universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

A Genetic Algorithm Approach to the Mobile Locker Location Problem (MLLP)

verfasst von | submitted by

Simon Urhofer Bakk.phil.

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 915
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Betriebswirtschaft
Betreut von | Supervisor:	Univ.-Prof. Dr. Jan Fabian Ehmke

Abstract/Abstract

This master thesis introduces a Genetic Algorithm (GA) approach to the Mobile Locker Location Problem (MLLP), a novel problem in the last-mile distribution sector concerning the allocation of Mobile Parcel Lockers (MPLs). An original GA has been developed and configured in a way to test its results with those of an exact solver. Through a series of experiments, the proposed GA demonstrated its capability to generate optimal solutions for smaller instances and reasonable results for larger, more complex instances, outperforming an exact solver under certain conditions in terms of runtime efficiency. The findings of this thesis highlight the GA's potential in real-world applications, offering valuable insights for future research in the area of logistics optimization.

In dieser Masterarbeit wird ein Genetischer Algorithmus (GA) Ansatz für das Mobile Locker Location Problem (MLLP) vorgestellt, ein neuartiges Problem im Last-Mile-Distributionssektor, das die Zuteilung von Mobile Parcel Lockers (MPLs) betrifft. Ein neuartiger GA wurde entwickelt und so konfiguriert, dass seine Ergebnisse mit denen eines exakten Solvers verglichen werden können. Durch eine Reihe von Experimenten hat der vorgeschlagene GA seine Fähigkeit unter Beweis gestellt, optimale Lösungen für kleinere Instanzen und akzeptable Ergebnisse für größere, komplexere Instanzen zu generieren, wobei er unter bestimmten Bedingungen einen exakten Solver in Bezug auf die Laufzeiteffizienz übertrifft. Die Ergebnisse dieser Arbeit unterstreichen das Potenzial des GA bei der Anwendung an realen Last-Mile-Problemen und bieten wertvolle Erkenntnisse für die zukünftige Forschung im Bereich der Logistikoptimierung.

Contents

List of Abbreviations

AHD…Attended Home Delivery

GA…Genetic Algorithm

FPL…Fixed Parcel Locker

MPL…Mobile Parcel Locker

MLLP…Mobile Locker Location Problem

TOP…Team Orienteering Problem

Introduction and Motivation

The concept of Mobile Parcel Lockers has been a rather recent addition to the wide array of scientific research regarding the so-called "last-mile distribution" in the ever-growing e-commerce market. Last-mile can be described as "the last stretch of a business-to-consumer (B2C) parcel delivery to the final consignee, who has to take reception of the goods at home or at a cluster/collection point" (Gevaers et al., 2009).

Last-Mile Distribution in General

Global e-commerce sales are ever-increasing, with their growth being accelerated by the COVID-19 pandemic and its implications. While the number of worldwide retail e-commerce sales in 2014 amounted to 1.3 trillion U.S. dollars, it has increased more than four-fold to 5.784 trillion U.S. dollars in 2023 and is projected to grow to 8.034 trillion U.S. dollars by the end of 2027. (Statista, 2024a) As of 2023, e-commerce sales account for over 19.4% of total retail sales worldwide, with an expected increase to 22.6% by the end of 2026. (Statista, 2024b)

This steep growth over the past years can at least partially be attributed to the COVID-19 pandemic. In the United States, overall online sales saw an increase of 25% within the first year of the pandemic. (Song, 2021) In a study in 2020, 29% of surveyed U.S. Americans declared their intention to never go back to shopping in person again. (Competera, 2021)

With e-commerce retail numbers rising to this extent, executing last-mile distribution efficiently is of high importance from a business point of view. Goodman (2005) states that not less than 28% of all transportation costs occur in this step of the supply chain. He quotes Steve Senkus, president and CEO of NonstopDelivery Inc.: "Last-mile logistics takes the most overhead and involves the most care (…) It takes a lot more overhead to get that delivery done, whether it's inside delivery with two men or unpack, haul away, set up and installation. Those are areas where very few providers have specialized, but they all know it's a very important part of the supply chain."

The last-mile distribution market has changed in the last 19 years, of course; varying different ways of connecting customers to their goods have been researched on and novel concepts are being developed. One of them being Mobile Parcel Lockers, for which scientific literature is still rather scarce.

The classic last-mile methods can be categorized by the following typology provided by Gevaers et al. (2011):

- Pick up at distribution center or shop (Place of delivery)
- Clustering (Place of delivery)
    - Reception boxes (Type of delivery)
    - Collection points (Type of delivery)
    - Post offices (Type of delivery)
- Home deliveries (Place of delivery)
    - Attended (Type of delivery)
    - Unattended (Type of delivery)

Introduction to Mobile Parcel Lockers

Mobile Parcel Lockers can be understood as delivery vehicles that are – in contrast to the more common stationary lockers - able to change their locations during the day, either autonomously or moved by a human driver. They can stop at so-called stopovers in the vicinity of customers' everyday walk of life (the customer's location data must be provided for this to work successfully) so that customers can pick up their parcels on their way to work, shopping trips, etc. within a preselected time window. The Mobile Parcel Lockers have to start at a depot and return to it after a working day is over. The optimization problem that is looking for the optimal locations of stops the Mobile Parcel Locker has to visit has been introduced as the Mobile Locker Location Problem (MLLP) by Schwerdfeger & Boysen (2020).

Contrary to Schwerdfeger & Boysen (2020), who aim to optimize the size of the fleet of Mobile Parcel Lockers by minimizing it, the objective of the MLLP within this thesis is to maximize the overall profit (i.e., the number of satisfied customers in total) with a fixed fleet size. While Schwerdfeger & Boysen (2020) commit themselves to satisfying every customer there is, in the interpretation of the MLLP that is used within this thesis, ignoring some customers temporarily is allowed. The unsatisfied customers are being served at a later date and are not considered while operating on the MLLP.

Another difference between Schwerdfeger's & Boysen's and this thesis' interpretation of the MLLP is the handling of time windows in which it is possible for customers to pick up their parcels. In this interpretation, whenever a certain customer is accepted at a certain

stopover, the Mobile Parcel Locker must remain at that stopover for the entirety of the time window to satisfy this customer. For this reason, every customer that can be served from a certain stopover and has not yet been served from another stopover will automatically be considered a satisfied customer, whenever an MPL has been routed to this stopover. An already satisfied customer cannot be served again by any other MPL.

The above-introduced interpretation of the MLLP can be viewed as a variant of the Team Orienteering Problem (TOP). The TOP originates from an outdoor sport, usually played in mountainous or heavily forested areas, called orienteering. In the single-competitor variant of orienteering, a competitor has to navigate through an area full of control points with the help of a compass and a map, tries to visit many different control points, and has to return to a specified control point within a time limit. The competitor can attain a varying score at each control point, and therefore has to determine which control points they want to visit to maximize their expected overall score beforehand. The competitor with the highest score is declared the winner of the competition. Team orienteering is an extension of the single-competitor variant of this sport, where instead of a single competitor, a team of several competitors participate. Each team member starts at the same specified control point and tries to visit different control points to increase the team score. The score of a control point that has already been visited cannot be attained by another member of the same team. For this reason, each team has to carefully plan ahead which control points should be visited by which team members to avoid overlapping. (Chao et al., 1996)

A considerable amount of the aspects of the TOP can be compared to the understanding of the MLLP that is applied for this thesis like the aspect of a team consisting of multiple contestants aiming to attain scores by visiting certain locations that diminish in value if they have already been visited by other team members. The TOP has been used to model numerous logistical problems similar to the MLLP in scientific literature for this reason. Time windows are usually not included in classical TOPs, but there can be found a considerable amount of research on the Team Orienteering Problem with Time Windows in the literature.

There exists a substantial number of different (meta-)heuristical approaches to the TOP in scientific literature. Besides more common approaches like Simulated Annealing or Local Search, there are interesting evolutionary algorithms like Particle Swarm Optimization, which is based on swarm intelligence observed in nature, or the Genetic Algorithm (GA). The GA is a meta-heuristic that is based on the process of natural selection. It aims to produce good solutions by applying so-called crossover and mutation operators on a predefined population of solutions. (Vansteenwegen & Gunawan, 2014)

Research Question and Motivation

It has been decided to put the central focus of this thesis on exploring the application of a GA approach to solving the MLLP. Thus, the following research question arises:

How can a Genetic Algorithm be configured to address the Mobile Locker Location Problem (MLLP) effectively, demonstrating competing solution quality and runtime efficiency compared to an exact solver?

The MLLP is interpreted as a TOP within this because the existing literature focuses on varying fleet sizes. In reality, companies providing MPL service might struggle to build a dynamic MPL fleet because of the cost-intensiveness of procuring new MPL. They might prefer working with a fixed fleet size to ensure planning security.

The idea to develop a GA approach to the MLLP stems from the complexity of the real-world MLLP. When key figures like fleet size, number of customers and possible MLLP halting positions increase, a solver might reach its limits when creating tour plans, whereas an algorithm is still able to produce viable tour plans. The goal of this thesis is to introduce a Genetic Algorithm that can outperform an optimal solver in this respect.

Theoretical Background

In the introduction, different last-mile methods have been presented. The most commonly used method at the moment is called Attended Home Delivery (AHD), as per Manerba et al. (2018). Mobile Parcel Lockers are a rather recent addition to the variety of last-mile methods and thus have not been the subject of a wide range of scientific research.

Literature Review on Traditional Approaches

Manerba et al. (2018) described AHD as a form of delivery that requires the presence of the customer during the delivery and includes time windows within which the delivery must take place. Their study aimed to analyze the influence of time window width on the distances traveled by delivery vehicles and its implications for environmental sustainability. To evaluate the impact of time window width, the authors formulated the problem as a Vehicle Routing Problem that was solved with a Mixed Integer Linear Programming solver. Their findings indicated that increasing the time window width leads to a reduction in the total kilometers traveled by delivery vehicles, thereby minimizing the environmental impact.

Another perspective on AHD, which is more closely related to the MLLP, is provided by Lang et al. (2021). The authors highlighted the issue of narrow delivery time slots, which lead to high costs for retailers as the flexibility of route planning is limited by them. Lang et al. (2021) focused on the opportunity cost of accepting customers early in the order horizon and thereby missing out on potential future customers. In order to calculate these opportunity costs, they introduced dynamic slotting techniques that compare the reward from accepting an order to the opportunity cost of not reserving the required delivery capacity for later orders. However, calculating this opportunity cost accurately involves solving a complex vehicle routing and scheduling problem. To tackle this issue, the paper proposes three dynamic slotting approaches that rely on an anticipatory, simulation-based preparation phase before the order horizon to approximate the opportunity cost. The proposed approaches involve creating anticipatory schedules solving the Team Orienteering Problem with Multiple Time Windows on samples of predicted demand to evaluate the feasibility of delivering order requests. After conducting an extensive computational study on the behavior of the proposed approaches, the authors concluded that all of them require only negligible run times within the order horizon and that they are applicable in practice.

