristic methods of the time for problem sizes up to 200 nodes. However, it was noted as being computationally expensive as a result of the exact methods used when calculating the routes.

Heuristic methods are arguably the most widely researched methods for solving variations of the VRP. These methods generally perform a relatively limited exploration of the solution space and employ construction or improvement procedures to produce high-quality solutions. In addition, these methods have also shown reduced computational requirements in comparison to the exact methods. Heuristics have shown to be an efficient solution method for solving variations of the VRP in most practical use cases, and as a result, they are among the most widely adopted.

### *c) Metaheuristic Methods*

Van Breedam (1995) explored the application of a traditional simulated annealing for solving the CVRP. The proposed method adopted a two-phased format, where a simple insertion heuristic was used to generate a set of feasible routes. A simulated annealing-based improvement method was subsequently used to traverse the solution space by exchanging and relocating nodes or sets of nodes between routes. Simulated annealing is a nature inspired algorithm that mimics the physical annealing process, which utilizes a controlled cooling process to obtain the ground state of a material. In simulated annealing, each feasible solution is representative of a material configuration at a specific state and the temperature is used to guide the selection of the next feasible solution (van Laarhoven & Aarts, 1987). For the VRP, this would reflect a new route configuration. The results of this research showed that simulated

annealing was comparable to other metaheuristics. It produced high-quality solutions for the CVRP, and for problem sizes up to 200 nodes. However, high CPU usage remained a drawback of this method (Van Breedam, 1995).

Baker & Ayechew (2003) presented one of the earliest implementations of a GA for solving variations of the VRP. Their solution method adopted a two-phased approach, similar to that of Fisher & Jaikumar (1981), where customers were first assigned to a vehicle before solving for each route. In this case, the GA was used to calculate the assignment of customers to vehicles and the routes were then calculated using a local search algorithm. The fitness of each candidate solution was calculated as the total route distance and the GA iterated through a reproductive process until the stopping criterion was satisfied. This approach was evaluated for variations of the VRP which included both capacity and distance constraints, namely the CVRP and Distance-Constrained Vehicle Routing Problem (DVRP). It was shown that the GA performed comparably to other metaheuristics and produced high-quality results for problem sizes up to 200 nodes.

Toth & Vigo (2003) proposed a metaheuristic algorithm referred to as granular tabu search for solving variations of the VRP. The algorithm was an adaptation of the well-known tabu search algorithm which made use of an effective intensification/diversification method, which may be applied to a variety of graph-theoretic and combinatorial optimization problems. Implementations of tabu search often require large computational resources to perform the thousands of iterations required to obtain a high-quality solution. As a result, several approaches have been investigated in literature to reduce the computing time of the neighbourhood exploration, including random sampling, candidate lists, and parallel implementations (Glover & Taillard, 1993). The proposed granular tabu search algorithm adopted a candidate list approach that restricted the neighbourhood search by focusing on a subset of the most promising solutions. The algorithm produced high-quality solutions with reduced computing times for problem sizes up to 483 nodes. It was evaluated for variations of the VRP which incorporated constraints for capacity and route distance, namely the CVRP and the DVRP (Toth & Vigo, 2003).

Bell & McMullen (2004) applied ant colony optimization techniques to solve the CVRP. Their research explored several configurations of ant colony optimization, which

incorporated candidate solution lists of varying sizes and multiple ant colonies to improve the solution quality. Ant colony optimization is a nature-inspired algorithm that is based on the behaviour of real ants. It utilizes the concept of pheromone trails to store information about the solution space as it is traversed. In the case of the CVRP, the ants represent the vehicles that construct routes to the customers with consideration for the feasibility and pheromone trails, which reflect the arcs that have been explored previously (Yu & Gen, 2010). The results of this research indicated that ant colony optimization was an effective method of solving the CVRP and obtained high-quality solutions for problem sizes up to 150 nodes (Bell & McMullen, 2004).

Metaheuristic methods are a more recent approach for solving variations of the VRP in comparison to heuristic and exact methods, and they have received a growing interest in recent years, particularly for highly constrained variations of the problem. These methods perform a wide and deep exploration of the solution space by utilizing more sophisticated search strategies, memory architectures and recombination techniques to escape local optima and produce high-quality solutions. However, they are subject to increased computational requirements in comparison to heuristic methods.

### *d) Hybrid Methods*

Yu, et al. (2011) proposed a hybrid approach that incorporated both ant colony optimization and tabu search for solving the VRPTW. Their solution approach adopted a multi-phase direct routing solution method where ant colony optimization was used to construct a route, which was augmented by tabu search to find the best neighbouring routes. Tabu search was performed once, and only when a set number of successive ant colony optimization searches did not yield an updated route. The proposed solution was evaluated for a dataset of 56 problems, each with 100 nodes. The results indicated that the hybrid ant colony optimization-tabu search algorithm was effective at generating high-quality solutions and achieved best-known solutions for 41 of the 56 problems evaluated.

Yousefikhoshbakht, et al. (2014) explored another hybrid ant colony optimization method for solving variations of the VRP, which incorporate capacity and distance constraints. Their solution approach incorporated a simple insert and swap algorithm,

to help escape from local optima, with a variation of the ant colony optimization algorithm referred to as the elite ant system. The proposed hybrid elite ant system, in contrast to classical elite ant system, also intentionally exaggerated the pheromone levels causing increased levels on the shortest paths and decreased levels on the longest paths. This increased the emphasis on the shortest and longest paths for every iteration when constructing the routes. The results of this work indicated that the proposed hybrid elite ant system was effective at generating high-quality solutions for both the CVRP and DVRP. It performed well for problem sizes of 50 to 199 nodes and achieved optimality for 6 of the 14 problems evaluated.

Prins (2004) presented a relatively simple hybrid GA implementation for solving variations of the VRP. The proposed method adopted a direct routing solution approach that calculated all routes simultaneously. Where earlier GA implementations adopted a direct translation from the chromosome to the routes, this research proposed a GA that encoded the chromosome as a graph structure and incorporated a hybridized local search procedure as part of the mutation operator. The proposed solution was evaluated on datasets which included capacity and distance constraints, and for problem sizes of 50 to 483 nodes. The results showed that this approach produced high-quality results and achieved optimality for 10 of the 14 problems evaluated in the smaller dataset. It also performed better than other popular methods, such as tabu search, on the larger dataset.

Hybrid methods are a relatively new approach for solving variations of the VRP and are recognized for their ability to combine the strengths of several techniques to produce high-quality results. These methods usually make use of either heuristic or metaheuristics algorithms that are augmented by other algorithms to help navigate the solution space and rapidly converge to and escape local optima.

# 3. Cluster Analysis

This chapter provides an overview of cluster analysis applicable to routing problems. The first section introduces cluster analysis and provides background information on clustering with respect to its significance, as well as listing some of the categories in which clustering techniques can be classified. Next, an overview of hierarchical and partitional clustering techniques is provided, and several popular methods are explored. A comparison of the various clustering methods is completed based on several studies from literature. Finally, various clustering methods that have been used for solving the CVRP and other variations of the basic VRP are listed. All figures provided in this chapter are the authors own interpretation.

## 3.1 Background

Cluster analysis is the generic name for a large set of statistical methods that aim to detect a set of groups, referred to as clusters, within a given dataset. It is characteristic of cluster analysis that the group structure is not known in advance, and so it is a tool which is considered both exploratory and descriptive (Wilmink & Uytterschaut, 1984).

The objective of clustering is to organize data that characterizes a population into an efficient representation by grouping instances into subsets such that similar instances are grouped together, and those which differ are separated into alternate groups. The measure of similarity is often expressed as some form of distance measure, such as the Euclidean distance between points, or in some cases the variance within each cluster (Rokach & Maimon, 2005).

With the growing interest in the application of machine learning globally, it is no surprise that many clustering methods and countless variations have been researched and developed. In general, there are many ways to categorize these methods. Jain (2010) grouped clustering methods into two main categories: hierarchical and partitional. Buhrmann (2016) listed five categories for clustering geographical data: hierarchical, iterative partitioning, graph-based, nearest neighbour, and density-

based. In this research, only the two broadest categories listed by Jain (2010) were considered for simplicity.

## 3.2 Hierarchical Clustering

Hierarchical clustering methods are generally considered deterministic and incorporate either an agglomerative or divisive technique. That is, the algorithm either starts with each data point in its own cluster and iteratively merges the clusters that are most similar, or it starts with all the data points in a single cluster and iteratively divides them into separate clusters. In both scenarios, the process usually runs until the desired number of clusters has been attained (Jain, 2010).

Bührmann (2016) noted that divisive techniques generally have higher time complexity in comparison to agglomerative techniques, and as a result, they are computationally more expensive. For this reason, only agglomerative techniques were considered in this research.

### *a) Single-linkage*

The single-linkage method, also referred to as the nearest neighbour method, determines the similarity of two clusters based on the minimum distance between all nodes of each cluster. The two clusters are merged if the closest node pairs between clusters are nearest to each other (Kopp, 1978 a). Figure 1 illustrates this principle for merging two clusters, each with three nodes. The linkage between the closest node pairs between each cluster is indicated.

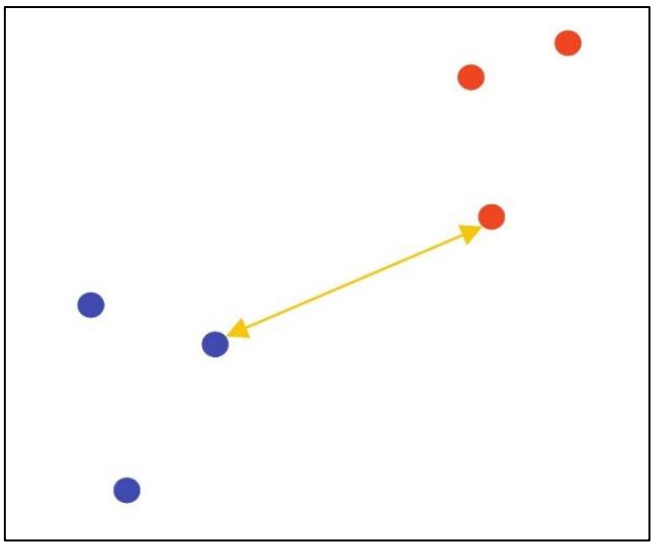


*Figure 1: Single-linkage criteria for merging two clusters.*

### *b) Complete-linkage*

The complete-linkage method, also referred to as the furthest neighbour method, determines the similarity of two clusters based on the maximum distance between all nodes of each cluster. The two clusters are merged if the furthest node pairs between clusters are nearest to each other. In contrast to the single-linkage method, the complete-linkage generates clusters which are more homogeneous (Kopp, 1978 b). Figure 2 illustrates this principle for merging two clusters, each with three nodes. The linkage of the farthest node pairs between each cluster is indicated.

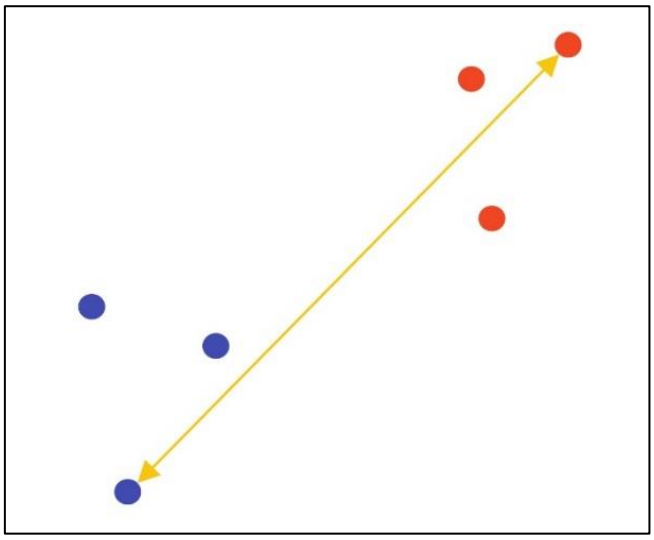


*Figure 2: Complete-linkage criteria for merging two clusters.*

### c) *Average-linkage*

The average-linkage method determines the similarity of two clusters based on the average distance between all the nodes of each cluster. The two clusters are merged if the average distance between the nodes of each cluster is reduced. The average-linkage stands between the single-linkage and complete-linkage and benefits from the stability and homogeneous characteristics of each method (Kopp, 1978 c). Figure 3 illustrates this principle for merging two clusters, each with three nodes. The linkage of all nodes between each cluster is indicated.

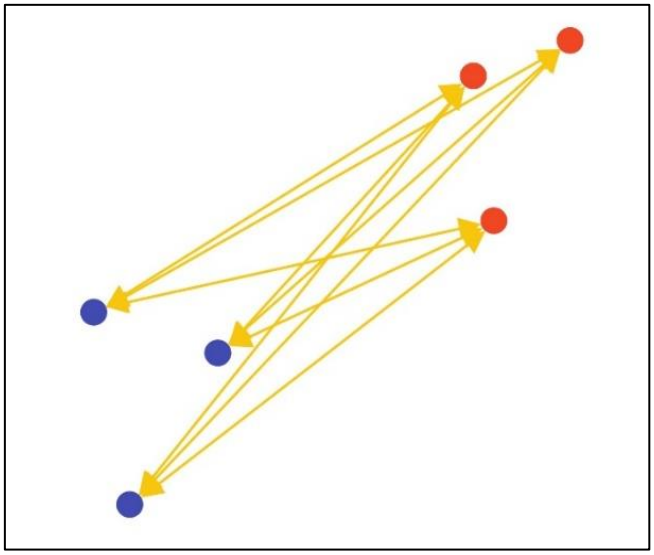


*Figure 3: Average-linkage criteria for merging two clusters.*

### d) *Centroid/Median-linkage*

The centroid-linkage method determines the similarity of two clusters based on the distance between the centroids of each cluster, and with consideration for the relative size. The centroids are simply calculated as the average co-ordinate in each dimension for all points within the cluster. The two clusters are merged if the centroids between clusters are nearest to each other. The median-linkage method is a special case of the centroid-linkage method which does not consider the relative size of each cluster. The selection of one method in favour of the other greatly depends on the problem. While larger clusters carry greater influence for the centroid-linkage method, the median-linkage considers all clusters as being equal (Kopp, 1978 c). Figure 4

illustrates this principle for merging two clusters, each with three nodes. The linkage of centroids between each cluster is indicated.

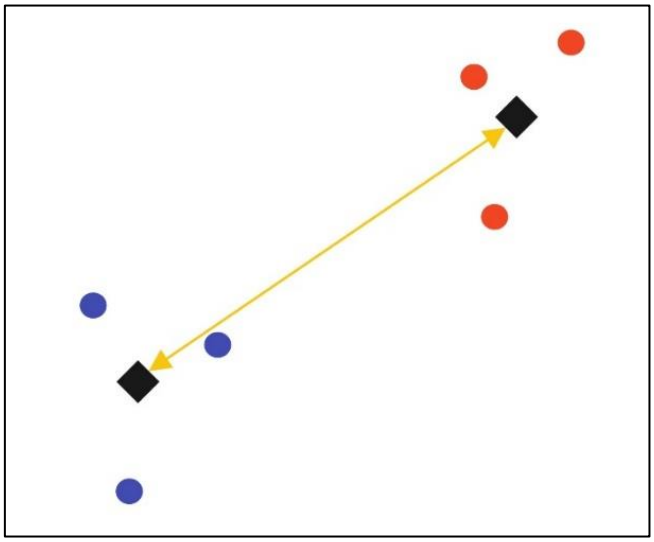


*Figure 4: Centroid/Median-linkage criteria for merging two clusters.*

### *e) Ward's method*

Ward's method, which is arguably the most widely adopted hierarchical method, aims to reduce the loss of information caused by the merger of two clusters. The similarity of the clusters is determined by calculating the sum of the squared distance for all nodes to the centroid of the resulting cluster. The two clusters are merged if the loss of information for two clusters being merged is reduced, that is the calculated sum of the squared distance weighted by the number of nodes is minimized (Kopp, 1978 c). Figure 5 illustrates this principle for merging two clusters, each with three nodes. The linkage of nodes from both clusters to the resulting centroid is indicated.

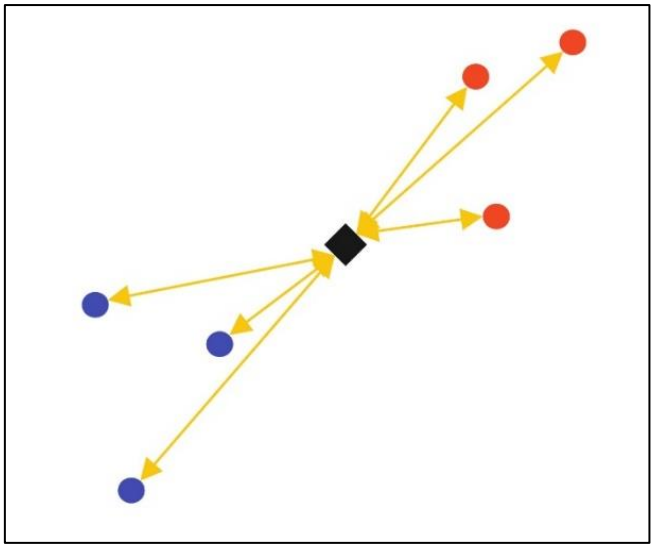


*Figure 5: Ward's method criteria for merging two clusters.*

An implementation of ward's method is provided in Appendix A.1 using the Python programming language.

## 3.3 Partitional Clustering

Partitional clustering methods are generally considered non-deterministic as they often have some degree of inherent randomness in their implementation. The positioning of the seeds, for example, may be presented randomly and ultimately influences the solution quality obtained. Partitional clustering algorithms also do not impose any fixed structure on the data, and the process of identifying clusters is completed simultaneously for all data points through an iterative process that converges to local optima (Omran, et al., 2007).

Unlike hierarchical clustering, partitional clustering methods are initiated with a set number of clusters, after which an iterative process of reassigning the points to the set of clusters is repeated to group similar points together. The similarity of the points is typically a function of Euclidean distance, where points that are closer together are considered more similar. The quality of the clusters identified is usually evaluated by an objective function that determines the similarity of the points within in each cluster based on the cumulative distance between each point and the centre of each cluster.

This is typically represented as the sum of the squared residual distances within each cluster (Bührmann, 2016). The clustering process is usually run until no improvement can be made or a defined exit condition, such as the maximum number of iterations, is met.

### *a) K-means*

The k-means method is one of the simplest and most efficient clustering algorithms in literature and remains among the most widely adopted. The algorithm makes use of a relatively simple centroid-based approach that determines the similarity between nodes and clusters based on Euclidean distance. The algorithm is initialized by selecting the centroids which represent the centre of each cluster. Nodes are then assigned to the cluster with the nearest centroid. Once the clusters are formed, the centroids are recalculated based on the mean of the assigned nodes and the nodes are then reassigned. The reassignment process is repeated until the centroids no longer change or an alternative relaxed convergence criterion is reached (Reddy & Vinzamuri, 2014). Figure 6 illustrates this principle for the evaluation of clusters from six nodes based on two centroids.

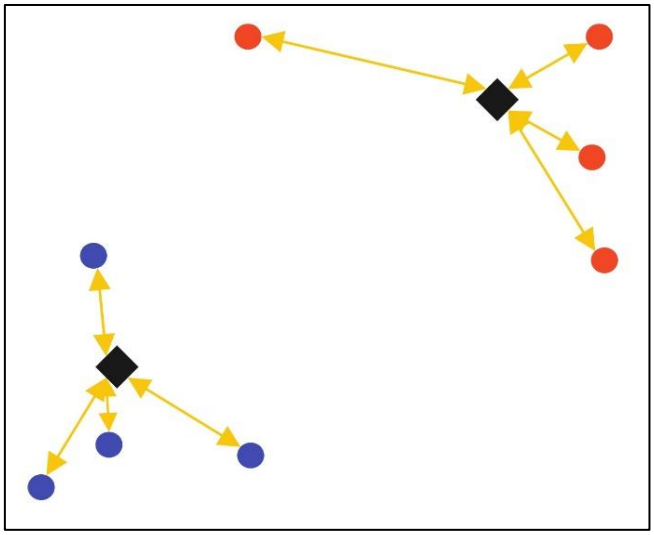


*Figure 6: K-means evaluation illustrated for two clusters.*

An implementation of k-means is provided in Appendix A.2 using the Python programming language.

### *b) K-medoids*

The k-medoids method is a variation of k-means that is more resilient to noise and outliers. Similarly, k-medoids adopts an iterative process that evaluates the similarity of nodes based on Euclidean distance. However, where k-means aims to reduce the sum of the squared errors, k-medoids aims to minimize the absolute error. The algorithm is initialized by selecting the medoids which represent the centre of each cluster. Nodes are then assigned to the cluster with the nearest medoid. Once the clusters are formed, a random non-representative node is selected to replace an existing medoid and the nodes are reassigned. If the selected replacement node decreases the absolute error, the change is accepted and the medoids are updated otherwise another random node is selected. The process is repeated until the medoids no longer change or an alternative relaxed convergence criterion is reached (Reddy & Vinzamuri, 2014). Figure 7 illustrates this principle for the evaluation of clusters from six nodes based on two medoids.

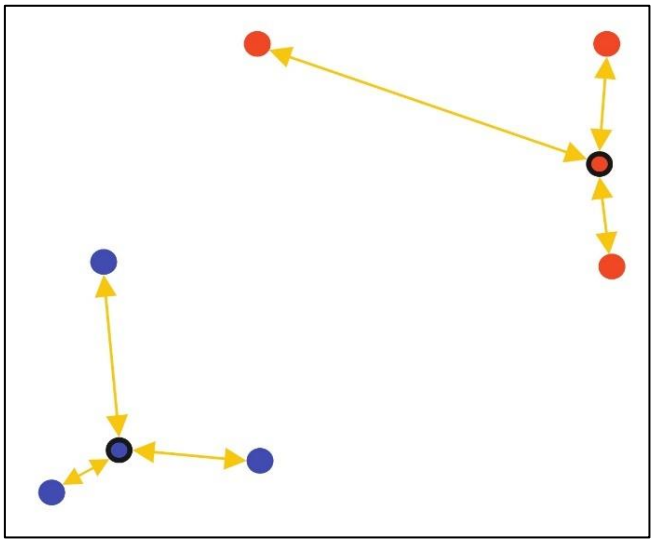


*Figure 7: K-medoids evaluation illustrated for two clusters.*

## 3.4 Comparison of Clustering Methods

Kuiper & Lloyd (1975) completed one of the earliest comparisons of several hierarchical clustering methods. The study compared the quality of the cluster formations of each method for several bivariate and multivariate random datasets generated from Monte Carlo samples. The quality of the clusters was calculated based on the percentage misclassification for all nodes in each dataset. The results indicated that ward's method performed the best.

A similar Monte Carlo comparison of six agglomerative hierarchical clustering methods was completed by Jain, et al. (1986). The study compared the quality of the cluster formations for several univariate random datasets of varying size, which were generated based on uniform and standard normal distributions. The quality of the clusters was calculated using a ratio of the number of within-cluster distances to the number of all distances at most equal to the maximum distance. The maximum distance in this case was considered as the maximum of the largest distance between two points in each successive cluster obtained. The results indicated that the complete-linkage method performed the best.

Bayne, et al. (1980) completed yet another Monte Carlo comparison of several clustering methods that included several partitional clustering and agglomerative hierarchical clustering techniques. The study compared the quality of the cluster formations for six parameterizations of two bivariate normal datasets. The quality of the clusters was calculated based on the probability of misclassification. The results indicated that partitional techniques, in general, outperformed hierarchical techniques, with the k-means method performing the best overall. Considering only the agglomerative methods, the results indicated that ward's method performed the best.

Buhrmann (2016) completed a comparative analysis of several clustering methods as part of a study into the effectiveness of clustering methods in solving various location-routing problems. The study evaluated each method for several medium and large-scale datasets. The quality of the clusters was calculated based on the sum of squared residuals. The results indicated that, except for ward's method, hierarchical clustering

did not perform as well as partitional clustering methods, such as k-median or k-means, which performed well.

Overall, the results indicate that partitional clustering techniques generally performed better than agglomerative techniques. Of the two broad categories evaluated, the best performing clustering methods for each category was generally k-means and ward's method. For this reason, only these methods are considered further in this research.

## 3.5 Application to VRP

The cluster-first route-second framework is a well-known method for solving the CVRP and was first introduced by Fisher & Jaikumar (1981). Essentially, this approach first formulates the problem as an assignment problem, where customers are grouped into clusters. Once the clusters have been calculated, the routes for each cluster are then solved independently. Table 1 lists some examples of this approach to variations of the VRP, as well as the respective clustering method used.

*Table 1: List of clustering methods incorporated into a VRP solution.*

| Author(s) | VRP Variation(s) | Clustering Algorithm(s) |
|---|---|---|
| Alfiyatin, et al. (2018) | VRPTW | k-means |
| Bührmann & Bruwer (2021) | CVRP | k-medoid |
| Nallusamy, et al. (2009) | VRP | k-means |
| Nurprihatin & Regent Montororing (2021) | CVRP | k-means, ward's method |

# 4. Genetic Algorithm

This chapter provides an overview of the GA. Firstly, background information of the GA is provided. Following this, the key features of a typical GA, as well as some well-known variations thereof, are explored, and the algorithmic implementation is described. Ultimately, this research project focuses on the integration of GAs into several CVRP solution methods and so this chapter will also serve to document the development of a simple GA implementation. Finally, various GA configurations used for solving several variations of the VRP are explored. All figures provided in this chapter are the authors own interpretation.

