Opleiding Informatica

Dynamic Multi-Objective Optimization for Real-Time Grocery Delivery Routing Using Natural Computing Algorithms

Mogyorósi Ádám

Supervisors:
Dr. Yingjie Fan & Robin van der Laag

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/06/2025

Abstract

The purpose of this paper is to explore the use of natural computing algorithms on real-time order allocation and routing problems in dynamic, same-day grocery delivery systems. The core of the problem was to balance two conflicting objectives: minimizing customer waiting times and minimizing total system-wide courier vehicle distance. The system operates under real-world constraints, such as vehicle capacity, fleet-size and dynamic, stochastic order arrival, moreover it is inspired by the operational model of the grocery delivery company Flink. Four natural computing algorithms (Simulated Annealing, Genetic Algorithm, Ant Colony Optimization, Artificial Bee Colony) and a baseline algorithm (Greedy) were implemented and compared across various scenarios, with six controlled variables. Based on the observed results, the Genetic Algorithm and the Artificial Bee Colony algorithm consistently outperform the rest of the algorithms by 3.7% on average, with the Artificial Bee Colony algorithm showing better solution quality and being more computationally efficient. Further analysis revealed that factors like depot location and demand distribution can significantly influence delivery performance, which can provide useful insights for practitioners, potentially improving such systems. For example, a strategically selected depot location could increase algorithmic performance by approximately 130% based on the observed results.

Contents

1 Introduction

1.1 Context and Background

Grocery delivery services have become an integral part of modern society, offering convenience, time efficiency and accessibility. By eliminating the need for physical store visits, these services save valuable time for individuals and families. Furthermore, during global health crises such as the COVID-19 pandemic, these services promote public safety by minimizing in-store interactions. According to recent data, the global revenue of the online food delivery market reached approximately $1.04 trillion in august 2024, with an estimated 1.4 billion people contributing to this market on a monthly basis [Wav24]. In the United States, the leading motivations to purchase groceries online include:

1. Time savings
2. Avoidance of impulse purchases
3. Ease of comparing products and prices

These compelling arguments have led the online grocery shopping market to demonstrate a continuous boom. By the end of 2024, the global market revenue had surged to $1.22 trillion [Del24], and it is projected to reach $1.4 trillion by the end of 2025, showing a year-on-year growth of approximately 14.75% [Sta25]. Moreover, the global market revenue is expected to show an annual growth rate of 7.83%, culminating in a $1.89 trillion market value by 2029 [Sta25]. As of 2025, an estimated 2.77 billion people, approximately 33% of the world's population, engage in online shopping, underlining the high demand for delivery services [Sel25]. This trend is further amplified by the increase in consumer demand for same-day delivery services [Kom24].Recent surveys indicate that approximately 62% of customers take the speed of the delivery into consideration, when shopping online, highlighting its vital role in enhancing customer satisfaction and repeat business [Shi23].

However, the implementation of same-day, last-mile delivery services present significant logistical challenges for e-commerce businesses, where efficient route planning is essential for maintaining customer satisfaction. The complexity of this problem is further amplified when factors such as traffic conditions, multiple delivery locations and narrow delivery windows are taken into account. Additionally, same-day deliveries require substantial investments in resources, including delivery vehicles and personnel, which can increase operational costs by a significant margin. Finding a balance between overcapacity and undercapacity is crucial to avoid unnecessary expenses and delays, further emphasizing the need for optimized delivery processes.

Given the inherent complexity of these challenges, natural computing algorithms have emerged as promising optimization algorithms for addressing last-mile delivery optimization problems [Vas14]. These algorithms are particularly well-suited for solving NP-hard problems, such as the vehicle routing problem (VRP), which plays a central role in efficient delivery planning. As a result, there has been a remarkable increase in the application of natural computing algorithms to optimize delivery routes and improve overall delivery efficiency in e-commerce businesses.

1.2 Problem Statement

Drawing inspiration from the operational framework of Flink, a leading grocery delivery company, this study addresses the complex challenge of real-time order allocation and routing in dynamic

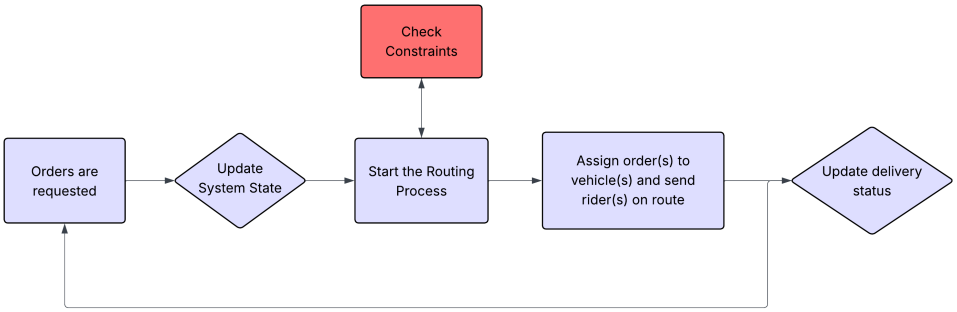


Figure 1: Flow diagram of the real-time order routing process.

grocery delivery systems. The problem at its core is the optimization of the delivery process in a highly dynamic environment, in which orders are continuously received, and decisions must be (re-)made within seconds to ensure a smooth and efficient service.
The problem is characterized by the following key features:

- **Dynamic Order Arrival**: Orders are requested stochastically.
- **Limited fleet size and capacity**: There are a fixed number of electric bikes available for delivery, each with limited carrying capacity, measured in number of bags they can carry.
- **Real-Time Decision Making**: Assignments must be made as soon as possible, no knowledge about future order requests.
- **Spatial and Temporal Considerations**: Order assignments are based on the current location of the vehicles and their next destinations.

The system does the following at each step:

1. Check if new orders are requested
    - If so, determine the new best order allocation and order based on the current knowledge.
2. Update vehicle and order states

Figure 1 depicts the overall workflow of the problem, starting from order requests and system state updates, through the constraint adhering routing process and order assignment, ending with delivery status updates before the cycle repeats.
This problem is inherently multi-objective, because the two main goals, customer waiting time and total rider distance, are not always aligned and may even conflict in certain scenarios. For instance:

- **Optimizing Courier Distance**: Batching some orders and delivering them in one trip can reduce the total distance traveled by riders. However, this approach might delay the delivery of some orders, which could lead to increased customer waiting times.
- **Minimizing Waiting Time**: Assigning riders to immediately deliver orders can reduce waiting times for some individual customers. However, this approach could result in fragmented routes, where couriers make multiple short trips, overall increasing the total distance traveled.

While it is true that doing multiple shorter trips may occasionally lead to faster deliveries, this is not guaranteed. For example, delivering orders to two customers sequentially in one trip may reduce the total distance traveled but force the second customer to wait longer. On the other hand, dispatching separate riders for each order will minimize individual waiting times, at the cost of increasing the total distance traveled by the fleet. This balance underlines the two focus points of the problem: customer waiting times reflect the service quality and customer satisfaction, while total rider distance relates to operational efficiency and cost-effectiveness. These objectives represent the interests of the two key stakeholders of the e-commerce ecosystem: customers and the company. This balance is what makes this problem both challenging and practically significant.
The environment of this problem is designed to closely mimic real-world conditions, it is stochastic and dynamic. Orders appear dynamically, which means that the system has no prior knowledge of them before the optimization begins, introducing stochasticity. In addition, variations in order size, delivery locations, and timing add an extra layer of complexity for the decision-making process.
In summary, this research aims to tackle a multi-objective problem characterized by real-time decision making, trade-offs between the stakeholders and a dynamic, stochastic environment. This study strives to contribute to the development of more efficient and customer-centric grocery delivery systems by addressing these challenges, ultimately benefiting both businesses and customers.

1.3 Research Objectives

The aim of this thesis is to develop and compare efficient multi-objective natural computing algorithms for a real-time order allocation and routing problem in dynamic grocery delivery systems. To address this issue, the following research questions were formulated:

1. What is the trade-off between customer waiting time and the system-wide distance traveled by the riders?
2. Which algorithm can solve the problem in the least amount of function evaluations?
3. How do these algorithms perform under different conditions, such as a sudden increase in the number of orders, different rider fleet sizes or varying order locations?

1.4 Methodology

This study will be structured using a three-phase approach, in order to address the research questions. The three phases will be: problem modeling, algorithm development and evaluation.

1.4.1 Problem Modeling and Simulation

At first, a simulation model will be developed to represent the dynamic grocery delivery system. The model will incorporate the following elements:

- **Orders**: Defined as nodes with the following attributes: order placement time, order size (the number of bags) and the geographic location of the order.
- **Riders**: Defined as agents with a maximum capacity constraint and dynamic availability.
- **System Dynamics**: These will simulate real-time order arrivals and rider movements.

A mathematical formula will formalize the multi-objective optimization problem, aiming to minimize the following two metrics:

- Customer waiting time (representing customer satisfaction).
- System-wide total distance traveled by riders

1.4.2 Algorithm Development and Implementation

Four natural computing algorithms will be adapted and implemented in Python in order to optimize the problem:

- Ant Colony Optimization (ACO)
- Genetic Algorithm (GA)
- Simulated Annealing (SA)
- Artificial Bee Colonies (ABC)

Every single one of these algorithms will have to be specifically adapted for this problem. Mainly how the solutions will be encoded (so the route representations for riders), how the fitness function will be designed in parallel with the weighted cost function, and the hyperparameters will have to be fine-tuned to achieve the balance between exploration and exploitation. Additionally, a simple greedy algorithm will also be implemented as a baseline for comparison. The greedy algorithm was selected due to its low computational requirements and simple, straightforward logic, making it a useful reference point when evaluating the performance and complexity of the more advanced natural computing approaches.

1.4.3 Evaluation and Analysis

To extensively evaluate the performance of these algorithms, datasets will be generated with diverse scenarios. The evaluation will be conducted based on specific criteria, which can be found in Table 1.

Table 1: Comparison Criteria

- Varying number of orders
- Varying spatial distributions of orders
- Different rider fleet sizes
- Average delivery time
- Total rider travel distance
- Computational efficiency

The computational efficiency metric will be measured using the number of function evaluations required to reach near-optimal solutions, since this way the results will show an accurate representation of the algorithms' performance regardless of hardware.
Statistical tests (paired/independent t-tests) will be used in order to determine significant differences in performance, mainly entertaining research questions 1 and 3. The computational efficiency metric will specifically address research question 2, highlighting the algorithms capable of real-time decision making.

1.5 Significance and Contributions

This study is significant from both practical and theoretical points of views for the modern grocery delivery systems. From a practical perspective, the findings strive to allow companies to make

data-driven decisions when selecting natural computing algorithms tailored to their operational needs, this study offers a broader empirical foundation by introducing extensive testing, diverse scenarios and deep performance analysis. Moreover, by quantifying the trade-offs between total delivery distance and customer waiting time, this study aims to provide insights for achieving an optimal balance between cost efficiency and service quality, thereby enabling more informed decision-making. For same-day delivery frameworks, this balance is crucial, especially to maintain customer satisfaction, reduce operational expenses and enhance the scalability of the operation.
From a theoretical perspective, the contributions of this research address a significant gap in existing literature. Certain natural computing algorithms have been explored (e.g., genetic algorithms, ant colony optimization, simulated annealing) individually in the context of Dynamic Vehicle Routing Problems (DVRPs) in previous studies [LLKT21] [HHBC22] [KBP+23] [ZW24] [CPPJ+24] [CZBC21] [RFB+24] [CSZ22] [YAJ+23], however a systematic comparison of their relative strengths and weaknesses in a multi-objective, real-time, same-day, grocery delivery context is yet to be explored. Existing literature mostly focuses on static or single objective optimization, neglecting the complexity of the real world, for example real-time order arrival, and competing objectives such as minimizing customer waiting time and total distance traveled by the couriers. This study plans to address this gap in the literature by creating a model for the problem of the same-day, modern, grocery delivery systems, based on the company Flink, then implementing and benchmarking four natural computing algorithms under real-time, multi-objective scenarios, analyzing and quantifying the results.
As opposed to previous studies, that optimize pre-planned delivery routes for future execution, this research focuses on the challenges of real-time decision-making, where potential solutions must be generated within seconds to accommodate dynamically incoming orders. This requirement imposes a significant constraint on potential algorithms, since even minor delays (even 2-7 seconds) in computation can lead to delivery bottlenecks and customer dissatisfaction. While computational speed may not directly affect customer satisfaction, especially for companies with greater computational resources, it remains a critical factor in real-time decision making. This study advances the understanding of how natural computing algorithms can be adapted for time-sensitive applications, by evaluating algorithms not only on solution quality, but also on computing speed. These insights are invaluable for e-commerce businesses operating in fast-paced environments.

1.6 Thesis Overview

This chapter contains the introduction, explaining the background of the problem, problem statement, methodology and significance. Section 2 discusses related work and gaps in the literature. Section 3 addresses model selection, key assumptions and simulation design. Section 4 contains the implementation of the algorithms. Section 5 describes the experiments. Section 6 discusses the experimental results and provides some practical insights. Section 7 concludes.
This study was conducted as a bachelor's thesis at LIACS, under the supervision of Dr. Yingjie Fan and Robin van der Laag.

2 Related Work

The vehicle routing problem has been extensively studied in logistics, with recent attention shifting towards dynamic and multi-objective versions, which better represent the complexities of the real world. This paper contributes to two key areas of the related literature: same-day delivery operations and dynamic vehicle routing with a specific focus on order allocation.

2.1 Static Approaches

A critical challenge in same-day delivery is optimizing routes in real time while balancing competing objectives. Early work on the VRP mostly focused on static and single objective optimization, often assuming perfect prior knowledge, highly simplifying this complex problem and ignoring real-world uncertainties. For example, Tao et al. [TLW22] addressed the joint optimization of vehicle routing and order split allocation using variable neighborhood search (VNS), but their model assumed multiple depots, a static environment and a singular objective, failing to account for dynamic order arrivals or multi-objective trade-offs. Similarly, in Schubert et al. [SKH21] sequential approaches to vehicle route optimization were implemented by balancing the trade-off between picking and delivery costs using VNS with mixed integer programming. While their study improved cost optimization, it remained confined to a static environment with a single objective.

Advanced techniques have been explored in other studies, but most of them retained similar limitations. Li et al. [LLKT21] researched the simultaneous product and service delivery using mixed integer programming and adaptive large neighborhood search with simulated annealing. In spite of their innovative approach, their model assumed a static environment with no uncertainties and an objective only focusing on cost reduction. Similarly, in Huang et al. [HHBC22] a variant of the ant colony optimization algorithm (ACO) was implemented to solve the VRP with drones, however their model assumed a static environment and a singular objective. An implementation of the particle swarm optimization algorithm based on roulette wheel selection for VRPs with time windows was explored in Kashyap et al. [KBP+23], yet their model also assumed a singular objective (the total travel costs) and a static environment without any uncertainties.