A different approach to last-mile distribution, on which a lot of scientific research is being performed, are classic Parcel Lockers which are, contrary to the mobile ones that are being dealt with in this thesis, stationary. These can be categorized as Reception Boxes in the typology of Gevaers et al. (2011).

In a paper by Deutsch and Golany (2018), a Parcel Locker is defined as a group of lockers, sited in apartment blocks, workplaces, railway stations etc. These lockers can be accessed by individual opening codes that allow customers but no one else to get ahold of their parcels. Deutsch and Golany (2018) claim their paper to be the first scientific attempt to develop a quantitative approach to determine the optimal number, location, and amount of individual lockers per locker location of these Parcel Lockers. They also aimed to calculate a sum of total profit that is equal to the revenue from customers who use the service, minus the facilities' fixed and operational setup costs, the discounts in the delivery costs for customers who need to travel to collect their parcels, and the loss of potential customers who are not willing to travel for service. The authors declared that they were able to transform their problem into the well-known Uncapacitated Facility Location Problem, which can be solved by a standard solver. Deutsch and Golany (2018) expressed their hope that the findings of their study will help companies that specialize in providing Parcel Lockers.

A mix of home deliveries and delivery to Parcel Lockers can be found in a paper by Jiang et al. (2019). It introduces a novel problem called the Traveling Salesman Problem with Time Windows for Last Mile Delivery in Online Shopping (TSPTWLMDOS). The objective of this problem is to find a minimum-cost tour that covers both home deliveries and deliveries to Parcel Lockers while assigning the deliveries of unvisited customers automatically to Parcel Lockers. The use of shared delivery facilities, where customers can pick up their parcels, is shown to reduce delivery costs and carbon emissions. The authors aimed to solve the TSPTWLMDOS with the help of a heuristic algorithm called General Variable Neighbourhood Search (GVNS). They were able to prove the efficiency of the proposed GVNS by comparing its results to computation results. This mix of home deliveries and delivery to Parcel Lockers proves to cause a significant reduction in daily logistics cost of the last-mile delivery, Jiang et al. concluded.

Literature Review on Mobile Parcel Locker Approaches

However, there has also been some scientific research conducted on Mobile Parcel Lockers, most importantly by Schwerdfeger & Boysen (2020), who, as already mentioned, introduced the Mobile Locker Location Problem (MLLP) that is being dealt with within this thesis. In their pioneering paper, Schwerdfeger & Boysen (2020) tried to assess the capability of Mobile Parcel Lockers for last-mile distribution by comparing their efficiency to stationary lockers. To accomplish this, they introduced the MLLP, a dynamic location problem that aims to optimize the fleet size (i.e., the number of MPLs) while feasibly serving a set number of customers. To solve this novel optimization problem, the authors developed three different Mixed-Integer Programs Models as well as a greedy heuristic which was used to find fleet size upper bounds for the MIPs. Finally, Schwerdfeger & Boysen (2020)

compared their results with those of the same data instances of stationary lockers and were thus able to prove the efficiency of MPLs. They concluded that the introduction of the MPL concept promises a "tremendous" reduction of fleet size compared to their commonly used stationary counterparts. Within their experiments, the needed fleet sizes to serve the same number of customers of stationary lockers were between 70% and 400% larger than those of MPLs; MPLs showed their advantage over their stationary lockers especially strongly when the distances the customers are willing to walk are small.

The authors recognized that a “serve all customers while reducing fleet size” approach might not be the most economical approach to the MLLP as the monetary costs of MPLs, especially autonomous ones, might be substantially higher than currently used concepts. Thus, Schwerdfeger & Boysen (2020) encouraged further research into the capability of MPLs for last-mile distribution with a focus on different key performance indicators.

Since the introduction of the MLLP, numerous researchers have come up with papers regarding the above-mentioned problem. One of them being Wang et al. (2020) who developed an approach to supplementing stationary lockers with MPLs. They integrated the operating problems of MPLs into a non-linear integer programming model, solved the location and route optimization problems of the MPLs with a two-layered Genetic Algorithm, and compared the results before and after integration. By testing the effects on the operating plan if mobile parcel lockers operated similarly to traditional home delivery in terms of demand aggregation, they were able to conclude that if this was the case, a great reduction in delivery time and labor costs could be achieved; on the other hand, not using demand aggregation for the MPLs leads to a decrease in fleet size.

Another paper was published by Li et al. (2021) which focuses on autonomously driving MPLs called Autonomous Mobile Lockers (AMLs) that transfer parcels from depots to couriers in the field. The authors tried to prove the capability of AMLs in city logistics by developing a cooperative system between robots (AMLs) and humans (couriers). They tested this system with a self-developed heuristic and concluded that the implementation of AMLs which are supported by human couriers can be able to improve overall operational efficiency in city logistics.

Peppel et al. (2024) tried to tackle the MLLP from an environmental perspective. They investigated the impact of MPLs on last-mile delivery networks, focusing on cost savings and environmental benefits. A non-linear integer programming model was developed to optimize MPL locations, incorporating factors like recipients' availability and route cost estimation, and was validated using a dataset of 742,457 parcel shipments. The authors were able to prove that the integration of MPLs can lead to an 8.7% reduction in operating

costs and up to 5.4% savings in CO2 emissions, particularly benefiting densely populated urban areas while potentially increasing emissions in rural settings.

Kötschau et al. (2023) explored the possibility of combining AHD with MPLs as well as with Fixed Parcel Lockers (FPL) which are the stationary equivalent of MPLs that has already been mentioned; they are installed at important locations throughout cities but, contrary to their mobile counterparts, cannot be moved. In order to do this, they introduced the Heterogeneous Locker Location Problem which aims to maximize the number of customers served while considering individual customer preferences regarding pickup distances and time windows. They also introduced two different kinds of customers:

- Restrictive customers, which are defined by a limited willingness for pickup efforts, accepting only closer stopovers, and having shorter pickup time windows. This type of customer is best served by AHD.
- Flexible customers, who are willing to travel further distances and have longer pickup time windows. This type of customer is best served by MPLs or FPLs.

The authors discovered that both types of customers benefit from the combination of AHD and MPLs or FPLs respectively. There have been tested out various combinations resulting in the following findings:

- 24% more restrictive customers were accepted when combining AHD and FPL
- 18% more restrictive customers were accepted when combining AHD and MPL
- 4% more flexible customers were accepted when combining AHD and FPL
- 6% more flexible customers were accepted when combining AHD and MPL

One of the most recent papers on MPLs is one by Liu et al. (2023). Liu and his co-authors formulated the so-called Mobile Parcel Locker Problem which is a variant of the MLLP. However, Liu et al. (2023) criticize the existing literature on MPLs for assuming static customer demands, while, in their opinion, stochastic customer demands correspond much more closely to real-world city logistics. Thus, the authors developed a novel Hybrid Q-Learning-Network-based Method (HQM) to solve the proposed MPLP and compared their results with the results of a Genetic Algorithm as well as an exact solver. Liu et al. (2023) can prove that their HQM can outperform both the exact solver as well as the Genetic Algorithm. Based on their results, the authors conclude that the capability of MPLs in city logistics is strongly dependent on time window constraints imposed by customers and parking spaces. Thus, they suggest that in the start-up phase of MPL implementation, MPLs should be supplemented with common last-mile delivery methods like stationary lockers.

Literature Review on a Genetic Algorithm Approach to the Team Orienteering Problem

All the above-mentioned papers on MPLs have in common that they are working with a self-imposed limitation of satisfying every single customer while minimizing key performance indicators like fleet size, traveling distance, or delay of service. However, it could make sense to formulate the MLLP with a fixed fleet size and the allowance to temporarily ignore customers; this approach to the MLLP might be closer to reality, as logistics companies often are not able to make changes in their fleet size flexibly because of monetary reasons. This understanding of the MLLP can be, as already mentioned in the introduction, understood as a variant of the Team Orienteering Problem. Furthermore, both the papers by Wang et al. (2020) and Liu et al. (2023) make use of Genetic Algorithms as a way to solve the MLLP which suggests that the application of this meta-heuristic can be promising when solving the MLLP.

For these reasons, a different interpretation of the MLLP than the pre-existing papers has been chosen. As already mentioned in the introduction, in the course of this thesis the MLLP is interpreted as a Team Orienteering Problem (TOP). For a more detailed explanation of this decision, see under Research Question and Motivation.

As the use of a Genetic Algorithm approach to the MLLP showed promising results in the existing literature, papers with a GA approach to the TOP were of particular interest for this thesis, with a paper by Karbowska-Chilinska & Zabielski (2013) being the most influential one for this thesis. Karbowska-Chilinska & Zabielski introduced a new GA that they tested on the so-called Orienteering Problem with Time Windows. This well-known routing problem shares similarities with the interpretation of the MLLP as a TOP. Their proposed GA is of a typical structure:

- **Initialization:** A solution (i.e., a sequence of locations, or a route) is coded into a chromosome. An initial population of 150 solutions is generated; each solution is initialized by picking a random vertex from a set of feasible vertices and adding it to the solution until further addition of vertices is no longer possible (i.e., another added solution's time windows would exceed the time horizon).
- **Selection:** Next, a new population is generated by performing tournament grouping selection. Random solutions are grouped and compete within each group, with the fittest solution based on the total profit^3/travel time being selected. This process is repeated to form a new population of solutions, aiming to improve the overall of the population.
- **Crossover:** Here, two solutions are randomly selected from the population. All the vertices from these two solutions with similar time windows are added to a set of

pairs (i.e., two similar vertices). One of these pairs is randomly selected, and the position of each vertex within its solution becomes the point of crossover. From this point on, the chromosome fragments of the two solutions are exchanged, and two new children tours with genes from both their parents are created. This crossover method was highly influential for the GA approach of this thesis.

- **Mutation:** Two types of mutation (each with a probability of 50%) are performed on a randomly selected solution from the population: Either a promising vertex is inserted into the selected solution, or a randomly selected vertex is deleted from the solution.

This GA is performed until a fixed number of generations is reached or if no improvement can be found after 100 iterations. The solution with the highest profit value of the final population of the GA is returned as a result.

Karbowska-Chilinska & Zabielski (2013) concluded that their proposed GA demonstrates competitive performance compared to Iterated Local Search.