## 4.1 Background

As combinatorial optimization problems become increasingly complex, exact methods of solution become less favourable and have limited application because of the increasing computational efforts required. As a result, the application of metaheuristics has gained widespread interest, particularly for highly constrained problems, which are not feasibly solved to optimality using the known exact methods (Bianchi, et al., 2009).

Metaheuristics are problem independent methods of efficiently exploring a discrete solution space, which is generally too large to be enumerated completely, with the goal of finding a near-optimal solution within a reasonable timeframe (Sorensen, et al., 2017). There are a wide variety of metaheuristics that are well documented in academic literature and there are several characteristics that may be used to classify them, including local search, global search, single solution, population based, nature inspired, and evolutionary amongst others (Bianchi, et al., 2009). This chapter focuses on a class of evolutionary population-based search algorithm referred to as the GA.

The GA was invented by John Holland in the 1960's as a mechanism of translating the process of natural adaptation for use in computer systems. It was developed further by Holland and his students at the University of Michigan throughout the 1960's and

1970's. In 1975, Holland presented a theoretical framework for the GA as a method of moving from one population of chromosomes to another by means of natural selection using genetics-inspired reproductive methods of crossover, mutation, and inversion. The implementation of natural selection was based on fitness such that, on average, fitter chromosomes will produce a greater number of offspring than those which are less fit (Mitchell, 1998).

## 4.2 Main Features

This section lists the key features of a GA and some possible variations thereof. The chromosome and population features are used to store candidate solutions. The fitness function evaluates the effectiveness of each candidate solution, and the selection, crossover and mutation operators facilitate the reproductive process of moving from one generation to another. Integration of these features into a complete GA will be described in section 4.3.

### 4.2.1 Chromosome

The chromosome represents a candidate solution to a given problem which reflects the genetic information that is characteristic of a particular candidate solution. Depending on the implementation, this information may be represented in various forms such as binary, integer and real encoding amongst others. Binary encoding is the earliest form and simply represents the genes of the chromosome as 1s and 0s. Similarly, integer and real encoding represent the genes of the chromosome as an integer or a floating-point number. Although binary encoding is preferred for its simplicity, more complex problems are not easily represented using binary values alone and this facilitates the need for alternative forms, such as integer encoding, which may be better suited to the problem and solution space (Pavai & Geetha, 2017). Certain problems do also exist that may require the use of real encoding however, it is not common in practice. An example of the different chromosome encodings mentioned is provided in figure 8.

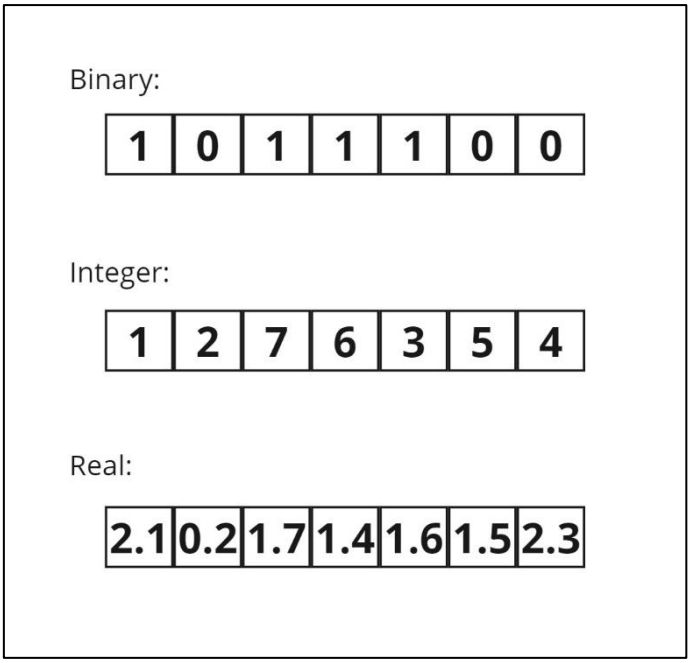



*Figure 8: Example of binary, integer, and real encoded chromosomes.*

### 4.2.2 Population

The population refers to a collection of chromosomes, each representing a candidate solution. It is a crucial component for any GA implementation that has two main aspects, the initial population and population size. The initial population is generally constructed from randomly generated chromosomes to ensure diversity amongst each candidate solution so that the solution space is widely explored. In some cases, the population may also be seeded with known good solutions to improve the rate of convergence. However, this may reduce diversity and limit the ability of the GA to find the global optimum. Population size is largely dependent on the complexity of the problem and also influences diversity and convergence. In general, a larger population increases the exploration of the solution space, however, it is more computationally expensive (Sivanandam & Deepa, 2007). An example population of integer encoded chromosomes is illustrated in figure 9.

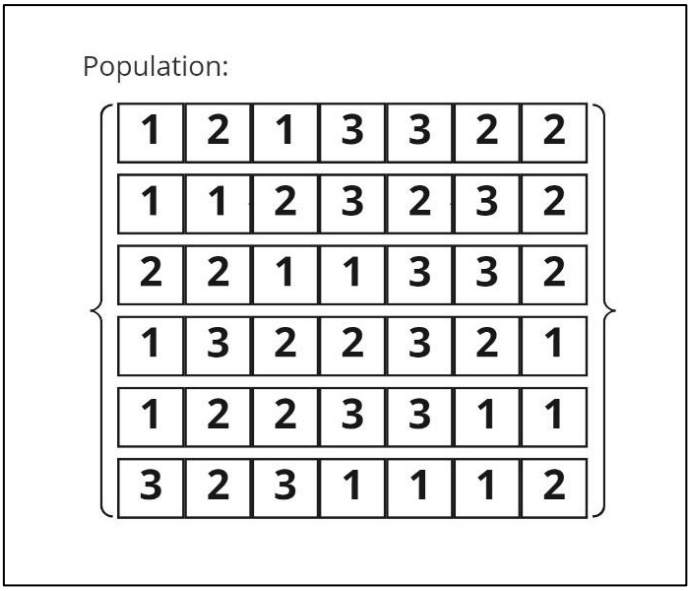



*Figure 9: Example population of integer encoded chromosomes.*

### 4.2.3 Fitness Function

The fitness function is a problem-specific function that is used to interpret the chromosome and evaluate the fitness of each candidate solution. It essentially defines the solution landscape being explored by the GA and assigns a fitness score to each chromosome in the current population, which correlates how well the chromosome solves the problem at hand (Mitchell, 1998).

GAs have a wide scope of application. Maulik & Bandyopadhyay (2000) presented a solution to perform clustering that made use of real encoding for the chromosomes of length $K \times N$, where $K$ is the number of clusters and $N$ is the number of dimensions. Each set of $N$ genes represent the coordinates of the centre of each cluster. The fitness of each candidate solution was calculated as the sum of the distances from each point in the clustering dataset to the nearest centre.

The well-known TSP is also a common application for GAs. Sivanandam & Deepa (2007) and Hussain, et al. (2017) both presented solutions that adopted integer encoding for the chromosomes of length $n$, where $n$ is the number of customers being serviced. The sequence of the route was interpreted directly from the chromosome where the gene represents each customer, and the sequence corresponds to the order

in which the customers were serviced. The fitness was calculated as the total distance of the routes, all of which begin and end at the depot.

### 4.2.4 Selection Operator

Selection refers to the process of identifying and selecting chromosomes in the population for reproduction. The intention of selection is to improve the population fitness, on average, for successive iterations of the GA. Typically, selection is completed with consideration for the fitness of each chromosome in the population as determined by the fitness function (Mitchell, 1998).

While numerous methods have been proposed in literature, this section will explore only the popular roulette wheel, rank, and tournament selection methods. The concept of elitism will also be discussed.

#### *a) Elite Selection*

Elite selection, otherwise referred to as elitism, was first proposed by De Jong (1975). It incorporates an additional parameter in the generation of each population called the generation gap $G$, where $0 \leq G \leq 1$, which specifies the proportion of the chromosomes that are directly carried over from one generation to the next. The remaining chromosomes in the new generation are calculated by means of crossover and mutation. Elitism is an effective way to preserve genetic information and significantly improve the performance of GAs (Sivanandam & Deepa, 2007).

Figure 10 provides a diagram that illustrates the concept of elitism, where a portion of the fittest chromosomes are carried over from the parent population. The balance of the chromosomes in the child population are calculated using conventional reproductive methods.

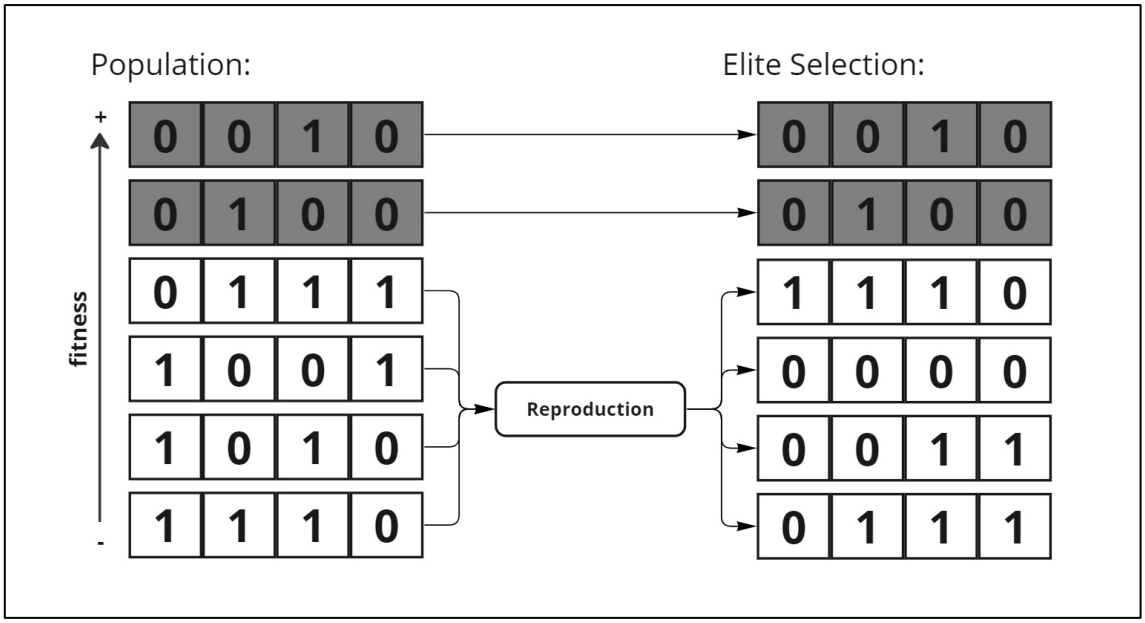



*Figure 10: Elite selection process for an example population.*

An implementation of elite selection integrated into the GA is provided in Appendix B.1 using the Python programming language.

### *b) Roulette Wheel Selection*

Goldberg (1989) presented a simple method of fitness-proportionate selection, where the probability of selecting a chromosome for reproduction is proportional to its fitness value. The process is analogous to a roulette wheel, where the slots in the wheel represent each chromosome in the population, and the size of the slot is weighted according to its fitness. For each selection, a simple “spin” of the wheel returns the chromosome identified for reproduction. The better the fitness, the more likely a given chromosome will be selected.

Mathematically, this may be represented by the following equation, where the probability $p$ may be calculated with respect to the fitness $w$ for each chromosome $i$ in the population $N$ (Lipowski 2011):

$$p(i) = \frac{w_i}{\sum_{i=1}^{N} w_i} \tag{8}$$

Figure 11 provides a pie chart of the roulette wheel selection probabilities that have been calculated using equation (8) for an example population of chromosomes based

on their respective fitness scores. Each slice represents the probability of selection for a particular chromosome, which is directly proportional to its fitness. Where the population has a high fitness variance, this will be reflected in the selection probabilities of the candidate solutions, which are significantly biased toward fitter chromosomes.

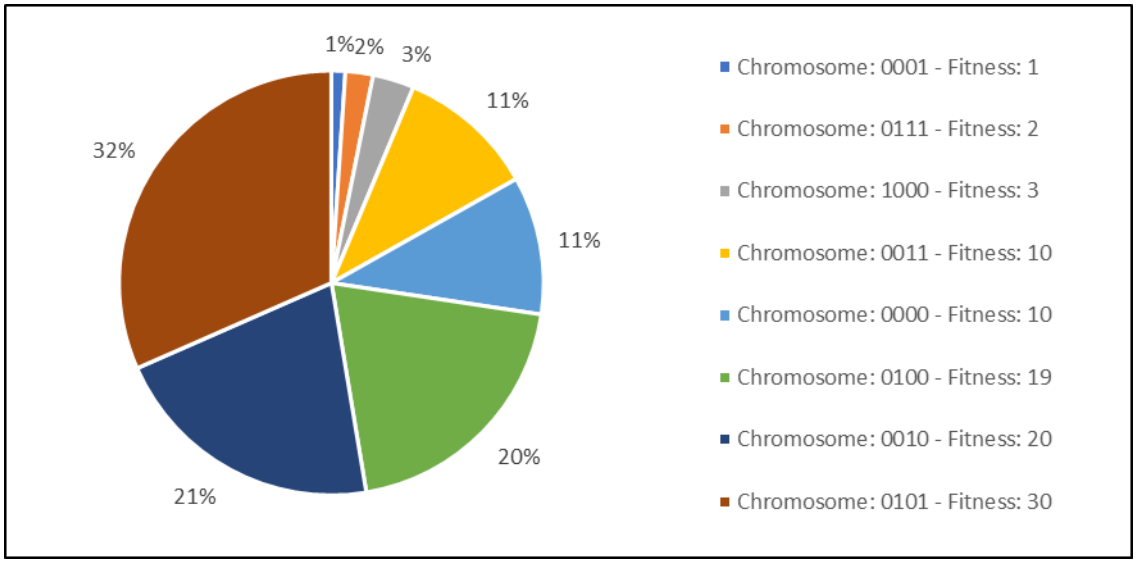



*Figure 11: Roulette wheel selection probabilities for an example population.*

An implementation of roulette wheel selection is provided in Appendix B.2 using the Python programming language.

### *c) Rank Selection*

Rank selection is an alternative method proposed by Baker (1985). It was intended to address the disadvantages of fitness-proportionate methods that cause the solution to converge too quickly, particularly when the fitness of the chromosomes in the population are skewed significantly. In rank selection, the probability of selecting a chromosome for reproduction is proportional to its rank within the population. Chromosomes that receive a higher rank are more likely to be selected for reproduction than those with a lower rank. While the original version proposed by Baker used a linear ranking scheme, other variations have also been proposed that made use of alternate ranking schemes, such as exponential (Mitchell, 1998).

Mathematically, the calculation of probabilities $p$ for linear ranking may be represented by the following equations, where $N$ is the population size and $r_i$ is the rank of the respective chromosome. Typically, $\eta^+$ and $\eta^-$ are the user selected probabilities that must satisfy the conditions $\eta^+ = 2 - \eta^-$ and $\eta^- \geq 0$ (Blickle 1997):

$$p(i) = \frac{1}{N}\left(\eta^- + (\eta^+ - \eta^-)\frac{r_i - 1}{N-1}\right) \tag{9}$$

Koza (1992) defines the gradient of the linear function in terms of a multiplication factor $r_m$. The user assigned probabilities, $\eta^+$ and $\eta^-$, are then calculated by the following equations:

$$\eta^- = \frac{2}{(r_m + 1)} \tag{10}$$

$$\eta^+ = \frac{2r_m}{(r_m + 1)} \tag{11}$$

Substituting equations (10) and (11) into (9), a special case where the value for $r_m$ is selected as $N$ exists, and the calculation of probabilities may be simplified to the following equation:

$$p(i) = \frac{r_i}{\sum_{i=1}^{N} r_i} \tag{12}$$

Figure 12 provides a pie chart of the rank selection probabilities that have been calculated using equation (12) for the same example population of chromosomes shown in Figure 11 based on their respective rank. Each slice represents the probability of selection for a particular chromosome, which is proportional to its rank within the population.

In comparison to roulette wheel selection, rank selection reduces the bias towards fitter chromosomes because the calculation of the selection probabilities is not reliant on the fitness scores. This becomes apparent when examining the example population using both methods. In roulette wheel selection, the top 4 chromosomes collectively account for 84% of the selection probability, whereas in rank selection, they only account for 72% of the selection probability. Similarly, the bottom 3 chromosomes account for 6% and 17% of the selection probability for roulette wheel selection and rank selection respectively. Ultimately, this prevents premature convergence of the

solution as genetic information from lower scoring candidate solutions are more likely to be retained in the population.

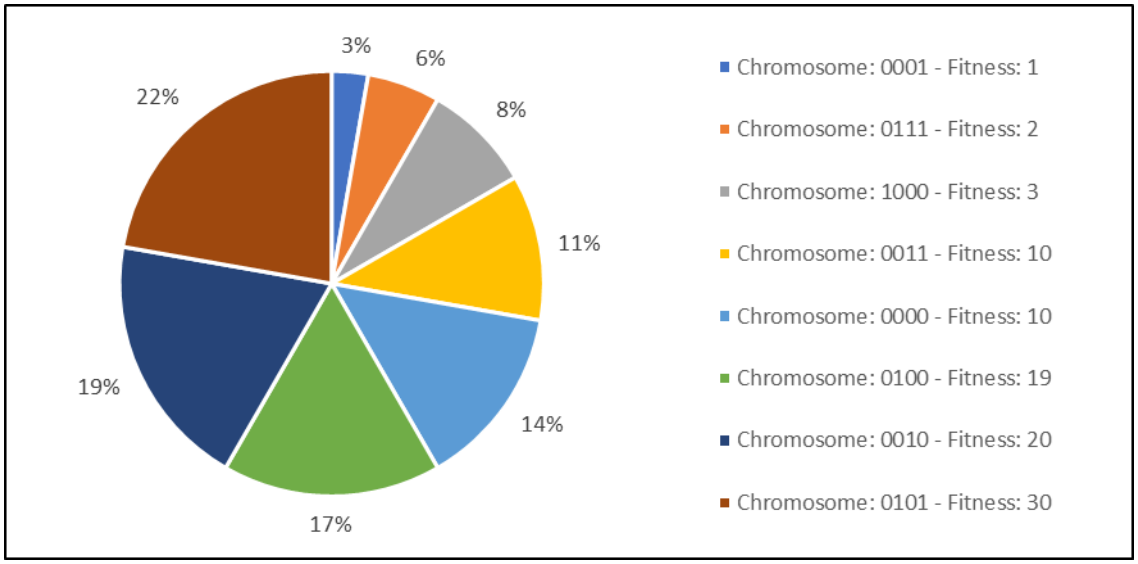



*Figure 12: Linear rank selection probabilities for an example population.*

An implementation of rank selection for the special case where $r_m = N$ is provided in Appendix B.3 using the Python programming language.

### *d) Tournament Selection*

Tournament selection is a relatively simple and time-efficient alternative to the fitness proportionate methods discussed previously. It is also favoured for its ease of implementation. The process consists of randomly choosing a set number of chromosomes to compete against one another in a tournament. For the typical case where two chromosomes are considered, the best chromosome as indicated by the fitness, is then selected with a fixed probability $p$ where $0.5 < p \leq 1$. Variations with a larger size tournament are also common (Goldberg & Deb, 1991).

Figure 13 provides a diagram that illustrates a typical example of a binary tournament, where two chromosomes are selected at random to complete for a position in the reproduction process. In this case, two tournaments are completed to satisfy the selection of a pair of parent chromosomes needed for reproduction. The first tournament indicates that the best chromosome was selected, while the second tournament indicates that the best chromosome was not selected.

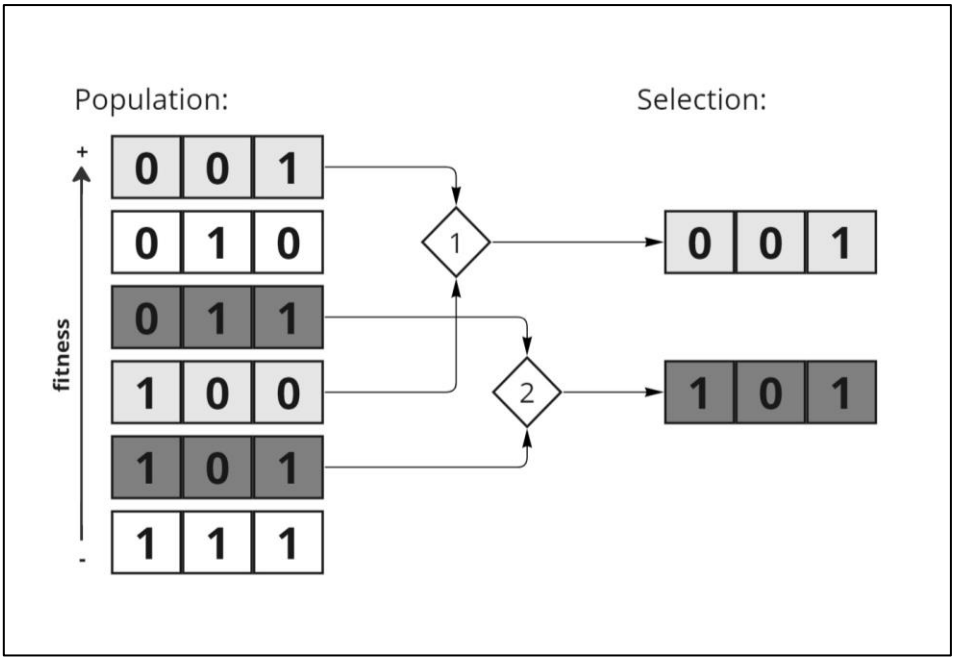



*Figure 13: Binary tournament selection process for an example population.*

An implementation of binary tournament selection is provided in Appendix B.4 using the Python programming language.

### 4.2.5 Crossover Operator

Crossover refers to the process of combining the parent chromosomes to produce the next generation of child chromosomes. This is a distinguishing feature of the GA that mimic's biological recombination (Mitchell, 1998). Typically, crossover is a stochastic process and there are several popular methods that are available.

#### *a) One-Point Crossover*

One-point crossover is the simplest method where the crossover region is determined by randomly selecting a single crossover point in both parent chromosomes. For each offspring, the set of genes before the crossover point are carried over from their respective parents and the set of genes after the crossover point are swapped (Mitchell, 1998).

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes. The randomly selected crossover points are indicated by "|":

$$P_1 = (1\ 0\ 1 \mid 0\ 1\ 1\ 0\ 1\ 1)$$

$$P_2 = (1\ 1\ 0 \mid 1\ 1\ 0\ 0\ 0\ 1)$$

The set of genes before the crossover point in $P_1$ and $P_2$ are repeated in $C_1$ and $C_2$ respectively:

$$C_1 = (1\ 0\ 1 \mid \times \times \times \times \times \times)$$

$$C_2 = (1\ 1\ 0 \mid \times \times \times \times \times \times)$$

The set of genes after the crossover point in $P_1$ and $P_2$ are copied directly to $C_2$ and $C_1$ respectively. The resulting offspring are as follows:

$$C_1 = (1\ 0\ 1 \mid 1\ 1\ 0\ 0\ 0\ 1)$$

$$C_2 = (1\ 1\ 0 \mid 0\ 1\ 1\ 0\ 1\ 1)$$

An implementation of one-point crossover is provided in Appendix B.5 using the Python programming language.

### *b) Two-Point Crossover*

Two-point crossover is essentially an extension of one-point crossover where the crossover region is determined by randomly selecting two unique crossover points in both parent chromosomes. For each offspring, the set of genes before and after the crossover points are carried over from their respective parents and the set of genes between the crossover points are swapped (Mitchell, 1998). The additional crossover point allows the solution space to be explored more extensively and preserves genetic information at both the head and tail of the chromosome (Sivanandam & Deepa, 2007).

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes. The randomly selected crossover points are indicated by "|":

$$P_1 = (1\ 0\ 1 \mid 0\ 1\ 1 \mid 0\ 1\ 1)$$

$$P_2 = (1\ 1\ 0 \mid 1\ 1\ 0 \mid 0\ 0\ 1)$$

The set of genes before and after the crossover points in $P_1$ and $P_2$ are repeated in $C_1$ and $C_2$ respectively:

$$C_1 = (1\ 0\ 1 \mid \times\ \times\ \times \mid 0\ 1\ 1)$$

$$C_2 = (1\ 1\ 0 \mid \times\ \times\ \times \mid 0\ 0\ 1)$$

The set of genes between the crossover points in $P_1$ and $P_2$ are copied directly to $C_2$ and $C_1$ respectively. The resulting offspring are as follows:

$$C_1 = (1\ 0\ 1 \mid 1\ 1\ 0 \mid 0\ 1\ 1)$$

$$C_2 = (1\ 1\ 0 \mid 0\ 1\ 1 \mid 0\ 0\ 1)$$

An implementation of two-point crossover is provided in Appendix B.6 using the Python programming language.

### *c) Uniform Crossover*

Uniform crossover differs from the n-point crossover methods discussed previously, where the crossover region is continuous. Here the crossover points are uniformly distributed throughout the chromosome and each gene has the possibility of being swapped (Syswerda, 1989). Typically, the probability for crossover is set such that each parent is considered equal, however, variations of this method may favour one parent. In this case, the probability for crossover and the crossover regions may be defined such that the offspring chromosomes are likely to have more genetic material from one parent (Vekaria & Clack, 1998).