Recent studies have introduced novel techniques, but remain constrained by static assumptions. The recent study, Zhu et al. [ZW24], improved the ACO method with a novel pheromone update mechanism for VRPs, but their model only considered the singular objective of minimizing the distance traveled by couriers and ignored the dynamic conditions of the real world. The research conducted by Ismail Hossain Tausif [Tau23] explored a variant of the VRP which involves drones and trucks using mixed integer linear programming and Gurobi solvers. While their research introduced innovative delivery methods, it was limited to a static environment and a single objective. Osaba et al. [OVRA24] produced a solution to the VRP using quantum annealers, optimizing fleet composition and route length, however their model also assumed a static environment. Research by Yu et al. [YAJ+23] proposed a new variant of the VRP involving simultaneous pickup and delivery with occasional drivers, implementing a solution using mixed integer linear programming, simulated annealing. However, their work did not consider dynamic scenarios and focused solely on cost minimization, leaving room for further research.

As real-world delivery environments are rarely ever static, they involve fluctuating demands and uncertainties that require more adaptive, real-time solutions, necessitating dynamic approaches.

2.2 Dynamic Approaches

In contrast to static models, recent research have begun exploring dynamic environments and multi-objective optimization, which are more representative of real-world delivery systems. For instance, Charlotte Ackva and Marlin W. Ulmer [AU23] implemented a mixed integer programming solution for same-day delivery VRPs, with their sole objective being to maximize the number of packages delivered daily. While their work did consider dynamic order arrivals, it deviated from this study by assuming multiple depots and a single objective. Similarly, Zhang et al. [ZLFV23] proposed and implemented a knapsack based linear model to approximate the expected reward of accepted requests in a dynamic environment. Although their approach considered a dynamic environment, it did not employ natural computing algorithms or consider multi-objective trade-offs. The recent paper by Côté et al. [CQGI21] compared efficient algorithms, ranging from a simple reoptimization heuristic to a sophisticated branch-and-regret heuristic, for same day delivery problems. Their model assumed a dynamic environment with uncertainties such as the arrival rate of customer requests, and the time items became available. This work focused on maximizing the number of requests served daily, while minimizing the traveled distance. A study by Hossein Fotouhi and Elise Miller-Hooks [FMH23] developed a scalable parallel algorithm with progressive hedging and a shrinking time horizon, creating a new mathematical formulation for the same-day delivery time guarantee problem. This study focused on maximizing customer satisfaction and minimizing delivery-related costs in a dynamic environment, where uncertainties included the likelihood of customers completing purchases and their satisfaction with delivery times.

A paper by Chandratreya et al. [CPPJ+24] approached the last-mile delivery as a traveling salesman problem (TSP) and implemented a genetic algorithm with local search heuristics. They considered traffic conditions and delivery windows as uncertainties, with objectives including minimizing travel distance, delivery times, and optimizing fleet size. In Bruni et al. [BFFP23] proposed an improved machine learning heuristic for two-stage optimization, minimizing total costs, and optimizing fleet composition. Their solution utilized machine learning, progressive hedging, Gurobi's exact solver, and Monte Carlo simulations, with uncertainties including order volume and size fluctuations. Research by Chu et al. [CZBC21] introduced a data-driven optimization approach combining machine learning techniques with capacitated vehicle routing optimization. Their smart predict-then-optimize framework minimized total travel time and costs, but they only considered stochastic travel time as an uncertainty, leaving stochastic order flow for future work.

A recent study by Rhouzali et al. [RFB+24] optimized the VRP using a branch-and-bound method and a genetic algorithm, considering fuel, tires, maintenance costs as uncertainties. Their sole objective was to reduce the overall operational costs. Chen et al. [CUT22] were among the first to implement deep Q-learning for same-day deliveries and for dynamic routing problems. Their objective was to maximize the number of accepted orders, with uncertainties such as the number, location, and timing of customer requests. In Cui et al. [CSZ22] traffic uncertainties were addressed and a novel customer satisfaction function was introduced. Their primary objective was to minimize the overall costs, and they implemented a memetic algorithm combining a genetic algorithm with a variable neighborhood search strategy. They identified multi-stage optimization as a future research direction.

While these studies focus on more realistic delivery models and environments, important gaps remain in the literature, especially in the application of natural computing algorithms.

2.3 Gaps in the Literature

Despite these advancements, significant gaps remain in the literature. Most existing studies either focus on static environments or single objectives, failing to address the dynamic, multi-objective nature of the modern real-time grocery delivery systems. Moreover, while some studies have explored dynamic conditions, they often rely on traditional optimization algorithms, or utilize overly complex algorithms, rather than natural computing algorithms, which offer greater flexibility and reliability especially for real-time decision making. Furthermore, few studies have comprehensively evaluated and compared algorithm performance under varying conditions, such as fluctuations in order size and volume or diverse spatial distribution of orders.
For example, while Chandratreya et al. [CPPJ+24] and Bruni et al. [BFFP23] addressed dynamic environments and multi-objective optimization, they employed genetic algorithms and machine learning techniques but did not compare their performance against other natural computing methods or evaluate them under a wide range of dynamic scenarios.
In summary, key gaps remain in the literature, particularly in the utilization of natural computing algorithms to dynamic grocery delivery systems. This thesis aims to bridge this gap by developing and comparing natural computing algorithms, specifically ACO, GA, SA, and ABC, for dynamic, multi-objective VRPs in the modern, same-day grocery delivery systems. Furthermore, this thesis is guided by three main research questions that aim to explore the core challenges of dynamic modern grocery delivery optimization: the balance between customer waiting times and system-wide vehicle travel distance, the efficiency of different algorithms in terms of function evaluations, and their robustness under varying conditions. With the use of these research questions, the aim of this study is to create a comprehensive and structured evaluation of these natural computing approaches in a highly dynamic, multi-objective environment.

3 Problem Modeling

The foundation of this study mainly relies on an accurate simulation of same-day delivery operations. An existing framework was selected and adapted, due to practical considerations, since it is better suited for rapid prototyping and experimentation than creating the framework custom, from scratch.

3.1 Key Assumptions and Constraints

Considering the practical limitations and assumptions of this study are critical for interpreting the results and scope of the findings.
Modeling assumptions

- **Travel Time Approximation**: The travel time is modeled as the Manhattan distance with a 10% Gaussian noise factor to simulate traffic variability, see Equation 2. While real-world traffic data, based on time and season could refine this estimate, this approximation gives a reasonable baseline for urban grid navigation.
- **Spatial Representation**: The service area is represented as a discrete grid, with customer locations restricted to integer coordinates. For calculating distance, the Manhattan distance was chosen, simply because it is well suited for urban environments with grid-like road networks. The travel time approximation is directly based on the distance metrics.

- **Vehicle Constraints**: As this research primarily focuses on electric bikes as delivery vehicles, their capacity is limited between 5-10 bags at once depending on the scenario and this number is assumed to be the same across all vehicles in the simulation. Additionally, the order pickup time is set to 2 minutes per order, this accounts for all loading procedures.
- **Scope**: This paper mostly focuses on dynamic order allocation and spatial clustering over detailed route optimization.
- **Order Size**: Order size (or the number of bags) is randomized between 1 and 5 throughout the experiments, therefore edge cases, where a single vehicle cannot deliver an order alone, are not considered.

3.2 Model selection and Customization

After evaluating several dynamic vehicle routing problem simulation frameworks, the dvrpsim python package was identified to be the most suitable match for this research [HT24]. This decision is mainly motivated by the following factors:

- Dvrpsim is specifically created for simulating dynamic vehicle routing problems.
- It provides a flexible, modular and customizable environment.
- It allows to connect external routing algorithms.

The core simulation environment is customized as shown in Listing 1. The algorithm variable was created for convenience, it allows different routing algorithms to be evaluated on the same model. A method for counting function evaluations is integrated for performance monitoring purposes. It can be found in Appendix A.

Listing 1: Model environment

```
class TheModel(Model):
    def __init__(self, algorithm) -> None:
        super().__init__()
        self.algorithm = algorithm
        self.function_call_counts = []

    def routing_callback(self):
        state = self.get_state()

        # For measuring function evaluations
        with FunctionCounter() as counter:
            result = self.algorithm(state, self)
        self.function_call_counts.append(counter.num_evals)

        return result
```

The fleet is modeled through an EBike class (which can be found in Listing 2) aiming to model realistic urban mobility constraints:

- The travel time of vehicles is calculated by the manhattan distance with 10% Gaussian noise, in order to account for traffic variability and urban grid navigation, see Figure 2.
- Some functions were created for checking capacity constraints in real time.
- Routing can be triggered when any vehicle enters the hub, however this is only necessary for the greedy algorithm.

$$\text{travel\_time}(o, d) = \mathcal{N}(1, 0.1^2) \cdot (|o_x - d_x| + |o_y - d_y|) \quad (1)$$

where: $o = (o_x, o_y)$ coordinates of the origin

$d = (d_x, d_y)$ coordinates of the destination

$|o_x - d_x| + |o_y - d_y|$ Manhattan distance between origin and destination

$\mathcal{N}(1, 0.1^2)$ Gaussian noise with mean 1 and variance 0.01

Figure 2: Equation of the travel time variable.

Listing 2: Vehicle class

```
class EBike(Vehicle):
    def __init__(self, id: str) -> None:
        super().__init__(id)
        self.total_distance = 0

    def travel_time(self, origin: Location, destination: Location) ->
       int | float:
        distance = manhattan_distance(origin.x, origin.y, destination.x,
            destination.y)
        self.total_distance += distance
        return random.gauss(1, 0.1) * distance

    @property
    def current_load(self):
        return sum(order.quantity for order in self.carrying_orders)

    def can_accept(self, order, accepted_order_ids):
        print(order.status)
        return (order.status != 'accepted') & (sum([o.quantity for o in
           list(self.model.orders) if o.id in accepted_order_ids]) +
           self.current_load + order.quantity <= self.capacity)
```

The orders are generated for each simulation with a random size and location based on several other variables which are further explained in Section 3.3.
The chosen framework does have some technical limitations:
The dvrpsim model does not natively support some critical functionalities, such as:

- Handling multiple order requests at the same timestamp
- State evaluation or state replication for evaluation, thus requiring a custom simulation freezing evaluation function, see Appendix B.
- Any kinds of visualization tools

Not handling multiple requests at once might bias results in high-load scenarios where orders arrive in bursts. On the other hand, the lack of evaluation and visualization tools only impact the development process but not the results.
After careful consideration, these constraints were deemed acceptable, since the main focus of this research is algorithmic comparison, rather than real-world deployment.

3.3 Simulation Design

In order to vigorously evaluate algorithm performance, a controlled experimental framework was developed. Six pivotal variables were identified for further analysis:

1. **Order volume** (number of orders): Evaluating the difference in performance across varying order volumes, will give insights into the scalability and stress tolerance of the algorithms.
2. **Service area size**: The size of the delivery area impacts the average delivery distance and customer wait times, which could amplify routing inefficiencies.
3. **Fleet size**: The optimal fleet size is a complex question, which is not the goal of this study, however testing different fleet sizes, can reveal how algorithms perform under under- or over-capacitated conditions.
4. **Demand distribution** (uniform vs fluctuating, see Figure 3): In the real-world, the demand distribution is rarely uniform, therefore by testing this variable with various fluctuating settings, the assessment of algorithm robustness is possible under more realistic conditions.
5. **Depot position** (center vs corner placement, see Figure 4): Depot placement influences the need for better order clustering and spatial adaptability.
6. **Vehicle capacity**: Varying the vehicle capacity variable allows for testing order clustering and routing under different logistical limitations.

The experimental design is based on the single-variable perturbation approach, where one variable is varied while the other variables remain fixed. This isolates and highlights the impact of each variable on algorithm performance. While this approach isolates effects, it ignores potential interactions

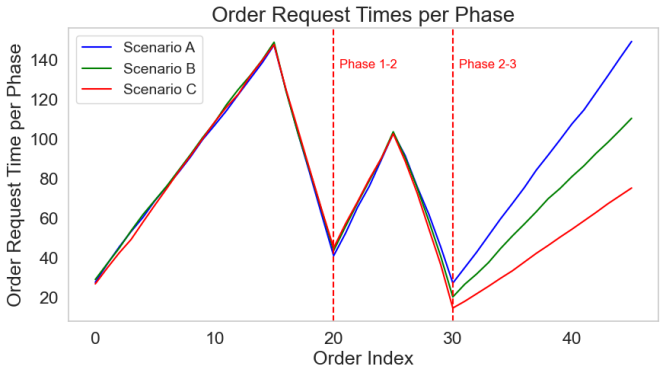
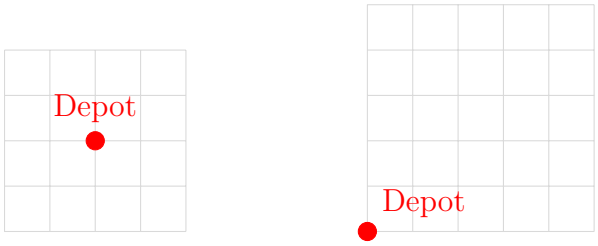


Figure 3: Different values for the demand distribution. The main difference between the three scenarios is the time it takes them to request all 50 orders.



Medium service area with the depot in the center

Larger service area with the depot in the corner

Figure 4: Depot positions, gray grid represents service area, red dot indicates depot location.

between variables. Exploring these interactions is left for future work due to lack of computational resources and time constraints.
The experimental design consists of two main phases:

1. **Selection phase**: All five algorithms are tested on three baseline scenarios to identify the best performing algorithms.

2. **Experimental phase**: The best two algorithms are evaluated across 30 scenarios (6 variables x multiple values each)

This experimental setup aims to combine thoroughness with computational efficiency, as the computational resources are limited.

3.3.1 Baseline Scenarios

Three scenarios were developed in order to represent common operational conditions while stressing different algorithmic capabilities, they can be found in Table 2. The purpose of the "Medium - Center" scenario was a balanced baseline for general performance. The "Bigger - Corner" scenario tests the scalability and depot placement sensitivity of the algorithms. Testing under complex, high-pressure conditions is conducted using the "Medium - Stressed" scenario.

Scenario	Depot Position	Orders (#)	Area Size	Fleet Size	Capacity (units)	Demand Distribution
Medium - Center	Center	50	20×20	5	5	Uniform, every 15-min
Bigger - Corner	Corner	150	50×50	10	5	Uniform, every 30-min
Medium - Stressed	Corner	50	50×50	3	5	Uniform, every 8-min

Table 2: Scenario parameters comparison

3.3.2 Experimental Scenarios

Each of these scenarios vary one critical variable while holding other variables fixed at baseline values. The reasoning behind the variable choices is based on real-world data from the company Flink. The variable choices can be found in Table 4. The baseline variables can be found in Table

3, these values remain unchanged throughout the experiments, except for the variable Number of Orders, as it is increased to 150 when testing the Fleet size parameter in order to necessitate the use of all vehicles.