Literature Comparison					
	Services			Additional Properties	
Paper	AHD	FPL	MPL	TOP	GA
Manerba et al. (2018)	X				
Lang et al (2021)	X			X	
Deutsch & Golany (2018)		X			
Jiang et al. (2019)	X	X			
Schwerdfeger & Boysen (2020)			X		
Wang et al. (2020)		X	X		X
Li et al. (2021)			X		
Peppel et al. (2024)			X		
Kötschau et al. (2023)	X	X	X		
Liu et al. (2023)			X		X
Karbowska-Chilinska & Zabielski (2013)				X	X

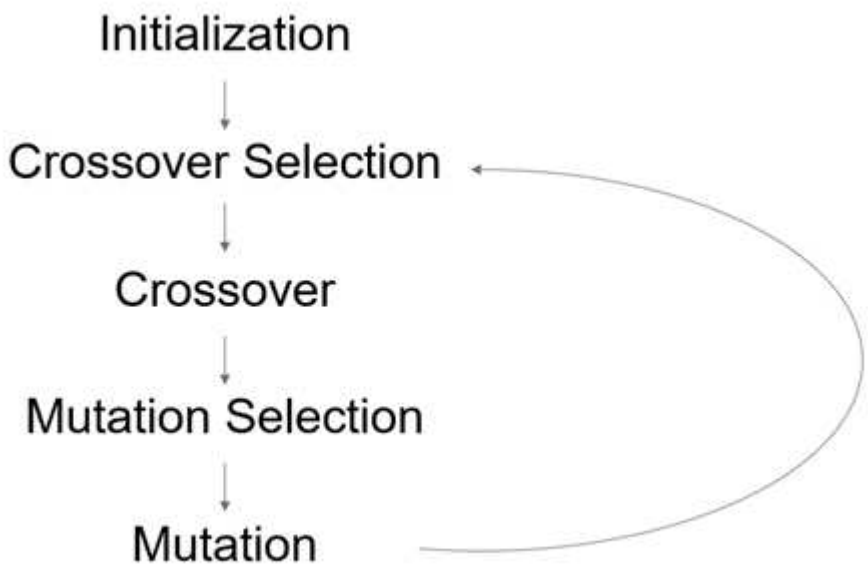
Table 1: Literature comparison of all papers discussed

Methodology

A novel GA approach that was influenced by the one introduced by Karbowska-Chilinska & Zabielski (2013) has been developed for this thesis. This GA approach follows the standard structure of a GA but its operators are mostly of a new formulation.

The Structure of the Proposed Genetic Algorithm

The GA's structure is based on a typical GA structure that can be found in the scientific literature and is the basis of Karbowska-Chilinska & Zabielski's GA as well. It can be summarized in the following way:



At first, an initial population with a size of 150 giant tours is generated, then two giant tours are selected for crossover, the crossover operation is carried out, and, finally, the mutation operations are performed. The individual steps of the GA are explained in more detail below.

The GA iterates through the crossover and mutation steps until a time limit is reached. After each iteration of the GA, the best solution of the whole population is looked for and saved as the best-found giant tour (in regard to the number of served customers), replacing the former best-found giant tour. After the complete GA run has finished, the best-found giant tour of all iterations is returned.

Necessary Adjustments

In order to transform a real-world problem like the MLLP into a theoretical one on which a GA can be performed, a few assumptions have to be made: We are operating on a two-dimensional plane, with the coordinates of the stopover and customer locations being known exactly. The MPLs are driving at a constant speed (in our case 30 km/h) and are

spawning at the first stopover of their tour and ending their route at the last stopover of their tour; the drive-time from and to the depot is not considered. We further assume that customers provide an exact radius of the maximum distance they are willing to walk to pick up their parcels as well as exact time windows during which they are available for being serviced. The time windows always open and end at full hours and can be assigned on a time horizon between hours 10 to 22. An MPL's halting time at any given stopover lasts from the earliest opening time of a customer's time window who is being served at this stopover until the latest closing time of a customer's time window who is being served at this stopover.

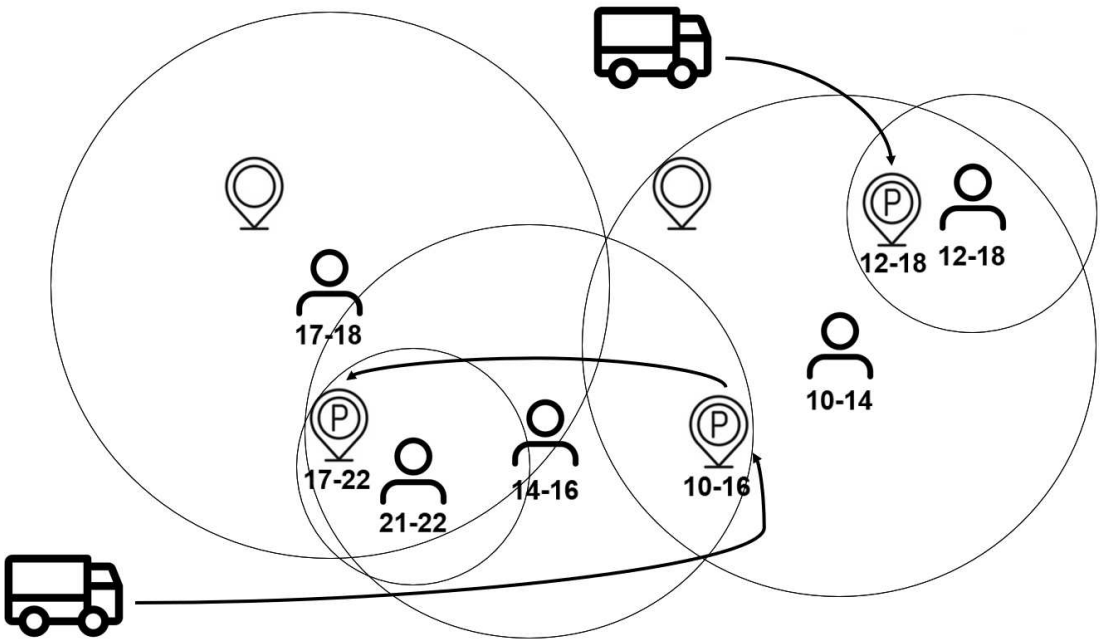


Figure 1: Depiction of a possible MPL tour plan

In this example (see Figure 1), there are five stopovers as well as five customers with individual walking radii and availability time windows. The right-most customer has a small walking distance radius and can only be served at one stopover. The upper MPL only travels to this stopover and halts there from hours 12 to 18, serving just one customer. The bottom-right stopover can be reached by two customers, with both customers being able to reach another stopover. However, these two customers with time windows of hours 10-14 and hours 14-16 are being served at the bottom-right stopover, making the lower MPL's halting time hours 10-16. Then, the lower MPL travels to the bottom-left stopover (note that service time can only start at full hours so there always has to be a difference of at minimum one hour between the end of service at one stopover and the beginning of service of the next stopover within the tour, no matter how small the travel time between those stopovers might be). At the bottom-left stopover, the remaining two customers are served, with the halting time of the MPL again being the combination of both these customers' time windows. The

remaining two available stopovers are not visited by any MPL as all customers have already been satisfied in this example.

The GA that is introduced within this thesis shares some similarities to the GA by Karbowska-Chilinska & Zabielski (2013) that has been described in the Literature Review section. One of these similarities is the interpretation of vehicle tours as chromosomes. The problem of treating the many individual tours of each vehicle within an MPL fleet as a single chromosome is combated by the introduction of giant tours. Every singular MPL tour is combined into one giant tour; the time windows of each stopover within this giant tour are adjusted to the overall time horizon. A visualization:

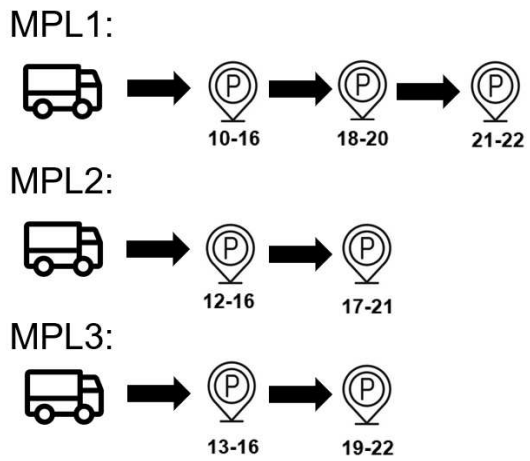


Figure 2: The singular tours of 3 MPLs

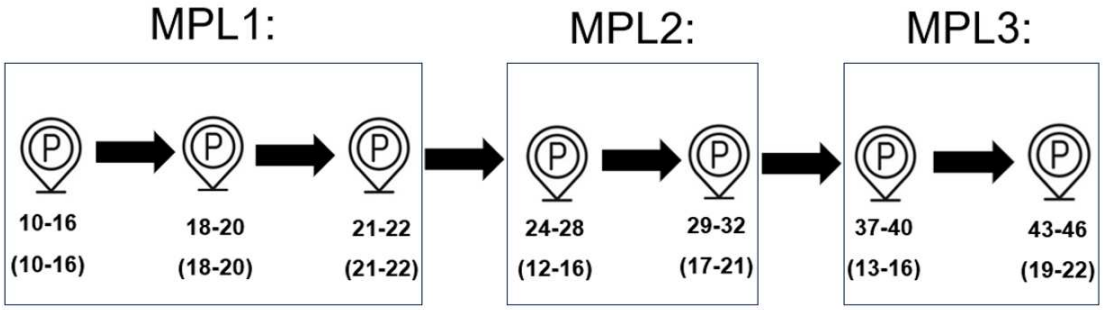


Figure 3: The 3 MPL tours combined into one giant tour

A pseudocode representation of the GA:

```
1:  population_size = 150
2:  best_tour = None
3:  best_fitness = infinity
4:
5:  population = []
6:  for i in range(population_size):
```

```
7:      tour = randomly_generate_tour()
8:      fitness = evaluate_tour(tour)
9:      if fitness < best_fitness:
10:         best_tour = tour
11:         best_fitness = fitness
12:     population.append(tour)
13:
14: while not time_limit_reached():
15:     tour_1 = tour_to_crossover1(population)
16:     tour_2 = tour_to_crossover2(population)
17:
18:     new_tour_1, new_tour_2 = crossover(tour_1, tour_2)
19:    replace_parents(population, tour_1, new_tour_1, tour_2, new_tour_2)
20:
21:     if should_mutate():
22:         tour_to_mutate = random_tour(population)
23:         mutate(tour_to_mutate)
24:
25:     for tour in population:
26:         fitness = evaluate_tour(tour)
27:         if fitness < best_fitness:
28:             best_tour = tour
29:             best_fitness = fitness
30:
31: return best_tour
```

The initialization step is represented in lines [1] to [12], the selection, crossover and mutations steps are represented in lines [15] to [23], and finally, in lines [25] to [29] the best-found tour is updated if applicable.