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes:

$$P_1 = (1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1)$$

$$P_2 = (1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1)$$

Let $R$ represent the random selection of genes that will be swapped (0) and repeated (1) from each respective parent chromosome:

$$R = (1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1)$$

The set of genes that are not selected for crossover in $P_1$ and $P_2$ are repeated in $C_1$ and $C_2$ respectively:

$$C_1 = (1\ 0 \times \times 1 \times 0 \times 1)$$

$$C_2 = (1\ 1 \times \times 1\ \times 0 \times 1)$$

The set of genes that are selected for crossover in $P_1$ and $P_2$ are copied directly to $C_2$ and $C_1$ respectively. The resulting offspring are as follows:

$$C_1 = (1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1)$$

$$C_2 = (1\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1)$$

An implementation of uniform-point crossover is provided in Appendix B.7 using the Python programming language.

### *d) Partially Mapped Crossover*

For some GA implementations, not all possible chromosomes represent a valid solution and so more specialized crossover methods may be applied to avoid violating the constraints of the problem. Goldberg & Lingle (1985) introduced partially mapped crossover in their GA implementation to solve the TSP. As with the two-point crossover method, the crossover region is determined by randomly selecting two unique crossover points in both parent chromosomes. For each offspring, the set of genes between the two crossover points is swapped and the remaining genes are filled in order of appearance in each parent such that the genes are not repeated within each chromosome.

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes. The randomly selected crossover points are indicated by "|":

$$P_1 = (9\ 3\ 6 \mid 2\ 5\ 8 \mid 7\ 4\ 1)$$

$$P_2 = (1\ 2\ 3 \mid 6\ 5\ 4 \mid 7\ 8\ 9)$$

The set of genes between the crossover points in $P_1$ and $P_2$ are copied directly to $C_2$ and $C_1$ respectively:

$$C_1 = (\times\ \times\ \times \mid 6\ 5\ 4 \mid \times\ \times\ \times)$$

$$C_2 = (\times\ \times\ \times \mid 2\ 5\ 8 \mid \times\ \times\ \times)$$

The set of genes before and after the crossover points in $P_1$ and $P_2$ are copied directly to $C_1$ and $C_2$ respectively, except for those which conflict the solution constraint:

$$C_1 = (9\ 3\ \times \mid 6\ 5\ 4 \mid 7\ \times\ 1)$$

$$C_2 = (1\ \times\ 3 \mid 2\ 5\ 8 \mid 7\ \times\ 9)$$

The remaining set of genes in $C_1$ and $C_2$ are carried across from $P_2$ and $P_1$ respectively based on the mapping of missing genes from the one parent to another. For $C_1$, the first missing gene is mapped from $P_2$ to $P_1$ and so 6 translates to 2 being selected. Similarly, for the second missing gene, 4 translates to 8 being selected. The same operation is completed for $C_2$, but the mapping is applied from $P_1$ to $P_2$. The resulting offspring are as follows:

$$C_1 = (9\ 3\ 2 \mid 6\ 5\ 4 \mid 7\ 8\ 1)$$

$$C_2 = (1\ 6\ 3 \mid 2\ 5\ 8 \mid 7\ 4\ 9)$$

An implementation of partially mapped crossover is provided in Appendix B.8 using the Python programming language.

### *e) Order Crossover*

Order crossover was proposed by Davis (1985). Like two-point crossover and partially mapped crossover discussed previously, the crossover region is determined by randomly selecting two unique crossover points in both parent chromosomes. For each offspring, the set of genes between the two crossover points are repeated from each parent and the remaining genes are filled in order of appearance from the other parent such that the genes are not repeated within each chromosome.

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes. The randomly selected crossover points are indicated by "|":

$$P_1 = (9\ 3\ 6 \mid 2\ 5\ 8 \mid 7\ 4\ 1)$$

$$P_2 = (1\ 2\ 3 \mid 6\ 5\ 4 \mid 7\ 8\ 9)$$

The set of genes between the crossover points in $P_1$ and $P_2$ are copied directly to $C_1$ and $C_2$ respectively:

$$C_1 = (\times\ \times\ \times \mid 2\ 5\ 8 \mid \times\ \times\ \times)$$

$$C_2 = (\times\ \times\ \times \mid 6\ 5\ 4 \mid \times\ \times\ \times)$$

The set of genes before and after the crossover points in $P_1$ and $P_2$ are copied to $C_2$ and $C_1$ respectively. The sequence of the genes is determined based on the order of appearance from the last crossover point, omitting those which already appear in the offspring. For $C_1$, the sequence is 7 → 8 → 9 → 1 → 2 → 3 → 6 → 5 → 4. After removing 2 → 5 → 8, the sequence becomes 7 → 9 → 1 → 3 → 6 → 4. Similarly, the same operation is completed for $C_2$. The resulting offspring are as follows:

$$C_1 = (3\ 6\ 4 \mid 2\ 5\ 8 \mid 7\ 9\ 1)$$

$$C_2 = (3\ 2\ 8 \mid 6\ 5\ 4 \mid 7\ 1\ 9)$$

An implementation of order crossover is provided in Appendix B.9 using the Python programming language.

### *f) Cycle Crossover*

Oliver, et al. (1987) proposed cycle crossover in their paper, which explored various permutation crossover operators. In this method, every gene is a direct crossover from one of the parents. The process of selecting the repeated genes from one parent is based on the corresponding gene indicated in the other parent. The cycle is completed when no valid selection can be made, and the remaining positions are swapped with the genes from the other parent.

Consider two parent chromosomes $P_1$ and $P_2$ that go through crossover and let $C_1$ and $C_2$ be the resulting offspring chromosomes:

$$P_1 = (9\ 5\ 6\ 2\ 1\ 8\ 7\ 3\ 4)$$

$$P_2 = (3\ 4\ 6\ 8\ 7\ 9\ 1\ 2\ 5)$$

The set of genes that are repeated from $P_1$ and $P_2$ to $C_1$ and $C_2$, respectively, are identified using the cyclic process, which identifies genes from one parent based on the corresponding gene in the other parent. For $C_1$, starting with position 1 in $P_1$, the cycle yields the following 9 → 3 → 2 → 8 → 9. The cycle ends because 9 is not valid and only 9 → 3 → 2 → 8 will be copied directly from $P_1$. The same exercise is completed for $C_2$ and yields the following:

$$C_1 = (9 \times \times 2 \times 8 \times\ 3 \times)$$

$$C_2 = (3 \times \times 8 \times 9 \times 2 \times)$$

The remaining set of genes in $C_1$ and $C_2$ are copied directly from $P_2$ and $P_1$ respectively. The resulting offspring are as follows:

$$C_1 = (9\ 4\ 6\ 2\ 7\ 8\ 1\ 3\ 5)$$

$$C_2 = (3\ 5\ 6\ 8\ 1\ 9\ 7\ 2\ 4)$$

An implementation of cycle crossover is provided in Appendix B.10 using the Python programming language.

### 4.2.6 Mutation Operator

Mutation refers to the process of diversifying the population from one generation to the next by altering a chromosome from its original state. The process typically occurs after crossover and the intention is to prevent the population from converging to local optima. It should however be noted that, while mutation does encourage exploration of alternative areas of the solution space, it can limit the rate at which the GA converges (Sivanandam & Deepa, 2007). Several popular mutation methods are available, some of which are discussed further in the sections that follow.

#### *a) Flip Mutation*

Flip mutation is commonly used for chromosomes that are binary encoded and is arguably the simplest method of mutation. The process is completed using a randomly generated mutation chromosome, which identifies the mutation points in the original chromosome with a set probability. For all the mutation points identified, the value of the corresponding gene in the chromosome is then flipped from 0 to 1 or 1 to 0 depending on the current value (Soni & Kumar, 2014).

Consider the input chromosome $I$ that is subject to mutation.

$$I = (1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1)$$

Let $R$ represent the random selection of genes that are unchanged (0) and inverted (1) for the chromosome:

$$R = (0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$$

Application of $R$ to $I$ yields the following output chromosome $O$:

$$O = (1\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 1)$$

An implementation of flip mutation is provided in Appendix B.11 using the Python programming language.

### *b) Swap Mutation*

Swap mutation is yet another relatively simple method of mutation however it is more commonly used in chromosomes that are not binary encoded. Here two genes are selected randomly from the chromosome, after which the mutation process is performed by simply swapping the values of the genes that were selected. This method is a popular choice for use in conjunction with permutation crossover methods (Soni & Kumar, 2014).

Consider the input chromosome $I$ that is subject to mutation.

$$I = (9\ 4\ 6\ 2\ 7\ 8\ 1\ 3\ 5)$$

Let $R$ represent the random selection of genes that are unchanged (0) and swapped (1) for the chromosome:

$$R = (1\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0)$$

Application of $R$ to $I$ yields the following output chromosome $O$:

$$I = (1\ 4\ 6\ 2\ 7\ 8\ 9\ 3\ 5)$$

An implementation of swap mutation is provided in Appendix B.12 using the Python programming language.

## 4.2.7 Generation

In the context of the GA, a generation represents the completion of a single cycle of the algorithm to move from one population to another. It is initiated using the current population, which undergoes fitness evaluation to determine the fitness score for each candidate solution. Next, the selection of the parent chromosomes from the current population is completed. Following this, the parent chromosomes go through the reproduction process, which utilizes the crossover and mutation operations to generate the next population of chromosomes, known as the offspring (Mitchell, 1998). Figure 14 provides a flowchart that highlights the key stages in this process.

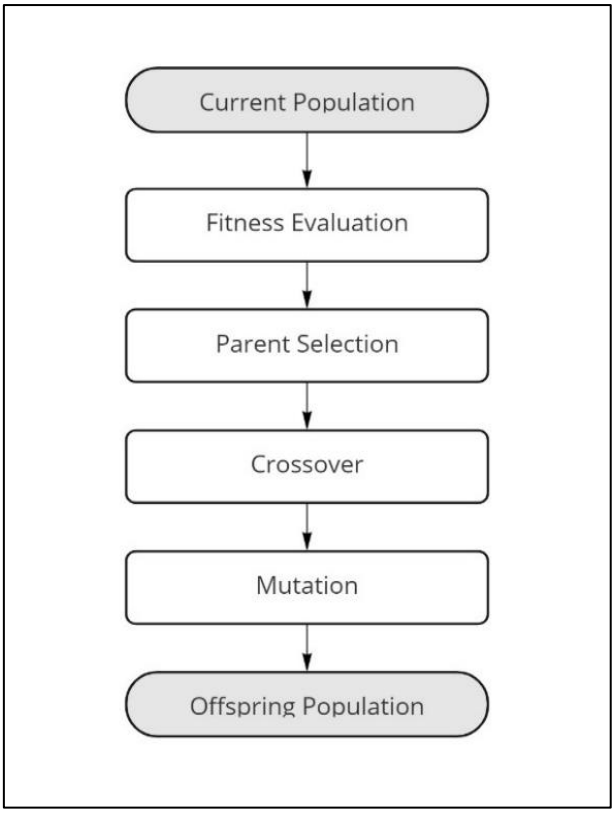



*Figure 14: Flowchart of a complete generation cycle for a typical GA (Mitchell, 1998).*

## 4.3 Algorithmic Implementation

The algorithmic implementation of a typical GA begins by randomly generating an initial parent population, after which the algorithm iterates through the generation cycle. For each iteration of the generation cycle, an offspring population is produced, and the exit condition of the algorithm is evaluated. If the exit condition is satisfied, the algorithm returns the best chromosome found, otherwise the offspring population replaces the parent population, and the next generation cycle is completed. Typically, the exit condition is defined in terms of the maximum number of generation cycles that the algorithm will iterate or a stalled rate of improvement in the fitness of the population from one generation to the next (Sivanandam & Deepa, 2007). Figure 15 provides a flow chart that illustrates this process.

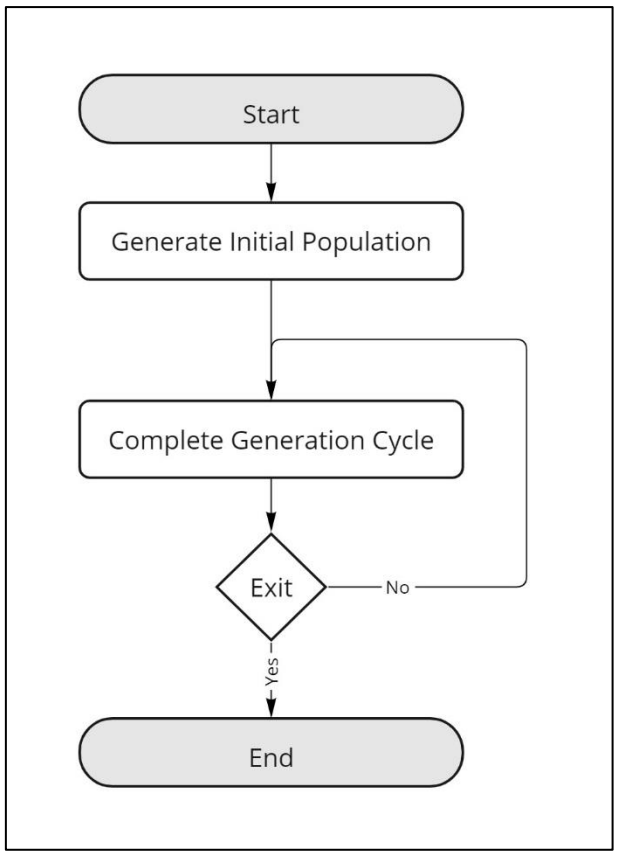



*Figure 15: Flowchart of the algorithmic implementation for a typical GA (Sivanandam & Deepa, 2007).*

An implementation of a simple GA is provided in Appendix B.1 using the Python programming language.

## 4.4 Application to VRP

GAs are a popular method of solving combinatorial optimization problems, such as the CVRP and other variations of the basic VRP. The versatility of metaheuristics, in general, means that the scope of integration for GAs into various VRP solution methods is wide. Table 2 lists some GA implementations for solving variations of the VRP from literature.

*Table 2: List of GA based solution methods for variations of the VRP.*

| Author(s) | Description | GA Configuration |
|---|---|---|
| Baker & Ayechew (2003) | The proposed solution integrated a GA into a conventional cluster-first route-second framework for solving the DVRP and CVRP. Here, a GA was used to calculate the assignment of customers to groups before calculating the routes. | Various GA implementations were explored. The best performing pure GA configuration made use of integer encoded chromosomes, binary tournament selection, two-point crossover, and swap mutation. |
| Biswas (2017) | A GA was integrated into a VRPTW direct routing solution format. In this case, the chromosome was evaluated such that the order of the genes corresponded to the route sequence, and the value corresponded to the customer node. | The GA configuration made use of integer encoded chromosomes, elitism and tournament selection, a problem specific cost-route crossover technique, and insertion mutation. |
| Prins (2004) | A framework to solve the DVRP and CVRP was presented in this report. The proposed direct routing solution incorporated a simple hybrid GA implementation with a local search mutation algorithm. | The GA configuration made use of integer encoded chromosomes, binary tournament selection, order crossover, and traditional mutation was replaced with a local search algorithm. |
| Geiger (2001) | The paper presented a general approach for utilizing GAs to solve multi-objective problems. Specifically, a GA was used to solve VRPTW by integrating it into a direct routing solution format. | The GA configuration made use of integer encoded chromosomes, roulette wheel selection, order crossover, and swap mutation. |

# 5. Research Method

This chapter provides an overview of the research method applied in this project. The first section provides an overview of the solution structure and framework in terms of the data preparation, model configuration and evaluation criteria. Following that, the datasets that have been selected to benchmark the experiments are presented. The final section describes the experiments and the configuration of the algorithms that are integrated into the CVRP solutions.

## 5.1 Framework

Figure 16 provides a flowchart which illustrates the framework that was used to integrate and evaluate the CVRP solutions of each experiment. The process consisted of three main steps necessary to compute the results for each dataset evaluated. The first step involved importing and preparing the data. The second step applied the CVRP solution being evaluated to calculate the routes for the given input data. Finally, the third step recorded the results and returned the route sequence and metrics of the experiment.

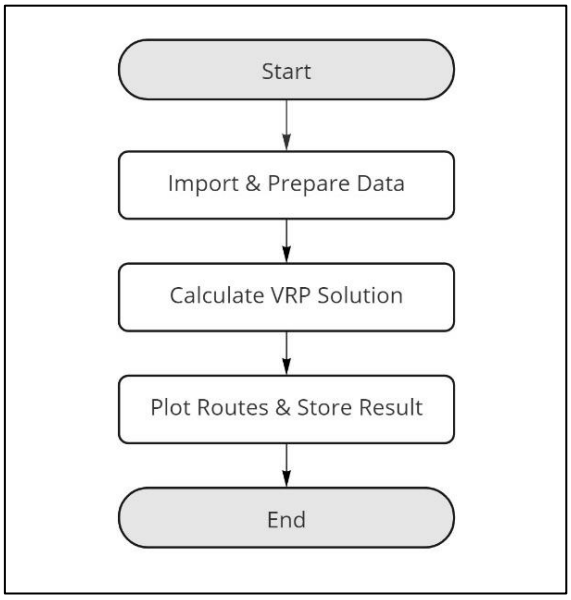



*Figure 16: Flowchart of the CVRP solution framework.*

### 5.1.1 Data Preparation

The datasets that were used in this research are available in a consistent format that contain the following fields:

**Customer**

- Two-dimensional Euclidean space co-ordinates (x, y)
- Demand (scalar)

**Depot**

- Two-dimensional Euclidean space co-ordinates (x, y)

**Vehicle**

- Capacity (scalar)

Prior to evaluating the experiments, each dataset was subject to a data preparation step to normalize and translate the data into a standard format for the model. This process entailed converting the input into a consistent data structure and shifting the origin of the Euclidean space such that the depot is centred at the origin.

### 5.1.2 Route Calculation

The calculation of routes was aligned to the classic CVRP interpretation, which approximates the cost of travelling from one customer to the next as the Euclidean distance between the two points. Other costs such as fixed vehicle costs were not considered. The objective was to minimize the total cost of servicing all customers (equation 1), while satisfying all the constraints of the problem (equations 2 to 7). Only solutions that met all constraints were considered.

The model was subject to the following constraints:

- All customers must be served exactly once and only by one vehicle.
- All customer demands must be satisfied.
- All vehicles start and end their respective tours at the depot.

The model was subject to the following assumptions:

- All vehicles are homogeneous and have uniform capacity.

All programs and calculations were executed on the same computer with the following specification:

- Operating System: Windows 10
- Processor: Intel Core i7-9750H (6 Core, 4.50 GHz)
- Memory: 16GB

### 5.1.3 Results

The following metrics were recorded for each experiment:

- Computing times for executing each model.
- Total cost/distance of the resulting tours (Equation 1).
- Sequence of the calculated routes.

Figure 17 is an example illustration of the routing sequence that provides a visual representation of the solution.

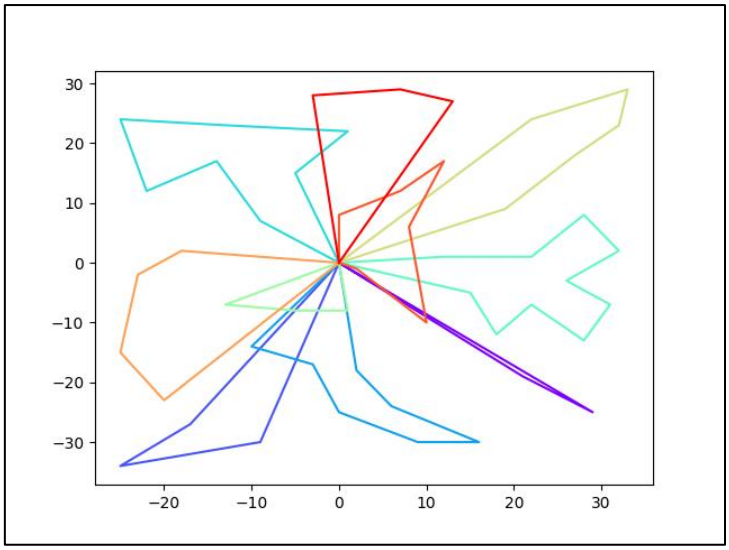



*Figure 17: Example solution output of the route sequences.*

### 5.1.4 Verification & Validation

Verification of the algorithms for each component of the solution was completed using unit testing, where the output was compared to a known result for a small-scale problem set. This allowed for easier debugging of the source code and ensured that the behaviour for each component of the solution was as expected.

Validation of the models was completed by following a consistent evaluation of the results with respect to the conventional CVRP interpretation for each dataset. Comparisons were made against some simple boundary parameters, which include the minimum number of vehicles, and the maximum route distance. The results for each experiment were deemed reasonable if the required number of vehicles was greater than or equal to the minimum threshold, and the calculated route distance was less than or equal to the maximum threshold. Moreover, solutions were considered favourable if they were in the same order as the best-known solutions. Finally, a visual examination of the generated routes was also performed to supplement the numerical evaluations.

## 5.2 Research Datasets

This section lists the publicly available datasets that were identified to evaluate the quality of the solution for each experiment. The datasets were selected with consideration for factors which include, the number of data points, the distribution of the data points, and the presence of visually distinguishable clusters. Ultimately, five datasets were selected to evaluate each experiment that provide a satisfactory level of coverage for the factors mentioned.

### 5.2.1 Christofides CMT Datasets

Figures 18 and 19 illustrate a subset of the dataset introduced by Christofides, et al. (1979) that was selected to evaluate the experiments. CMT01 is a relatively small dataset that was selected due to the even distribution of its data points across the Euclidean space. CMT12 is a medium sized dataset that was selected for the visually distinguishable clusters within its data points, which could highlight the performance of solutions that incorporate a clustering algorithm.

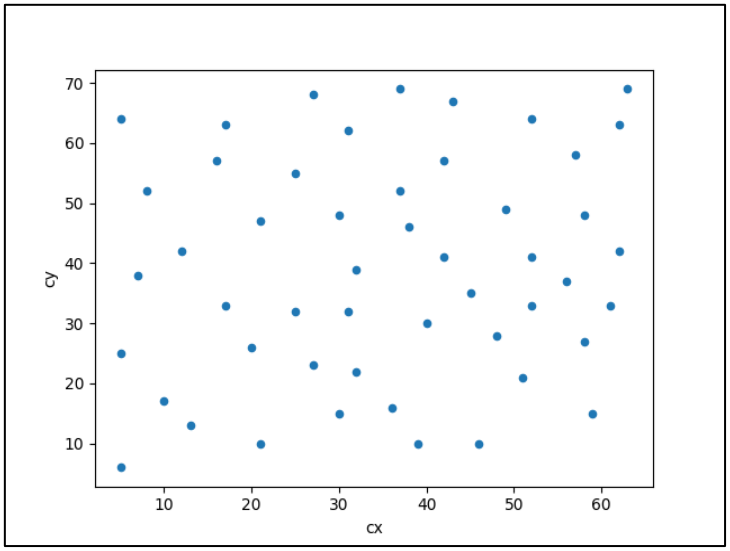



*Figure 18: Customer nodes of the CMT01 dataset.*

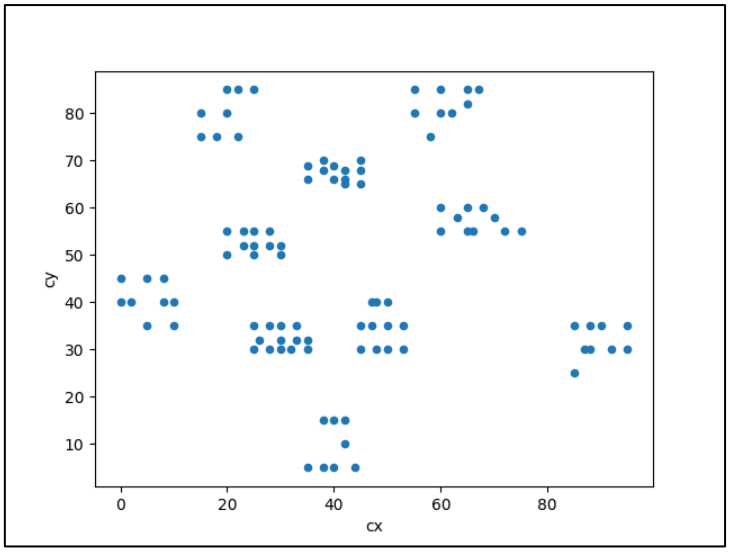



*Figure 19: Customer nodes of the CMT12 dataset.*

### 5.2.2 Fisher F Dataset

Figures 20 and 21 illustrate a subset of the dataset provided by Fisher (1994) that were selected to evaluate the experiments. F072 is a relatively small dataset with data points that are evenly distributed from each other, but not across the entire Euclidean space. F135 is a medium sized dataset where the distribution of the data points is skewed and concentrated around the depot. These datasets were selected to assess the model performance where the clustering is not trivial and requires a balance of distance and demand.