Scenario	Depot Position	Orders (#)	Area Size	Fleet Size	Capacity (units)	Demand Distribution
Baseline Values	Center	50	50×50	5	5	Uniform

Table 3: Baseline parameter values for the experimental scenarios

Table 4: Experimental Scenarios and Relevance

Scenario	Tested Values	Relevance
num_orders	200 (A), 500 (B)	Peak vs off-peak demand volumes
area_size	100×100 (A), 200×200 (B)	smaller vs larger service areas
fleet_size	5 (A), 20 (B), 50 (C)	Small vs large operation scales
layout	Center (A), Corner (B)	Hub placement strategies
capacity	5 (A), 8 (B), 10 (C)	E-bike capacity configurations
distributions	Demand spikes (A–C)	Rush hour patterns (see Appendix D and Figure 3)

4 Algorithm Implementation

4.1 Cost Function

As the goal of this research is to find a balance between the total customer wait time and the total system wide travel distance, a mathematical formula was created for the algorithms to use. It is the weighted average of the total wait time (W) and the total travel distance (D), it can be found in Equation 2. The values for both α and β were set to be 0.5 throughout the experiments, since this way customer satisfaction and company resources share an equal amount of weight.

$$C = \alpha W + \beta D \tag{2}$$

4.2 Greedy Algorithm

This algorithm was implemented in order to provide a baseline for comparing the more complex algorithms on solution quality and computational speed, the reason for choosing the greedy algorithm specifically can be found in Section 1.4.2. The greedy algorithm is a heuristic approach, that makes locally optimal decisions at each step to minimize delivery costs.
It processes open orders sequentially without global optimization, and assigns them to the first available vehicle that can accommodate them. In case there are no unassigned orders or no idle vehicles at the hub, it terminates without routing and waits for the next vehicle to arrive at the hub. The pseudocode for the greedy algorithm can be found in Listing 3 and the full implementation can be found in Appendix E.

Listing 3: Pseudocode for the Greedy Algorithm

```
WHILE unassigned orders exist:
    FOR each idle vehicle at depot
        FOR each unassigned order:
            IF vehicle can carry order (based on capacity):
                ADD to order to vehicle route
                ACCEPT order
    APPEND depot to vehicle route
RETURN assignments
```

Drawbacks of the greedy algorithm include:

- Possibility of missing globally efficient routes due to only focusing on the immediate gain.
- Already created routes cannot be revised dynamically.

4.3 Mutation Operators

In metaheuristic algorithms mutation operators serve as the primary mechanism for the balance between exploration, searching new areas of the solution space, and exploitation, refining existing, promising solutions by introducing variation. Smaller mutations usually promote exploitation, while larger mutations tend to contribute to exploration.

4.3.1 Order Swap

The Order Swap operator randomly selects two eligible vehicles and swaps orders with each other. Vehicle eligibility means that the vehicle has at least one planned route that has not yet been initiated. This operation allows the algorithm to dynamically revise upcoming trips potentially uncovering more efficient vehicle-order pairings. The use of this operator might result in more trips planned, if due to capacity constraints the swapped order cannot fit into the planned trip. This operation introduces moderate exploration, while preserving most of the existing route structure. An example workflow of the order swap operator can be found in Figure 5.

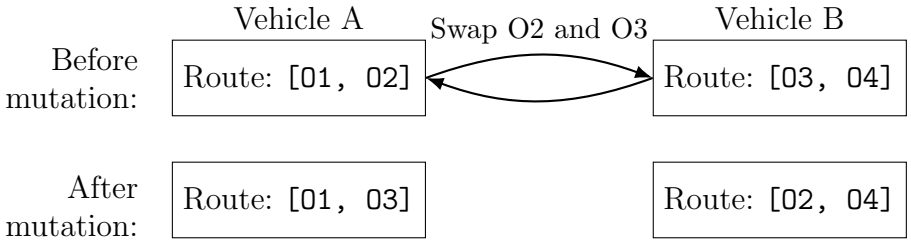


Figure 5: An example workflow of the order swap mutation operator. (The selected trips are not yet initiated.)

4.3.2 Order Reassignment

The Order Reassignment operator randomly selects an eligible order, and reassigns it to a different vehicle. Order eligibility means that the order is not yet picked up by any vehicle. With the use of

this operator the workload balance can be explored, potentially leading to better route efficiency and load balancing. This operator assumes that there is a plan for every order. An example workflow or this mutation operator can be found in Figure 6.

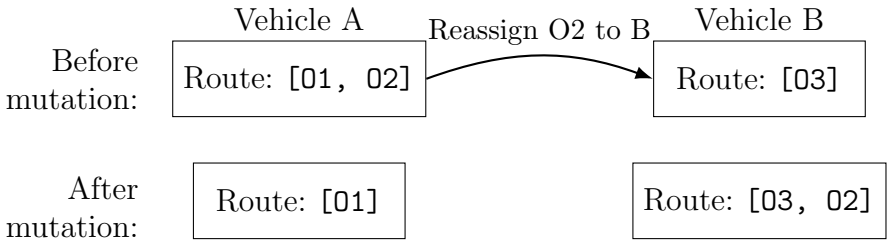


Figure 6: An example workflow of the order reassignment mutation operator. (The selected trips are not yet initiated.)

4.3.3 Route Shuffling

The Route Shuffling operator modifies the delivery sequence within a single vehicle's upcoming trip. By simply shuffling the order of deliveries, it explores alternative visitation sequences that could result in shorter trips or shorter waiting times. As this operator maintains the same set of orders per vehicle, it is mainly exploitative. Figure 7 depicts and example workflow for this mutation operator.

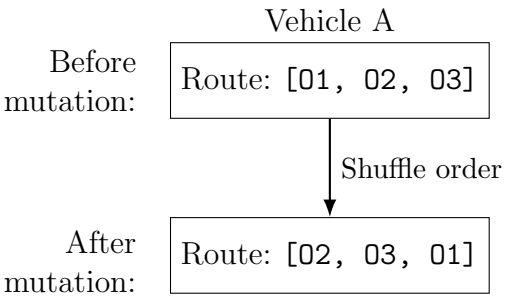


Figure 7: An example workflow of the route shuffling mutation operator. (The selected trip is not yet initiated.)

4.4 Simulated Annealing

This metaheuristic approach aims to find near-optimal solutions by utilizing controlled randomization. It balances solution exploration and exploitation by temperature decay, as the temperature decays to zero, the acceptance probability also decays to zero, see Figure 8. The algorithm is able to overcome getting stuck in local optima by accepting worse solutions with a probability based on the solution quality and the current temperature (see Figure 9).
The main steps of the algorithm can be seen in Listing 4. The mutation operators used for the Simulated Annealing algorithm are order swap, order reassign and route shuffle, these are explained in Section 4.3.

$$T_k = T_{k-1} \cdot \alpha \tag{3}$$

where T_k : Temperature at the current iteration k

T_{k-1} : Temperature at the previous iteration

α : Decay factor $(0 < \alpha < 1)$

Figure 8: Temperature decay equation in simulated annealing.

$$\text{prob} = e^{-\frac{\Delta f}{T}} \tag{4}$$

where: Δf = fitness difference (neighbor - current)

T = current temperature

Figure 9: Probability of accepting a solution in simulated annealing.

Listing 4: Pseudocode for Simulated Annealing

```
GENERATE random feasible solution
EVALUATE solution

WHILE Temperature > Min_Temperature AND max iterations not reached:
    MUTATE solution
    EVALUATE solution
    IF quality_difference < 0 OR random [0, 1) < exp(-quality_difference
       /Temperature):
        ACCEPT mutated solution
    UPDATE best solution if improved
    DECAY Temperature

RETURN best solution found
```

The full implementation of the Simulated Annealing algorithm can be found in Appendix F.

4.4.1 Parameters

The parameters were adapted from [ZIR+21]. These values can be found in Table 5.

Table 5: Simulated Annealing Parameters

Parameter	**Value**	**Role**
Initial Temp (T)	1000	Controls early-stage randomness
Min Temp	0.001	Stopping threshold
Cooling Rate (α)	0.999	Exponential decay speed
Max Iterations	100	Computation budget

4.5 Genetic Algorithm

This metaheuristic algorithm is inspired by the process of natural selection. It evolves its solutions through three main components:

1. Population based search with an elitist selection mechanism
2. Route recombination through a problem-specific crossover operator (see Section 4.5.1)
3. Mutation operators, see Section 4.3

The pseudocode for the Genetic Algorithm can be found in Listing 5.
No clear signs of premature convergence were observed throughout the experiments, and given that the GA consistently produced top ranking results, it is assumed that the population diversity was maintained at a sufficient level.

Listing 5: Pseudocode for Genetic Algorithm

```
INITIALIZE population with random generated feasible solutions
FOR generation = 1 to max generations:
    EVALUATE fitness for all individuals (using the simulate_evaluate
       function)
    SELECT parents through tournament selection (best out of 3)
    APPLY crossover with probability pc
    MUTATE offspring using mutation operators
    COMBINE parent and offspring populations
    SELECT next generation using elitism
RETURN best solution found
```

4.5.1 Crossover Operator

For this study, a problem specific crossover operator was implemented to recombine two parent solutions into a single child solution. As the application of traditional, permutation based crossover operators would produce invalid solutions (as they may duplicate or omit orders), the implemented solution respects the constraints of ongoing trips and dynamic order handling.
The main steps of the crossover operator are:

1. Initialize the child solution by deep-copying the first parent
2. Identify all orders eligible for reassignment (i.e., not currently being serviced)
3. Randomly select a small subset of these orders to be swapped
4. Remove the selected orders from the child solution
5. Reassign the selected orders from the second parent to the corresponding vehicles

To formalize this process, let P_1 and P_2 denote the two parent solutions. Let $A = \{a_1, a_2, \ldots, a_n\}$ be the set of all eligible orders in P_1 and let $B = \{b_1, b_2, \ldots, b_n\}$ be the set of eligible orders in P_2. Let $I \subseteq \{1, 2, \ldots, n\}$ be a randomly selected index subset of the orders being swapped. The equation for the child solution can be found in Equation 5.

$$C = (P_1 \setminus \{a_i \mid i \in I\}) \cup \{b_i \mid i \in I\} \tag{5}$$

This operation promotes genetic diversity without violating the constraints of the environment, keeping all solutions feasible.

4.5.2 Parameters

The parameters were adopted from a closely related study [AYGP22], and can be found in Table 6.

Table 6: Genetic Algorithm Parameters

Parameter	Value	Role
Population Size	120	Number of individuals in each generation
Generations	200	Total number of evolutionary cycles
Crossover Probability	0.3	Likelihood crossing over of two individuals
Swap Probability	0.1	Probability of using the swap mutation
Reassignment Probability	0.1	Probability of using the reassign order mutation
Shuffle Probability	0.1	Probability of using the shuffle route mutation

4.6 Ant Colony Optimization

The Ant Colony Optimization (ACO) is a metaheuristic algorithm inspired by the foraging behavior of real ants. It is aimed at solving optimization problems by simulating artificial ants that iteratively construct solutions based on pheromone trails and heuristic information. The algorithm is able to explore promising regions of the solution space, since (artificial) ants prefer paths with higher pheromone intensity.
In the context of the DVRP, each ant represents a potential reassignment of alterable orders to vehicles. The ants can be guided by two types of pheromones:

- **Route Pheromone** (τ_{ij}): represents the desirability of moving from location i to j.
- **Pairing Pheromone** (Φvo): represents the desirability of assigning order o to vehicle v.

However the algorithm only utilizes the pairing pheromone for now, due to two reasons:

1. Adding another pheromone loop would have made the algorithm even more computationally expensive
2. Time constraints

The pseudocode for the algorithm is shown in Listing 6.

Listing 6: Pseudocode for Ant Colony Optimization

```
INITIALIZE pheromone levels for route and vehicle-order pairings
FOR generation = 1 to max generations:
    FOR ant = 1 to num_ants:
        GENERATE solution by assigning modifiable orders to vehicles
        EVALUATE solution fitness (simulate_evaluate)
        IF solution is best so far:
            UPDATE best_solution
    EVAPORATE all pheromones by factor rho
    UPDATE pheromones using best solution of this generation
RETURN best solution found
```

4.6.1 Solution Construction

The core of the Ant Colony Optimization algorithm is the generate_ant_solution function, which assigns modifiable orders to vehicles in a greedy, but probabilistic way. For every modifiable order:

1. All valid order-vehicle combinations are evaluated for their cost.
2. A probability, shown in Figure 10, is computed for each option, proportional to a combination of:
   - The pheromone level (raised to the power of α)
   - The heuristic value (inverse of estimated cost, raised to the power of β)
3. One option is sampled using roulette-wheel selection.

$$\text{probability} = (\text{pheromone})^{\alpha} \cdot (\text{heuristic})^{\beta} \tag{6}$$

where:
- pheromone : amount of pheromone
- heuristic : inverse of estimated cost
- α : sensitivity to pheromone value ($\alpha \geq 0$)
- β : sensitivity to heuristic value ($\beta \geq 0$)

Figure 10: Probability equation in the ACO algorithm.

As this solution construction algorithm loops over every modifiable order, every order will be assigned, thus yielding a valid solution. However it must be mentioned that looping through every order-vehicle combination is very expensive computationally.

4.6.2 Pheromone Update

At the end of every generation, the pheromone matrices are updated, first by the evaporation of pheromones set by the variable ρ, this can be seen in Equations 7 and 8, and then by the best solution of the generation, which is used to deposit new pheromones, rewarding higher quality solutions. The pheromone update equations can be found in Equations 9 and 10.

$$\tau_{i,j} \leftarrow (1 - \rho) \cdot \tau_{i,j} \tag{7}$$

$$\Phi_{v,o} \leftarrow (1 - \rho) \cdot \Phi_{v,o} \tag{8}$$

$$\tau_{i,j} \leftarrow \tau_{i,j} + \frac{q}{f_{\text{best}}} \tag{9}$$

$$\Phi_{v,o} \leftarrow \Phi_{v,o} + \frac{q}{f_{\text{best}}} \tag{10}$$

4.6.3 Parameters

Due to the lack of directly comparable research to this problem, parameter values were collected from recent, relevant literature, and further refined through fine-tuning [WGHH24].

The explored variable options and the selected final settings are listed in the Table 7. The "Medium - Center" base scenario was selected for fine-tuning (see Table 2), due to its moderate computational requirements and balanced structure, making it perfect for evaluating general baseline performance. The experiment consisted of 10 iterations over every combination of the variable options found in Table 7. The code for the fine tuning can be found in fine_tuning.py.

Table 7: ACO Algorithm Parameter Options

Parameter	**Options**	**Selected Value**
Number of Ants	10, 50, 100	**100**
Generations	50, 125, 250	**125**
Alpha_ACO	1, 2, 4	**2**
Beta_ACO	3, 4, 5	**5**
Evaporation Rate	0.1, 0.45, 0.9	**0.45**
Pheromone Deposit	1, 50, 100	**1**

4.7 Artificial Bee Colony

The Artificial Bee Colony (ABC) algorithm is a metaheuristic inspired by the foraging behavior of honey bees. The artificial bee colony consists of three groups of bees:

- Employed Bees
- Onlooker Bees
- Scout Bees

Each type of bee contributes to the exploration and exploitation of the solution space, in different ways.
In the implementation of this algorithm, each feasible solution is referred to as a "food source". The algorithm takes the following steps in order to thoroughly explore and exploit the solution space:

1. **Initialization**: A pool of randomized feasible solutions is generated to form the initial food sources. Each food source is evaluated using the simulate_evaluate function, see Appendix B.

2. **Employed Bee Phase**: Each food source is mutated by one of the three previously discussed mutation operators, see Section 4.3. If the new solutions yield better fitness values, they replace the original food source.

3. **Onlooker Bee Phase**: Solutions are probabilistically selected based on fitness values, then they are mutated similarly as in the previous step. Better solutions similarly replace their predecessors.