Stopover Update

The Stopover Update operator is not a part of the GA per se and is thus not mentioned in the structure section but acts as an integral helper operator in the crossover and mutation step.

A pseudocode representation of the stopover update:

```
1: def StopoverUpdate(giantTour):
2:     tour.TotalServedCustomersCount = 0
3:
4:     ServedCustomersSet = empty set
5:
6:     for each stopover in tour.VisitedPoints:
7:         ResetStopoverToOriginalState(stopover)
8:
9:         stopover.ServedCustomersCount = 0
10:
11:        customersToRemove = empty list
12:
13:        for each customerID in stopover.IDreachableCustomers:
14:            if customerID is in ServedCustomersSet:
15:                add customerID to customersToRemove
16:
17:        for each customerID in customersToRemove:
18:            remove customerID from stopover.IDreachableCustomers
19:
20:        stopover.ServedCustomersCount = length of
stopover.IDreachableCustomers
```

```
21:
22:         tour.TotalServedCustomersCount +=
stopover.ServedCustomersCount
23:
24:     return tour
```

This operator updates each stopover of a giant tour that is passed to it. First, each stopover is reset to its original state (i.e., all customers that can be served at this stopover in theory are considered served at said stopover) [7]. Then, the operator loops through the whole giant tour and keeps track of all the customers that have been served already. Whenever a customer would be served a second time in the same giant tour, said customer is removed from the corresponding stopover [11] to [18]. Finally, the count of served customers of each stopover and eventually the total count of served customers of each giant tour is updated [20] to [22].

Initialization

In the first step of the GA, the population generation is carried out. The initialization step is an integral part of the GA to generate an initial population which will be the basis for all the steps that follow. The population size is and stays constant at 150 giant tours. One by one, these giant tours are created in one of the two following ways:

Random initialization:

A pseudocode representation of the random initialization:

```
1: population_size = 150
2: population = []
3: for _ in range(population_size) do
4:     giant_tour = []
5:     all_possible_stopovers = get_all_possible_stopovers()
6:     while all_possible_stopovers do
7:         current_MPL = []
8:         unfeasible_stopovers = []
9:         while all_possible_stopovers do
10:                                                 selected_stopover    =
select_random_stopover(all_possible_stopovers)
11:             if is_feasible(selected_stopover, current_MPL) then
12:                 insert_stopover(selected_stopover, current_MPL)
13:             else
14:                 unfeasible_stopovers.append(selected_stopover)
15:             end if
16:             all_possible_stopovers.remove(selected_stopover)
17:         end while
18:         for stopover in unfeasible_stopovers do
19:             stopover.adjust_time_window(12)
20:         end for
21:         all_possible_stopovers = unfeasible_stopovers[:]
22:         giant_tour.append(current_MPL)
23:     end while
24:     population.append(giant_tour)
25: end for
26: return population
```

At every iteration of the random initialization, a random stopover is selected from the list of all possible stopovers and checked for its feasibility [10] to [16]. A stopover is considered feasible if its time window does not overlap with any already inserted stopovers of the current MPL, if it can be reached before its time window opens (considering the driving time from the preceding stopover), and if the capacity of the current MPL will not be violated by inserting it. If the stopover is feasible, it will be inserted into the current MPL according to its time windows; if the stopover is not feasible, it is removed from the list of all possible stopovers and added to a new list of unfeasible stopovers. This process is repeated until there are no more stopovers in the list of all possible stopovers; hereby, the first MPL is considered finished, and we move on to the second MPL [22]. The list of unfeasible stopovers becomes the new list of all possible stopovers [21] (note that stopovers that already have been inserted in the previous MPL will therefore not be eligible for random selection) and the same insertion method as for the first MPL is performed. However, there is one difference: In order to treat the entirety of all the singular tours of each MPL as a giant tour with corresponding time windows, 12 hours will be added to the time window of every stopover in the new list of possible stopovers whenever we move to the next MPL [18] & [19]. When all MPLs are filled in this way, a giant tour is considered initialized and added to the population [24].

Fitness value-based initialization:

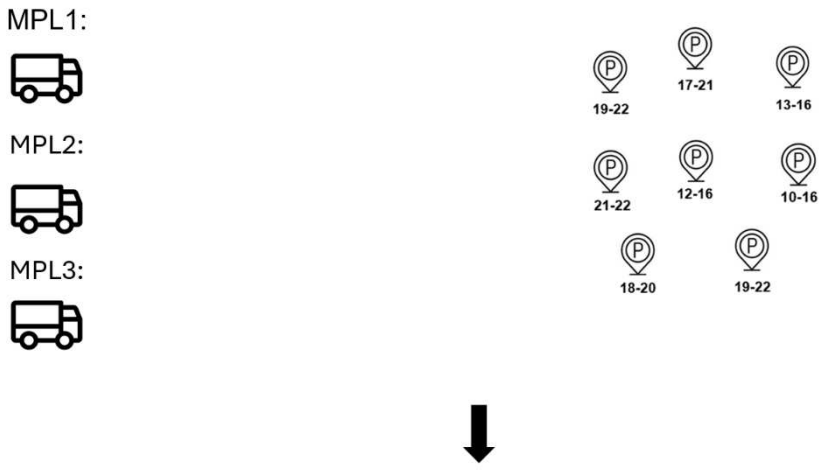
A pseudocode representation of the fitness initialization:

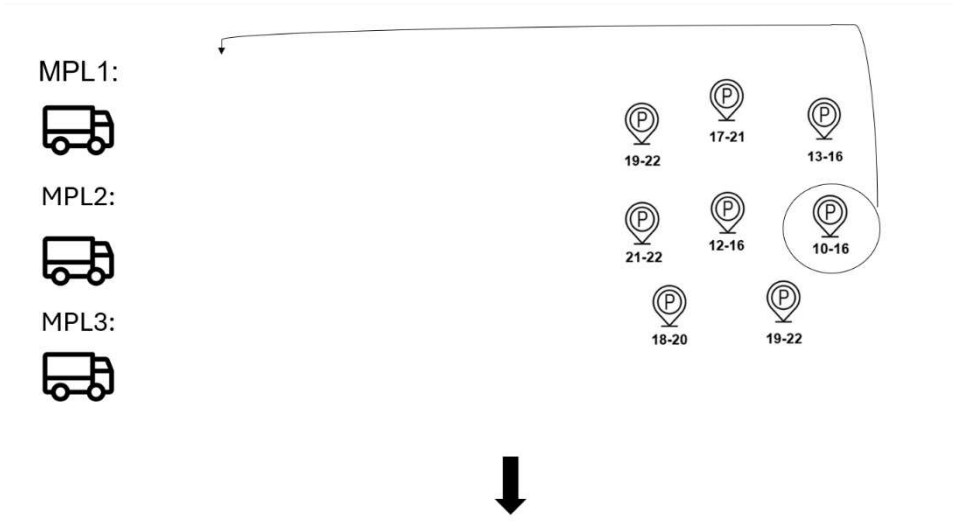
```
1: population_size = 150
2: population = []
3: for _ in range(population_size) do
4:     giant_tour = []
5:      all_possible_stopovers = get_all_possible_stopovers()  # Function
to get all possible stopovers
6:     while all_possible_stopovers do
7:         current_MPL = []
8:         unfeasible_stopovers = []
9:         while all_possible_stopovers do
10:                total_fitness = sum(stopover.fitness for stopover in
all_possible_stopovers)
11:            roulette_value = random.uniform(0, total_fitness)
12:            cumulative_fitness = 0
13:            selected_stopover = None
14:            for stopover in all_possible_stopovers do
15:                cumulative_fitness += stopover.fitness
16:                if cumulative_fitness >= roulette_value then
17:                    selected_stopover = stopover
18:                    break
19:            if is_feasible(selected_stopover, current_MPL) then
20:                insert_stopover(selected_stopover, current_MPL)
21:            else
22:                unfeasible_stopovers.append(selected_stopover)
23:            end if
24:            all_possible_stopovers.remove(selected_stopover)
```

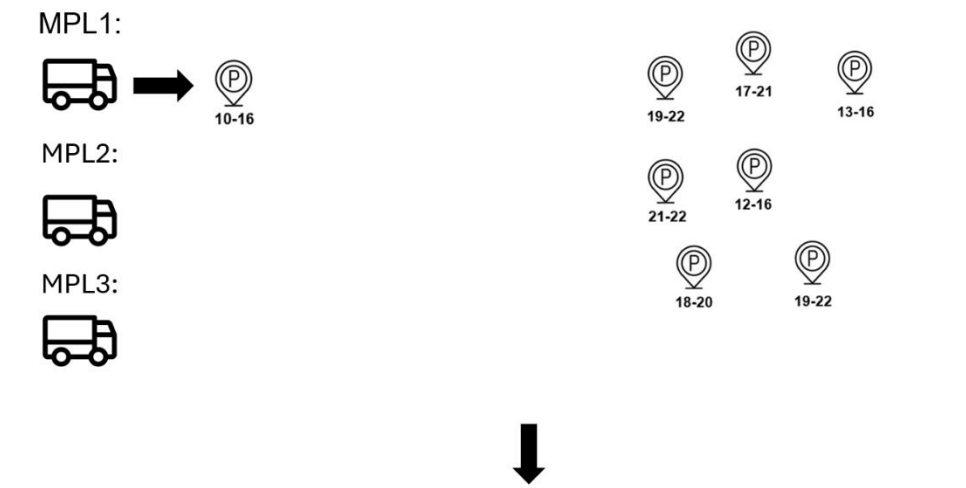
```
25:         end while
26:         for stopover in unfeasible_stopovers do
27:             stopover.adjust_time_window(12)
28:         end for
29:         all_possible_stopovers = unfeasible_stopovers[:]
30:         giant_tour.append(current_MPL)
31:     end while
32:     population.append(giant_tour)
33: end for
34: return population
```

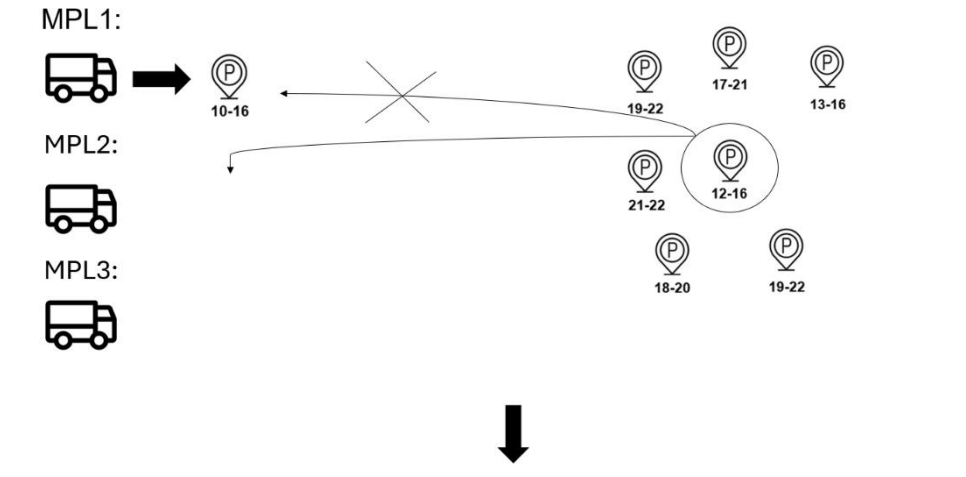
This is a different approach to the initialization which aims to generate better GA outputs by working with a better (in terms of number of served customers) initial population. Everything is orchestrated the same as in the random initialization with one exception: no random picking of stopovers but one based on fitness values. Each stopover from the list of all possible stopovers gets assigned a fitness value based on its number of served customers compared to all the other stopovers. After a stopover has been inserted into a giant tour, the fitness values of all stopovers from the list of all possible stopovers are newly calculated [10] to [18].