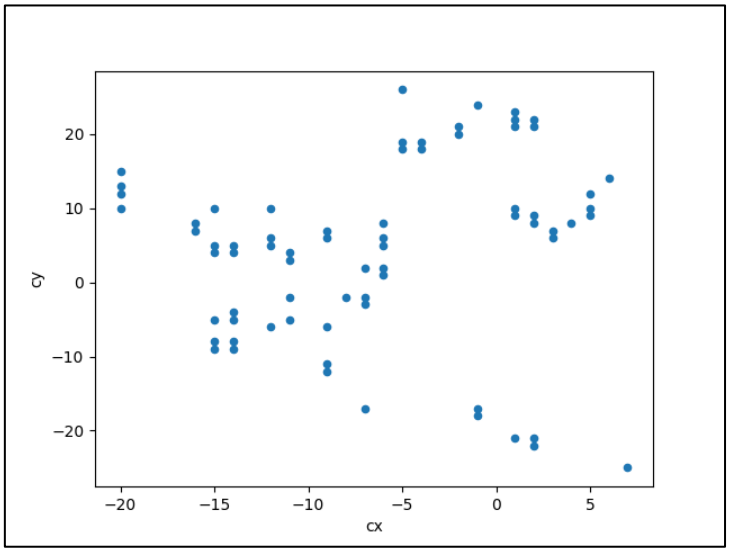



*Figure 20: Customer nodes of the F072 dataset.*

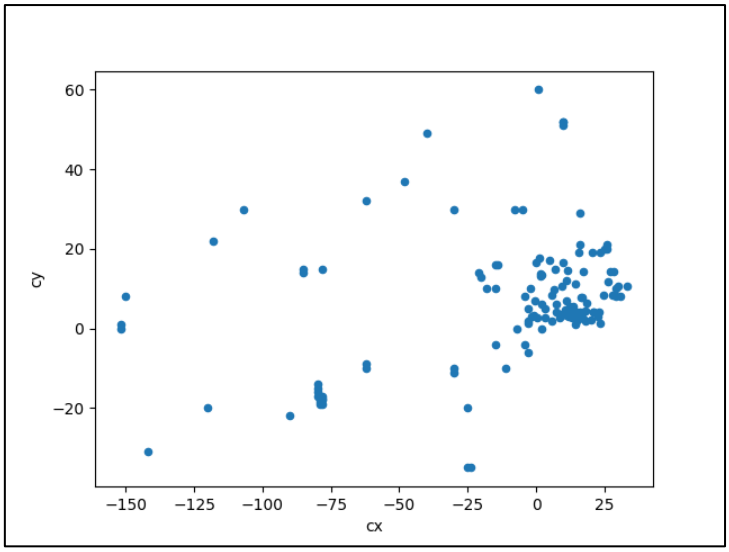



*Figure 21: Customer nodes of the F135 dataset.*

### 5.2.3 Uchoa X Dataset

Figure 22 illustrates a subset of the dataset presented by Uchoa, et al. (2017) that was selected to evaluate the experiments. X491 is a large dataset where the data points are distributed across the Euclidean space with a varied concentration, which may be representative of a typical real-world routing problem.

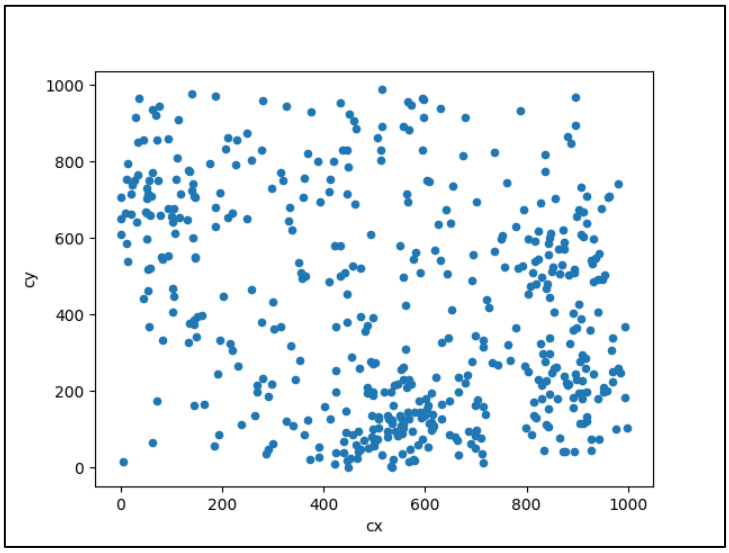



*Figure 22: Customer nodes of the X491 dataset.*

### 5.2.4 Summary of Datasets

Table 3 presents some key parameters of the selected datasets that were used as a reference for evaluating the solutions. It includes the number of customer nodes in the dataset, the minimum number of vehicles required to service all customers, the maximum route distance that could be travelled to service all customers, and the best-known solution. The minimum number of vehicles $K$ required to service the total demand $D$ for $n$ number of customers was determined by $K \geq \frac{\mathrm{D}}{n}$, $K \in \mathbb{Z}$. The maximum route distance was approximated using a simple heuristic that calculates the distances travelled when vehicles service exactly one customer.

*Table 3: Summary of parameters for the CVRP datasets selected.*

| Dataset | Customer Nodes | Vehicle Capacity | Total Demand | Minimum Vehicles Required | Maximum Route Distance | Best-Known Solution |
|---|---|---|---|---|---|---|
| **CMT01** | 50 | 160 | 777 | 5 | 2402.35 | 524.61 |
| **CMT12** | 100 | 200 | 1810 | 10 | 5770.96 | 819.56 |
| **F072** | 71 | *30000* | *114840* | *4* | *2169.33* | *237.00* |
| **F135** | 134 | 2210 | 14620 | 7 | 9768.11 | 1162.00 |
| **X491** | 490 | 428 | 24957 | 59 | 479050.28 | 66483.00 |

## 5.3 Experiment Setup

This section presents the configurations of the CVRP solution methods that were evaluated in each experiment. There were nine experiments across three broad configurations, the conventional direct routing solution, the conventional cluster-first route-second solution, and the proposed hybrid GA cluster-first route-second solution, all of which are explained further in this section.

### 5.3.1 Direct Routing Solution

Figure 23 provides a flow chart that illustrates the process of a conventional direct routing solution for the CVRP. The process received the customer, depot, and vehicle capacity as input and calculated all routes in a single process step. The route sequences calculated were returned as the output.

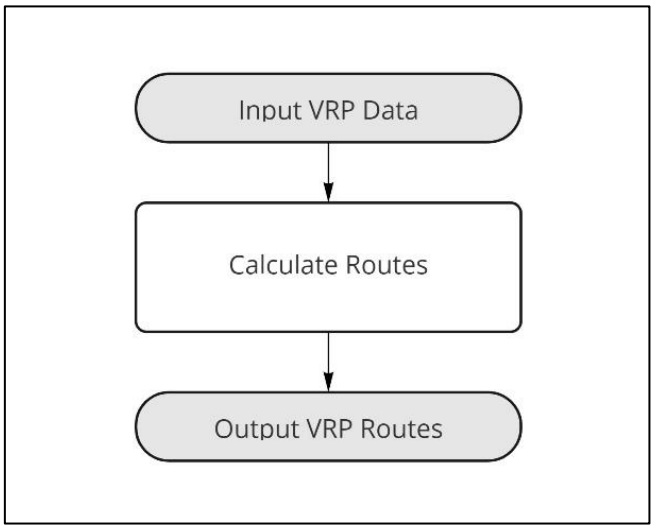



*Figure 23: Flowchart of the direct routing CVRP solution (Geiger, 2001).*

For experiments 1-3, a GA was used to solve the CVRP by calculating the routes directly. Table 4 lists the parameters of the GAs that were used in each of the experiments to calculate the route sequences for all datasets evaluated. All parameters except for the crossover method were kept consistent across the experiments.

It is important to note that the selection of parameters for any GA is not an exact science, and there are no universally defined values for these parameters. The selection of parameters is highly dependent on the specific implementation, problem size, problem complexity, and computational requirements. Consequently, the process of selecting the appropriate parameters was iterative and required upfront experimentation, and refinement to determine a satisfactory set of parameters for the problem at hand.

Epochs represent the number of times the algorithm was run to completion, where the starting population for each epoch was reinitialized. By running multiple epochs, a wide

exploration of the solution space was achieved. Initial testing across the datasets indicated that the solution yielded consistent results for each epoch. Consequently, it was determined that 10 epochs provided a satisfactory level of exploration.

The maximum number of iterations represents the stopping condition of the algorithm. Ultimately, the algorithm cannot run indefinitely, and the number of iterations were limited such that the solution had sufficiently converged before terminating. Initial testing indicated that 50 000 iterations provided a good balance of convergence and computational effort.

The chromosome encoding and length are dependent on the fitness function. For routing, the fitness function was designed such that the genes represent the customer nodes as well as the order in which they are serviced. This will be discussed further later in this section.

Population size and elitism are also parameters that are implementation and problem specific. Thorough testing of the solution across all datasets indicated that a population size of 20 and elitism ratio of 0.1 achieved a satisfactory level of convergence and exploration of the solution space. It was evident that the solution could escape local optima.

The selection, crossover and mutation methods were selected based on the literature provided in chapter 4. Rank selection was selected to reduce the significant bias toward candidate solutions that had a higher fitness. Each of the crossover methods listed in chapter 4 relevant to routing were evaluated across all three experiments. Swap mutation was selected due to the limitations of the chromosome encoding, which are not compatible with flip mutation. As with other parameters, the mutation probability was selected based on initial testing where a satisfactory balance of convergence and exploration was achieved.

*Table 4: Experiment 1-3 GA parameters for solving CVRP route sequences.*

| Feature | Parameter | Experiment 1 | Experiment 2 | Experiment 3 |
|---|---|---|---|---|
| **Iteration** | **Epochs** | 10 | | |
| | **Max Iterations** | 50000 | | |
| **Population** | **Chromosome Encoding** | Integer | | |
| | **Chromosome Length** | *Number of Customers** | | |
| | **Population Size** | 20 | | |
| | **Elitism Ratio** | 0.1 | | |
| **Selection** | **Selection Method** | Rank Selection | | |
| **Crossover** | **Crossover Method** | Partially Mapped Crossover | Cycle Crossover | Order Crossover |
| **Mutation** | **Mutation Method** | Swap Mutation | | |
| | **Mutation Probability** | 0.1 | | |
| ** Indicates variables that have a dependent value.* | | | | |

Figure 24 provides a flowchart that describes the algorithm used to evaluate the fitness of each candidate solution based on the work of Geiger (2001). The algorithm interprets the chromosome and returns the fitness score equivalent to the total distance travelled for all routes. The process of evaluating each chromosome is described by the following steps:

1. Calculate the cost matrix for travelling from one node to the next for all node pairs in the dataset and initialize the first route.
2. Iterate through each gene in the chromosome. Here, the value of the gene corresponds to the customer and the position corresponds to the order of service.
    a. If the demand of the current route does not exceed the capacity of the vehicles:
        i. If the gene corresponds to the first node in the tour:
            - Add the cost of travelling from the depot to the current node to the fitness score.
        ii. Otherwise:
            - Add the cost of travelling from the previous node to the current node to the fitness score.
    b. Otherwise:
        - Add the cost of travelling from the current node to the depot to the fitness score.
        - Start a new route and repeat the current iteration.
    c. If the gene corresponds to the last node in the tour:
        - Add the cost of travelling from the current node to the depot to the fitness score.
3. Return the total fitness score.

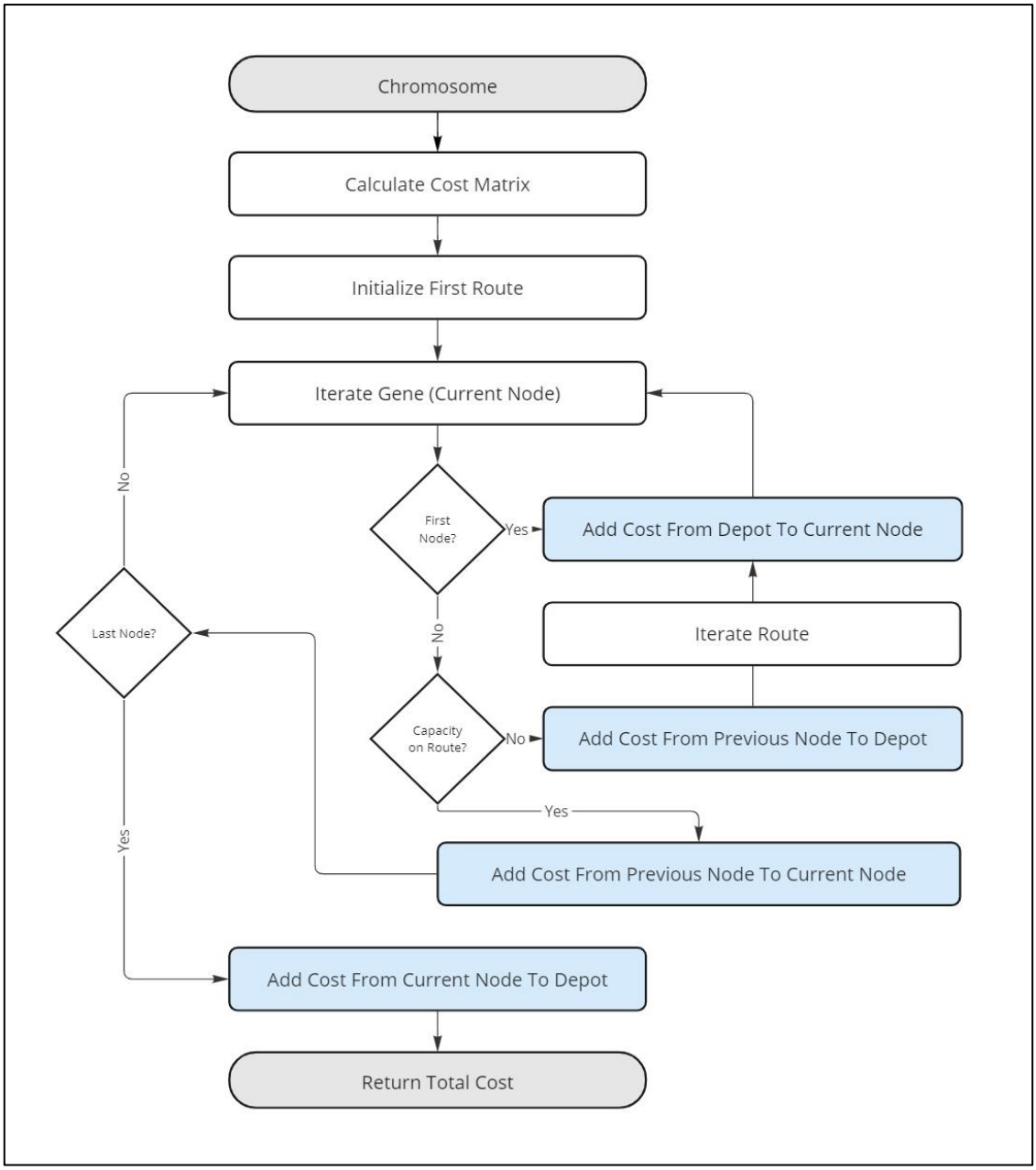



*Figure 24: Flowchart of the fitness function algorithm for routing (Geiger, 2001).*

### 5.3.2 Cluster-First Route-Second Solution

Figure 25 provides a flowchart of the process for a conventional cluster-first route-second CVRP solution. The process received the customer, depot, and vehicle capacity as input, grouped the customers into separate clusters and solved for the routes of each cluster separately. The route sequences were returned as the output.

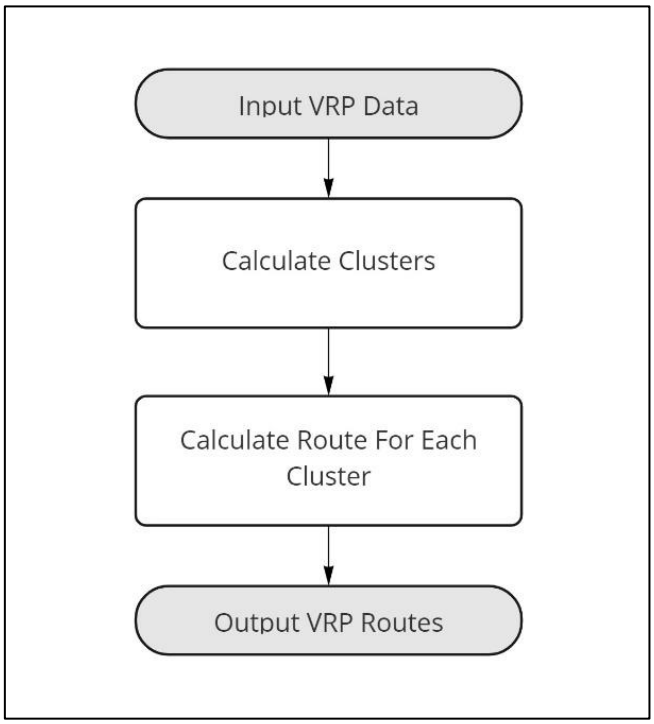



*Figure 25: Flowchart of the cluster-first route-second CVRP solution (Fisher & Jaikumar, 1981).*

For experiments 4 and 5, the assignment of customers to clusters was completed using conventional clustering techniques. Experiments 6, 7, and 8 incorporated a GA in place of the conventional clustering techniques. To ensure that the capacity constraints of the problem were always satisfied, clusters that exceeded the capacity of the fleet were divided and their clusters recalculated. This ensured that only valid solutions were generated. Once the assignment of customers to their respective clusters was completed, the routes were calculated using the GA configuration from experiment 1.

Table 5 lists the parameters of the conventional clustering techniques that were used to calculate the clusters. Preliminary testing of the k-means algorithm across the datasets indicated that consistent results were obtained for each epoch completed. As a result, 10 epochs were deemed to provide a satisfactory level of performance. In contrast, ward's method is a deterministic algorithm that does not require multiple epochs.

The maximum number of iterations was also determined using upfront testing. The convergence rate for the k-means algorithm was high for the relatively small datasets evaluated in this research. 20 iterations were found to provide the desired level of performance. The number of iterations for ward's method is directly related to the targeted number of clusters. This was consequently used as the stopping condition.

*Table 5: Experiment 4-5 parameters for clustering.*

| Feature | Parameter | Experiment 4 | Experiment 5 |
|---|---|---|---|
| **Iteration** | **Epochs** | 10 | 1 |
| | **Max Iterations** | 20 | ** |
| | **Clusters** | *Target** | *Target** |
| | **Clustering Technique** | K-Means | Ward's Method |
| ** Indicates variables that have a dependent value.*<br>*** Indicates variables that are not required.* | | | |

Table 6 lists the parameters of the GAs that were used to calculate the clusters for all datasets. The number of epochs and maximum number of iterations were kept consistent with the first three experiments at 10 and 50 000 respectively. Based on the upfront testing completed, a satisfactory level of performance was maintained for clustering.

As with the previous experiments, the chromosome encoding, and length are dependent on the fitness function. For clustering, integer encoded chromosomes were used where the value of each gene represents the assigned cluster, and the position corresponds to the customer node. This will be discussed further later in this section.

Based on initial testing, the population size and elitism ratio were unchanged from the previous configurations and delivered a satisfactory level of performance. This was also the case for the selection and mutation operators.

The crossover operator was however changed for experiments 6, 7, and 8. Each of the methods evaluated were selected with consideration for the literature presented in chapter 4 and are relevant to clustering.

*Table 6: Experiment 6-8 GA parameters for clustering*

| Feature | Parameter | Experiment 6 | Experiment 7 | Experiment 8 |
|---|---|---|---|---|
| **Iteration** | **Epochs** | 10 | | |
| | **Max Iterations** | 50000 | | |
| **Population** | **Chromosome Type** | Integer | | |
| | **Chromosome Length** | *Number of Customers** | | |
| | **Population Size** | 20 | | |
| | **Elitism Ratio** | 0.1 | | |
| **Selection** | **Selection Method** | Rank Selection | | |
| **Crossover** | **Crossover Method** | One-Point Crossover | Two-Point Crossover | Uniform Crossover |
| | **Crossover Probability** | ** | ** | 0.5 |
| **Mutation** | **Mutation Method** | Swap Mutation | | |
| | **Mutation Probability** | 0.1 | | |
| ** Indicates variables that have a dependent value.*<br>*** Indicates variables that are not required.* | | | | |

Figure 26 provides a flowchart that describes the algorithm that was used to calculate the fitness score for each candidate solution. It is similar to the fitness evaluation process used by Biswas (2017), however, a penalty factor is used in place of the weighting factor per objective. The algorithm interprets the chromosome and calculates the sum of the residual distances between points and their respective centroids, as well as a penalty factor based on the demands in each cluster with respect to the fleet capacity. The process of evaluating the fitness of each chromosome is described by the following steps:

1. Iterate through each gene in the chromosome and assign the customer nodes to their respective clusters. Here the value of the gene corresponds to the cluster and the position corresponds to the customer.
2. For each cluster, calculate the total demand for the respective customers within the cluster as well as the residual distance from each point to the centroid.
    a. If the total demand for a cluster does not exceed the capacity of the vehicle, set the penalty factor to 1.
        - Add the residual distance multiplied by the penalty factor to the fitness score.
    b. If the total demand for a cluster exceeds the capacity of the vehicle, set the penalty factor to the proportion of capacity that exceeds that of the vehicle.
        - Add the residual distance multiplied by the penalty factor to the fitness score.
3. Return the total fitness score.

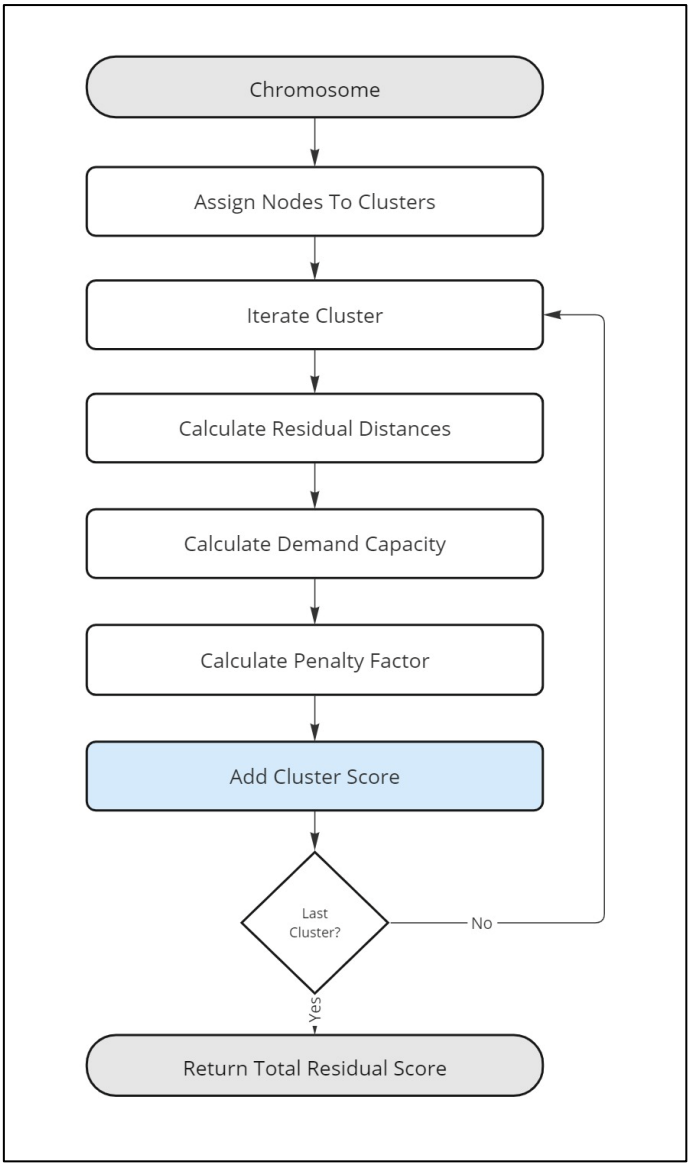



*Figure 26: Flowchart of the fitness function algorithm for clustering (Biswas, 2017).*

### 5.3.3 Hybrid GA Cluster-First Route-Second Solution

Figure 27 provides a flowchart of the process for the proposed hybrid GA cluster-first route-second CVRP solution. The process received the customer, depot, and vehicle capacity as input, grouped the customers into separate clusters and solved for the route in each cluster separately, after which all routes were returned as the output.

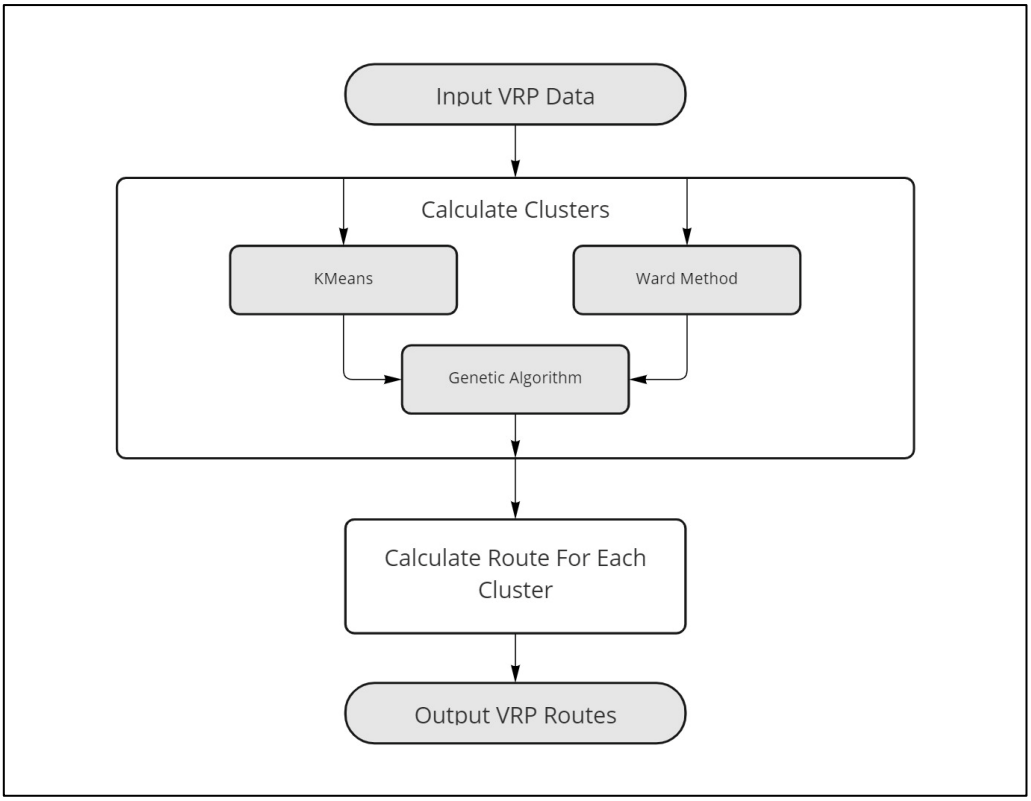



*Figure 27: Flowchart of the hybrid GA cluster-first route-second CVRP solution.*

Within the clustering phase of the proposed hybrid solution, both the ward's method and k-means clustering algorithms were used to generate two initial clusters. These clusters were then encoded and used to seed a portion of the initial population of chromosomes in a GA. The GA was then used to further explore the solution space and attempt to improve on the quality of the clusters. Calculation of the initial clusters was completed using the clustering configurations from experiments 4 and 5. The GA configuration from experiment 8 was used to improve on the initial clusters calculated. As before, the routes were calculated using the GA configuration from experiment 1.