4. **Scout Bee Phase**: If a food source fails to improve after a defined number of attempts (limit), it is abandoned and replaced by a new randomly generated feasible solution, encouraging exploration and preventing stagnation.

5. **Selection**: The previous three steps is repeated max_iterations number of times, the best found solution across all iterations is kept track of and is returned at the end.

4.7.1 Parameters

Due to the lack of directly comparable research to this problem, parameter values were collected from recent, relevant literature, and further refined through fine-tuning [NWSD22].
The explored variable options and the selected final settings are listed in the Table 8. The fine-tuning process followed the same approach as described for the Ant Colony Optimization algorithm, this can be found in Section 4.6.3.

Table 8: Artificial Bee Colony Algorithm Parameter Options

Parameter	Options	Selected Value
Colony Size	10, 50, 100	**50**
Max Iterations	10, 100, 200	**100**
Limit	20, 100, 500, 1000	**100**

5 Experiments

5.1 Variable Logging

As explained in Section 1.4, the use of IOHAnalyzer was considered to monitor algorithm performance, due to its complete framework for analyzing iterative optimization heuristics and extensive visualization capabilities. However, due to the intricate and highly customized nature of the dvrpsim environment, integration with IOHAnalyzer is not trivially feasible.
A custom logging model was developed, in order to solve this issue. The DataLogger class was implemented to capture both during-run and per-generation data. It tracks fitness progression, as well as operational metrics specific to the problem this study considers.
The following data is recorded by the logger:

- **Per generation**: Best, average, and worst fitness, standard deviation (if applicable), simulation time, number of open and delivered orders, total wait time, and total planned distance.
- **Per run summary**: Final best fitness, cumulative wait time and distance, total number of orders delivered, and number of function evaluations.
- **Order-level data**: For each order delivered, the generation, release time, delivery time, and wait time are tracked.
- **Vehicle-level data**: For each vehicle, the total distance traveled per generation is logged.

Table 9 provides a more detailed overview of the logged variables including their units and logging frequency.
Each run of the simulation writes into uniquely named files, which ensures clean separation and reproducibility, even if runs are parallelized.
With the use of this customized logging infrastructure, detailed insights can be extracted from the algorithm runs, which can be used for algorithm tuning and evaluation.

Table 9: Summary of logged variables, their units, and logging frequency.

Variable	**Unit**	**Frequency**
Main log (per generation)		
generation	count	per generation
best_fitness	cost	per generation
avg_fitness	cost	per generation
worst_fitness	cost	per generation
std_dev	cost	per generation
time	minutes	per generation
orders_open	count	per generation
orders_delivered	count	per generation
total_wait_time	minutes	per generation
total_planned_distance	kilometers	per generation
Summary log (per run)		
run_id	integer	per run
final_best_fitness	cost	per run
final_wait_time	minutes	per run
final_distance	kilometers	per run
total_orders_delivered	count	per run
total_function_calls	count	per run
Order-level log (per run)		
generation	count	per run
order_id	integer	per run
release_date	timestamp	per run
delivery_time	timestamp	per run
wait_time	minutes	per run
Vehicle-level log (per run)		
generation	count	per run
vehicle_id	integer	per run
total_distance	kilometers	per run

5.2 Statistical Testing

In order to compare the performance of the proposed algorithms, statistical tests were conducted using the final best fitness values obtained across all scenarios. A series of pairwise comparisons were performed between every algorithm, using paired t-tests. This type of testing accounts for the paired nature of the data, as each algorithm was evaluated on identical scenario and run combination, with the same random seeds.
These tests were performed using the ttest_rel function from the scipy.stats package in Python. A significance threshold of $\alpha = 0.05$ was used to determine whether observed differences were meaningful.
The aim of these tests was to provide useful insight into the relative performance of the algorithms, taking variability across scenarios into account.

5.3 Experimental Setup

The experiment was conducted in two phases. The first phase evaluated all five algorithms on the baseline scenarios (see Section 3.3.1). Each algorithm was executed 10 times with different random seeds, to ensure comparability the same set of random seeds was used across all algorithms, resulting in paired data for statistical analysis. The number of runs (10) was chosen due to time and computational resource constraints. The script used for conducting this phase can be found in selection.py.
In the second phase of the experiment the focus shifted onto the experimental scenarios (see Section 3.3.2). Based on performance in phase one, the top two algorithms were chosen (Genetic Algorithm and Artificial Bee Colony) for further analysis. Each scenario - algorithm combination was again run 10 times with a consistent set of random seeds per run. The script used for conducting this phase can be found in experiments.py.
Both scripts for the two phases and the logging methodology were set up with parallelization in mind to improve efficiency (see Section 5.1 for more information about logging).
The results from both phases are presented and analyzed in Section 6.

5.3.1 Hardware Used

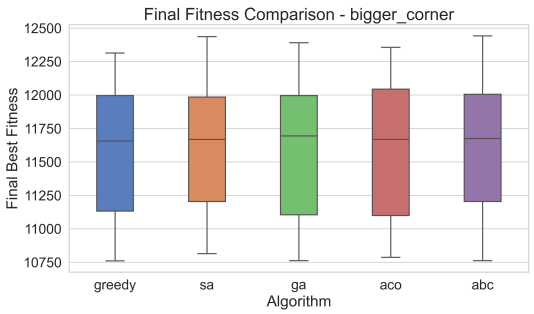
The experiments were conducted on computers provided by Leiden University. The hardware specifications are:

- **Processor:** Intel Core i7-14700 (20 cores: 8 performance cores with HyperThreading, 12 efficiency cores)
- **Memory:** 32 GB RAM
- **Graphics Card:** NVIDIA GeForce GTX 4060, 8 GB VRAM
- **Storage:** 1 TB NVMe SSD

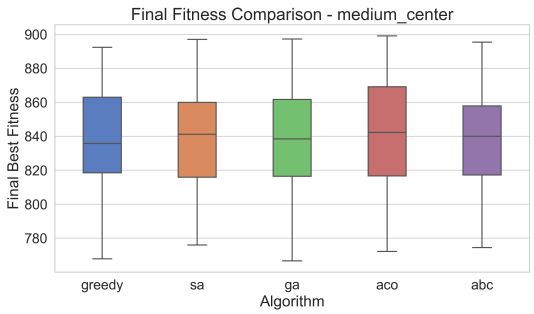
6 Results and Analysis

6.1 Phase One

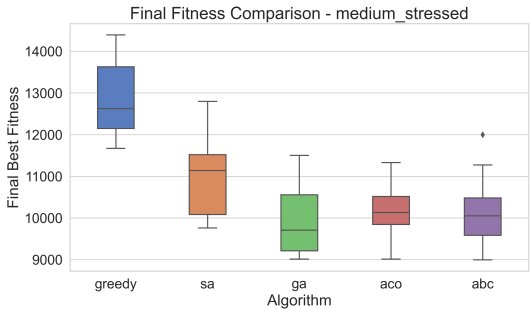
In phase one, all five algorithms were compared on the baseline scenarios (see Section 3.3.1).



(a) Bigger-Corner Scenario



(b) Medium-Center Scenario



(c) Medium-Stressed Scenario

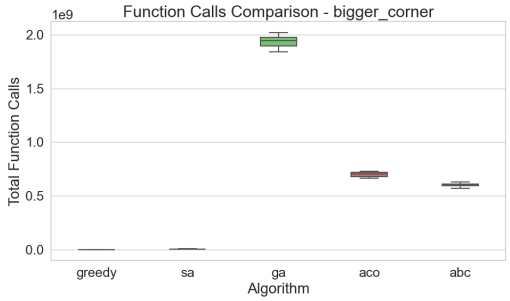
Figure 11: Comparison of final fitness values across baseline scenarios. The final fitness value was approximately the same in Scenarios Bigger-Corner and Medium-Center, however in Scenario Medium-Stressed the greedy algorithm performed clearly the worst, while the GA was best.

6.1.1 Final Fitness

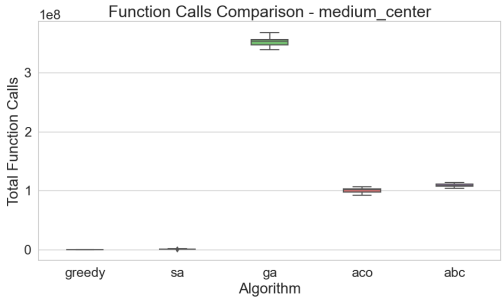
There were minimal differences between the final fitness of the algorithms in the Bigger-Corner and Medium-Center scenarios, with a maximum difference of only 0.5%, as seen in Figures 11a and 11b. This is primarily because, in neither scenario, the hub received an overwhelming number of orders simultaneously. However, the Medium-Stressed scenario, see Figure 11c, was specifically designed to create a high pressure situation. The differences between algorithms in this scenario are more pronounced: the greedy algorithm performing the worst, while the GA performing the best on average across runs. Notably, the GA has a significantly larger variance than the ACO and ABC algorithms.

6.1.2 Efficiency

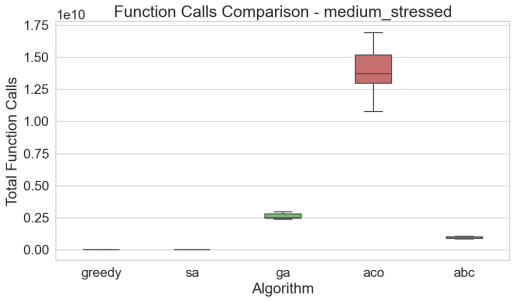
From Figure 12, it can be observed that the GA is the least efficient on average, as it requires a high amount of function evaluations across all scenarios, averaging 1.62×10^9 function calls. It can also be seen that the greedy algorithm requires the fewest function evaluations, averaging 3.2×10^4 evaluations, followed closely by SA, which averaged 5.72×10^6 function calls. The ABC algorithm falls in the middle, in terms of function evaluation count, as it averages 4×10^8 function evaluations between the Bigger-Corner and Medium-Center scenarios. However, the ACO algorithm's number of function evaluation increases significantly when the number of simultaneously open orders increases, like in the Medium-Stressed scenario, where it averages 1.4×10^{10} function evaluation calls. This is



(a) Bigger-Corner Scenario



(b) Medium-Center Scenario



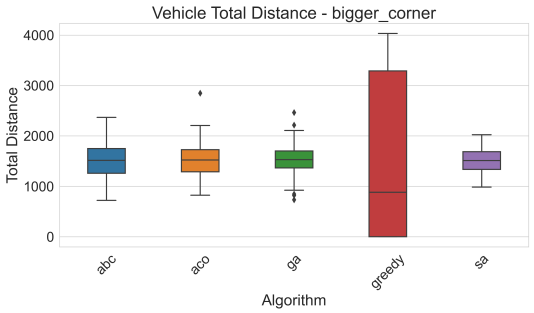
(c) Medium-Stressed Scenario

Figure 12: Comparison of function calls across baseline scenarios. It can be seen that the GA required the most function evaluations on average across scenarios, however in Scenario C the ACO algorithm required 3-4 times as much as the GA.

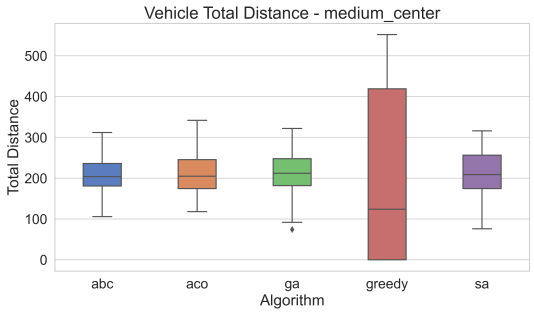
explained by the solution construction process of the ACO algorithm (see in Section 4.6.1).

6.1.3 Wait Time and Total Distance Metrics

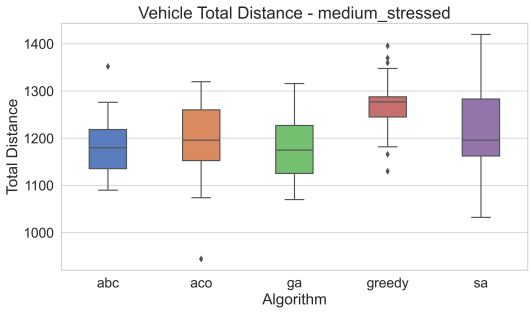
Customer wait times across the algorithms in Bigger-Corner and Medium-Center scenarios showed minimal differences, however, vehicle distance statistics show some variation, see Figure 13 and 14. The greedy algorithm is highly inconsistent across runs, resulting in a large spread of total travel distance, seen in Figure 13. While on average it achieves lower total travel distances than the other algorithms, this is because the greedy algorithm, sends as many orders at once with one vehicle as possible, optimizing for total travel distance, rather than customer wait times.
Figure 14 highlights that while the more complex algorithms tend to utilize all vehicles evenly, the greedy algorithm primarily uses the first few vehicles. This is simply due to the fact that the greedy algorithm sends as many orders as possible with the first vehicle, thus failing to consider distributing the orders and utilizing all available vehicles at once.
In the Medium-Stressed scenario the GA and ABC algorithms maintain lower total distances than the other algorithms, with average reductions of 3.9% and 4.1% respectively, see Figure 13. Interestingly, in the high pressure situation all five algorithms utilize all vehicles evenly, including the greedy algorithm. This is likely due to necessity.



(a) Bigger-Corner Scenario

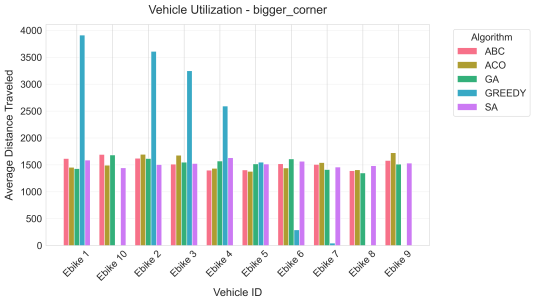


(b) Medium-Center Scenario

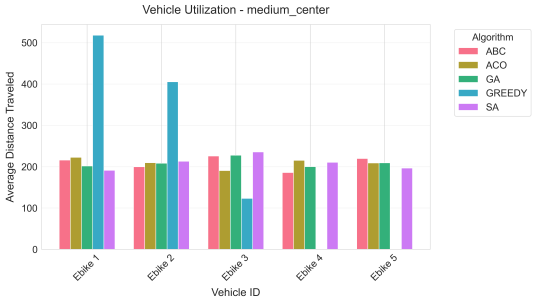


(c) Medium-Stressed Scenario

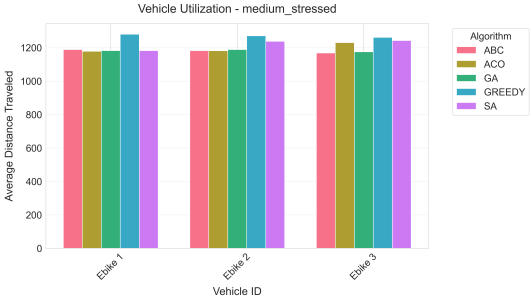
Figure 13: Average total distance traveled across baseline scenarios. The greedy algorithm varies the most in scenarios A and B, however it does achieve the shortest travel distances. In scenario C the GA and the ABC algorithms achieve the shortest travel distances.