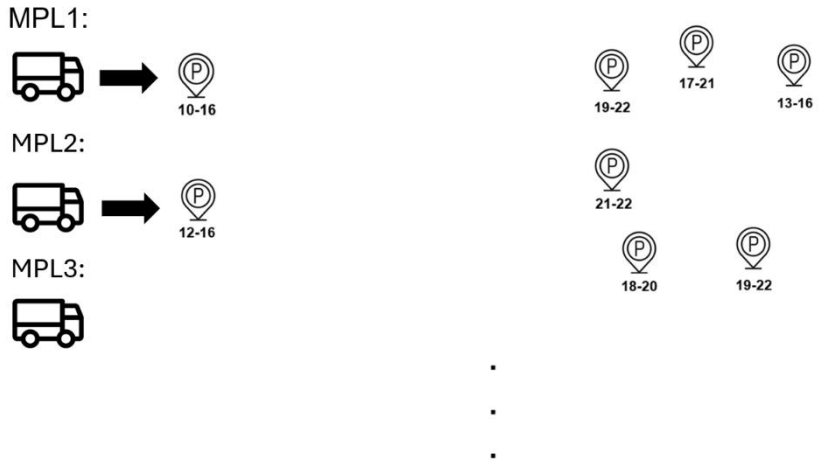
A visual representation of the initialization step:

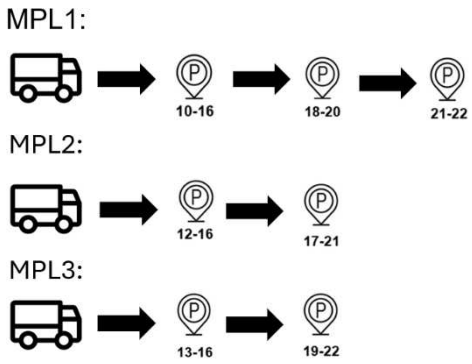



MPL1:
MPL2:
MPL3:
19-22
17-21
13-16
21-22
12-16
10-16
18-20
19-22


MPL1:
10-16
MPL2:
MPL3:
19-22
17-21
13-16
21-22
12-16
18-20
19-22


MPL1:
10-16
MPL2:
MPL3:
19-22
17-21
13-16
21-22
12-16
18-20
19-22





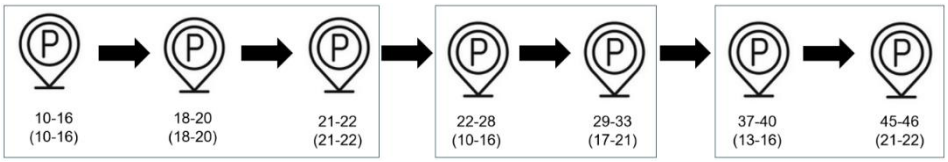


Figure 4: A visual representation of the initialization step

Selection

Next follows the search for two giant tours to perform the crossover operation on. The selection step is integral for the following crossover step. There are three different selection methods used in the experimental phase of this thesis. Two of them are based on a roulette wheel selection, the third one is based on random selection.

- Random Selection:

  A pseudocode representation of the random selection:

```
1: selected_parents = []
2: parent1 = SelectRandomGiantTour(population)
3: repeat
4:     parent2 = SelectRandomGiantTour(population)
5: until parent2 ≠ parent1
6: selected_parents.append(parent1)
7: selected_parents.append(parent2)
8: return selected_parents
```

  Firstly, a simple random selection is used, where two giant tours are selected completely randomly. The first giant tour is selected at random from the population [2]. The second giant tour is selected at random as well but cannot be the same as the first giant tour [3] to [5].

- Standard Fitness Selection:

  A pseudocode representation of the fitness selection:

```
1: selected_parents = []
2: fitness_values = []
3: cumulative_fitness = 0
4: cumulative_probabilities = []
5: total_customers_served = 0
6: for each giant_tour in population do
7:     served_customers = CalculateServedCustomers(giant_tour)
8:     fitness_values.append(served_customers)
9:     total_customers_served += served_customers
10:for i from 1 to size of fitness_values do
11:    fitness_values[i] /= total_customers_served
12:    cumulative_fitness += fitness_values[i]
13:    cumulative_probabilities.append(cumulative_fitness)
14:parent1            =            RouletteWheelSelect(population,
cumulative_probabilities)
15:repeat
16:                parent2    =    RouletteWheelSelect(population,
cumulative_probabilities)
17:until parent2 ≠ parent1
18:selected_parents.append(parent1)
19:selected_parents.append(parent2)
20:return selected_parents
```

  Here, two "good" giant tours are selected with the help of roulette wheel selection [14] to [16]. Each giant tour in the current population gets assigned a fitness value based on its number of served customers compared to all the other giant tours [6] to [13]. The higher the number of served customers of a giant tour is, the higher the fitness value of the giant tour is. The higher the fitness value of the giant tour is, the higher the chance that the giant tour is selected for crossover. The idea behind this is that two parent giant tours that are serving a lot of customers should produce good

or even better offspring giant tours. The two selected giant tours cannot be the same [17].

- Similar Customers Selection:

  A pseudocode representation of the similar customers selection:

```
1: selected_parents = []
2: parent1 = SelectRandomGiantTour(population)
3: fitness_values = []
4: shared_customers = []
5: for each giant_tour in population do
6:                                    shared_customers_count       =
CalculateSharedServedCustomers(parent1, giant_tour)
7:     shared_customers.append(shared_customers_count)
8: max_shared_customers = maximum value in shared_customers
9: for each shared_count in shared_customers do
10:    fitness_value = max_shared_customers - shared_count
11:    fitness_values.append(fitness_value)
12:repeat
13:    parent2 = RouletteWheelSelect(population, fitness_values)
14:until parent2 ≠ parent1
15:selected_parents.append(parent1)
16:selected_parents.append(parent2)
17:return selected_parents
```

  Here, a roulette wheel selection is performed again [13]. However, the fitness values are assigned differently. First, a random giant tour from the population is selected as the first parent [2]. Then, fitness values for all the remaining giant tours of the population are calculated by their "similarity" to the first parent [6] to [12]. They are considered similar if they serve a lot of the same customers that are already served in the first giant tour. Their fitness value is higher the lower the similarity is. By doing this, it could be possible better offspring giant tours can be produced because the overlap of served customers, which can get high by picking two already good giant tours, gets reduced and completely new solution areas of the solution space are explored.

Crossover

The crossover step is the most important step of any GA, as its function is to take already created new offspring solutions with features from both parent solutions.

A pseudocode representation of the crossover operator:

```
1: def perform_crossover(parent1, parent2):
2:     synchronous_pairs = find_synchronous_pairs(parent1, parent2)
3:
4:     if synchronous_pairs:
5:         selected_pair = random.choice(synchronous_pairs)
6:
7:         new_giant_tour1, new_giant_tour2 = swap_sequences(parent1,
parent2, selected_pair)
```

```
8:          StopoverUpdate(new_giant_tour1)
9:          StopoverUpdate(new_giant_tour2)
10:
11:         TimeWindowUpdate(new_giant_tour1)
12:         TimeWindowUpdate(new_giant_tour2)
13:
14:         if is_feasible_for_crossover(new_giant_tour1,
new_giant_tour2):
15:             replace_parent_giant_tours(new_giant_tour1,
new_giant_tour2)
16:     else:
17:         pass
18:
19: def is_feasible_for_crossover(new_giant_tour1, new_giant_tour2):
20:     if not is_drive_time_feasible(new_giant_tour1) or not
is_drive_time_feasible(new_giant_tour2):
21:         return False
22:
23:     if not is_capacity_feasible(new_giant_tour1) or not
is_capacity_feasible(new_giant_tour2):
24:         return False
25:
26:     return True
27:
28: def TimeWindowUpdate(selected_giant_tour):
29:     for stopover in selected_giant_tour.stopovers:
30:         stopover.time_window_start =
stopover.original_time_window_start + 12 * (stopover.mpl_number - 1)
31:         stopover.time_window_end = stopover.original_time_window_end
+ 12 * (stopover.mpl_number - 1)
32:
33: def is_drive_time_feasible(new_giant_tour):
34:     for mpl in new_giant_tour.mpls:
35:         for i in range(len(mpl.stopovers) - 1):
36:             stopover1 = mpl.stopovers[i]
37:             stopover2 = mpl.stopovers[i + 1]
38:
39:             distance =
calculate_euclidean_distance(stopover1.location, stopover2.location)
40:
41:             drive_time = distance / mpl.speed
42:
43:             if stopover2.time_window_start -
stopover1.time_window_end > drive_time:
44:                 return False
45:     return True
46:
47: def is_capacity_feasible(new_giant_tour):
48:     for mpl in new_giant_tour.mpls:
49:         if mpl.served_customers > mpl.capacity:
50:             return False
51:     return True
52:
53: def swap_sequences(parent1, parent2, selected_pair):
54:     stopover1, stopover2 = selected_pair
55:
56:     index_stopover1_parent1 = parent1.index(stopover1)
57:     index_stopover2_parent2 = parent2.index(stopover2)
58:
59:     sequence1_parent1 = parent1[:index_stopover1_parent1 + 1]
60:     sequence2_parent1 = parent1[index_stopover1_parent1 + 1:]
61:
```

```
62:     sequence1_parent2 = parent2[:index_stopover2_parent2 + 1]
63:     sequence2_parent2 = parent2[index_stopover2_parent2 + 1:]
64:
65:     new_giant_tour1 = sequence1_parent1 + sequence2_parent2
66:     new_giant_tour2 = sequence1_parent2 + sequence2_parent1
67:
68:     return new_giant_tour1, new_giant_tour2
69:
70: def replace_parent_giant_tours(new_giant_tour1, new_giant_tour2,
population):
71:     index_parent1 = population.index(parent1)
72:     index_parent2 = population.index(parent2)
73:
74:     population[index_parent1] = new_giant_tour1
75:     population[index_parent2] = new_giant_tour2
76:
77: def find_synchronous_pairs(parent1, parent2):
78:     synchronous_pairs = []
79:
80:     for stopover1 in parent1:
81:         for stopover2 in parent2:
82:             if are_synchronous(stopover1, stopover2):
83:                 synchronous_pairs.append((stopover1, stopover2))
84:
85:     return synchronous_pairs
86:
87: def are_synchronous(stopover1, stopover2):
88:     return stopover1.time_window_start == stopover2.time_window_start
and stopover1.time_window_end == stopover2.time_window_end
```