# 6. Results & Analysis

This chapter discusses the results of the experiments evaluated in this research. First, the results of experiments 1-3 are presented, which evaluate three GA configurations integrated into a direct routing solution format. Next, the results of experiments 4-8 are presented. These experiments evaluate two conventional clustering algorithms and three GA configurations respectively, which are integrated into a cluster-first route-second solution format. The results of experiment 9 are then presented, which evaluate the proposed hybrid clustering GA method. Finally, a summary and discussion of the results and key observations from all the experiments is completed.

## 6.1 Direct Routing Solution

### 6.1.1 Parameter Evaluation

Figure 28 depicts the convergence of the GA configuration used for routing in experiment 1 for all datasets. In general, the convergence rate decreased with an increase in the size of the problem, as expected. For all datasets evaluated, approximately 95% of the convergence occurred within the first 20 000 iterations.

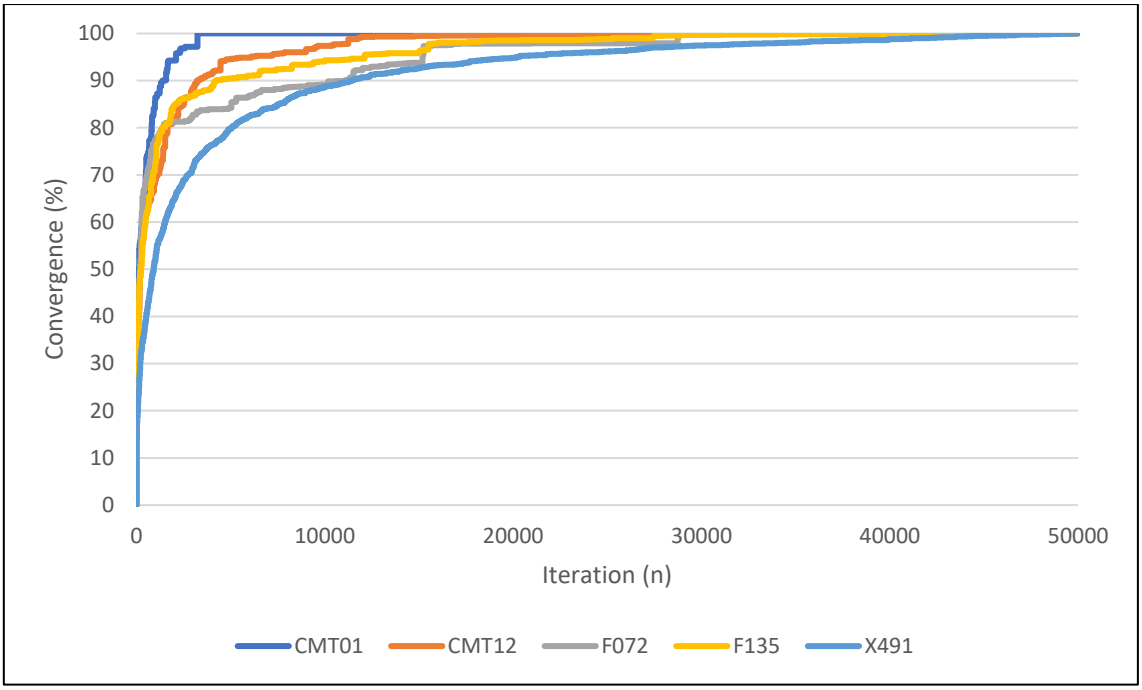



*Figure 28: GA routing solution convergence for experiment 1.*

### 6.1.2 Experiment 1 to 3 Results

Table 7 provides an overview of the total route distance results for experiments 1-3, which are expressed as a percentage relative to the best-known solutions for all datasets evaluated. Table 8 lists the detailed results for each experiment. On average, experiment 1 performed the best, however, all three experiments deviated significantly from the benchmark. Experiment 2 required the least computational effort.

*Table 7: Overview of results for experiments 1-3.*

| Dataset | Best-Known Solution | % Greater Than Best-Known Solution | | |
|---|---|---|---|---|
| | | Exp. 1 | Exp. 2 | Exp. 3 |
| **CMT01** | **524.61** | 18.02 | 19.79 | 25.58 |
| **CMT12** | **819.56** | 31.32 | 26.51 | 48.37 |
| **F072** | **237.00** | 47.31 | 59.55 | 59.87 |
| **F135** | **1162.00** | 24.52 | 17.60 | 31.66 |
| **X491** | **66483.00** | 36.69 | 37.35 | 37.37 |
| **Avg.** | | 31.57 | 32.16 | 40.57 |

*Table 8: Detailed results for experiments 1-3.*

| Dataset | Exp. | Routes (k) | Time (s) | Total Route Distance |
|---|---|---|---|---|
| **CMT01** | **1** | 6 | 42.91 | 619.12 |
| **CMT01** | **2** | 6 | 30.38 | 628.45 |
| **CMT01** | **3** | 6 | 59.83 | 658.83 |
| **CMT12** | **1** | 10 | 85.16 | 1 076.21 |
| **CMT12** | **2** | 10 | 51.43 | 1 036.80 |
| **CMT12** | **3** | 10 | 122.19 | 1 216.00 |
| **F072** | **1** | 5 | 58.96 | 349.12 |
| **F072** | **2** | 5 | 52.29 | 378.13 |
| **F072** | **3** | 5 | 85.29 | 378.89 |
| **F135** | **1** | 7 | 119.19 | 1 446.97 |
| **F135** | **2** | 7 | 90.54 | 1 366.55 |
| **F135** | **3** | 7 | 216.10 | 1 529.93 |
| **X491** | **1** | 63 | 746.88 | 90 876.08 |
| **X491** | **2** | 63 | 279.46 | 91 314.99 |
| **X491** | **3** | 63 | 1 206.40 | 91 329.79 |

The route sequences calculated for all datasets are illustrated in figures 29 to 33. In general, there was significant overlapping of the routes between groups of customers for all three experiments, and across all datasets evaluated. While some overlap should be expected in the optimal solution for some datasets, significant overlap in others may indicate that the algorithm struggled to escape from local optima. This was particularly evident for the results of the CMT12 dataset, which has visually distinct clusters that do not exceed the capacity of the fleet. Here, the GA implementations were not able to effectively separate the obvious customer groups. In addition to this, the presence of crossing arcs within subtours suggested that the individual routes were also not optimal.

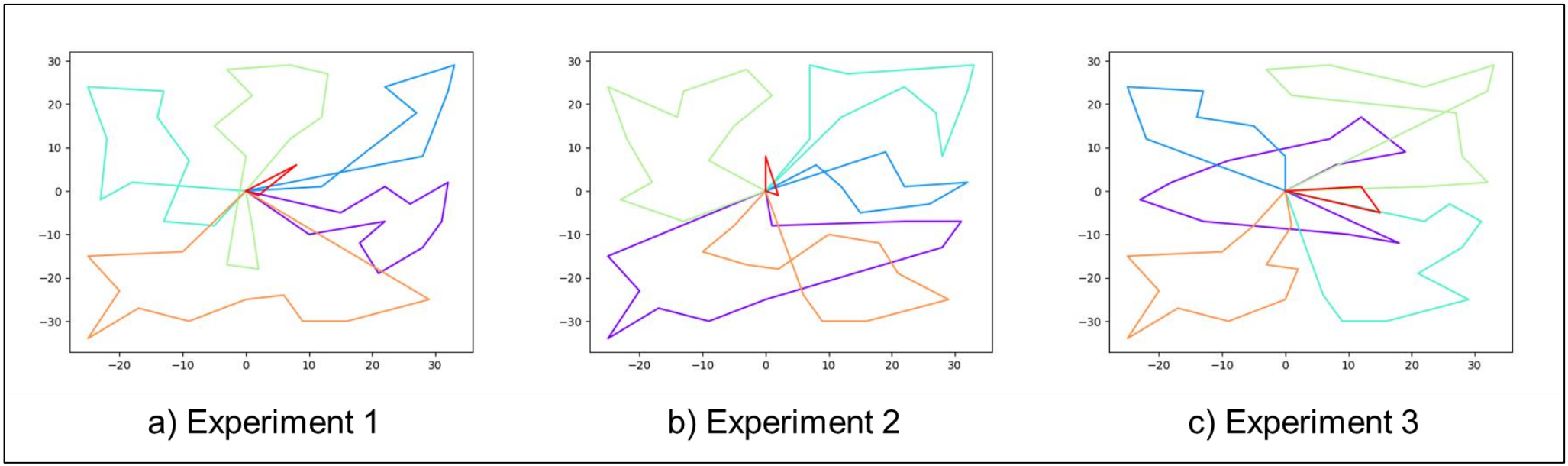



*Figure 29: CMT01 calculated routes for experiments 1-3.*

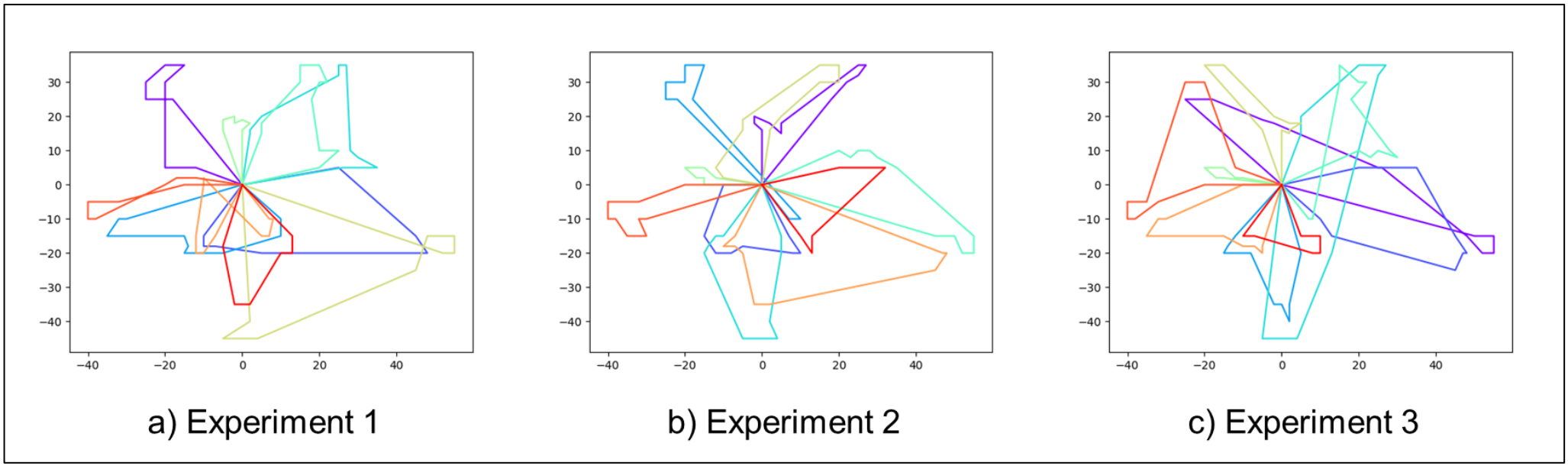



*Figure 30: CMT12 calculated routes for experiments 1-3.*

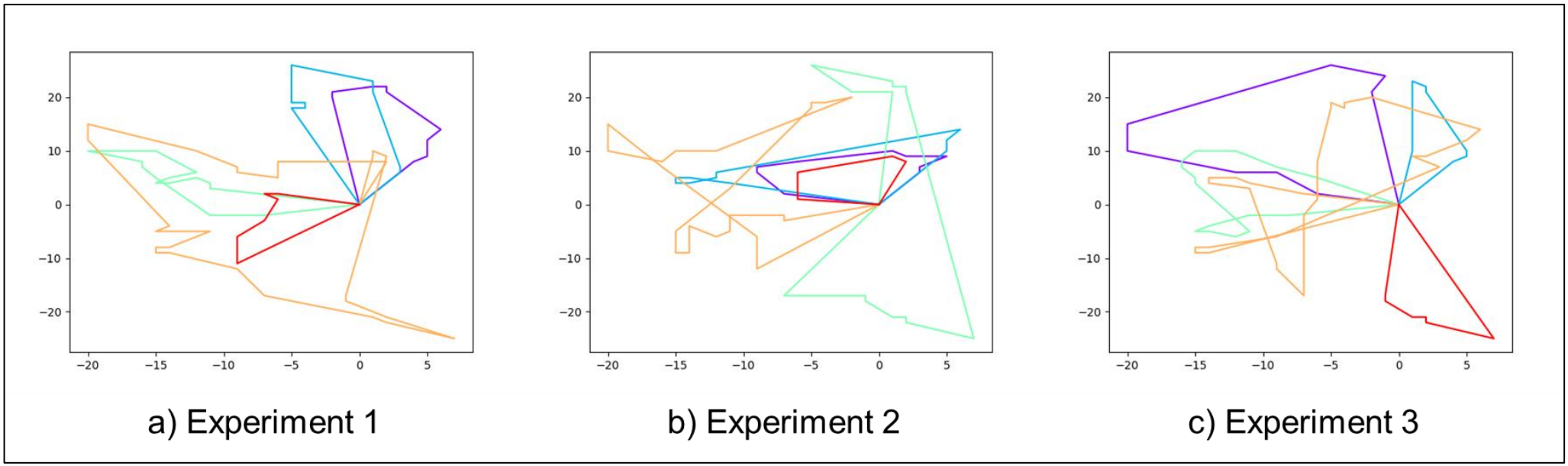



*Figure 31: F072 calculated routes for experiments 1-3.*

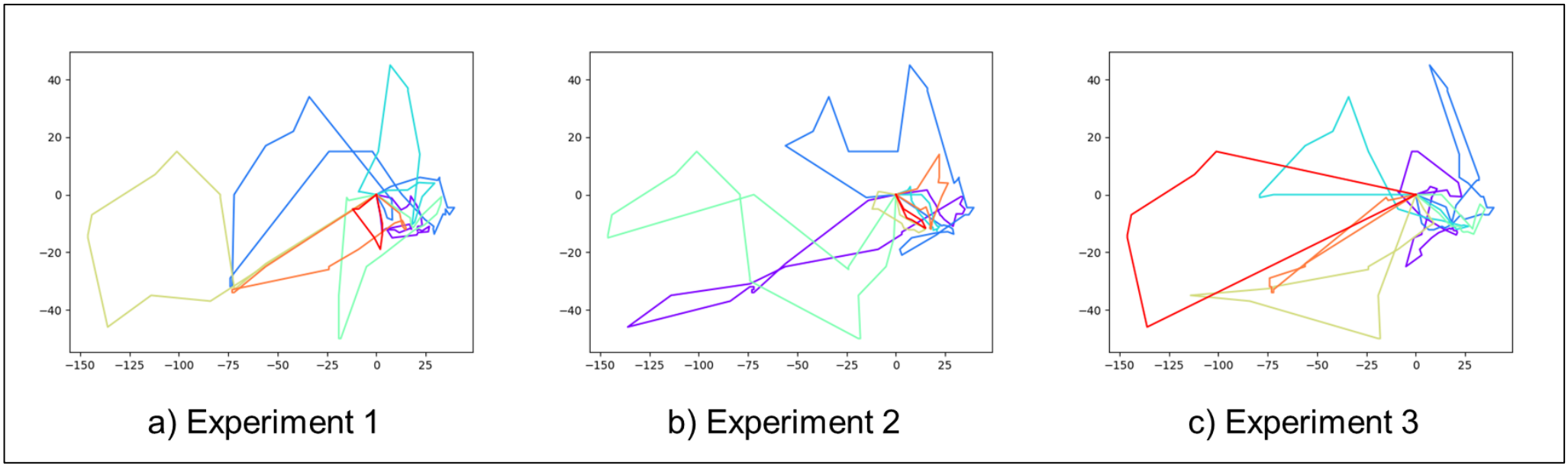



*Figure 32: F135 calculated routes for experiments 1-3.*

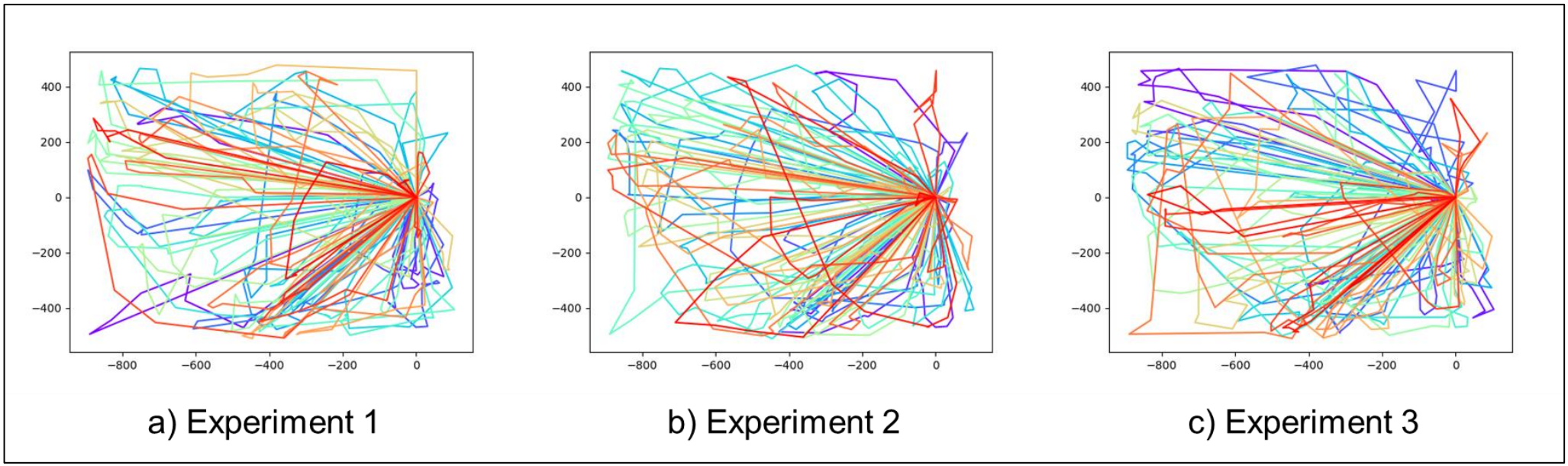



*Figure 33: X491 calculated routes for experiments 1-3.*

## 6.2 Cluster-First Route-Second

### 6.2.1 Parameter Evaluation

Figure 34 depicts the convergence of the k-means clustering algorithm used in experiment 4. For all datasets, the algorithm converged within the first 10 iterations.

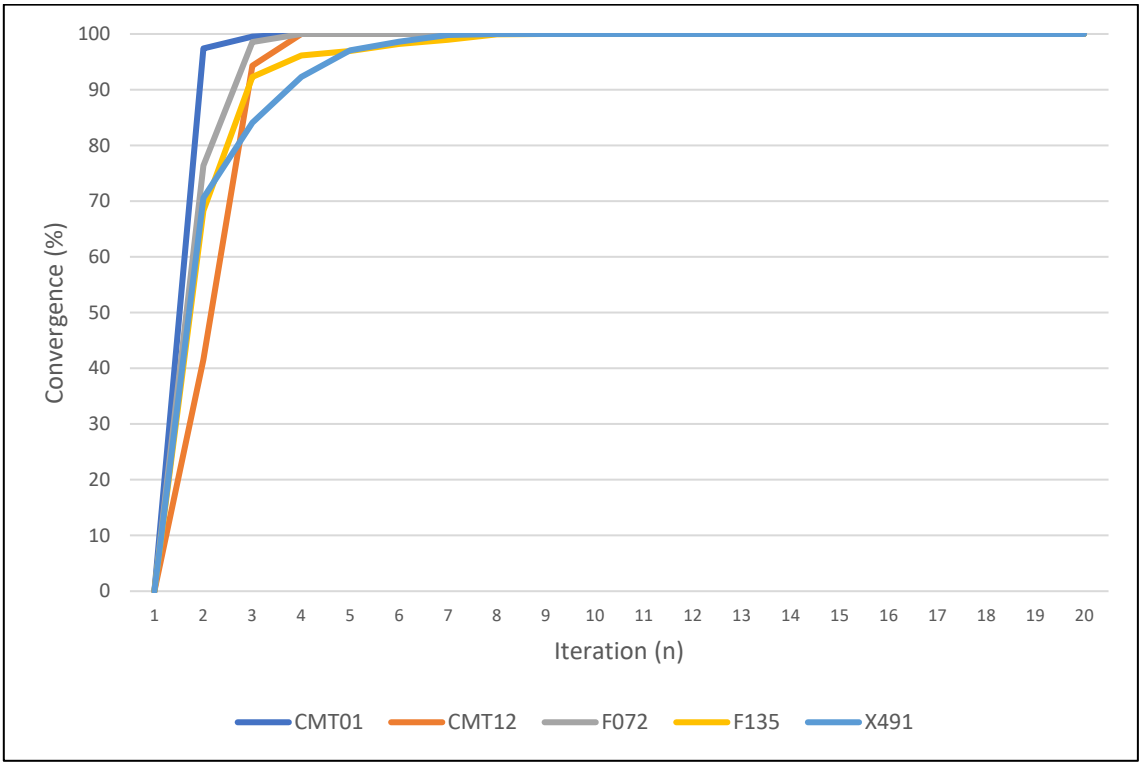



*Figure 34: K-means clustering solution convergence for experiment 4.*

Figure 35 depicts the convergence of the GA configuration used for clustering in experiment 8 for all datasets. As with the previous configurations, the convergence rate decreased with an increase in the size of the problem. For all datasets evaluated, approximately 95% of the convergence occurred within the first 10 000 iterations.

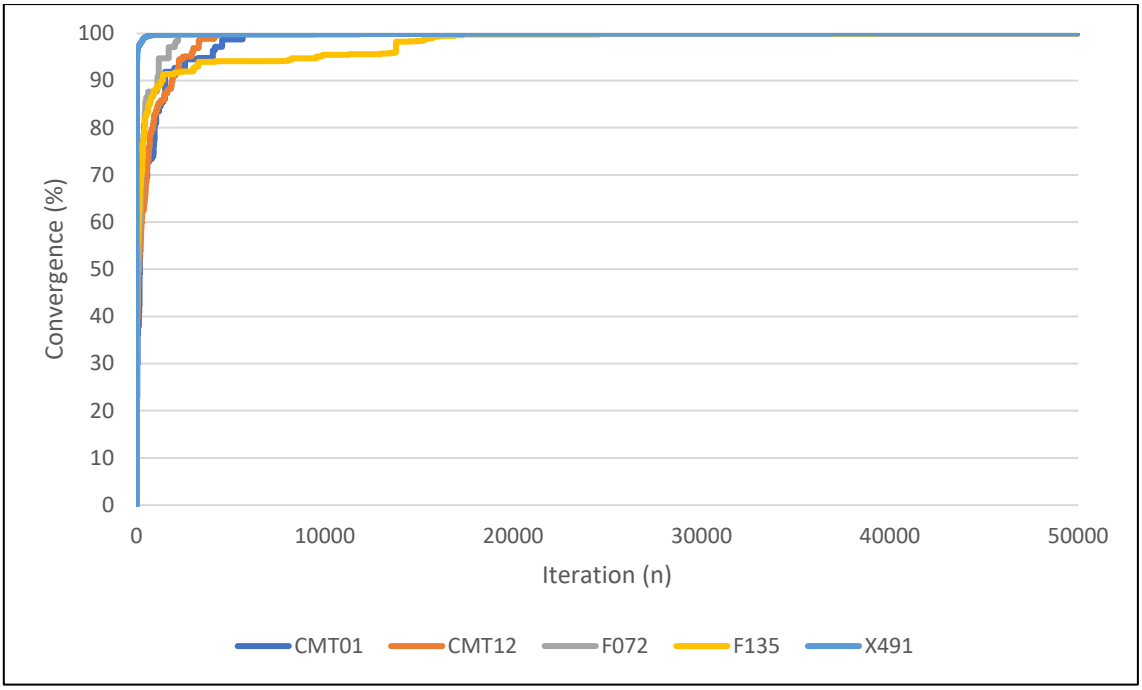



*Figure 35: GA clustering solution convergence for experiment 8.*

### 6.2.2 Experiment 4 to 8 Results

Table 9 provides an overview of the total route distance results for experiments 4-8, which are expressed as a percentage relative to the best-known solutions for all datasets evaluated. Table 10 lists the detailed results for each experiment. In general, the GA clustering methods outperformed the conventional clustering methods, however, they were also more computationally expensive. On average, experiment 8 was the best performing method overall, and it scaled better with an increasing problem size in comparison to the other clustering methods.