(a) Bigger-Corner Scenario



(b) Medium-Center Scenario



(c) Medium-Stressed Scenario

Figure 14: Vehicle utilization across baseline scenarios. The greedy algorithm does not utilize all available vehicles in scenarios A and B, however it does so in scenario C.

6.1.4 Phase One Conclusion

Based on these observations, the GA and ABC algorithms were selected as the best two. This decision is not solely due to their superior performance across the baseline scenarios, but also because they maintained reasonable efficiency, as opposed to the promising, but costly ACO algorithm.
In order to confirm this selection, statistical significance testing was conducted, see Section 5.2, to compare the GA and ACO algorithms with the other algorithms in these scenarios. Table 10 shows that the only two non significant differences were found in comparisons: the GA vs ACO and ABC vs ACO, which suggests that ACO performs on a similar level as the GA and ABC, however due to its inefficiency in high demand scenarios with many simultaneous orders, ACO was ruled out of the top two algorithms.

Comparison	Mean Difference	t-statistic	p-value	Significant
GA vs ABC	-94.15	-2.193825	0.0364	True
GA vs GREEDY	-963.08	-3.611470	0.0011	True
GA vs SA	-378.79	-3.558657	0.0013	True
GA vs ACO	-93.49	-1.891111	0.0686	False
ABC vs GREEDY	-868.93	-3.597153	0.0012	True
ABC vs SA	-284.64	-3.469142	0.0017	True
ABC vs ACO	0.66	0.017498	0.9862	False
ACO vs GREEDY	-869.59	-3.619015	0.0011	True
ACO vs SA	-285.3	-3.030819	0.0051	True
GREEDY vs SA	584.29	3.279949	0.0027	True

Table 10: Comparison algorithm performance across baseline scenarios

6.2 Phase Two

In phase two the Genetic Algorithm (GA) and the Artificial Bee Colony algorithm (ABC) were compared using the experimental scenarios (see Section 3.3.2). As described in Section 3.3.2, 6 different scenarios were designed, all of which with several configuration options. For ease of reference, these options will be referred to by their letter identifiers, which can be found in Table 4.

6.2.1 Final Fitness

Over the 30 scenario combinations, a statistically significant performance difference was only found in one scenario: the Layout - Option B (see Table 4), in which the ABC algorithm outperformed the GA, shown in Figure 15 and Table 11. In this scenario configuration the depot is located in the corner of the service area, consequently making the clustering of routes more important. Suboptimal clustering can lead to disproportionally long routes and increased waiting times. The ABC algorithm appears to be able to adapt to this environment more effectively than the GA, as it achieved a 2.8% lower final fitness value on average in this scenario. This is potentially due to its solution construction and local search phases, which enable it to better explore and exploit the search space around promising solutions, leading to more refined clustering.

scenario	option	GA_mean	ABC_mean	mean_diff	t_stat	p_value	significant
area_size	A	5410.55	5393.95	16.60	0.24	0.81	False
area_size	B	14412.41	14636.34	-223.92	-1.58	0.15	False
capacity	A	2454.28	2501.98	-47.70	-1.38	0.20	False
capacity	B	2135.47	2177.98	-42.51	-1.57	0.15	False
capacity	C	2098.68	2107.67	-8.99	-0.39	0.70	False
distributions	A	2209.25	2204.25	4.99	0.30	0.77	False
distributions	B	2318.58	2306.55	12.03	0.63	0.54	False
distributions	C	2387.28	2366.93	20.36	0.64	0.54	False
fleet_size	A	6074.08	6073.41	0.67	0.05	0.96	False
fleet_size	B	5878.32	5872.88	5.44	1.14	0.29	False
fleet_size	C	5867.15	5867.30	-0.15	-0.01	0.99	False
layout	A	2454.28	2489.36	-35.08	-1.28	0.23	False
layout	B	5776.94	5622.19	154.75	5.66	0.00	True
num_orders	A	8107.63	8082.06	25.57	0.89	0.40	False
num_orders	B	20135.30	20104.16	31.14	1.08	0.31	False

Table 11: Scenario - Algorithm combinations, and their respective results in terms of final fitness. The only statistically significant result is the Layout - Option B, in which the ABC algorithm outperformed the GA.

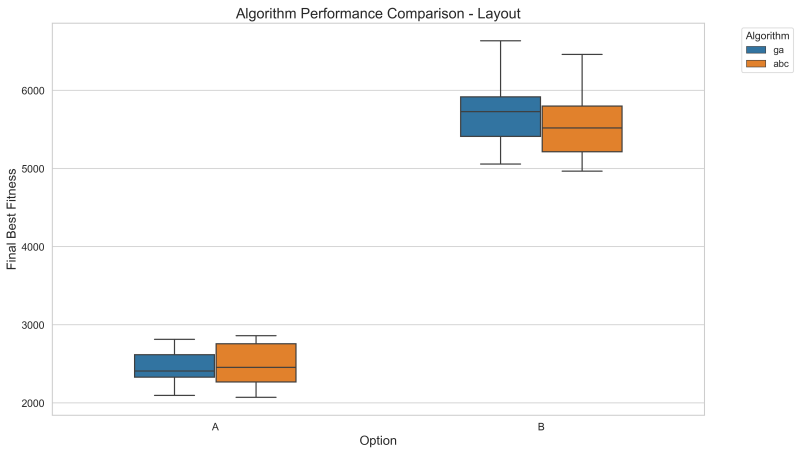


Figure 15: Average Final Fitness for Scenario Layout. The ABC algorithm achieves a significantly better final fitness in Scenario B.

Scenario	Option	GA mean	ABC mean	Mean Difference	t-statistic	p-value	Significant
area_size	A	811.88	817.20	-5.32	-1.07	0.31	False
area_size	B	1534.28	1523.76	10.52	1.94	0.08	False
capacity	A	443.72	444.20	-0.48	-0.15	0.88	False
capacity	B	411.56	411.12	0.44	0.20	0.85	False
capacity	C	407.44	406.00	1.44	0.40	0.70	False
distributions	A	464.80	462.64	2.16	0.69	0.51	False
distributions	B	453.72	457.16	-3.44	-1.30	0.23	False
distributions	C	446.04	443.44	2.60	1.17	0.27	False
fleet_size	A	1506.12	1508.72	-2.60	-1.07	0.31	False
fleet_size	B	380.53	380.53	0.00	NaN	NaN	False
fleet_size	C	152.16	152.21	-0.05	-1.00	0.34	False
layout	A	443.72	443.76	-0.04	-0.01	0.99	False
layout	B	724.08	719.36	4.72	1.38	0.20	False
num_orders	A	2011.52	2006.72	4.80	1.66	0.13	False
num_orders	B	5009.64	5011.08	-1.44	-0.39	0.70	False

Table 12: Scenario - Algorithm combinations, and their respective results in terms of total system-wide vehicle distance. There were no significant differences observed between the two algorithms.

6.2.2 Efficiency

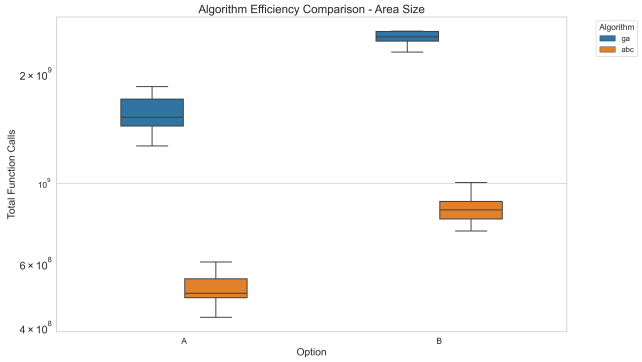
Throughout every experimental scenario, the GA consistently required a significantly higher number of function evaluations compared to the ABC algorithm, as seen in Figure 16. This is expected, as both algorithms use the same initial solution generator, and mutation operators, thus the dominant factors in terms of time complexity are:

- Number of Iterations or Generations
- Colony or Population size
- Problem size (i.e., number of modifiable orders)

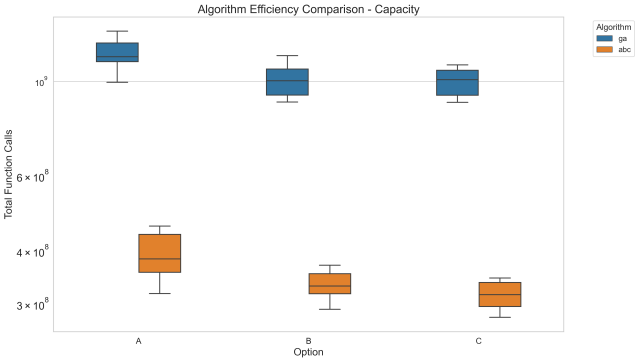
The main difference being that the GA sorts the entire population each generation in order to select the next generation, which is quite expensive computationally. On average, the GA required about 3 to 3.3 times more function evaluations than the ABC algorithm across scenarios, this meant approximately 58 seconds of difference per decision on average on the hardware used (see Section 5.3.1).

6.2.3 Wait Time and Total Distance Metrics

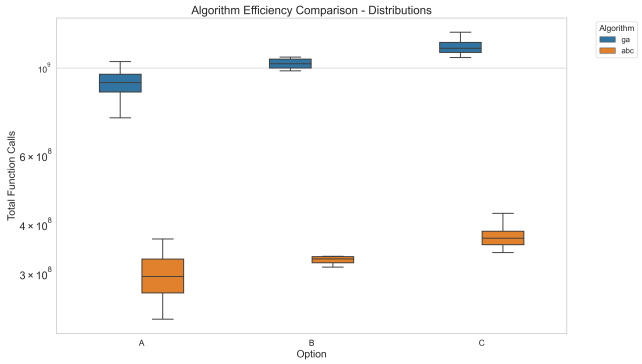
The two algorithms perform similarly with respect to total travel distances and vehicle utilization, as shown in Table 12. The only statistically significant difference was found in the total customer wait time variable in scenario Layout - B, see Table 13, similarly as with the final fitness value, see Section 6.2.1, the ABC algorithm achieves a lower mean customer wait time than the GA. These results suggest that the ABC algorithm, due to its superior exploration and exploitation capabilities, is better at finding more efficient customer visit sequences, achieving lower wait times without increasing the total travel distance, in spatially challenging scenarios.



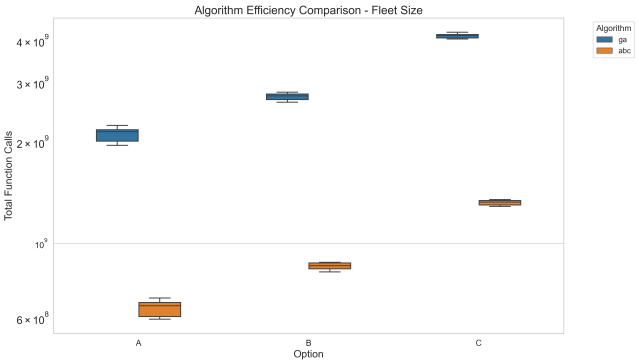
(a) Scenario Area Size



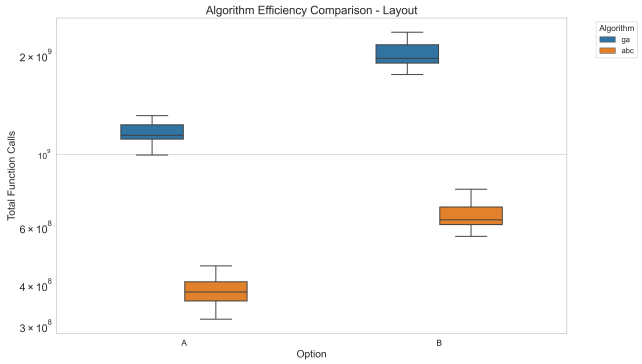
(b) Scenario Capacity



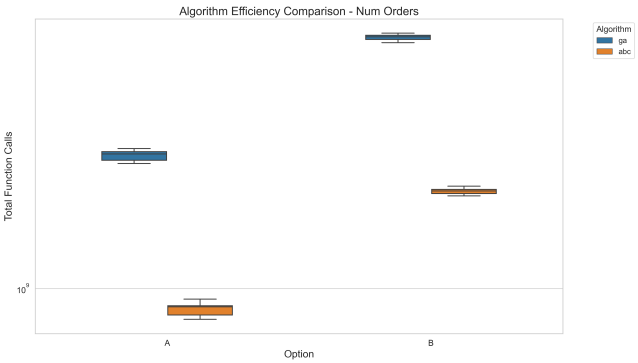
(c) Scenario Distributions



(d) Scenario Fleet Size



(e) Scenario Layout



(f) Scenario Number of Orders

Figure 16: Average Number of Function Evaluations Per Scenario. The GA consistently requires significantly more function evaluations than the ABC algorithm.

Scenario	Option	GA mean	ABC mean	Mean Difference	t-statistic	p-value	Significant
area_size	A	135.23	134.04	1.20	0.49	0.64	False
area_size	B	423.07	433.08	-10.01	-1.81	0.10	False
capacity	A	53.80	55.66	-1.86	-1.50	0.17	False
capacity	B	44.26	46.01	-1.74	-1.77	0.11	False
capacity	C	43.20	43.71	-0.50	-0.57	0.59	False
distributions	A	41.89	41.91	-0.02	-0.02	0.98	False
distributions	B	47.37	46.55	0.83	1.22	0.25	False
distributions	C	50.89	50.33	0.55	0.49	0.64	False
fleet_size	A	30.78	30.69	0.10	0.45	0.66	False
fleet_size	B	27.64	27.57	0.07	1.14	0.29	False
fleet_size	C	27.51	27.49	0.02	0.11	0.91	False
layout	A	53.80	55.20	-1.40	-1.59	0.15	False
layout	B	158.67	152.95	5.72	6.34	0.00	True
num_orders	A	30.79	30.65	0.14	0.54	0.61	False
num_orders	B	30.44	30.31	0.14	1.08	0.31	False

Table 13: Scenario - Algorithm combinations, and their respective results in terms of total customer wait time. The only statistically significant result is the Layout - Option B, in which the ABC algorithm outperformed the GA.

6.2.4 Phase Two Conclusion

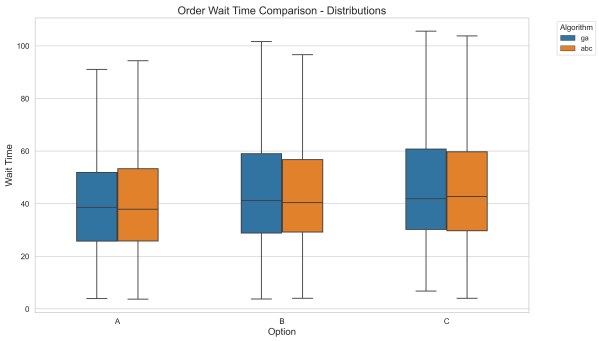
In summary, the ABC algorithm showed a statistically significant advantage ($p < 0.001$), both in terms of average final fitness and total customer waiting time, in a single scenario, where the location of the depot increased the weight of optimal spatial clustering of orders. In all of the other scenarios both algorithms achieved similar results in terms of final fitness, total travel distance, and total customer wait times. However the GA consistently required a significantly higher number of function evaluations than the ABC algorithm. Therefore, if high quality solutions in spatially challenging environment layouts are the primary concern, the ABC algorithm is the preferable option. Furthermore, if computational efficiency is pivotal, the ABC algorithm can provide comparable solution quality to the GA while being significantly more computationally efficient.