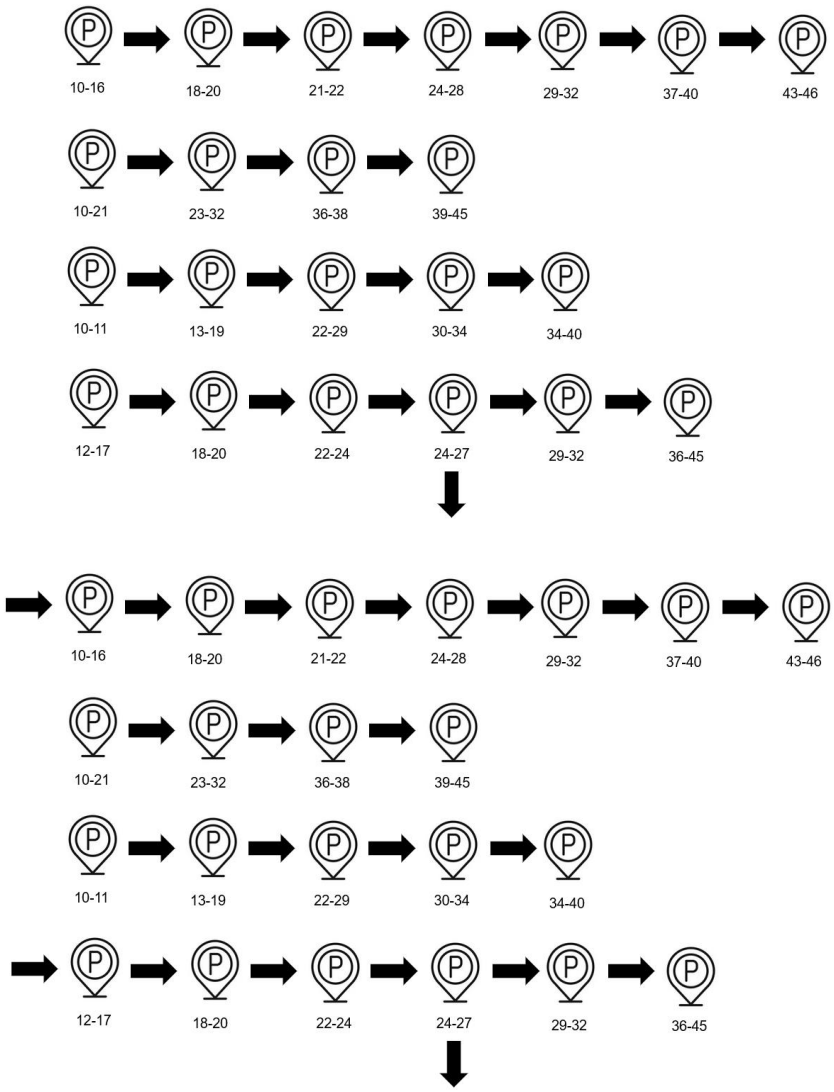
In this step, the two in the crossover selection step selected giant tours are now checked for synchronous pairs of stopovers (at this point the adapted time window notation from the initialization step takes effect) [77] to [88]. If two or more synchronous stopovers occur between the two selected giant tours, one of these synchronous stopover pairs is randomly selected as a so-called "point of crossover".

From this point on, the remaining sequence of the giant tour is swapped with that of the paired partner, resulting in two completely new giant tours that replace the two worst giant tours of the whole population [53] to [69].

However, a point of crossover can only be considered as such, if after swapping the dissected parts of the giant tours, the stopovers at exactly the point of crossover of either giant tour (i.e., the synchronous stopovers) can still be reached before their time windows open (considering the driving time from the preceding stopovers), and if the maximum capacity of every MPL of either giant tour is still not violated. If one or both of these feasibility conditions are not fulfilled, the corresponding synchronous pair is no possible candidate for becoming the point of crossover [19] to [52]. Before this feasibility check occurs, both offspring giant tours are updated with the stopover update operator [8] to [9].

If a crossover operation happened, the resulting two offspring giant tours would replace the parent giant tours in the population [70] to [76].

A visual representation of the selection + crossover step:



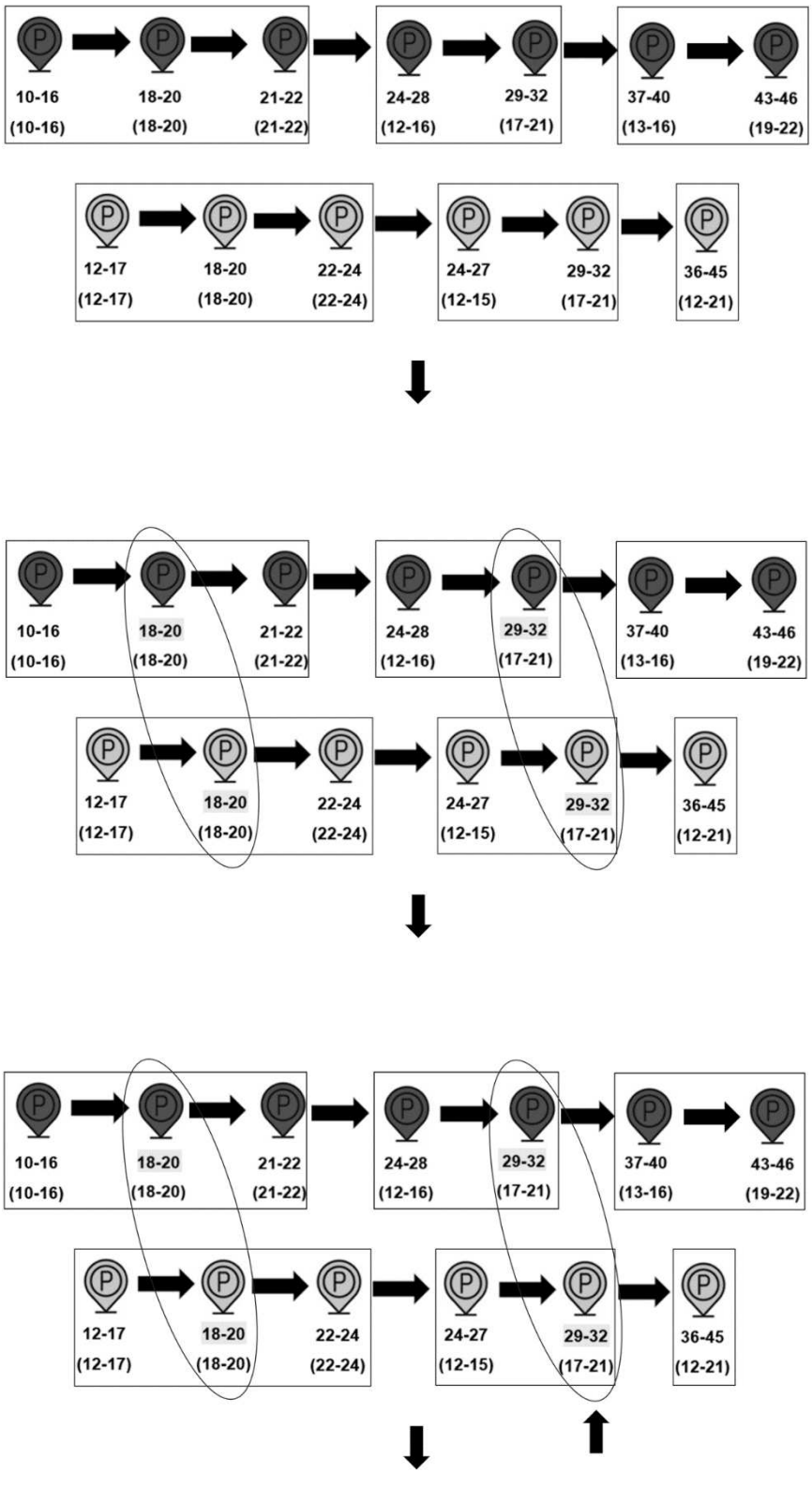

10-16
(10-16)
18-20
(18-20)
21-22
(21-22)
24-28
(12-16)
29-32
(17-21)
37-40
(13-16)
43-46
(19-22)
12-17
(12-17)
18-20
(18-20)
22-24
(22-24)
24-27
(12-15)
29-32
(17-21)
36-45
(12-21)

Figure 5: A visual representation of the selection + crossover step

Mutation

In the final step of each GA iteration, three mutation operators are introduced. By performing mutation operations on either certain solutions from the population or on the population itself, new areas of the solution space can be explored that would likely not be accessible otherwise. These mutation operators are classic examples of the mutation operators of a GA and were mainly inspired by Bouly et al. (2008). With a certain probability, a randomly selected giant tour of the population is subjected to these operators:

- Swap operator:

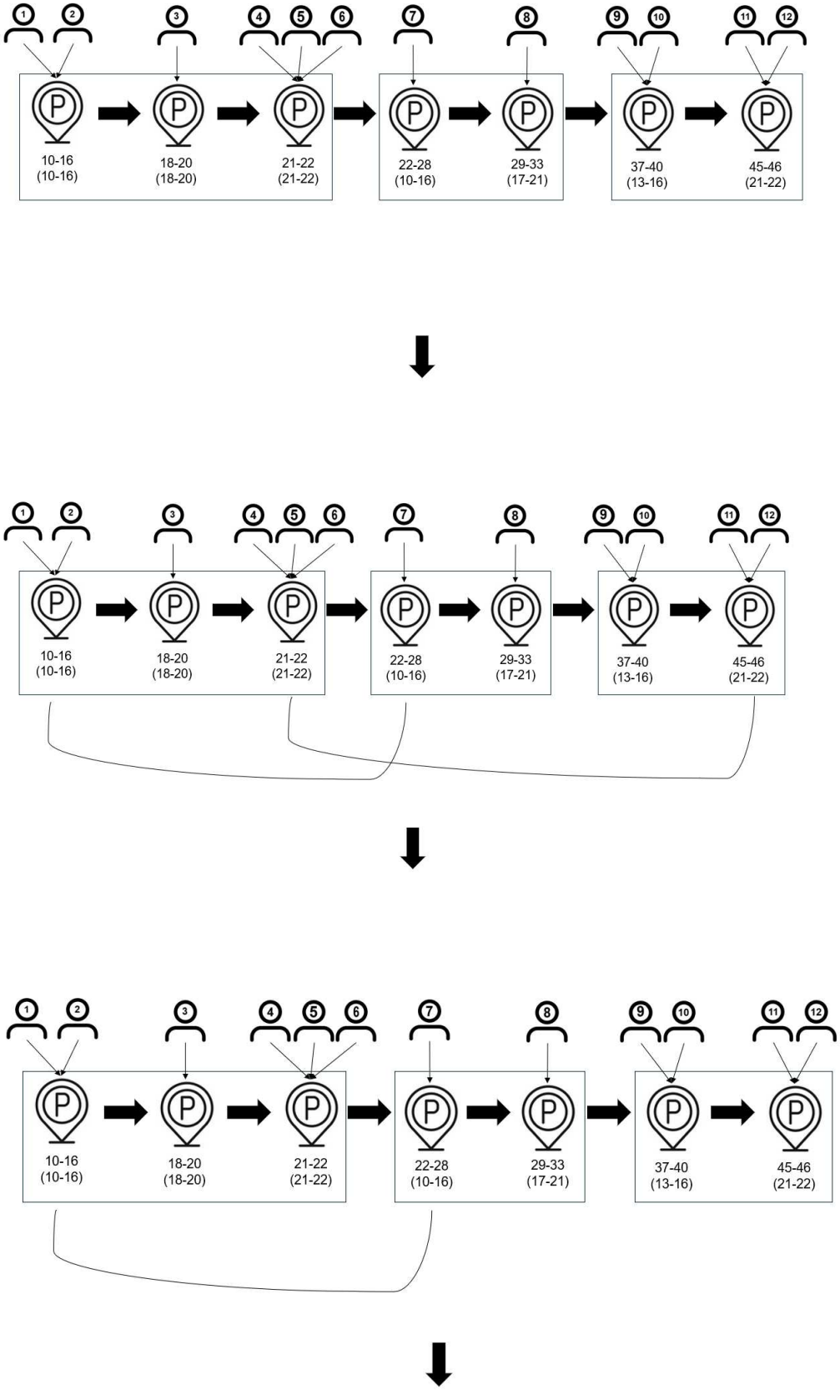
  A pseudocode representation of the swap operator:

```
1: def swap_operator(selected_giant_tour):
2:     stopover_to_swap =
random.choice(selected_giant_tour.stopovers)
3:
4:     stopovers_with_same_time_windows = []
5:     for stopover in selected_giant_tour.stopovers:
6:         if stopover.original_time_window.start ==
stopover_to_swap.original_time_window.start and
stopover.original_time_window.end ==
stopover_to_swap.original_time_window.end:
7:             stopovers_with_same_time_windows.append(stopover)
8:
9:     if len(stopovers_with_same_time_windows) == 0:
10:        return
11:
12:    feasible_swapping_options = []
13:    for stopover_to_swap_with in
stopovers_with_same_time_windows:
14:        StopoverUpdate(selected_giant_tour)
15:        TimeWindowUpdate(selected_giant_tour)
16:
17:        if is_feasible_for_swap(selected_giant_tour,
stopover_to_swap, stopover_to_swap_with):
18:
feasible_swapping_options.append(stopover_to_swap_with)
```

```
19:
20:         if len(feasible_swapping_options) == 0:
21:             return
22:
23:         stopover_to_swap_with =
random.choice(feasible_swapping_options)
24:
25:         index1 =
selected_giant_tour.stopovers.index(stopover_to_swap)
26:         index2 =
selected_giant_tour.stopovers.index(stopover_to_swap_with)
27:         selected_giant_tour.stopovers[index1],
selected_giant_tour.stopovers[index2] = stopover_to_swap_with,
stopover_to_swap
28:
29:         StopoverUpdate(selected_giant_tour)
30:         TimeWindowUpdate(selected_giant_tour)
31:
32:     return selected_giant_tour
33:
34: def TimeWindowUpdate(selected_giant_tour):
35:     for stopover in selected_giant_tour.stopovers:
36:         stopover.time_window_start =
stopover.original_time_window_start + 12 * (stopover.mpl_number -
1)
37:         stopover.time_window_end =
stopover.original_time_window_end + 12 * (stopover.mpl_number - 1)
```

A stopover within the randomly selected giant tour is determined as a possible swapping stopover randomly [2]. Then stopovers with the exact same time windows within the same giant tour are being searched for (the time-window notation introduced above has to be reversed temporarily to allow a feasible swap) [4] to [8]. If there exists one or more stopovers with the exact same time window within the giant tour, the randomly selected stopover is being swapped with one of its time window twins, if the drive time does not get violated by doing so [12] to [27]. If there exists no stopover with the exact same time window within the giant tour, this mutation operator will not be performed [9] to [10] & [20] to [21]. Before the feasibility check as well as after the swapping has been performed, there occurs the stopover update for the giant tour [14] & [29]. Also, the time windows of all stopovers in the mutated giant tour are updated with the GA notation [34] to [37]. This is a shake-up operator whose intention it is to escape local optima; if a giant tour on which the swap operator has been performed on is selected for crossover, areas of the solution space might be explored that would otherwise not have been reachable.

A visual representation of the swap mutation operator:



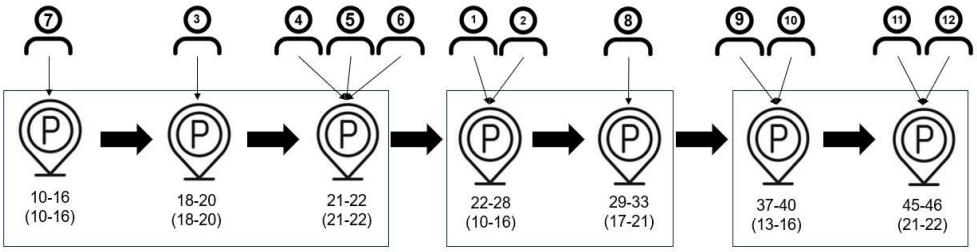


Figure 6: A visual representation of the swap operator

- Destruct-and-Repair operator:

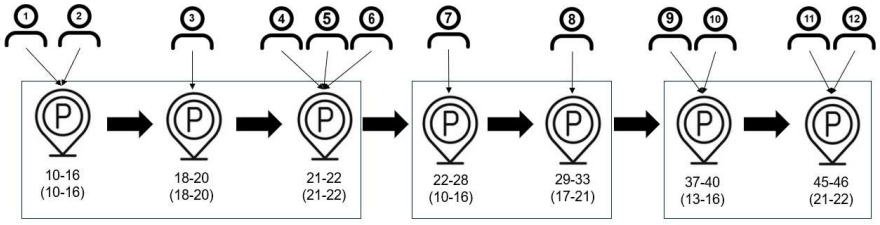
  A pseudocode representation of the destruct-and-repair operator:

```
1: def destruct_and_repair_operator(selected_giant_tour):
2:     stopover_to_delete =
random.choice(selected_giant_tour.stopovers)
3:
4:     selected_giant_tour.stopovers.remove(stopover_to_delete)
5:
6:     unused_stopovers = get_unused_stopovers()
7:
8:     feasible_insertion_points = []
9:     for unused_stopover in unused_stopovers:
10:
selected_giant_tour.stopovers.insert(selected_giant_tour.stopovers.
index(stopover_to_delete), unused_stopover)
11:
12:         StopoverUpdate(selected_giant_tour)
13:         TimeWindowUpdate(selected_giant_tour)
14:
15:         if is_drive_time_feasible(selected_giant_tour) and
is_capacity_feasible(selected_giant_tour):
16:             total_served_customers =
sum(len(stopover.served_customers) for stopover in
selected_giant_tour.stopovers)
17:             feasible_insertion_points.append((unused_stopover,
total_served_customers))
18:
19:         selected_giant_tour.stopovers.remove(unused_stopover)
20:
21:     feasible_insertion_points.sort
22:
23:    best_insertion_point = feasible_insertion_points[0][0] if
feasible_insertion_points else None
24:
25:     if best_insertion_point:
26:
selected_giant_tour.stopovers.insert(selected_giant_tour.stopovers.
index(stopover_to_delete), best_insertion_point)
```

```
27:
28:         StopoverUpdate(selected_giant_tour)
29:         TimeWindowUpdate(selected_giant_tour)
30:
31:     return selected_giant_tour
32
33: def TimeWindowUpdate(selected_giant_tour):
34:      for stopover in selected_giant_tour.stopovers:
35:          stopover.time_window_start =
stopover.original_time_window_start + 12 * (stopover.mpl_number -
1)
36:          stopover.time_window_end =
stopover.original_time_window_end + 12 * (stopover.mpl_number - 1)
```

A randomly selected stopover is removed from the randomly selected giant tour [2]. In its place, the unused stopover that fits the resulting gap, whose insertion would not lead to drive-time or capacity violations and that leads to the highest overall increase of the total number of served customers of the giant tour is inserted. This is done by trying to insert every unused stopover in the gap that results from the destruction step, performing a feasibility check on the adapted tour, sorting all viable insertion options, and then picking the best one (if there are any viable insertion options) [9] to [26]. Before the feasibility check is performed and again after the destruct and repair operation has been performed, there occurs the stopover update for the giant tour [12] & [28]. Also, the time windows of all stopovers in the mutated giant tour are updated with the GA notation [33] to [36]. This is a local search operator, which ensures a small improvement of the whole population in every iteration by improving one of the giant tours.