*Table 9: Overview of results for experiments 4-8.*

| Dataset | Best-Known Solution | % Greater Than Best-Known Solution | | | | |
|---|---|---|---|---|---|---|
| | | Exp. 4 | Exp. 5 | Exp. 6 | Exp. 7 | Exp. 8 |
| **CMT01** | **524.61** | 0.40 | 31.80 | 2.80 | 2.43 | 2.43 |
| **CMT12** | **819.56** | 1.24 | 1.14 | 1.63 | 1.26 | 1.25 |
| **F072** | **237.00** | 40.32 | 61.70 | 9.44 | 16.52 | 11.84 |
| **F135** | **1162.00** | 26.09 | 83.69 | 13.54 | 14.76 | 14.76 |
| **X491** | **66483.00** | 45.29 | 65.06 | 40.97 | 38.17 | 33.66 |
| **Avg.** | | 22.67 | 48.68 | 13.68 | 14.63 | 12.79 |

*Table 10: Detailed results for experiments 4-8.*

| Dataset | Exp. | Routes (k) | | Times (s) | | Total Route Distance |
|---|---|---|---|---|---|---|
| | | Number of Clusters | Residual Distance | Clustering | Routing | |
| **CMT01** | **4** | 5 | 561.62 | 0.01 | 17.44 | 526.69 |
| **CMT01** | **5** | 8 | 459.60 | 0.03 | 24.19 | 691.44 |
| **CMT01** | **6** | 5 | 542.49 | 73.70 | 16.75 | 539.30 |
| **CMT01** | **7** | 5 | 542.49 | 73.41 | 18.00 | 537.37 |
| **CMT01** | **8** | 5 | 542.49 | 78.98 | 20.78 | 537.37 |
| **CMT12** | **4** | 10 | 437.96 | 0.03 | 35.90 | 829.73 |
| **CMT12** | **5** | 10 | 437.96 | 0.13 | 35.46 | 828.94 |
| **CMT12** | **6** | 10 | 437.96 | 146.15 | 35.81 | 832.93 |
| **CMT12** | **7** | 10 | 437.96 | 147.56 | 37.45 | 829.89 |
| **CMT12** | **8** | 10 | 437.96 | 162.63 | 34.84 | 829.80 |
| **F072** | **4** | 7 | 332.89 | 0.01 | 27.04 | 332.55 |
| **F072** | **5** | 9 | 291.18 | 0.08 | 31.28 | 383.23 |
| **F072** | **6** | 4 | 399.81 | 88.14 | 17.41 | 259.37 |
| **F072** | **7** | 4 | 399.81 | 92.24 | 18.37 | 276.14 |
| **F072** | **8** | 4 | 399.81 | 92.98 | 21.87 | 265.07 |
| **F135** | **4** | 10 | 1125.78 | 0.06 | 44.56 | 1465.14 |
| **F135** | **5** | 16 | 675.71 | 0.73 | 53.04 | 2134.51 |
| **F135** | **6** | 7 | 1458.00 | 170.26 | 32.82 | 1319.39 |
| **F135** | **7** | 7 | 1433.16 | 175.88 | 39.37 | 1333.56 |
| **F135** | **8** | 7 | 1420.82 | 180.38 | 30.87 | 1333.52 |
| **X491** | **4** | 89 | 15039.72 | 0.96 | 266.79 | 96594.10 |
| **X491** | **5** | 102 | 12517.19 | 16.30 | 303.28 | 109735.11 |
| **X491** | **6** | 59 | 56082.89 | 1652.02 | 207.59 | 93722.63 |
| **X491** | **7** | 59 | 53314.90 | 1620.38 | 202.26 | 91858.92 |
| **X491** | **8** | 59 | 51942.82 | 1723.93 | 202.67 | 88859.36 |

Figures 36 to 40 provide an illustration of the route sequences calculated for all datasets. In comparison to the direct routing approach used for experiments 1-3, the two phased cluster-first route-second approach was expected to improve customer grouping and reduce the overlapping of routes. The results presented for the CMT01 and CMT12 datasets indicate that all five experiments were effective at segmenting customers that have visually distinct clusters or a uniform distribution of customer

nodes. With that being said, the optimal solution requires a balance between segmentation and capacity utilization of the fleet. For the F072 and F135 datasets, which are characterized by a skewed distribution of customer nodes, the conventional clustering approaches used in experiments 4 and 5 proved to be less successful, resulting in more tours. Here the advantage of the GA implementations, which consider the capacity when calculating the clusters, was evident. Overall, the two-phased cluster-first route-second approach simplifies the route-calculation because the routes for each customer grouping are evaluated separately. This is evident by the reduced presence of crossing arcs within subtours when compared to the first three experiments.

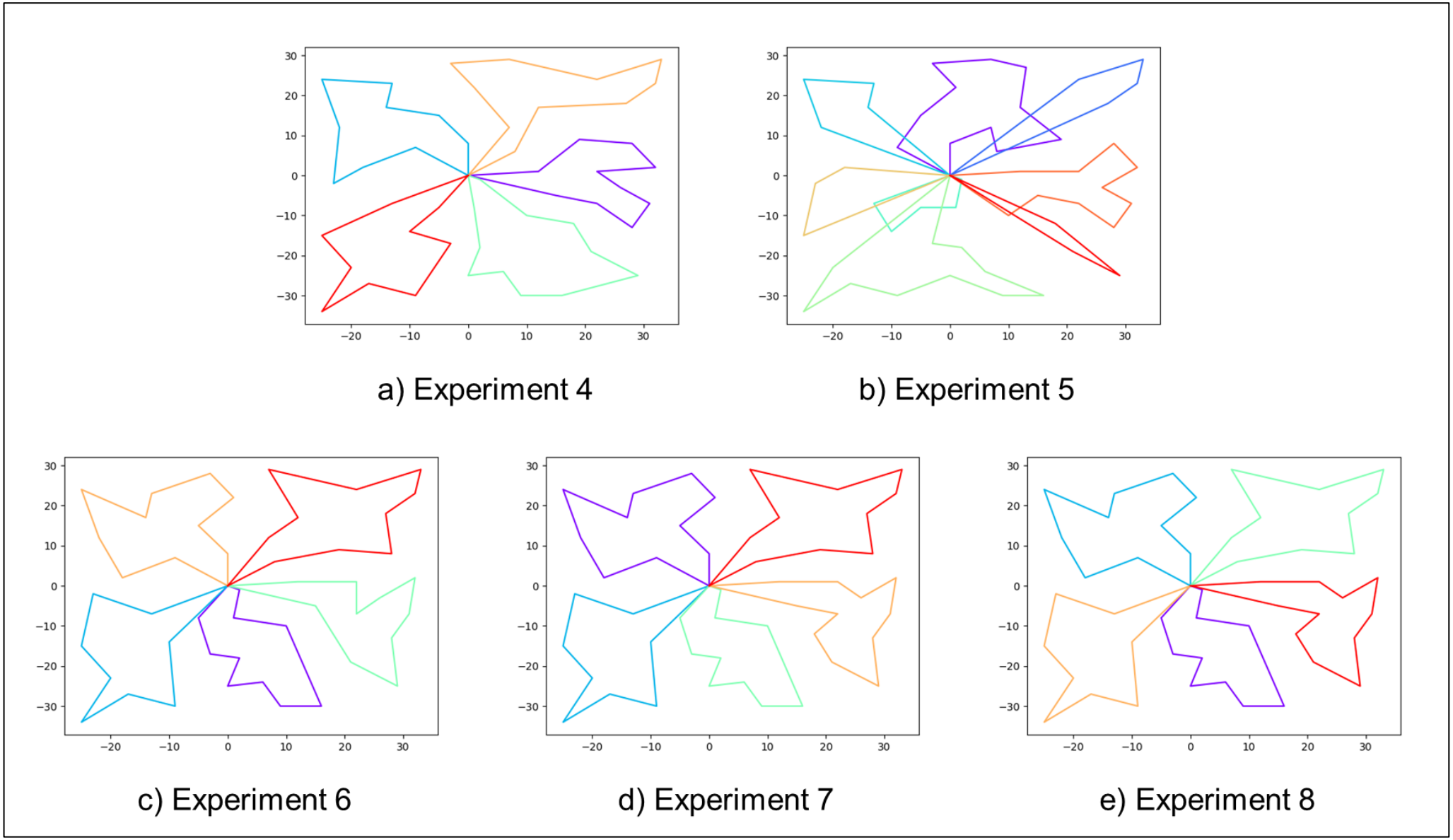



*Figure 36: CMT01 calculated routes for experiments 4-8.*

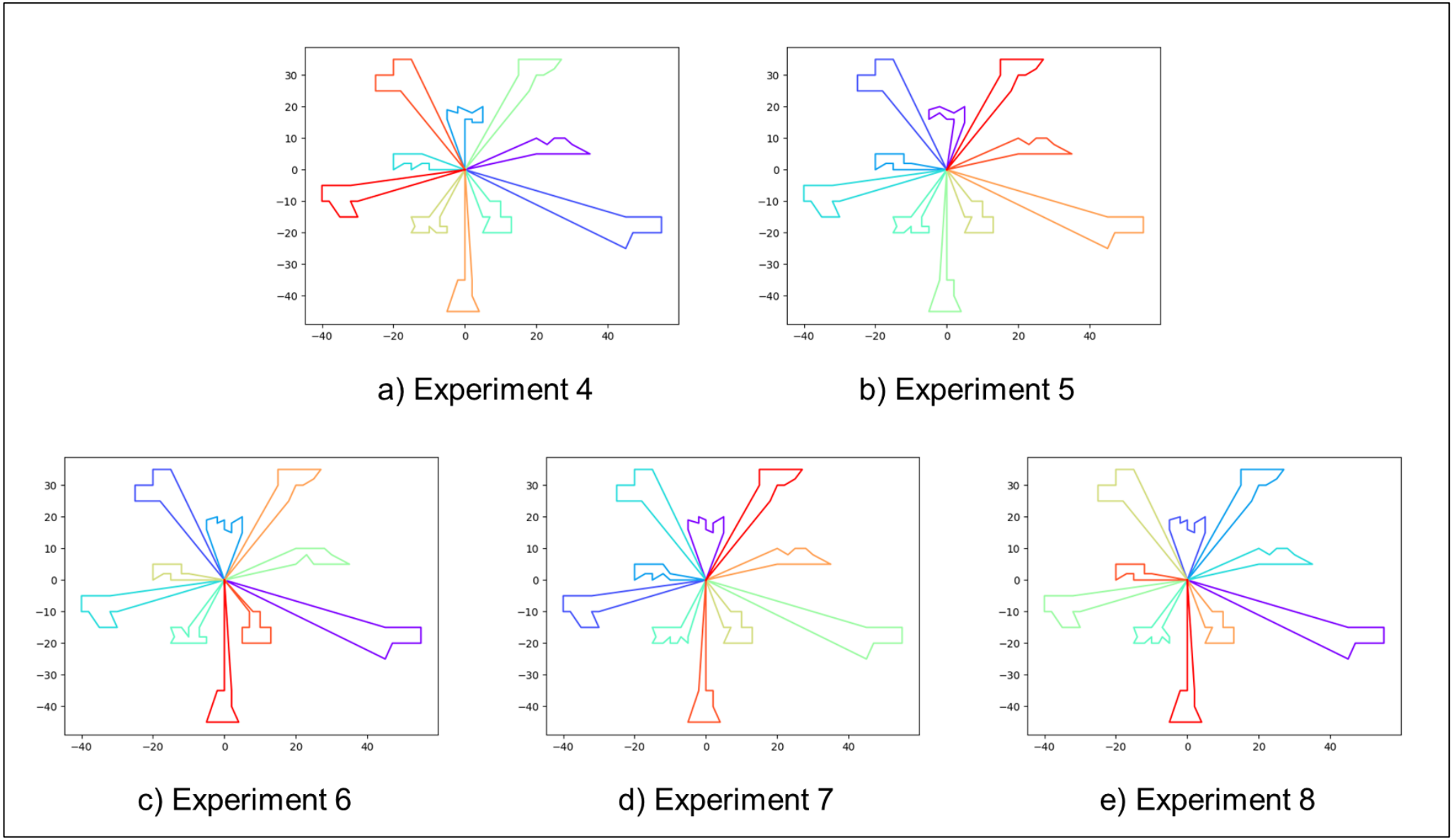



*Figure 37: CMT12 calculated routes for experiments 4-8.*

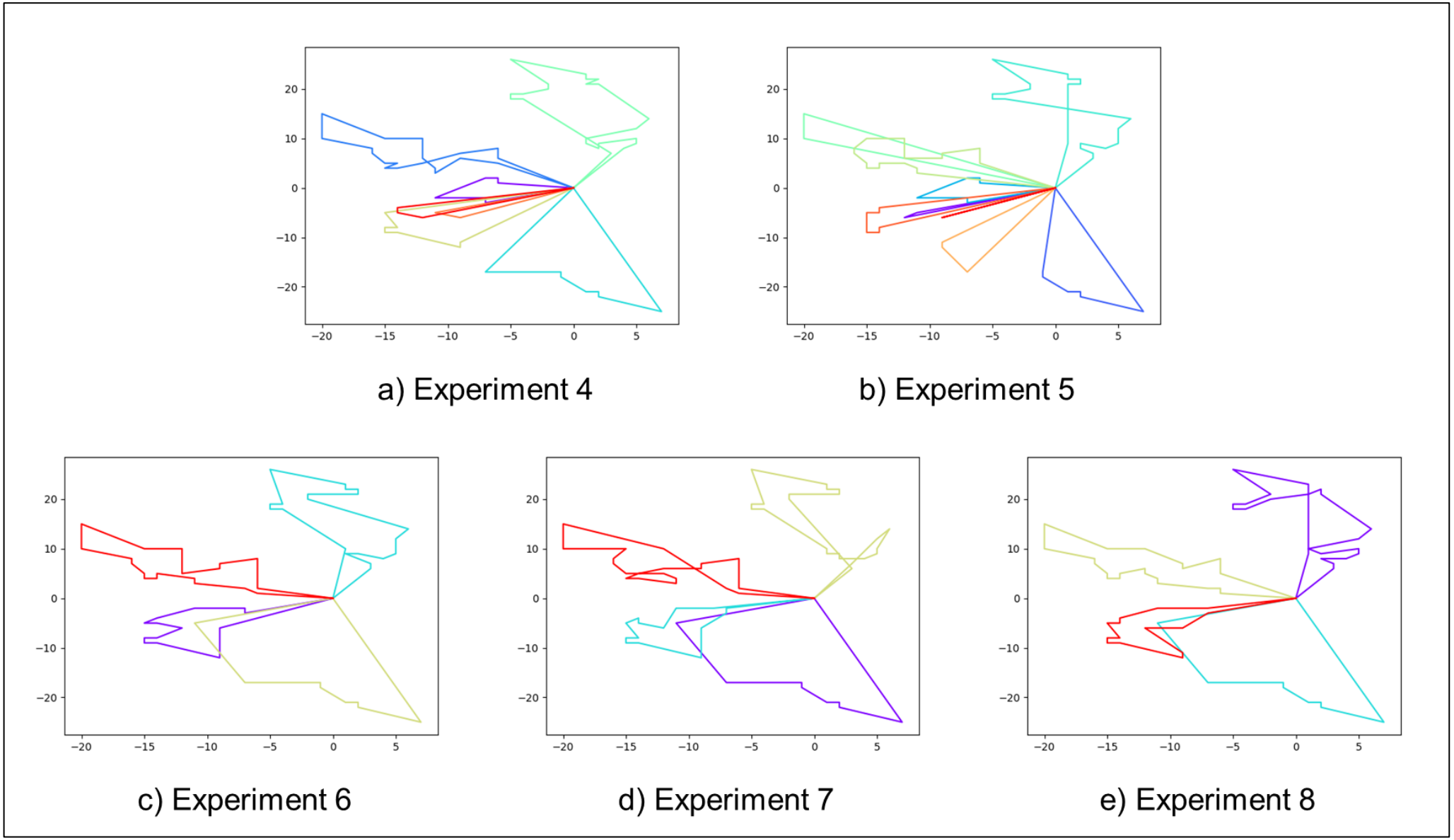



*Figure 38: F072 calculated routes for experiments 4-8.*

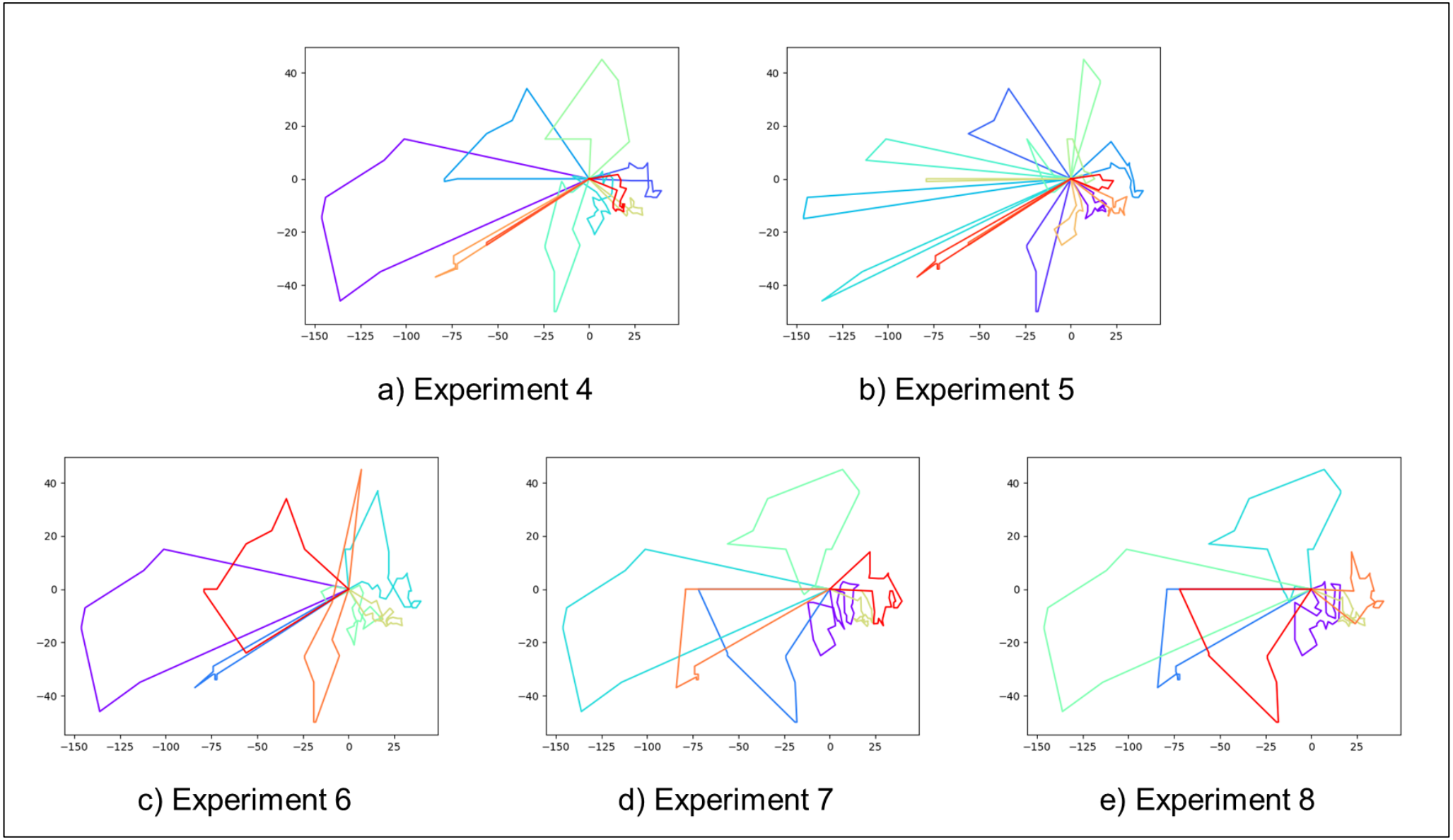



*Figure 39: F135 calculated routes for experiments 4-8.*

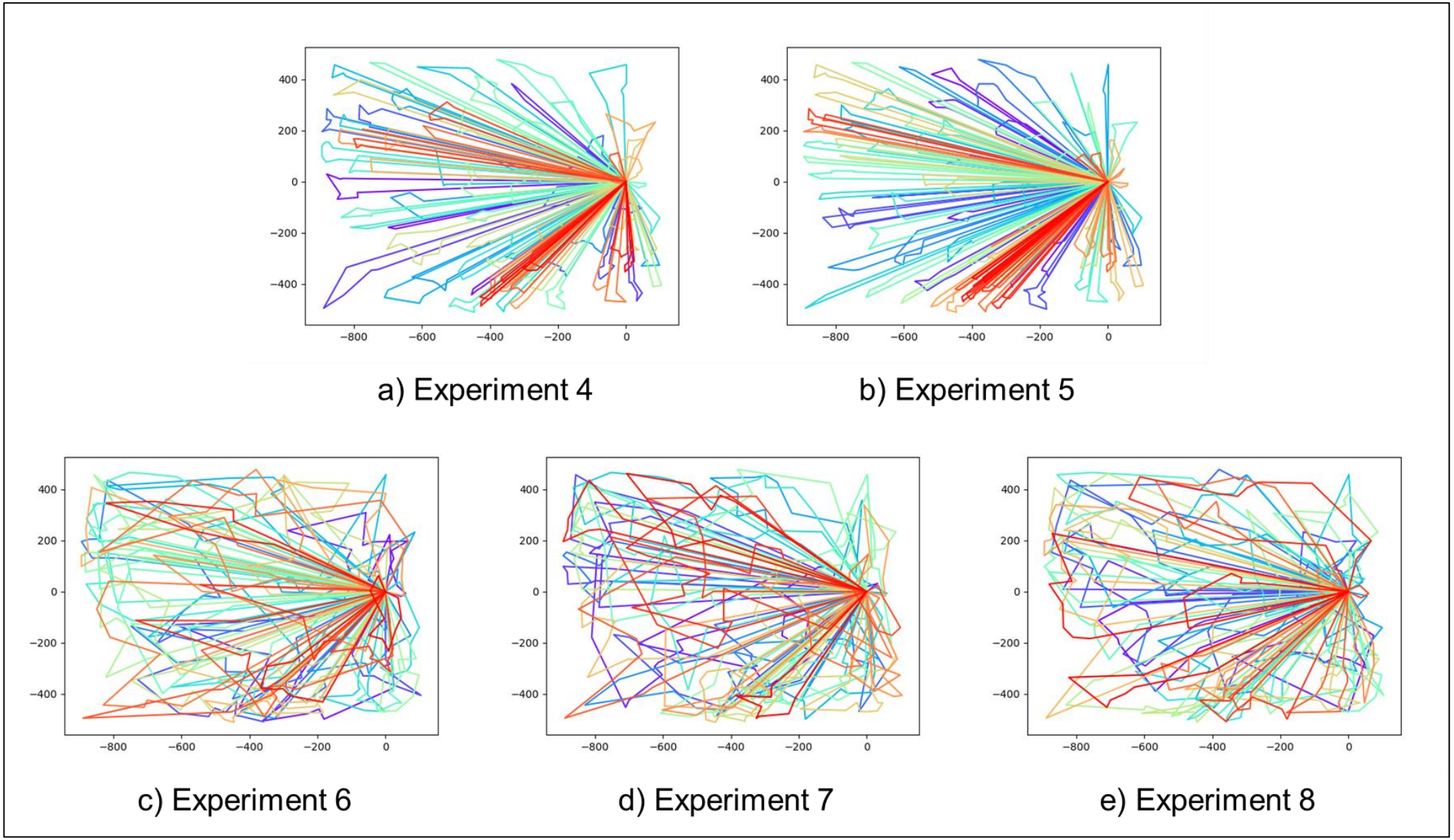



*Figure 40: X491 calculated routes for experiments 4-8.*

## 6.3 Hybrid GA Cluster-First Route-Second

### 6.3.1 Experiment 9 Results

Table 11 provides an overview of the total route distance results for experiment 9, which are expressed as a percentage relative to the best-known solutions for all datasets evaluated. Table 12 lists the detailed results for the experiment.

*Table 11: Overview of results for experiment 9.*

| Dataset | Best-Known Solution | % Greater Than Best-Known Solution |
|---|---|---|
| | | Exp. 9 |
| **CMT01** | **524.61** | 0.64 |
| **CMT12** | **819.56** | 1.15 |
| **F072** | **237.00** | 7.02 |
| **F135** | **1162.00** | 15.06 |
| **X491** | **66483.00** | 28.69 |
| **Avg.** | | 10.51 |

*Table 12: Detailed results for experiment 9.*

| Dataset | Exp. | Routes (k) | | Times (s) | | Total Route Distance |
|---|---|---|---|---|---|---|
| | | Number of Clusters | Residual Distance | Clustering | Routing | |
| **CMT01** | **9** | 5 | 561.03 | 73.92 | 21.96 | 527.95 |
| **CMT12** | **9** | 10 | 437.96 | 150.96 | 41.04 | 828.98 |
| **F072** | **9** | 4 | 399.81 | 109.98 | 22.73 | 253.65 |
| **F135** | **9** | 7 | 1425.16 | 193.93 | 37.12 | 1337.00 |
| **X491** | **9** | 59 | 44608.10 | 1806.16 | 220.90 | 85558.90 |

The route sequences calculated for all datasets are illustrated in figures 41 to 45. In general, the segmentation of customer groups was comparable to the previous cluster-first route-second solution methods. The results presented for the CMT01 and CMT12 again proved to be effective with minimal route overlap and crossing of arcs within subtours. This was again reflected in the results of the F072 and F135 datasets.