6.3 Practical Insights

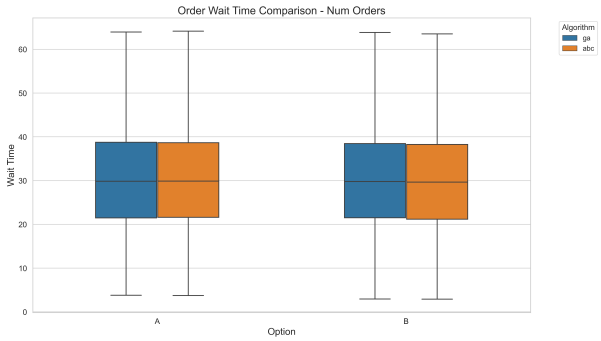
Beyond algorithmic performance, these simulations provide valuable insights for practitioners into dynamic delivery systems.

6.3.1 Customer-Centric Insights (Reducing Wait Times)

- **Order volume**: As expected, the higher number of order volumes, the larger the overall distance covered by all delivery vehicles. However the customer wait times depend more on the distribution of the demand, as the system can be overwhelmed by too many orders, this difference can be seen in Figure 17.



(a) Demand Distributions Impact



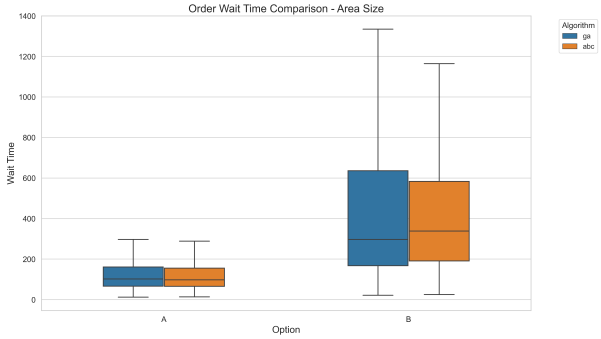
(b) Order Volume Impact

Figure 17: The customer wait times do not differ between the two scenarios changing the number of orders, however there are differences between the scenarios varying the distribution of the orders.

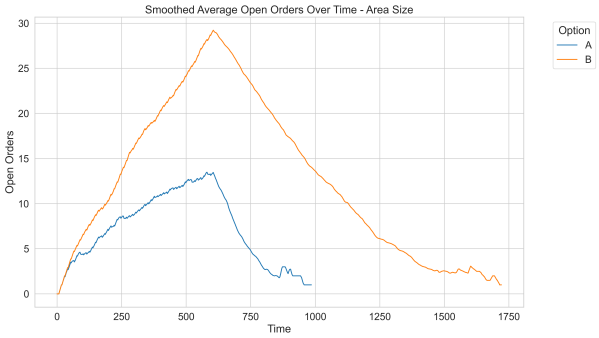
- **Service area size**: Larger service areas increased the total distance length, average customer wait time and introduced higher variance in the latter, as shown in Figure 18. This can be explained by the number of open orders over time, since with more distance to cover, orders on average took more time to deliver, which increases variance and overall wait time.
- **Fleet size**: Smaller fleet sizes resulted in longer customer wait times and increased total distances per vehicle, as seen in Figure 19a. On the other hand, large fleet sizes did not significantly reduce customer wait times nor total distance, therefore finding the optimal fleet size is crucial, this is highlighted by Figure 19b.
- **Vehicle capacity**: On average lower vehicle capacities required more trips, therefore increasing total travel distance and customer wait times, as depicted in Figure 21a. However as the capacity was increased better clustering solutions were possible, thus these variables were decreased. It is important to note that there was no significant improvement in total travel distance or customer wait time between the capacity setting 8 and 10, see Figure 21b, which suggests that such small increase in capacity does not allow for significantly better clustering, therefore increasing the capacity in larger increments is recommended.

6.3.2 Operator-Centric Insights (Reducing Distance and Improving Efficiency)

- **Demand distribution**: Fluctuating distribution did not pose a challenge for these algorithms, it was rather the sudden increase in order volume. If these surges in order volume could be forecasted, it would be interesting to develop forecasting tools. However this is left for future research.
- **Depot position**: A centrally located depot decreased total distances and customer wait times, whereas a corner placement made it harder to form efficient route clusters, thus increased both variables and their variance, as shown in Figures 20a and 20b. This further highlights the need for strategic depot placement in dynamic delivery systems.

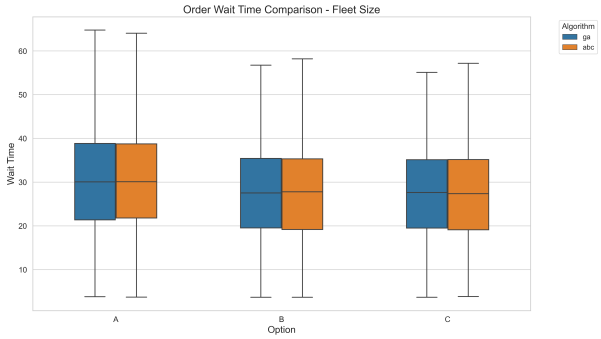


(a) Area Size Impact

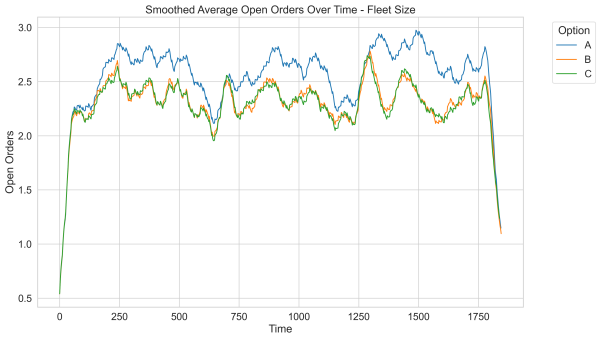


(b) Open Orders Over Time

Figure 18: The customer wait times increased as the area size was increased. The increase in area size introduced high variance in customer wait times. This can be explained by the open orders over time graph.

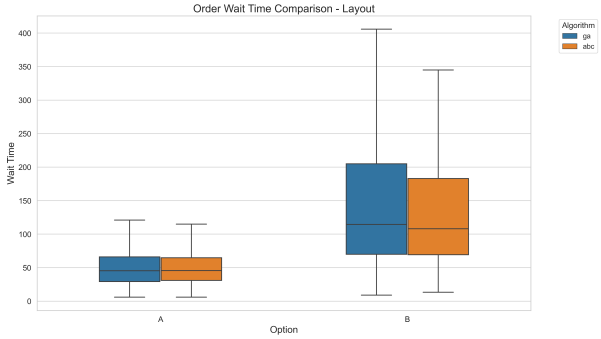


(a) Fleet Size Impact

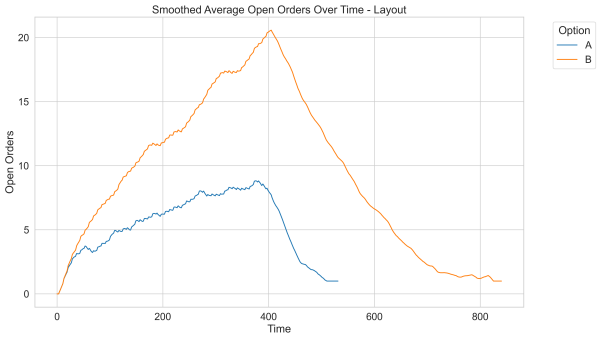


(b) Open Orders Over Time

Figure 19: An increase in fleet size reduces customer wait times, however beyond a certain point additional increase has minimal effect. This is further underlined by the average open orders over time graph, where fleet sizes of 20 and 50 perform almost identically in the given scenario, both outperforming the scenario where the fleet size was set to 5.

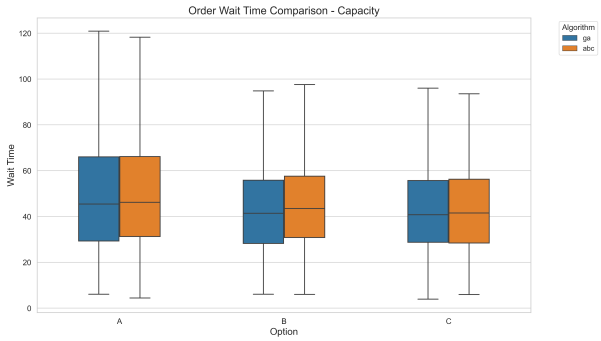


(a) Layout Impact

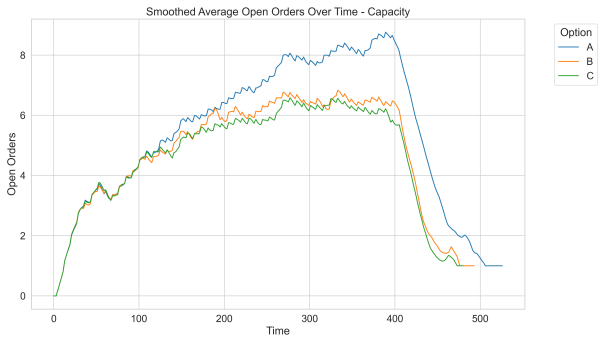


(b) Open Orders Over Time

Figure 20: Customer wait times and their variances increased significantly when the depot was placed in the corner. As seen in the open orders over time graph, this is due to the fact that a centrally placed depot enables faster deliveries due to its strategic placement.



(a) Vehicle Capacity Impact



(b) Open Orders Over Time

Figure 21: Customer wait times dropped significantly when the vehicle capacity was increased from 5 to 8, but the further increase to 10 had little impact. This trend is also shown in the open orders over time graph.

7 Conclusions and Further Research

The aim of this paper was to implement and evaluate natural computing approaches for a real-time order allocation and routing problem in dynamic, same-day grocery delivery systems. The primary objective was to analyze the trade-offs between competing objectives, customer waiting times and total vehicle travel distances, while also keeping algorithmic efficiency and robustness in mind across various realistic operational scenarios.

7.1 RQ1: Trade-off Between Customer Wait Time and Total Distance

The results show that it is possible to optimize both customer wait times and total vehicle travel distances effectively, by using natural computing approaches. Across the five algorithms considered, the Genetic Algorithm (GA) and Artificial Bee Colony (ABC) algorithms emerged as top performers in terms of these metrics. On average, the GA achieved a 0.71% reduction in total distance and a 10.12% decrease in customer wait time, while the ABC algorithm achieved a 0.75% reduction in total distance and 7.85% decrease in customer wait time, compared to the other three algorithms (Simulated Annealing, Ant Colony Optimization and Greedy).
Future work could explore varying the weights of these metrics in the cost function (described in Section 4.1), in order to get a better understanding of the control trade-offs between these competing objectives.

7.2 RQ2: Algorithmic Efficiency in Terms of Function Evaluations

Regarding efficiency, while the simpler algorithms, like Greedy and Simulated Annealing, required substantially fewer function evaluations as the more complex algorithms, they did not achieve comparable solution quality. Between the two top performing algorithms, ABC clearly outperformed the GA in terms of function evaluations, as the GA required about 3.16 times more function evaluations than the ABC algorithm. This can be explained by the lower computational overhead of the ABC algorithm, as it avoids sorting the population each generation, unlike the GA. In

contexts where both high-quality solutions and efficiency are critical, the ABC algorithm is the more practical choice.

7.3 RQ3: Robustness Across Dynamic Conditions

When comparing the two top performing algorithms over various realistic scenarios, they both exhibited stable, comparable performance, except for a single scenario (Layout - B). In this scenario the depot is located in the corner of the service area, thus increasing the impact of suboptimal order clustering. While placing the depot at the center of the service area is ideal for minimizing distances and reducing waiting times, practical considerations often make it difficult or even impossible (for example land availability, regulations or transportation infrastructure) [WLB02] [SG23] [QZ24]. In this case the ABC algorithm significantly outperformed the GA, implicating that it is more equipped to handle spatially challenging environments.

7.4 Overall Findings

In summary, the GA and ABC algorithms emerged as the most effective approaches across the first phase of the evaluation. While both delivered high quality solutions, the ABC algorithm demonstrated a greater ability to adapt to spatially challenging environments, as it achieved a 2.8% lower final fitness value than the GA in the spatially challenging scenario (Layout - B, see Section 3.3.2), and it consistently outperformed the GA in terms of computational efficiency, as the GA required about 3 - 3.3 times as many function evaluations as the ABC algorithm, making the Artificial Bee Colony algorithm the more practical choice for dynamic delivery environments.
Additionally, some practical insights, into the intricacies of the examined parameters were given in Section 6.3. These insights can help practitioners decide which parameters to focus on, based on their own business needs.

7.5 Future Work

Future work may explore hybridization of these strategies, to combine the respective strengths of these approaches. Travel time estimation could be improved by using real traffic data, instead of estimates. Further research could investigate, the development of order surge prediction algorithms, the potential interactions between variables using a multi-variable perturbation approach, the effect of irregular service area shapes or more dynamic constraints, for example heterogeneous vehicle capacities, handling multiple order requests simultaneously, and changes in fleet size throughout the day to account for driver availability. These additions could improve the practical applicability and realisticness of the proposed algorithms in real-world delivery systems.

References

[AU23] Charlotte Ackva and Marlin Ulmer. Consistent routing for local same-day delivery via micro-hubs. *OR Spectrum*, 46:1–35, 12 2023.

[AYGP22] Anil Kumar Agrawal, Susheel Yadav, Amit Ambar Gupta, and Suchit Pandey. A genetic algorithm model for optimizing vehicle routing problems with perishable products under time-window and quality requirements. *Decision Analytics Journal*, 5:100139, 2022.

[BFFP23] Maria Elena Bruni, Edoardo Fadda, Stanislav Fedorov, and Guido Perboli. A machine learning optimization approach for last-mile delivery and third-party logistics. *Computers Operations Research*, 157:106262, 2023.

[CPPJ+24] Abhijit Chandratreya Phd, Sanjaykumar Patil, Nitin Joshi, Sandeep Londhe, and Hemant Anbhule. Last-mile delivery optimization for sustainable environment. *Community practitioner: the journal of the Community Practitioners' Health Visitors' Association*, 21:1184–1205, 07 2024.

[CQGI21] Jean-François Côté, Thiago Queiroz, Francesco Gallesi, and Manuel Iori. Dynamic optimization algorithms for same-day delivery problems, 04 2021.

[CSZ22] Shuxian Cui, Qian Sun, and Qian Zhang. A time-dependent vehicle routing problem for instant delivery based on memetic algorithm. *Computational Intelligence and Neuroscience*, 2022:1–11, 08 2022.

[CUT22] Xinwei Chen, Marlin W. Ulmer, and Barrett W. Thomas. Deep q-learning for same-day delivery with vehicles and drones. *European Journal of Operational Research*, 298(3):939–952, 2022.

[CZBC21] Hongrui Chu, Wensi Zhang, Pengfei Bai, and Yahong Chen. Data-driven optimization for last-mile delivery. *Complex Intelligent Systems*, 9, 02 2021.

[Del24] Deliverect. The state of the food delivery industry in the us 2024, 2024. Retrieved 09.03.2025.

[FMH23] Hossein Fotouhi and Elise Miller-Hooks. Same-day delivery time-guarantee problem in online retail. *Communications in Transportation Research*, 3:100105, 2023.