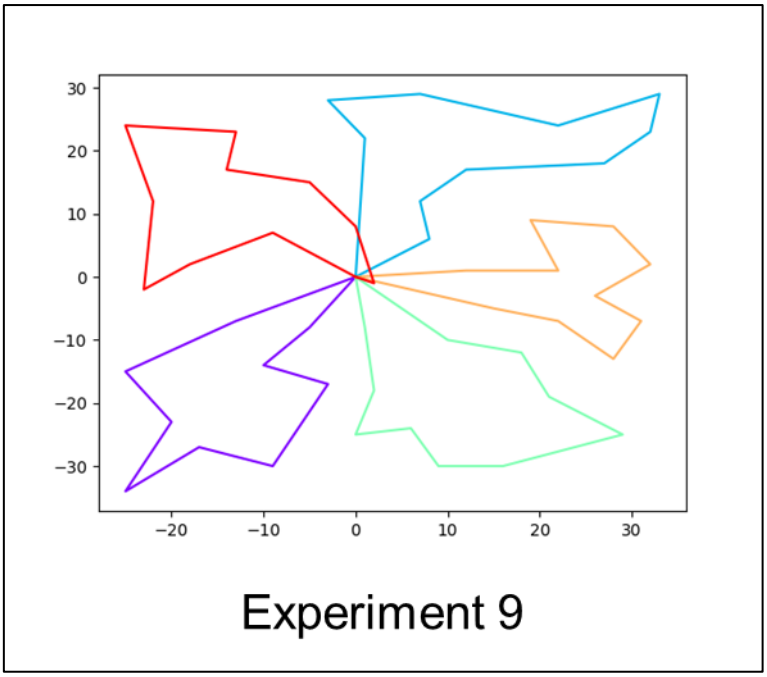



*Figure 41: CMT01 calculated routes for experiment 9.*

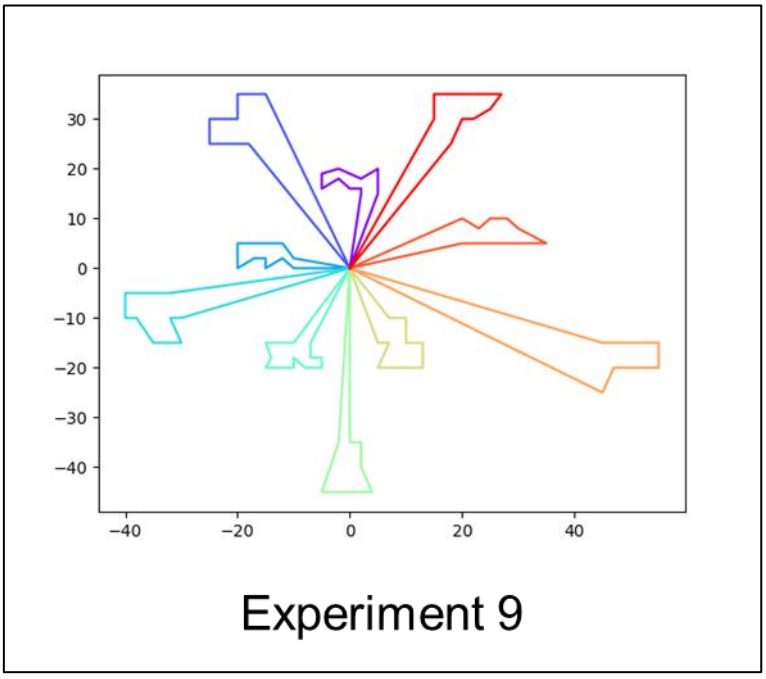



*Figure 42: CMT12 calculated routes for experiment 9.*

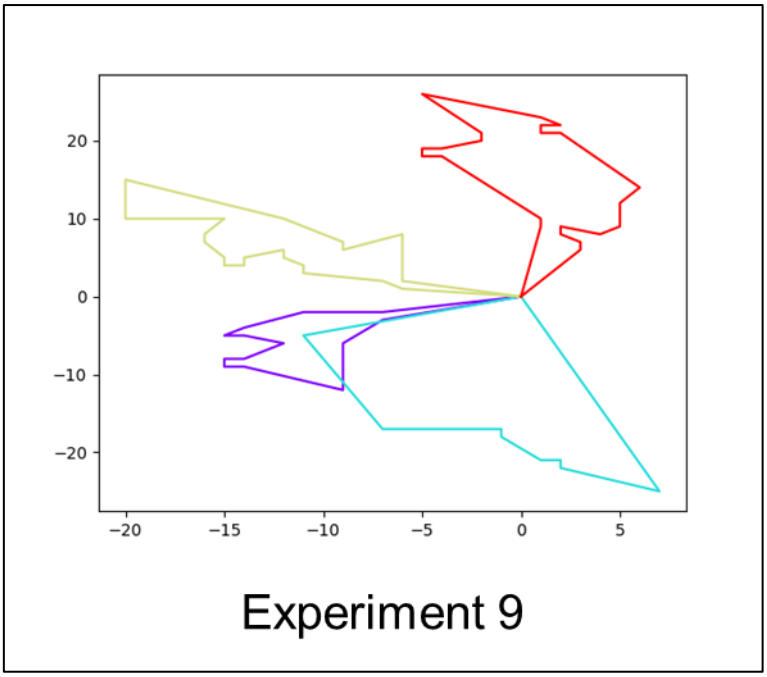



*Figure 43: F072 calculated routes for experiment 9.*

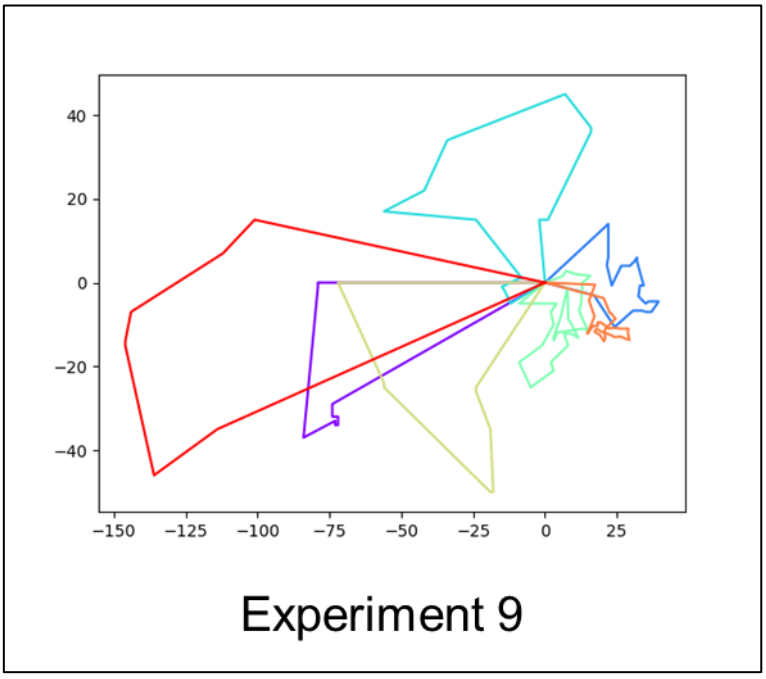



*Figure 44: F135 calculated routes for experiment 9.*

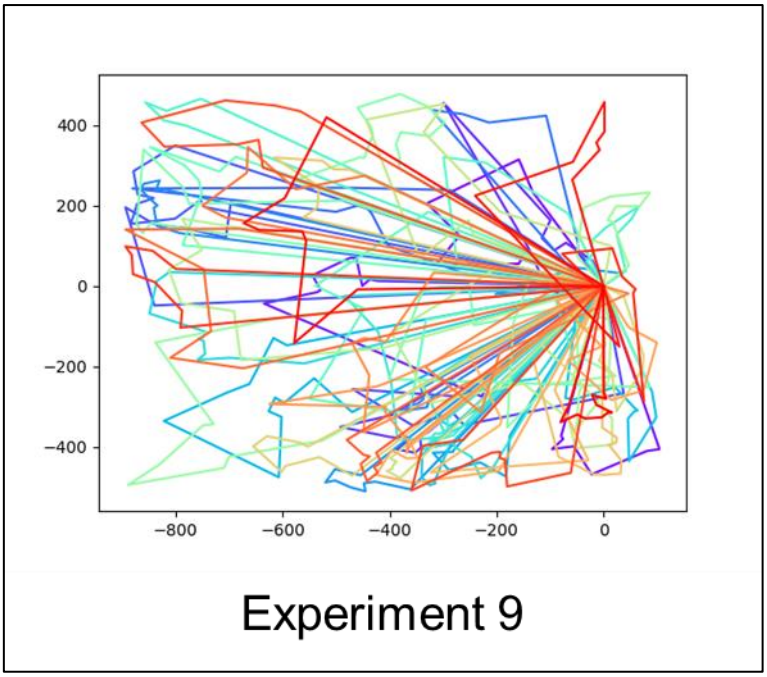



*Figure 45: X491 calculated routes for experiment 9.*

## 6.4 Discussion

From the results presented in the previous sections, it is evident that the best performing GA configuration for the direct routing solution format (experiments 1-3) was experiment 1. It produced the lowest overall route distance on three of the five datasets evaluated, as well as on average. This improvement did however incur a small computation time penalty over the experiment 2 GA configuration, which performed similarly. Experiment 3 was the worst performing GA configuration tested, both in terms of total route distance and computation time. In general, the direct routing solution approach did not perform well, and all three experiments struggled to segment customers effectively.

Experiments 4 and 5 integrated conventional clustering methods into a cluster-first route-second solution approach and carried over the GA configuration used to calculate routes from experiment 1. As expected, these experiments improved the customer segmentation, however, they did not perform well overall except in niche scenarios such as the CMT12 dataset. This dataset is characterized by visually distinct clusters that do not exceed the fleet capacity. In general, this behaviour was expected

because the demand had no influence on the formation of clusters, which were subdivided as required to produce a valid solution. This inevitably increased the number of clusters and consequently the number of tours required to service all customers while decreasing the utilization of the vehicles in the fleet.

Experiment 6 to 8 expanded on the previous experiments and substituted the conventional clustering method for a GA. On average, this method achieved a significant improvement over the conventional clustering methods by considering the capacities of the customer nodes when calculating the clusters. For all datasets, the GA implementations for these experiments yielded the minimum number of clusters possible without exceeding the capacity constraints of the fleet. From the results, it was also evident that, all three GA configurations performed similarly across all datasets except for the large X491 dataset. Here, the GA configuration for experiment 8 performed significantly better in comparison to the others. In general, incorporating GAs into the clustering process had a positive impact on the quality of the solutions. However, they required significantly more computationally effort in comparison to the conventional clustering techniques.

The proposed hybrid approach of experiment 9 utilized conventional clustering techniques to seed the initial population of the GA configuration used for clustering. As before, this solution adopted a cluster-first route-second approach and carried over the configuration of the conventional clustering methods and GA from experiments 4, 5, and 8 respectively. On average, the solution quality of experiment 9 was marginally improved or comparable to the best results of the previous experiments across all datasets except for the large X491 dataset. For this dataset, a significant improvement was achieved in the clustering performance that was reflected in a lower overall route distance. Once again, however, the improvement in solution quality did incur a computational time penalty. While experiment 9 was noted as being the best overall experiment in terms of solution quality, it was also the most computationally expensive solution evaluated. This was expected due to the increased complexity of the proposed solution.

Table 13 summarizes the route distances presented in the previous sections for each experiment, and across all datasets. The results are expressed as a percentage

relative to the best-known solutions. The average percentage gap for each experiment across all datasets is also provided.

*Table 13: Summary of results for experiments 1-9.*

| Experiment | % Greater Than Best-Known Solution | | | | | |
|---|---|---|---|---|---|---|
| | **CMT01** | **CMT12** | **F072** | **F135** | **X491** | **Avg.** |
| **1** | 18.02 | 31.32 | 47.31 | 24.52 | 36.69 | 31.57 |
| **2** | 19.79 | 26.51 | 59.55 | 17.6 | 37.35 | 32.16 |
| **3** | 25.58 | 48.37 | 59.87 | 31.66 | 37.37 | 40.57 |
| **4** | **0.4*** | 1.24 | 40.32 | 26.09 | 45.29 | 22.67 |
| **5** | 31.8 | **1.14*** | 61.7 | 83.69 | 65.06 | 48.68 |
| **6** | 2.8 | 1.63 | 9.44 | **13.54*** | 40.97 | 13.68 |
| **7** | 2.43 | 1.26 | 16.52 | 14.76 | 38.17 | 14.63 |
| **8** | 2.43 | 1.25 | 11.84 | 14.76 | 33.66 | 12.79 |
| **9** | 0.64 | 1.15 | **7.02*** | 15.06 | **28.69*** | **10.51**** |
| ** Indicates the best result achieved for each dataset.*<br>***Indicates the best result on average for all datasets.* | | | | | | |

Although successive solution configurations were effective at addressing some of the shortcomings of those previously evaluated, it is important to note that the performance of these solutions were not able to match the best-known solutions. In general, the quality of the solutions explored decreased with the size of the problem, which may indicate premature termination of the algorithms. The results also emphasized the pivotal role of the solution framework. Here, it was evident that the cluster-first route-second approach benefitted significantly from a reduced solution space in comparison to the direct routing framework, albeit with the risk of slightly limiting the potential solution quality. Despite not being able to match the best-known solutions under the imposed conditions, the results do highlight the effectiveness of augmenting metaheuristics, whether through solution framework or the integration of conventional techniques, as a strategy to improve the rate of convergence. This does, however, introduce a trade-off between solution quality and computational efficiency that necessitates careful consideration in the pursuit of an optimal balance for practical applications of the CVRP.

# 7. Conclusion

The objective of this research was to evaluate the impact of integrating metaheuristics, specifically a GA, at various stages of a CVRP solution. In the completion of this study, nine experiments that comprised two basic and well-known solution frameworks, namely direct routing, and cluster-first route-second, were evaluated against five datasets from literature. These datasets were selected to benchmark the performance of each experiment over a wide range of conditions.

During the literature review, it was established that early techniques for solving the CVRP, such as exact methods, have a limited scope of application and are not suitable for larger and more complex problems. For this reason, heuristic, metaheuristic, and hybrid approaches were identified as the focus for this research. Furthermore, it was established that these solution methods are generally integrated into either a direct routing or cluster-first route-second solution format.

The findings of this research indicated that the experiments which adopted the cluster-first route-second solution format produced higher quality results in comparison to those which made use of the direct routing solution format. Moreover, the utilization of GAs to perform the clustering operation improved the results in comparison to the conventional clustering methods. Finally, the proposed hybrid method, which incorporated conventional clustering methods to seed the initial population of the GA used for clustering, further improved on the results. Overall, improvements to the solution quality when integrating the conventional clustering methods with the GA were expected, as it allowed the strengths of each approach to be leveraged.

In conclusion, metaheuristics such as the GA have proven to be an effective approach for tackling different stages of the CVRP. Furthermore, combining metaheuristics with more conventional heuristic methods, such as the k-means clustering algorithm, to create a hybrid approach, have also proven to be an effective strategy for guiding the search process. The results of this research indicate that the integration of metaheuristics with complementary techniques enhance the exploration of the solution space. Additionally, the use of frameworks which constrain the solution space proves to be a valuable strategy for enhancing computational efficiency.

## 7.1 Future Work

Based on the results of this research, several areas have been identified for future work. These include:

- Hybrid methods for solving the CVRP are effective and should be investigated for other variations of the VRP which include additional constraints, such as time windows, backhauls, multi-depot, or a combination of these.
- Alternative hybrid methods should also be explored. This includes integrating alternative heuristic methods, such as local search, with a GA, or combining other metaheuristics, such as simulated annealing, with conventional clustering techniques.
- Considering the deterioration of solution quality with the increase in problem size, optimization of the parameters, such as the alternate stopping criteria, or reproduction operators for the GA, should be investigated.

# 8. References


Alfiyatin, A. N., Mahmudy, W. F. & Anggodo, Y. P., 2018. K-Means Clustering and Genetic Algorithm to Solve Vehicle Routing Problem with Time Windows Problem. The Indonesian Journal of Electrical Engineering and Computer Science (IJEECS), 11(2), pp. 462-468. doi: 10.11591/ijeecs.v11.i2.pp462-468.

Baker, B. M. & Ayechew, M. A., 2003. A genetic algorithm for the vehicle routing problem. Computers & Operations Research, 30(5), pp. 787-800. doi: 10.1016/S0305-0548(02)00051-5.

Baker, J. E., 1985. Adaptive Selection Methods for Genetic Algorithms. In: Proceedings of the 1st International Conference on Genetic Algorithms. New York: L. Erlbaum Associates Inc., pp. 101-111. doi: 10.4324/9781315799674.

Baldacci, R., Christofides, N. & Mingozzi, A., 2008. An exact algorithm for the vehicle routing problem based on the set partitioning formulation with additional cuts. Mathematical Programming, 115(2), pp. 351-385. doi: 10.1007/s10107-007-0178-5.

Bayne, C. K., Beauchamp, J. J., Begovich, C. L. & Kane, V. E., 1980. Monte Carlo comparisons of selected clustering procedures. Pattern Recognition, 12(2), pp. 51-62. doi: 10.1016/0031-3203(80)90002-3.

Bell, J. E. & McMullen, P. R., 2004. Ant colony optimization techniques for the vehicle routing problem. Advanced Engineering Informatics, 18(1), pp. 41-48. doi: 10.1016/j.aei.2004.07.001.

Bianchi, L., Dorigo, M., Gambardella, L. M. & Gutjahr, W. J., 2009. A survey on metaheuristics for stochastic combinatorial optimization. Natural Computing, 8(2), pp. 239-287. doi: 10.1007/s11047-008-9098-4.

Biswas, S., 2017. Multi-objective Genetic Algorithms for Multi-depot VRP with Time Windows, Ontario: Brock University. doi: 10464/11528.

Bührmann, J. & Bruwer, F., 2021. K-Medoid Petal-Shaped Clusering For The Capacitated Vehicle Routing Problem. The South African Journal of Industrial Engineering, 32(2), pp. 33-41. doi: 10.7166/32-3-2610.

Bührmann, J. H., 2016. PhD Thesis: The effects of clustering on the medium and large-scale capacitated location-routing problem, Johannesburg: University of Witwatersrand. doi: 10539/20701.

Christofides, N., Mingozzi, A. & Toth , P., 1981. State space relaxation procedures for the computation of bounds to routing problems. Networks, 11(2), pp. 145-164. doi: 10.1002/net.3230110207.

Christofides, N., Mingozzi, A. & Toth, P., 1979. The Vehicle Routing Problem. In: Combinatorial Optimization. Chichester: Wiley, pp. 315-338. doi: unavailable.

Clarke, G. & Wright, J., 1964. Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. Operations research, 12(4), pp. 568-581. doi: 10.1287/opre.12.4.568.

Dantzig, G. B. & Ramser, J. H., 1959. The Truck Dispatching Problem. Management Science, 6(1), pp. 80-91. doi: 10.1287/mnsc.6.1.80.

Davis, L., 1985. Applying Adaptive Algorithms to Epistatic Domains. In: IJCAI. San Francisco: Morgan Kaufmann Publishers Inc., pp. 162-164. doi: unavailable.

De Jong, K. A., 1975. PhD Thesis: An Analysis of the Behavior of a Class of Genetic Adaptive Systems., Michigan: University of Michigan. doi: 2027.42/4507.

Dethloff, J., 2001. Vehicle routing and reverse logistics: The vehicle routing problem with simultaneous delivery and pick-up. OR Spektrum, 23(1), pp. 79-96. doi: 10.1007/PL00013346.

Escobar, J. W., Linfati, R., Toth, P. & Baldoquin, M. G., 2014. A hybrid Granular Tabu Search algorithm for the Multi-Depot Vehicle Routing Problem. Journal of Heuristics, 20(5), pp. 483-509. doi: 10.1007/s10732-014-9247-0.

Fisher, M., 1994. Optimal solution of vehicle routing problems using minimum K-trees. Operations Research, 42(4), pp. 626-642. doi: 10.1287/opre.42.4.626.

Fisher, M. L. & Jaikumar, R., 1981. A generalized assignment heuristic for vehicle routing. Networks, 11(2), pp. 109-124. doi: 10.1002/net.3230110205.

Geiger, M. J., 2001. Report: A Genetic Algorithm applied to multiple objective vehicle routing problems, University of Hohenheim: Department of Productions and Operations Management. doi: unavailable.

Glover, F. & Taillard, E., 1993. A user's guide to tabu search. Annals of Operations Research, 41(1), pp. 1-28. doi: 10.1007/BF02078647.

Goldberg, D. E., 1989. Genetic Algorithms in Search, Optimization & Machine Learning. Reading: Addison-Wesley. doi: unavailable.

Goldberg, D. E. & Deb, K., 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: Foundations of Genetic Algorithms. Illinois: Elsevier, pp. 69-93. doi: 10.1016/B978-0-08-050684-5.50008-2.

Goldberg, D. E. & Lingle, R., 1985. Alleles, Loci and the Traveling Salesman Problem. In: Proceedings of the 1st International Conference on Genetic Algorithms. New York: L. Erlbaum Associates Inc., pp. 154-159. doi: 10.4324/9781315799674

Hadjiconstantinou, E., Christofides, N. & Mingozzi, A., 1995. A new exact algorithm for the vehicle routing problem based on q-paths and k-shortest paths relaxations. Annals of Operations Research, 61(1), pp. 21-43. doi: 10.1007/BF02098280.

Haque, M. & Magld, K., 2012. Improving the solution of traveling salesman problem using genetic, memetic algorithm and edge assembly crossover. International Journal of Advanced Computer Science and Applications, 3(7). doi: 10.14569/IJACSA.2012.030715.

Hussain, A., Muhammad, Y., Nauman S., Hussain, I., Mohamd Shoukry, A., Gani, S., 2017. Research Article: Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator, Islamabad: Computational Intelligence and Neuroscience. doi: 10.1155/2017/7430125.

Jain, A. K., 2010. Data Clustering: 50 Years Beyond K-Means. Pattern Recognition Letters, 31(8), pp. 651-666. doi: 10.1016/j.patrec.2009.09.011.

Jain, N. C., Indrayan, A. & Goel, L. R., 1986. Monte Carlo comparison of six hierarchical clustering methods on random data. Pattern Recognition, 19(1), pp. 95-99. doi: 10.1016/0031-3203(86)90038-5.

Jozefowiez, N., Semet, F. & Talbi, E.-G., 2008. Multi-objective vehicle routing problems. European Journal of Operational Research, 189(2), pp. 293-309. doi: 10.1016/j.ejor.2007.05.055.

Kopp, B., 1978 a. Hierarchical Classification I: Single-Linkage Method. Biometrical journal, 20(5), pp. 495-501. doi: 10.1002/bimj.4710200506.

Kopp, B., 1978 b. Hierarchical Classification II: Complete-Linkage Method. Biometrical journal, 20(6), pp. 597-602. doi: 10.1002/bimj.4710200607.

Kopp, B., 1978 c. Hierarchical classification III: Average-linkage, median, centroid, WARD, flexible strategy. Biometrical journal, 20(7-8), pp. 703-711. doi: 10.1002/bimj.197800010.

Kuiper, F. K. & Lloyd, F., 1975. A Monte Carlo Comparison of Six Clustering Procedures. Biometrics, 31(3), pp. 777-783. doi: 10.2307/2529565.

Kumar, S. N. & Panneerselvam, R., 2012. A Survey on the Vehicle Routing Problem and Its Variants. Intelligent Information Management, 4(3), pp. 66-74. doi: 10.4236/iim.2012.43010.

Laporte, G., 1992. The vehicle routing problem: An overview of exact and approximate algorithms. European Journal of Operational Research, 59(3), pp. 345-358. doi: 10.1016/0377-2217(92)90192-C.

Laporte, G., 2009. Fifty Years of Vehicle Routing. Transportation Science, 43(4), pp. 408-416. doi: 10.1287/trsc.1090.0301.

Laporte, G., Toth, P. & Vigo, D., 2013. Vehicle routing: historical perspective and recent contributions. EURO Journal on Transportation and Logistics, 2(1-2), pp. 1-4. doi: 10.1007/s13676-013-0020-6.

Maulik, U. & Bandyopadhyay, S., 2000. Genetic algorithm-based clustering technique. Pattern Recognition, 33(9), pp. 1455-1465. doi: 10.1016/S0031-3203(99)00137-5.

Mitchell, M., 1998. An Introduction to Genetic Algorithms. 1 ed. Massachusetts: The MIT Press. doi: 10.7551/mitpress/3927.001.0001.

Nallusamy, R., Duraiswamy, K., Dhanalaksmi, R. & Parthiban, P., 2009. Optimization of Multiple Vehicle Routing Problems Using Approximation Algorithms. International Journal of Engineering Science and Technology, 1(3), pp. 129-135. doi: 10.48550/arXiv.1001.4197.

Nurprihatin, F. & Regent Montororing, Y. D., 2021. Improving vehicle routing decision for subsidized rice distribution using linear programming considering stochastic travel times. In: The 2nd International Conference on Sciences and Technology Applications (ICOSTA). Medan City: IOP Publishing. doi: 10.1088/1742-6596/1811/1/012007.

Oliver, I. M., Smith, D. J. & Holland, J. R. C., 1987. A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In: Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application. Massachusetts: L. Erlbaum Associates Inc., pp. 224-230. doi: 10.4324/9780203761595.

Omran, M. G. H., Engelbrecht, A. P. & Salman, A., 2007. An Overview of Clustering Methods. Intelligent Data Analysis, 11(6), pp. 583-605. doi: 10.3233/IDA-2007-11602.

Palhazi Cuervo, D., Goos, P., Sörensen, K. & Arráiz, E., 2014. An iterated local search algorithm for the vehicle routing problem with backhauls. European Journal of Operational Research, 237(2), pp. 454-464. doi: 10.1016/j.ejor.2014.02.011.

Papadimitriou, C. H. & Steiglitz, K., 1982. Combinatorial Optimization: Algorithms and Complexity. New York: Dover Publications Inc. doi: unavailable.

Pavai, G. & Geetha, T. V., 2017. A Survey on Crossover Operators. ACM Computing Surveys, 49(4), pp. 1-43. doi: 10.1145/3009966.

Prins, C., 2004. A simple and effective evolutionary algorithm for the vehicle routing problem. Computers & Operations Research, 31(12), pp. 1985-2002. doi: 10.1016/S0305-0548(03)00158-8.

Reddy, C. K. & Vinzamuri, B., 2014. A Survey of Partitional and Hierarchical Clustering Algorithms. In: C. C. Aggarwal & C. K. Reddy, eds. Data Clustering Algorithms and Applications. London: Taylor & Francis Group, pp. 87-110. doi: 10.1201/9781315373515-4.

Sivanandam, S. N. & Deepa, S. N., 2007. Introduction to genetic algorithms. New York: Springer. doi: 10.1007/978-3-540-73190-0.

Soni, N. & Kumar, T., 2014. Study of Various Mutation Operators in Genetic Algorithms. International Journal of Computer Science and Information Technologies, 5(3), pp. 4519-4521. doi: unavailable.

Sorensen, K., Sevaux, M. & Glover, F., 2017. A History of Metaheuristics. In: Handbook of Heuristics. Cham: Springer, pp. 791-808. doi: 10.1007/978-3-319-07124-4_4

Syswerda, G., 1989. Uniform Crossover in Genetic Algorithms. In: Proceedings of the 3rd International Conference on Genetic Algorithms. San Francisco: Morgan Kaufmann Publishers Inc., pp. 2-9. doi: unavailable.

Thangiah, S., 1995. Vehicle Routing with Time Windows using Genetic Algorithms. In: L. Chambers, ed. The Practical Handbook of Genetic Algorithms. Boca Raton: CRC Press. doi: 10.1201/9780429128332.

Toth, P. & Vigo, D., 2003. The Granular Tabu Search and Its Application to the Vehicle-Routing Problem. INFORMS Journal on Computing, 15(4), pp. 331-429. doi: 10.1287/ijoc.15.4.333.24890.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A., 2017. New benchmark instances for the Capacitated Vehicle Routing Problem. European Journal of Operational Research, 257(3), pp. 845-858. doi: 10.1016/j.ejor.2016.08.012.

Van Breedam, A., 1995. Improvement heuristics for the Vehicle Routing Problem based on simulated annealing. European Journal of Operational Research, 86(3), pp. 480-490. doi: 10.1016/0377-2217(94)00064-J.

van Laarhoven, P. & Aarts, E., 1987. Simulated Annealing: Theory and Applications. 1 ed. Dordrecht: Springer Netherlands. doi: 10.1016/0377-2217(94)00064-J.

Vekaria, K. & Clack, C., 1998. Selective crossover in genetic algorithms: An empirical study. In: Parallel Problem Solving from Nature. Berlin, Heidelberg: Springer, pp. 438-447. doi: 10.1007/BFb0056886.

Wilmink, F. & Uytterschaut, H., 1984. Cluster Analysis, History, Theory and Applications. In: Multivariate Statistical Methods in Physical Anthropology. Dordrecht: D. Reidel Publishing Company, pp. 135-175. doi: 10.1007/978-94-009-6357-3_11.

Yousefikhoshbakht, M., Didehvar, F. & Rahmati, F., 2014. An Efficient Solution for the VRP by Using a Hybrid Elite Ant System. International Journal of Computers Communications & Control, 9(2), pp. 56-62. doi: 10.15837/ijccc.2014.3.161.

Yu, B., Yang, Z. & Yao, B., 2011. A hybrid algorithm for vehicle routing problem with time windows. Expert Systems with Applications, 38(1), pp. 435-441. doi: 10.1016/j.eswa.2010.06.082.

Yu, X. & Gen, M., 2010. Introduction to evolutionary algorithms. London: Springer. doi: 0.1007/978-1-84996-129-5.

# Appendix A – Implementation of Clustering Techniques

## 1. Ward's Method

The code below is an implementation of the ward's method clustering algorithm. The function receives two variables as input, the nodes to be clustered, and the target number of clusters. The iterative process begins by assigning each node to its own cluster, and only merges two clusters for each iteration of the algorithm. To reduce time complexity and prevent recalculation of known values, a variance matrix is implemented to store the variance penalty of merging two clusters. The variance matrix is updated for each iteration of the algorithm as is the calculation of the centroids. When the desired number of clusters has been attained, the exit condition is satisfied, and the classification is returned.

```
def wards(nodes, num_clusters):
    # Set Variables
    clusters = len(nodes)
    classification = {}
    centroids = {}
    iteration = 0

    # Iterate Algorithm
    while clusters > num_clusters:
        if classification == {}:
            # Assign Nodes To Individual Clusters
            for n in nodes:
                classification[n] = n
                centroids[n] = nodes[n]

            # Set Variance Matrix
            variance_matrix = {}
            for i in centroids:
```

```
            centroid_1_nodes = [nodes[c] for c in classification if c == i]
            variances = {}
            for j in centroids:
                if j == i:
                    variances[j] = None
                elif j > i:
                    centroid_2_nodes = [nodes[c] for c in classification if c == j]
                    centroid_nodes = centroid_1_nodes + centroid_2_nodes
                    variances[j] = variance(centroid_nodes)
            variance_matrix[i] = variances
    else:
        # Find Cluster Pair With Minimum Variance Penalty
        cluster_1 = None
        cluster_2 = None
        min_variance = None
        for i in centroids:
            for j in centroids:
                if j > i:
                    variance = variance_matrix[i][j]
                    if (min_variance == None):
                        cluster_1 = i
                        cluster_2 = j
                        min_variance = variance
                    elif (variance != None) and (variance < min_variance):
                        cluster_1 = i
                        cluster_2 = j
                        min_variance = variance
        # Merge Cluster Pair
        cluster_nodes = {}
        for c in classification:
            if classification[c] == cluster_2:
                classification[c] = cluster_1
                cluster_nodes[c] = nodes[c]
```

```
            elif classification[c] == cluster_1:
                cluster_nodes[c] = nodes[c]
        # Remove And Update Centroids
        cluster_nodes_x = sum([cluster_nodes[n][0] for n in cluster_nodes])/len(cluster_nodes)
        cluster_nodes_y = sum([cluster_nodes[n][1] for n in cluster_nodes])/len(cluster_nodes)
        centroids.pop(cluster_2)
        centroids[cluster_1] = [cluster_nodes_x, cluster_nodes_y]
        # Update Variance Matrix
        variance_matrix.pop(cluster_2)
        for v in variance_matrix:
            if v < cluster_2:
                variance_matrix[v].pop(cluster_2)
        for i in centroids:
            centroid_1_nodes = [nodes[c] for c in classification if classification[c] == i]
            for j in centroids:
                if (j > i):
                    if (i == cluster_1) or (j == cluster_1):
                        centroid_2_nodes = [nodes[c] for c in classification if classification[c] == j]
                        centroid_nodes = centroid_1_nodes + centroid_2_nodes
                        variance_matrix[i][j] = variance(centroid_nodes)
    clusters = len(centroids)
    # Calculate Total Residual
    residual = sum([euclidean(centroids[classification[i]], nodes[i]) for i in classification])

# Return Best Classification, Centroids and Residual
return classification, centroids, residual
```

## 2. K-Means

The code below is an implementation of the k-means clustering algorithm. The function receives three variables as input, the nodes to be clustered, the target number of clusters, and the maximum number of iterations. The iterative process begins by calculating a random set of initial centroids, after which the nodes are assigned to each cluster. For each iteration of the algorithm, the centroids are recalculated, and all the nodes are reassigned. When the maximum number of iterations has elapsed, the exit condition is satisfied, and the classification is returned.

```
def kmeans(nodes, num_clusters, max_iteration):
    # Set Variables
    classification = {}
    centroids = {}
    iteration = 0

    # Iterate Algorithm
    while iteration < max_iteration:
        if iteration == 0:
            # Calculate Initial Centroids
            cen = 0
            iteration_centroids = {}
            for node in random.sample(list(nodes.keys()), num_clusters):
                cx = nodes[node][0]
                cy = nodes[node][1]
                iteration_centroids[cen] = [cx, cy]
                cen += 1
        else:
            # Recalculate New Centroids
            iteration_centroids = {}
            for i in range(num_clusters):
                points = [j for j in classification if classification[j] == i]

                if len(points) == 0:
```

```
                cx = 0
                cy = 0
            else:
                cx = float(sum([nodes[k][0] for k in points])/len(points))
                cy = float(sum([nodes[k][1] for k in points])/len(points))

            iteration_centroids[i] = [cx, cy]

    # Set Iteration Centroids
    centroids = iteration_centroids
    # Calculate Node Classification
    classification = {}
    for i in nodes:
        distance = [euclidean(nodes[i], centroids[j]) for j in centroids]
        classification[i] = distance.index(min(distance))

    # Calculate Total Residual
    residual = sum([euclidean(centroids[classification[i]], nodes[i]) for i in classification])

    iteration += 1

# Return Best Classification, Centroids, Residual and Iteration
return classification, centroids, residual
```

# Appendix B – Implementation of the Genetic Algorithm

## 1. Genetic Algorithm

The code below is an implementation of a simple GA that incorporates the concept of elitism. The function receives four variables as input, the objective function used to evaluate the fitness of each chromosome, the maximum number of iterations the algorithm will repeat, the number of chromosomes in the population, and the ratio of chromosomes that are retained from one generation to the next. The process begins by generating an initial population, after which the fitness of each chromosome is evaluated before iteratively looping through the reproductive process of selection, crossover, and mutation, which utilizes the default operators discussed in Chapter 4. The process ends when the maximum number of iterations has elapsed, and the best chromosome is returned.

```
def genetic_algorithm(objective_function, max_iteration, population_size, elite_ratio):
    # Set Variabes
    best_chromosome = None
    best_score = None
    best_iteration = None

    # Iterate Algorithm
    iteration = 0
    while iteration < max_iteration:
        if iteration == 0:
            # Generate Initial Population
            population = generate_population()
        else:
            # Calculate New Population
            population = {}
            if elite_ratio > 0:
                elite_scores = sorted(((a,b) for b,a in iteration_scores.items()))
```

```
        for i in range(0,population_size,2):
            if (i < (population_size*elite_ratio)):
                # Repeat Elite Chromosomes
                chromosome_1 = iteration_population[elite_scores[i][1]]
                chromosome_2 = iteration_population[elite_scores[i+1][1]]
            else:
                # Calculate New Chromosomes
                # Selection
                parent_1,parent_2 = selection_operator(iteration_population, iteration_scores)
                # Crossover
                offspring_1,offspring_2 = crossover_operator(parent_1, parent_2)
                # Mutation
                chromosome_1, chromosome_2 = mutation_operator(offspring_1),mutation_operator(offspring_2)
            # Store New Chromosomes
            population[i] = chromosome_1
            population[i+1] = chromosome_2
    # Set Generation Population
    iteration_population = population

    # Calculate Population Fitness
    iteration_scores = evalulate_fitness(iteration_population, objective_function)

    # Store Best Chromosome
    if (best_score == None) or (min(iteration_scores.values()) < best_score):
        best_iteration = iteration
        best_score = min(iteration_scores.values())
        best_chromosome = iteration_population[min(iteration_scores, key=iteration_scores.get)]

    iteration += 1

# Return Best Chromosome, Score and Iteration
return best_chromosome, best_score, best_iteration
```

## 2. Roulette Wheel Selection

The code below is an implementation of roulette wheel selection. The function receives two variables as input, a dictionary of the population of chromosomes being evaluated, and a dictionary with the corresponding fitness of each chromosome in the population. The process randomly selects and returns two parent chromosomes based on a probability, which is proportional to the fitness of each chromosome.

```
def roulette_wheel_selection(population, scores):
    # Set Variables
    max_score = max(scores.values())
    min_score = min(scores.values())
    sum_score = sum(scores.values())

    # Calculate Probabilities
    probabilities = [(min_score-(i-max_score)/sum_score) for i in scores.values()]
    if (sum(probabilities) == 0):
        probabilities =[1 for _ in scores.values()]

    # Select Parent Chromosomes
    parent_1, parent_2 = random.choices(population, weights=probabilities, k=2)

    # Return Selection
    return parent_1, parent_2
```

### 3. Rank Selection

The code below is an implementation of rank wheel selection. The function receives two variables as input, a dictionary of the population of chromosomes being evaluated, and a dictionary with the corresponding fitness of each chromosome in the population. The process randomly selects and returns two parent chromosomes based on a probability, which is proportional to the rank of each chromosome.

```
def rank_selection(population, scores):
    # Calculate Ranking
    ranking = [None for _ in range(len(scores))]
    rank = len(scores)
    for i in sorted(range(len(scores)), key=lambda j: [i for i in scores.values()][j]):
        ranking[i] = rank
        rank -= 1

    # Select Parent Chromosomes
    parent_1, parent_2 = random.choices(population, weights=ranking, k=2)

    # Return Selection
    return parent_1, parent_2
```

## 4. Tournament Selection

The code below is an implementation of tournament selection. The function receives two variables as input, a dictionary of the population of chromosomes being evaluated, and a dictionary with the corresponding fitness of each chromosome in the population. The process randomly selects and returns two parent chromosomes based on a random selection of two chromosomes, which compete for position in the output based on the fitness, for each parent.

```
def tournament_selection(population, scores):
    # Select Parent 1 Chromosomes For Tournament
    index_1a = random.randint(0, len(population)-1)
    index_1b = random.randint(0, len(population)-1)

    # Calculate Parent 1
    if scores[index_1a] < scores[index_1b]:
        parent_1 = population[index_1a]
    else:
        parent_1 = population[index_1b]

    # Select Parent 2 Chromosomes For Tournament
    index_2a = random.randint(0, len(population)-1)
    index_2b = random.randint(0, len(population)-1)

    # Calculate Parent 2
    if scores[index_2a] < scores[index_2b]:
        parent_2 = population[index_2a]
    else:
        parent_2 = population[index_2b]

    # Return Selection
    return parent_1, parent_2
```

## 5. One-Point Crossover

The code below is an implementation of one-point crossover. The function receives three variables as input, two parent chromosomes, and the chromosome length. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the randomly selected crossover point.

```
def one_point_crossover(parent_1, parent_2, chromosome_length):
    # Set Variables
    chromosome_1 = parent_1.copy()
    chromosome_2 = parent_2.copy()

    # Set Random Crossover Point
    crossover_point = random.randint(0, chromosome_length-1)
    crossover_range = range(crossover_point)

    # Calculate Offspring
    offspring_1 = [None for _ in range(chromosome_length)]
    offspring_2 = [None for _ in range(chromosome_length)]

    for i in range(chromosome_length):
        if (i in crossover_range):
            offspring_1[i] = chromosome_1[i]
            offspring_2[i] = chromosome_2[i]

        else:
            offspring_1[i] = chromosome_2[i]
            offspring_2[i] = chromosome_1[i]

    # Return Offspring
    return offspring_1, offspring_2
```

## 6. Two-Point Crossover

The code below is an implementation of two-point crossover. The function receives three variables as input, two parent chromosomes, and the chromosome length. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the two randomly selected crossover points.

```
def two_point_crossover(parent_1, parent_2, chromosome_length):
    # Set Variables
    chromosome_1 = parent_1.copy()
    chromosome_2 = parent_2.copy()

    # Set Random Crossover Points
    crossover_points = [random.randint(0,chromosome_length-1),random.randint(0,chromosome_length-1)]
    crossover_point_1 = min(crossover_points)
    crossover_point_2 = max(crossover_points)
    crossover_range = range(crossover_point_1,crossover_point_2)

    # Calculate Offspring
    offspring_1 = [None for _ in range(chromosome_length)]
    offspring_2 = [None for _ in range(chromosome_length)]

    for i in range(chromosome_length):
        if (i not in crossover_range):
            offspring_1[i] = chromosome_1[i]
            offspring_2[i] = chromosome_2[i]
        else:
            offspring_1[i] = chromosome_2[i]
            offspring_2[i] = chromosome_1[i]

    # Return Offspring
    return offspring_1, offspring_2
```

## 7. Uniform Crossover

The code below is an implementation of uniform crossover. The function receives four variables as input, two parent chromosomes, the chromosome length, and the probability threshold of crossover for each gene in the chromosome. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the sequence of randomly selected crossover points.

```
def uniform_crossover(parent_1, parent_2, chromosome_length, crossover_probability):
        # Set Variables
        chromosome_1 = parent_1.copy()
        chromosome_2 = parent_2.copy()

        # Set Random Crossover Points
        crossover_points = random.choices([1,0], weights=[crossover_probability,(1-crossover_probability)], k=chromosome_length)

        # Calculate Offspring
        offspring_1 = [None for _ in range(chromosome_length)]
        offspring_2 = [None for _ in range(chromosome_length)]

        for i in range(chromosome_length):
            if (crossover_points[i] == 1):
                offspring_1[i] = chromosome_1[i]
                offspring_2[i] = chromosome_2[i]
            else:
                offspring_1[i] = chromosome_2[i]
                offspring_2[i] = chromosome_1[i]

        # Return Offspring
        return offspring_1, offspring_2
```

## 8. Partially Mapped Crossover

The code below is an implementation of partially mapped crossover. The function receives three variables as input, two parent chromosomes and the chromosome length. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the two randomly selected crossover points.

```
def partially_mapped_crossover(parent_1, parent_2, chromosome_length):
    # Set Variables
    chromosome_1 = parent_1.copy()
    chromosome_2 = parent_2.copy()

    # Set Random Crossover Points
    crossover_points = [random.randint(0,chromosome_length-1),random.randint(0,chromosome_length-1)]
    crossover_point_1 = min(crossover_points)
    crossover_point_2 = max(crossover_points)

    # Calculate Offspring
    offspring_1 = [None for _ in range(chromosome_length)]
    offspring_2 = [None for _ in range(chromosome_length)]

    for i in range(chromosome_length):
        if (i in range(crossover_point_1,crossover_point_2)):
            offspring_1[i] = chromosome_2[i]
            offspring_2[i] = chromosome_1[i]
        elif (i in range(0,crossover_point_1)) or (i in range(crossover_point_2,chromosome_length)):
            offspring_1[i] = chromosome_1[i]
            map = False
            while map == False:
                if offspring_1[i] in chromosome_2[crossover_point_1:crossover_point_2]:
                    offspring_1[i] = chromosome_1[chromosome_2.index(offspring_1[i])]
                else:
```

```
                map = True

        offspring_2[i] = chromosome_2[i]
        map = False
        while map == False:
            if offspring_2[i] in chromosome_1[crossover_point_1:crossover_point_2]:
                offspring_2[i] = chromosome_2[chromosome_1.index(offspring_2[i])]
            else:
                map = True

    # Return Offspring
    return offspring_1, offspring_2
```

## 9. Order Crossover

The code below is an implementation of order crossover. The function receives three variables as input, two parent chromosomes and the chromosome length. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the two randomly selected crossover points.

```
def order_crossover(parent_1, parent_2, chromosome_length):
    # Set Variables
    chromosome_1 = parent_1.copy()
    chromosome_2 = parent_2.copy()

    # Set Random Crossover Points
    crossover_points = [random.randint(0,chromosome_length-1),random.randint(0,chromosome_length-1)]
    crossover_point_1 = min(crossover_points)
    crossover_point_2 = max(crossover_points)

    # Calculate Offspring
    sequence_1 = []
    sequence_2 = []
    for i in range(chromosome_length):
        if (i in range(0,chromosome_length-crossover_point_2)):
            if (chromosome_2[i+crossover_point_2] not in chromosome_1[crossover_point_1:crossover_point_2]):
                sequence_1.append(chromosome_2[i+crossover_point_2])
            if (chromosome_1[i+crossover_point_2] not in chromosome_2[crossover_point_1:crossover_point_2]):
                sequence_2.append(chromosome_1[i+crossover_point_2])
        else:
            if (chromosome_2[i-(chromosome_length-crossover_point_2)] not in chromosome_1[crossover_point_1:crossover_point_2]):
                sequence_1.append(chromosome_2[i-(chromosome_length-crossover_point_2)])
            if (chromosome_1[i-(chromosome_length-crossover_point_2)] not in chromosome_2[crossover_point_1:crossover_point_2]):
                sequence_2.append(chromosome_1[i-(chromosome_length-crossover_point_2)])
```

```
    offspring_1 = [None for _ in range(chromosome_length)]
    offspring_2 = [None for _ in range(chromosome_length)]
    for i in range(chromosome_length):
        if (i in range(0,crossover_point_1)):
            offspring_1[i] = sequence_1[i+(chromosome_length-crossover_point_2)]
            offspring_2[i] = sequence_2[i+(chromosome_length-crossover_point_2)]
        elif (i in range(crossover_point_1,crossover_point_2)):
            offspring_1[i] = chromosome_1[i]
            offspring_2[i] = chromosome_2[i]
        elif (i in range(crossover_point_2,chromosome_length)):
            offspring_1[i] = sequence_1[i-crossover_point_2]
            offspring_2[i] = sequence_2[i-crossover_point_2]

    # Return Offspring
    return offspring_1, offspring_2
```

## 10. Cycle Crossover

The code below is an implementation of cycle crossover. The function receives three variables as input, two parent chromosomes and the chromosome length. The process returns two offspring chromosomes that are calculated based on the parent chromosomes and the two randomly selected crossover points.

```
def cycle_crossover(parent_1, parent_2, chromosome_length):
    # Set Variables
    chromosome_1 = parent_1.copy()
    chromosome_2 = parent_2.copy()

    # Calculate Offspring
    repeat = []
    for i in range(chromosome_length):
        if (i == 0):
            repeat.append(chromosome_1[i])
        else:
            if (chromosome_2[chromosome_1.index(repeat[i-1])] not in repeat):
                repeat.append(chromosome_2[chromosome_1.index(repeat[i-1])])
            else:
                break
    offspring_1 = [None for _ in range(chromosome_length)]
    offspring_2 = [None for _ in range(chromosome_length)]
    for i in range(chromosome_length):
        if (chromosome_1[i] in repeat):
            offspring_1[i] = chromosome_1[i]
        else:
            offspring_1[i] = chromosome_2[i]

        if (chromosome_2[i] in repeat):
            offspring_2[i] = chromosome_2[i]
```

```
        else:
            offspring_2[i] = chromosome_1[i]
    # Return Offspring
    return offspring_1, offspring_2
```

## 11. Flip Mutation

The code below is an implementation of flip mutation. The function receives three variables as input, the chromosome to be mutated, the chromosome length, and probability threshold of mutation for each gene in the chromosome. The process returns a mutated chromosomes that is calculated based on the input chromosome and the randomly selected mutation points.

```
def flip_mutation(chromosome, chromosome_length, mutation_probability):
    # Set Variables
    chromosome_in = chromosome.copy()

    # Set Random Mutation Points
    mutation_points = random.choices([1,0], weights=[mutation_probability,(1-mutation_probability)], k=chromosome_length)

    # Calculate Mutation
    chromosome_out = [None for _ in range(chromosome_length)]

    for i in range(chromosome_length):
        if (mutation_points[i] == 1):
            chromosome_out[i] = abs(chromosome_in[i]-1)

        else:
            chromosome_out[i] = chromosome_in[i]

    # Return Mutation
    return chromosome_out
```

## 12. Swap Mutation

The code below is an implementation of swap mutation. The function receives three variables as input, the chromosome to be mutated, the chromosome length, and probability threshold of mutation for the chromosome. The process returns a mutated chromosomes that is calculated based on the input chromosome and the randomly selected mutation points.

```
def swap_mutation(chromosome, chromosome_length, mutation_probability):
    # Set Variables
    chromosome_in = chromosome.copy()

    if random.random() < mutation_probability:
        # Set Random Mutation Points
        mutation_points = [random.randint(0,chromosome_length-1),random.randint(0,chromosome_length-1)]
        mutation_point_1 = min(mutation_points)
        mutation_point_2 = max(mutation_points)

        # Calculate Mutation
        chromosome_out = [None for _ in range(chromosome_length)]
        for i in range(chromosome_length):
            if (i == mutation_point_1):
                chromosome_out[mutation_point_1] = chromosome_in[mutation_point_2]
            elif (i == mutation_point_2):
                chromosome_out[mutation_point_2] = chromosome_in[mutation_point_1]
            else:
                chromosome_out[i] = chromosome_in[i]
    else:
        chromosome_out = chromosome_in

    # Return Mutation
    return chromosome_out
```