[HHBC22] Shan-Huen Huang, Ying-Hua Huang, Carola A. Blazquez, and Chia-Yi Chen. Solving the vehicle routing problem with drone for delivery services using an ant colony optimization algorithm. *Advanced Engineering Informatics*, 51:101536, 2022.

[HT24] Markó Horváth and Tímea Tamási. A general modeling and simulation framework for dynamic vehicle routing, 2024.

[KBP+23] Gautam Siddharth Kashyap, Alexander E. I. Brownlee, Orchid Chetia Phukan, Karan Malik, and Samar Wazir. Roulette-wheel selection-based pso algorithm for solving the vehicle routing problem with time windows, 2023.

[Kom24] Komal Puri. How same-day delivery is changing last mile logistics, 2024. Retrieved 09.03.2025.

[LLKT21] Wenli Li, Kunpeng Li, P N Ram Kumar, and Qiannan Tian. Simultaneous product and service delivery vehicle routing problem with time windows and order release dates. *Applied Mathematical Modelling*, 89:669–687, 2021.

[NWSD22] Zsuzsanna Nagy, Ágnes Werner-Stark, and Tibor Dulai. An artificial bee colony algorithm for static and dynamic capacitated arc routing problems. *Mathematics*, 10(13), 2022.

[OVRA24] Eneko Osaba, Esther Villar-Rodriguez, and Antón Asla. Solving a real-world package delivery routing problem using quantum annealers, 2024.

[QZ24] Yiru Huang Wenxin Li Yongxiang Zhang Xu Yan Qingwei Zhong, Yingxue Yu. Collaborative optimization of depot location, capacity and rolling stock scheduling considering maintenance requirements. *Sci Rep 14*, 2024.

[RFB+24] Mohamed Rhouzali, Hicham Fouraiji, Adil Barra, Wafaa Dachry, and Messaoudi Najat. Vehicle routing optimization for sustainable last-mile delivery. *International Journal of Engineering Trends and Technology*, 72:390–404, 09 2024.

[Sel25] Sellers Commerce. 51 ecommerce statistics in 2025 (global and u.s. data), 2025. Retrieved 09.03.2025.

[SG23] Patrick Stokkink and Nikolas Geroliminis. A continuum approximation approach to the depot location problem in a crowd-shipping system. *Transportation Research Part E: Logistics and Transportation Review*, 176:103207, 2023.

[Shi23] Shippo. 2023 state of shipping report, 2023. Retrieved 09.03.2025.

[SKH21] Daniel Schubert, Heinrich Kuhn, and Andreas Holzapfel. Same-day deliveries in omnichannel retail: Integrated order picking and vehicle routing with vehicle-site dependencies. *Naval Research Logistics (NRL)*, 68(6):721–744, 2021.

[Sta25] Statista. Online food delivery - worldwide, 2025. Retrieved 09.03.2025.

[Tau23] Ismail Tausif. Last mile delivery optimisation model for drone-enabled vehicle routing problem. *Emerging Minds Journal for Student Research*, 1:39–73, Jul. 2023.

[TLW22] Yi Tao, Changhui Lin, and Lijun Wei. Metaheuristics for a large-scale vehicle routing problem of same-day delivery in e-commerce logistics system. *Journal of Advanced Transportation*, 2022(1):8253175, 2022.

[Vas14] Vassilios Vassiliadis. Algorithms and methods inspired from nature for solving supply chain and logistics optimization problems. *International Journal of Natural Computing Research*, 2014.

[Wav24] Wave Grocery. Online grocery shopping statistics (2024): Latest data summary, 2024. Retrieved 09.03.2025.

[WGHH24] Minghui Wu, Bo Gao, Heping Hu, and Konglin Hong. Research on path planning of tea picking robot based on ant colony algorithm. *Measurement and Control*, 57(8):1051–1067, 2024.

[WLB02] Tai-Hsi Wu, Chinyao Low, and Jiunn-Wei Bai. Heuristic solutions to multi-depot location-routing problems. *Computers Operations Research*, 29(10):1393–1415, 2002.

[YAJ+23] Vincent F. Yu, Grace Aloina, Panca Jodiawan, Aldy Gunawan, and Tsung-Chi Huang. The vehicle routing problem with simultaneous pickup and delivery and occasional drivers. *Expert Systems with Applications*, 214:119118, 2023.

[ZIR+21] F. Zahra, P. Islami, N. Rachmawati, A. Redi, R. Nadlifatin, and A. Supranartha. Simulated annealing algorithm for vehicle routing problem with simultaneous pick up and delivery: A case study of liquid petroleum gas distribution. In *2nd Asia Pacific International Conference on Industrial Engineering and Operations Management*, September 2021.

[ZLFV23] Jian Zhang, Kelin Luo, Alexandre M. Florio, and Tom Van Woensel. Solving large-scale dynamic vehicle routing problems with stochastic requests. *European Journal of Operational Research*, 306(2):596–614, 2023.

[ZW24] Shouliang Zhu and Chao Wang. An interaction-enhanced co-evolutionary algorithm for electric vehicle routing optimization. *Applied Soft Computing*, 165:112113, 2024.

A Function Evaluation Counter Class

Listing 7: Custom class for counting the number of function calls during the routing process

```python
class FunctionCounter:
    def __init__(self):
        self.counts = defaultdict(int)
        self.num_evals = 0

    def tracer(self, frame, event, arg):
        if event == "call":
            self.num_evals += 1

    def __enter__(self):
        self.counts.clear()
        sys.setprofile(self.tracer)
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.setprofile(None)
```

B Model Evaluation Mid Simulation

Listing 8: Custom function for evaluating model during simulation

```python
def simulate_evaluate(assignments, state, locations_dict, orders_dict):
    total_wait_time = 0
    total_distance = 0
```

```
    current_time = state.get('time', 0)

    # Simulate the rest of the simulation for every vehicle
    for vehicle_id in assignments['vehicles']:
        local_time = current_time

        vehicle_assignments = assignments['vehicles'][vehicle_id]
        vehicle_state = state['vehicles'][vehicle_id]

        # Check vehicle current location
        if vehicle_state['current_visit'] is None:
            prev_loc_id = vehicle_state['previous_visit']['location']
            prev_loc = locations_dict[prev_loc_id]
            local_time = vehicle_state['previous_visit']['departure_time
                ']
        else:
            current_loc_id = vehicle_state['current_visit']['location']
            current_loc = locations_dict[current_loc_id]
            prev_loc = current_loc

        # Simulate the delivery of all orders already planned for the
            vehicle
        for visit in vehicle_assignments['next_visits']:

            next_loc_id = visit['location']
            next_loc = locations_dict[next_loc_id]

            distance = manhattan_distance(prev_loc.x, prev_loc.y,
                next_loc.x, next_loc.y)
            total_distance += distance
            travel_time = distance
            arrival_time = local_time + travel_time

            # Simulate service
            service_time = 0
            if 'pickup_list' in visit:
                pickup_orders = [orders_dict[o_id] for o_id in visit['
                    pickup_list']]
                service_time = sum(o.pickup_duration for o in
                    pickup_orders)

            local_time = arrival_time + service_time

            if 'delivery_list' in visit:
                for order_id in visit['delivery_list']:
                    order = orders_dict[order_id]
                    wait_time = max(arrival_time - order.release_date,
                        0)
```

```
                    total_wait_time += wait_time

            prev_loc = next_loc
    # Once the total wait time and distance is added up, use the cost
       function to get the simulated fitness score
    return cost_function(0.5, 0.5, total_wait_time, total_distance)
```

C Model Initialization

Listing 9: Initializing the model (basic scenario)

```
def initialize_model(algorithm):
    # Initialize the model
    model = TheModel(algorithm)

    # Set depot location
    depot = Location(id='DEPOT', x=0, y=0)
    model.add_location(depot)

    # Iterate through the orders
    for i in range(20):
        # Randomize customer location
        x = random.randint(-10, 10)
        y = random.randint(-10, 10)
        customer_location = Location(f'CUSTOMER {i + 1} at ({x},{y})', x
           =x, y=y)
        model.add_location(customer_location)

        # Set order variables
        order = Order(id=f'O-{i + 1}')
        order.pickup_location = depot
        order.delivery_location = customer_location
        order.release_date = (i + 1) * 8
        order.pickup_duration = 2
        order.quantity = random.randint(1, 5)
        model.request_order(order, decision_point_on_request=True)

    # Iterate through the vehicles
    for i in range(2):
        # Set vehicle variables
        vehicle = EBike(f'Ebike {i+1}')
        vehicle.initial_location = depot
        vehicle.capacity = 5
        model.add_vehicle(vehicle)

    return model
```

D Distributions Scenario Code

Listing 10: Design of the fluctuating order distributions

```
# Set period lengths (in number of orders)
p_1_l = 20
p_2_l = 10
p_3_l = 20

# Choose scenario
if scenario == 'A':
    p_1 = 8
    p_2 = 12
    p_3 = 8
elif scenario == 'B':
    p_1 = 8
    p_2 = 12
    p_3 = 6
elif scenario == 'C':
    p_1 = 8
    p_2 = 12
    p_3 = 4
else:
    print("Scenario invalid")
    return

phase_1 = []
phase_2 = []
phase_3 = []

# Order request times have to be different, as the model cannot handle
   multiple order requests at once
used_times = set()

# Create the time stamps for the first phase
for i in range(p_1_l):
    while True:
        time = (i + 1) * p_1 + random.randint(0, p_1)
        if time not in used_times:
            phase_1.append(time)
            used_times.add(time)
            break
# Needs to be sorted, cause the model cannot handle non-sequential input
phase_1 = sorted(phase_1)

# Create the time stamps for the second phase
for i in range(p_2_l):
    while True:
```

```
        time = phase_1[-1] + (i + 1) * p_2 + random.randint(0, p_2)
        if time not in used_times:
            phase_2.append(time)
            used_times.add(time)
            break
phase_2 = sorted(phase_2)

# Create the time stamps for the third phase
for i in range(p_3_1):
    while True:
        time = phase_2[-1] + (i + 1) * p_3 + random.randint(0, p_3)
        if time not in used_times:
            phase_3.append(time)
            used_times.add(time)
            break
phase_3 = sorted(phase_3)

# Concatenate the phases
order_request_times = phase_1 + phase_2 + phase_3
```

E Greedy Algorithm Implementation

Listing 11: The complete greedy routing algorithm

```
def greedy_routing_algorithm(state, model, logger=None):

    # Logging various variables
    if logger:
        time = state.get('time', 0)
        num_open_orders = len(state['open_orders'])
        num_delivered_orders = sum([1 for o in model.orders if o.
            is_delivered])
        total_customer_wait_time = sum([(o.delivery_time - o.
            release_date) for o in model.orders if o.is_delivered])
        total_planned_distance = sum([v.total_distance for v in model.
            vehicles])

    # Find unassigned orders
    unassigned_orders = [order_id for order_id in state['open_orders'].
        keys() if
                         state['open_orders'][order_id]['
                            assigned_vehicle'] is None]

    # If there are no unassigned orders accept orders and return
        previous assignments
    if len(unassigned_orders) == 0:
```

```
        assignments = {'vehicles': {}, 'orders': {}}
        for vehicle_id in state['vehicles'].keys():
            assignments['vehicles'][vehicle_id] = {'next_visits': state['vehicles'][vehicle_id]['next_visits']}

        # Log various variables
        if logger:
            fitness = simulate_evaluate(assignments, state, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders})
            logger.log(None, None, fitness, None, None, time, num_open_orders, num_delivered_orders, total_customer_wait_time, total_planned_distance)

        return assignments

    # Find idle vehicles
    idle_vehicles = [vehicle_id for vehicle_id, vehicle_state in state['vehicles'].items()
                     if vehicle_state['status'] == 'IDLE' and vehicle_state['current_visit']['location'] == 'DEPOT'
                     ]

    # If no idle vehicles, accept orders and return previous assignments
    if len(idle_vehicles) == 0:
        assignments = {'vehicles': {}, 'orders': {order_id: {'status': 'accepted'} for order_id in unassigned_orders}}

        for vehicle_id in state['vehicles'].keys():
            assignments['vehicles'][vehicle_id] = {'next_visits': state['vehicles'][vehicle_id]['next_visits']}

        # Log various variables
        if logger:
            fitness = simulate_evaluate(assignments, state, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders})
            logger.log(None, None, fitness, None, None, time, num_open_orders, num_delivered_orders, total_customer_wait_time, total_planned_distance)

        return assignments

    assignments = {'vehicles': {}, 'orders': {}}

    # Fill up assignments with previous assignments in order to preserve state
```

```
    for vehicle_id in state['vehicles'].keys():
        assignments['vehicles'][vehicle_id] = {'next_visits': state['
            vehicles'][vehicle_id]['next_visits']}

    for vehicle_id in idle_vehicles:
        vehicle = next(v for v in model.vehicles if v.id == vehicle_id)

        vehicle_route = []
        accepted_order_ids = []

        for order_id in unassigned_orders:
            order = next(o for o in model.orders if o.id == order_id)

            # If vehicle can accept order, assign it to vehicle
            if vehicle.can_accept(order, accepted_order_ids):
                accepted_order_ids.append(order_id)
                vehicle_route.append({
                    'location': order.delivery_location.id,
                    'delivery_list': [order_id],
                })
                # Make sure to update the model
                assignments['orders'][order_id] = {'status': 'accepted'}
                order.status = 'accepted'

        # If a new route exists, make it complete by starting at the
            depot, and finishing there as well
        if vehicle_route:
            vehicle_route.insert(0, {'location': 'DEPOT', 'pickup_list':
                accepted_order_ids})
            vehicle_route.append({'location': 'DEPOT'})
            assignments['vehicles'][vehicle_id] = {'next_visits':
                vehicle_route}

    # Log various variables
    if logger:
        fitness = simulate_evaluate(assignments, state, {loc.id: loc for
            loc in model.locations}, {order.id: order for order in model
            .orders})
        logger.log(None, None, fitness, None, None, time,
            num_open_orders, num_delivered_orders,
            total_customer_wait_time, total_planned_distance)

    return assignments
```

F Simulated Annealing Algorithm Implementation

Listing 12: The complete SA implementation

```
def sa(state, model, logger=None):
    # Gets capacity from the model (assumes equal capacities for all
       vehicles)
    capacity = next(v.capacity for v in model.vehicles)

    # Generate an initial random valid solution
    initial_solution = generate_random_solution(state, capacity)

    # Check fitness of said random solution
    current_fitness = simulate_evaluate(initial_solution, state, {loc.id
       : loc for loc in model.locations}, {order.id: order for order in
       model.orders})

    # Log various values
    if logger:
        time = state.get('time', 0)
        num_open_orders = len(state['open_orders'])
        num_delivered_orders = sum([1 for o in model.orders if o.
           is_delivered])
        total_customer_wait_time = sum([(o.delivery_time - o.
           release_date) for o in model.orders if o.is_delivered])
        total_planned_distance = sum([v.total_distance for v in model.
           vehicles])

    # Make sure not to disturb previous solutions
    best_solution = copy.deepcopy(initial_solution)
    state_copy = copy.deepcopy(state)
    best_fitness = current_fitness

    # Parameters
    T = 1000
    T_min = 0.001
    alpha = 0.999
    max_iterations = 100
    iteration = 0

    # Enter the main SA loop
    while T > T_min and iteration < max_iterations:
        iteration += 1

        # Making sure not to alter solutions it's not supposed to
        neighbour_solution = copy.deepcopy(best_solution)

        # Make use of the perturbation functions
        operation = random.choice(['swap', 'reassign', 'shuffle'])

```

```
        if operation == 'swap' and len(state_copy['open_orders']) >= 2:
            neighbour_solution = swap_orders(neighbour_solution, state,
                capacity)
        elif operation == 'reassign':
            neighbour_solution = shuffle_orders(neighbour_solution,
                state)
        else:
            neighbour_solution = reassign_order(neighbour_solution,
                state, capacity)

        # Evaluate perturbed solution
        neighbour_fitness = simulate_evaluate(neighbour_solution,
            state_copy, {loc.id: loc for loc in model.locations}, {order.
            id: order for order in model.orders})

        difference = neighbour_fitness - current_fitness

        # If better update
        if difference < 0:
            current_solution = neighbour_solution
            current_fitness = neighbour_fitness

            if neighbour_fitness < best_fitness:
                best_solution = copy.deepcopy(neighbour_solution)
                best_fitness = neighbour_fitness

        # Else update with a certain probability
        else:
            prob = math.exp(-difference / T)
            if random.random() < prob:
                current_solution = neighbour_solution
                current_fitness = neighbour_fitness

        # Decrease the temperature gradually
        T *= alpha

        # Log various values
        if logger:
            logger.log(iteration, best_fitness, current_fitness, 0, 0,
                time, num_open_orders, num_delivered_orders,
                total_customer_wait_time, total_planned_distance)

    return best_solution
```

G Genetic Algorithm Implementation

```
def ga(state, model, pop_size=120, generations=200, p_crossover=0.3, p_swap=0.1, p_reassign=0.1, p_shuffle=0.1, logger=None):
    # Gets capacity from the model (assumes equal capacities for all vehicles)
    capacity = next(v.capacity for v in model.vehicles)

    # Generates pop_size number of random solutions
    population = [generate_random_solution(copy.deepcopy(state), capacity) for _ in range(pop_size)]

    # Logs various variables
    if logger:
        time = state.get('time', 0)
        num_open_orders = len(state['open_orders'])
        num_delivered_orders = sum([1 for o in model.orders if o.is_delivered])
        total_customer_wait_time = sum([(o.delivery_time - o.release_date) for o in model.orders if o.is_delivered])
        total_planned_distance = sum([v.total_distance for v in model.vehicles])

    # Here it enters the main loop of the GA
    for gen in range(generations):

        # It calculates the individual fitnesses of the randomly generated solutions
        fitness = []
        for individual in population:
            score = simulate_evaluate(individual, state, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders})
            fitness.append(score)

        # Tournament selection
        parents = []
        for _ in range(pop_size):
            candidates = random.sample(list(zip(population, fitness)), k=3)
            parents.append(copy.deepcopy(min(candidates, key=lambda x: x[1])[0]))

        # Crossing over
        offspring = []
        for i in range(0, pop_size, 2):
            p1 = parents[i]
            p2 = parents[i + 1]

```

```
            if random.random() < p_crossover:
                child1 = crossover(p1, p2, state, capacity)
                child2 = crossover(p2, p1, state, capacity)
            else:
                child1, child2 = p1, p2
            offspring.extend([child1, child2])

        # Mutating
        for i in range(pop_size):
            if random.random() < p_swap:
                o = copy.deepcopy(offspring[i])
                offspring[i] = swap_orders(o, state, capacity)

            if random.random() < p_reassign:
                o = copy.deepcopy(offspring[i])
                offspring[i] = reassign_order(o, state, capacity)

            if random.random() < p_shuffle:
                o = copy.deepcopy(offspring[i])
                offspring[i] = shuffle_orders(o, state)

        # Checks the offspring's fitness
        offspring_fitness = []
        for individual in offspring:
            score = simulate_evaluate(individual, state, {loc.id: loc
                for loc in model.locations}, {order.id: order for order
                in model.orders})
            offspring_fitness.append(score)

        fitness_combined = fitness + offspring_fitness

        # Logs various variables
        if logger:
            best = min(fitness_combined)
            avg = (sum(fitness_combined) / len(fitness_combined))
            worst = max(fitness_combined)
            std = statistics.stdev(fitness_combined)
            logger.log(gen, best, avg, worst, std, time, num_open_orders
                , num_delivered_orders, total_customer_wait_time,
                total_planned_distance)

        # Selects the next population, by keeping the best solutions
        combined = list(zip(population + offspring, fitness_combined))
        combined.sort(key=lambda x: x[1])
        population = [x[0] for x in combined[:pop_size]]

    # Returns the best found solution
```

```
    best = min(population, key=lambda x: simulate_evaluate(x, copy.deepcopy(state), {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders}))
    return best
```

H Ant Colony Optimization Implementation

Listing 13: Complete ACO implementation

```
def aco(state, model, num_ants=100, generations=125, alpha_aco=2, beta_aco=5, rho=0.45, q=1, logger=None):
    # Gets capacity from the model (assumes equal capacities for all vehicles)
    capacity = next(v.capacity for v in model.vehicles)

    # Logs various variables
    if logger:
        time = state.get('time', 0)
        num_open_orders = len(state['open_orders'])
        num_delivered_orders = sum([1 for o in model.orders if o.is_delivered])
        total_customer_wait_time = sum([(o.delivery_time - o.release_date) for o in model.orders if o.is_delivered])
        total_planned_distance = sum([v.total_distance for v in model.vehicles])

    # Set initial pheromone levels
    pheromone_route = defaultdict(lambda: defaultdict(lambda: 1.0))
    pheromone_pairing = defaultdict(lambda: defaultdict(lambda: 1.0))

    # Set initial best fitness to infinite
    best_solution = None
    best_fitness = float('inf')

    # Enter main GA loop
    for gen in range(generations):
        solutions = []
        fitnesses = []

        # Generate randomized population
        for _ in range(num_ants):
            solution = generate_ant_solution(state, pheromone_route, pheromone_pairing, alpha_aco, beta_aco, capacity, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders})

```

```
            # Evaluate individuals
            fitness = simulate_evaluate(solution, state, {loc.id: loc
                for loc in model.locations}, {order.id: order for order
                in model.orders})

            solutions.append(solution)
            fitnesses.append(fitness)

            # If better update
            if fitness < best_fitness:
                best_solution = solution
                best_fitness = fitness

        # Evaporate pheromones
        for i in pheromone_route:
            for j in pheromone_route[i]:
                pheromone_route[i][j] *= (1 - rho)

        for i in pheromone_pairing:
            for j in pheromone_pairing[i]:
                pheromone_pairing[i][j] *= (1 - rho)

        best_this_gen_id = fitnesses.index(min(fitnesses))
        best_this_gen = solutions[best_this_gen_id]

        # Update pheromones
        for v_id, route in best_this_gen['vehicles'].items():
            visits = route["next_visits"]
            if not visits:
                continue

            pickup_lists = [v["pickup_list"] for v in visits if v['
                pickup_list']]
            for i in range(len(route) - 1):
                a, b = visits[i]["location"], visits[i + 1]["location"]
                pheromone_route[a][b] += q / best_fitness

            for pickup_list in pickup_lists:
                for o_id in pickup_list:
                    pheromone_pairing[v_id][o_id] += q / best_fitness

        # Log various variables
        if logger:
            logger.log(gen, min(fitnesses), statistics.mean(fitnesses),
                max(fitnesses), statistics.stdev(fitnesses), time,
                num_open_orders, num_delivered_orders,
                total_customer_wait_time, total_planned_distance)

```

```
    return best_solution
```

Listing 14: Implementation of the generate ant solution function

```
def generate_ant_solution(state, pheromone_route, pheromone_pairing,
    alpha_aco, beta_aco, capacity, locations_dict,
                              orders_dict):
    # Find modifiable orders
    # So orders that are not delivered nor on route nor being picked up
    orders_being_picked_up = set()
    for vehicle_id, vehicle_data in state["vehicles"].items():
        if (
                vehicle_data["status"] == "UNDER_SERVICE"
                and vehicle_data.get("current_visit", {}).get("location"
                    ) == "DEPOT"
        ):
            pickup_list = vehicle_data["current_visit"].get("pickup_list
                ", [])
            orders_being_picked_up.update(pickup_list)

    orders_waiting_for_pickup = [
        o_id for o_id, o_data in state["open_orders"].items()
        if o_data.get("pickup_time") is None
           and o_id not in orders_being_picked_up
    ]

    assignments = {'vehicles': {}, 'orders': {}}

    # Fill up the assignments dict to maintain the state
    for vehicle_id in state['vehicles'].keys():
        assignments['vehicles'][vehicle_id] = {'next_visits': state['
           vehicles'][vehicle_id]['next_visits']}

    # If no alterable orders, return
    if not orders_waiting_for_pickup:
        return assignments

    vehicle_ids = list(state['vehicles'].keys())
    random.shuffle(orders_waiting_for_pickup)

    # If there are modifiable orders assign them
    for o_id in orders_waiting_for_pickup:
        # Remove them first
        assignments, v = remove_order_from_vehicle(assignments, o_id)

        # Prepare data for reassignment
        o_data = state["open_orders"][o_id]
        values = []
```

```
41
42          order_rep = {
43              'id': o_id,
44              'pickup_location': o_data['pickup_location'],
45              'delivery_location': o_data['delivery_location'],
46              'quantity': o_data['quantity'],
47              'earliest_pickup_start': o_data['earliest_pickup_start'],
48              'latest_pickup_start': o_data['latest_pickup_start'],
49              'pickup_duration': o_data['pickup_duration'],
50              'earliest_delivery_start': o_data['earliest_delivery_start'
                    ],
51              'latest_delivery_start': o_data['latest_delivery_start'],
52              'delivery_duration': o_data['delivery_duration'],
53          }
54          # Check how good each assignment is per vehicle
55          for v_id in vehicle_ids:
56              current_route = assignments['vehicles'][v_id]['next_visits']
57              new_route = assign_order_to_vehicle(order_rep, state['
                    vehicles'][v_id], v_id, o_id, current_route, capacity,
58                                                  state)
59
60              cost_increase = estimated_extra_cost(current_route,
                    new_route, locations_dict, orders_dict)
61
62              pheromone = pheromone_pairing[v_id][o_id]
63              heuristic = 1 / (cost_increase + 0.000001)
64              value = (pheromone ** alpha_aco) * (heuristic ** beta_aco)
65              values.append((v_id, new_route, value))
66
67          total_values = sum(value for _, _, value in values)
68          r = random.random()
69          cumulative_sum = 0
70
71          # Roulette wheel selection
72          for v_id, new_route, value in values:
73              cumulative_sum += value / total_values
74              if r <= cumulative_sum:
75                  assignments['vehicles'][v_id]['next_visits'] = new_route
76                  assignments['orders'][o_id] = {'status': 'accepted'}
77                  break
78
79      return assignments
```

I Artificial Bee Colony Implementation

Listing 15: Complete implementation of the ABC algorithm

```python
def abc(state, model, colony_size=50, max_iterations=100, limit=100, logger=None):
    # Gets capacity from the model (assumes equal capacities for all vehicles)
    capacity = next(v.capacity for v in model.vehicles)

    # Logging various variables
    if logger:
        time = state.get('time', 0)
        num_open_orders = len(state['open_orders'])
        num_delivered_orders = sum([1 for o in model.orders if o.is_delivered])
        total_customer_wait_time = sum([(o.delivery_time - o.release_date) for o in model.orders if o.is_delivered])
        total_planned_distance = sum([v.total_distance for v in model.vehicles])

    # Generating the pool of random solutions
    initial_food_sources = [generate_random_solution(state, capacity) for _ in range(colony_size)]
    initial_food_fitnesses = [simulate_evaluate(food_source, state, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders}) for food_source in initial_food_sources]
    trial_counter = [0 for _ in range(colony_size)]

    best_solution = None
    best_fitness = float('inf')

    # Enter main loop of ABC
    for iteration in range(max_iterations):

        # Employed bees loop
        for i in range(colony_size):
            r = random.randint(1, 3)
            # Make sure it doesn't alter solutions it shouldn't
            sol = copy.deepcopy(initial_food_sources[i])
            # Mutate solution
            if r == 1:
                neighbour_food_source = swap_orders(sol, state, capacity)
            elif r == 2:
                neighbour_food_source = reassign_order(sol, state, capacity)
            else:
                neighbour_food_source = shuffle_orders(sol, state)

            # Evaluate solution
```

```
            neighbour_food_fitness = simulate_evaluate(
                neighbour_food_source, state, {loc.id: loc for loc in
                model.locations}, {order.id: order for order in model.
                orders})

            # If better, update
            if neighbour_food_fitness < initial_food_fitnesses[i]:
                initial_food_sources[i] = neighbour_food_source
                initial_food_fitnesses[i] = neighbour_food_fitness
                trial_counter[i] = 0

            # Else update trial count
            else:
                trial_counter[i] += 1

        # Onlooker bees section

        # Decide probabilities based on employed bees solution
        prob = [1/f for f in initial_food_fitnesses]
        total = sum(prob)
        prob = [p / total for p in prob]

        for _ in range(colony_size):
            # Choose solution based on probabilities
            i = np.random.choice(range(len(prob)), p=prob)
            r = random.randint(1, 3)
            sol = copy.deepcopy(initial_food_sources[i])
            # Mutate chosen solution
            if r == 1:
                neighbour_food_source = swap_orders(sol, state, capacity
                    )
            elif r == 2:
                neighbour_food_source = reassign_order(sol, state,
                    capacity)
            else:
                neighbour_food_source = shuffle_orders(sol, state)

            # Check fitness
            neighbour_food_fitness = simulate_evaluate(
                neighbour_food_source, state, {loc.id: loc for loc in
                model.locations}, {order.id: order for order in model.
                orders})

            # Update accordingly
            if neighbour_food_fitness < initial_food_fitnesses[i]:
                initial_food_sources[i] = neighbour_food_source
                initial_food_fitnesses[i] = neighbour_food_fitness
                trial_counter[i] = 0
```

```
            else:
                trial_counter[i] += 1

        # Scout bees section
        for i in range(colony_size):
            # If solution is not promising, randomize a new solution instead
            if trial_counter[i] >= limit:
                initial_food_sources[i] = generate_random_solution(state, capacity)
                initial_food_fitnesses[i] = simulate_evaluate(initial_food_sources[i], state, {loc.id: loc for loc in model.locations}, {order.id: order for order in model.orders})
                trial_counter[i] = 0

        # Find best fitness
        best_fitness_this_iteration = min(initial_food_fitnesses)
        if best_fitness_this_iteration < best_fitness:
            best_fitness = best_fitness_this_iteration
            best_solution = initial_food_sources[initial_food_fitnesses.index(best_fitness_this_iteration)]

        # Log various variables
        if logger:
            logger.log(iteration, min(initial_food_fitnesses), statistics.mean(initial_food_fitnesses), max(initial_food_fitnesses), statistics.stdev(initial_food_fitnesses), time, num_open_orders, num_delivered_orders, total_customer_wait_time, total_planned_distance)

    return best_solution
```