A visual representation of the destruct-and-repair operator:



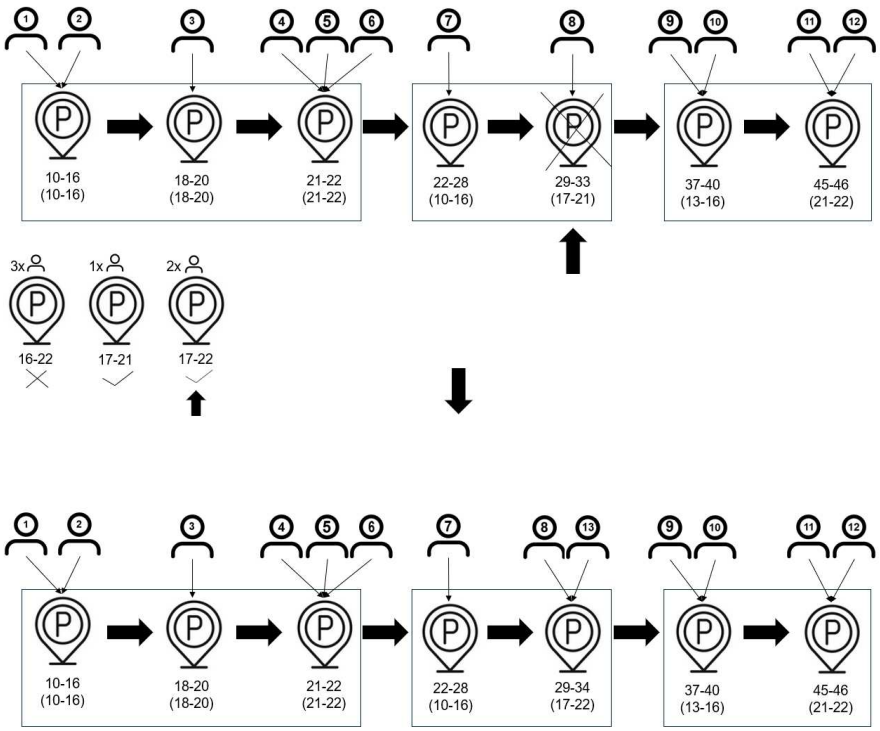


Figure 7: A visual representation of the destruct-and-repair operator

Furthermore, with a certain probability, the following mutation operator is being performed on the population:

- Repeated initialization operator (worst giant tour removed):
  A pseudocode representation of the repeated initialization operator (worst giant tour removed):

```
1: def repeated_initialization_operator(population):
2:     min_served_customers_tour =
find_giant_tour_with_min_served_customers(population)
3:
4:     population.remove(min_served_customers_tour)
5:
6:     new_giant_tour = RandomInitialization()
7:
8:     population.append(new_giant_tour)
9:
10:    return population
```

  A completely new giant tour is created in the same way as in the random initialization step (see page 16 for a detailed explanation) [6]. It replaces the giant tour with the lowest overall profit of the population [2] to [4]. This is, like the swap operator, also a shake-up operator; by initializing a new completely random giant tour and having

it replace the currently worst giant tour in the population new options for the crossover operation become possible and, therefore, different areas of the solution space can be explored.

- Repeated initialization operator (random giant tour removed):

  A pseudocode representation of the repeated initialization operator (random giant tour removed):

```
1: def repeated_initialization_operator(population):
2:     random_tour = random.choice(population.tours)
3:     population.remove(min_served_customers_tour)
4:
5:     new_giant_tour = RandomInitialization()
6:
7:     population.append(new_giant_tour)
8:
9:    return population
```

  A completely new giant tour is created in the same way as in the random initialization step (see page 16 for a detailed explanation) [5]. It replaces a random giant tour within the population [2] to [4].

Experimental Design

The GA presented in the methodology part was implemented using the C# programming language, making use of its versatility to develop an efficient solution for the MLLP. C# was chosen due to its common usage in scientific literature, as well as the availability of powerful libraries for numerical computation and parallel processing within the .NET framework.

The development of the GA was conducted using Microsoft Visual Studio, taking advantage of its variety of tools for code editing and debugging. The implementation was carried out on a desktop workstation equipped with an AMD Ryzen 5 2600 Six-Core Processor operating at a clock speed of 3.40 GHz, 32GB of RAM, and running the Windows 10 operating system.

Dataset Handling

The instances, on which the experiments of this thesis are being performed, are based on locations provided by Gehring and Homberger (1999). These instances consist of an area of 100 km² which is populated by a certain number of customer locations (coordinates, availability time windows, and maximum travel distance are included). These customer locations follow a so-called RC1 distribution which is a mixture of an R1 distribution (customer locations are randomly distributed) and a C1 distribution (customer locations are tightly clustered). For a visual representation of the instance with 200 customer locations, which has mostly been operated on for this thesis, see Figure 13 in the appendix.

Each instance also contains a certain number of stopover locations (with coordinates). The stopover locations always share their coordinates with the customer locations (meaning that if there were 200 stopover locations their coordinates would be identical with those of the 200 customer locations; if there were 50 stopover locations, their coordinates would be identical with 50 of the 200 customer locations). As already mentioned in the “Necessary Adjustment” section of this thesis, the driving speed of each MPL is considered constant at 30 km/h. This ensures that no drive time between two stopover locations exceeds one hour.

After an instance has been read into C#, all the possible stopovers (note the difference between a stopover, i.e., an object with coordinates, fixed halting times, and a certain number of customers that are served there, and a stopover location, i.e., an object with coordinates only) are generated. This is done by first creating a stopover with every possible time window combination possible for each stopover location (i.e., a stopover with time windows 10-11, 11-12, 11-13,…, 10-12, 11-13,…, 10-21, 11-22, 10-22 for each stopover location).

Then, it is checked which customers can be served at which stopover by means of the maximum travel distance as well as the availability time windows of each customer. All the

stopovers at which no customer can be served (i.e., no customer that could reach the stopover location can do so within the time windows of said stopover) are removed at this step, as they would not be able to contribute to a good solution. All the remaining viable stopovers are considered for the GA.

These stopovers are, as was explained in the Introduction, of a dynamic nature in relation to the customers that are served at them. Whenever a stopover is added to a giant tour, all the customers that are served at this stopover are checked for whether they are already served at an earlier point of the giant tour. If this is the case, these customers in question are removed from this occurrence of the stopover (only within the scope of this certain giant tour, the original version of the stopover remains unchanged). By doing this, it can happen that stopovers, at which not a single customer is served, end up in giant tours of the population.

Parameter Tuning

A multitude of initial experiments have been run to test the performance of the GA approach to the MLLP formulated within this thesis and to perform the necessary parameter tuning for further experiments. The objective of the first set of experiments is to find promising settings for the GA in which further experiments are performed; less promising settings are dismissed.

To ensure comparability throughout these experiments, a fixed population size (i.e., the number of solutions the GA works on at all times) and fixed runtimes for the GA are defined. The population size that has been chosen is 150 and the runtimes of GA iterations the initial experiments are being performed with are 10, 30, and 60 minutes.

The determined population size results from an influence of GAs found in the scientific literature as well as trial and error. Whenever the population size has been set too low or too high, the solution quality suffers, as there is either too small of a number of solutions to work within the population or too large of a number.

In general, the experimental process looks like this:

- Step 1: The initial population is generated by the means described in the methodology section.
- Step 2: The best giant tour of the initial population (=the giant tour that serves the most customers) as well as the average number of served customers in the initial population are printed.
- Step 3: The Genetic Algorithm is being performed for the duration of input runtime with the initial population as a baseline.

- Step 4: The best giant tour (=the giant tour that serves the most customers) that has been found during the whole GA process is printed together with the iteration in which this giant tour was found as well as the total number of GA iterations that have been performed during the runtime.

The initial experiments are performed on a medium-sized instance with 200 customers, 50 possible stopover locations, and 10 MPLs. Each MPL has a capacity of 20 (i.e., a maximum of 20 customers can be served by each MPL).

On the next page, you can see the optimal tour plan for this instance (50 stopover locations/10 MPLs/MPL capacity 20):

MPL	Stopover	X	Y	Time Window starts	Time Window ends	Customers served
MPL1	50	7,79	5,57	10	14	5
MPL1	34	1,21	3,07	15	22	11
MPL2	2	2,29	7,57	10	12	1
MPL2	44	1,07	0,64	13	15	1
MPL2	4	6,93	2,07	16	22	13
MPL3	5	6,57	1,57	10	14	7
MPL3	16	2,14	1,5	15	16	1
MPL3	1	6,14	2,64	17	19	2
MPL3	28	10	3	20	22	1
MPL4	45	4,29	6	10	14	6
MPL4	42	7,79	5,07	15	17	2
MPL4	36	2,79	0,93	18	22	8
MPL5	10	9,79	9,07	10	12	1
MPL5	13	4	9,93	13	14	1
MPL5	17	1,43	4,93	15	16	1
MPL5	2	2,29	7,57	17	22	13
MPL6	1	6,14	2,64	10	11	1
MPL6	20	7,5	0,14	12	13	1
MPL6	12	8,36	6,57	14	22	18
MPL7	30	8,57	1	10	11	1
MPL7	32	7,71	2,14	12	13	1
MPL7	29	4,43	4,14	14	22	15
MPL8	43	3,57	7,57	10	15	3
MPL8	5	6,57	1,57	16	17	1
MPL8	15	8,07	3,21	18	22	12
MPL9	3	8,79	8	10	22	19
MPL10	35	0,29	3,14	10	16	5
MPL10	37	2,57	9,79	17	18	1
MPL10	10	9,79	9,07	19	20	1
MPL10	1	6,14	2,64	21	22	1
	Served Customers Total					154

Table 2: Tour Plan of the optimal solution for 200 customers/50 stopover locations/10 MPLs/MPL capacity 20

Random Initialization Experiments

Experiments 1-4

Experiment	Initialization	Selection	Crossover Threshold	Destruct and Repair Mutation	Shake-up Mutations	Best Tour Original Population	Best Found Tour	Compared to Optimal Solution	Found in Iteration	Total Iterations
1	Random	Random	0	0%	0%	92 (60 on average)	117	75.97%	7,195,510	9,999,447
2	Random	Fitness	0	0%	0%	92 (60 on average)	116	75.32%	3,471,417	9,904,291
3	Random	Similarity	0	0%	0%	92 (60 on average)	114	74.03%	105,803	1,804,760
4	Random	Random	1.01	0%	0%	92 (60 on average)	100	64.94%	9,212	1,923,870

Table 3: GA Results for Experiments 1-4 after 60 minutes runtime

In the first set of experiments, the crossover operator is tested (see Table 3). The GA is initialized with a completely random population and no mutation operators are performed. The only difference within the first four experiments can be found in the selection process and crossover process respectively.

The results for the random selection and fitness value-based selection that the GA was able to find similarly good and the GA managed to achieve around the same number of iterations within 60 minutes for both those settings. The only difference is that the GA run with fitness value-based selection gets stuck in a local optimum after around one-third of total iterations whereas the random selection performs almost 4 million iterations more until it finds its returned solution.

Both these selection methods outperform the similar customers selection method. This method seems to slow the GA down considerably, as the GA run of Experiment 3 managed around 8 million iterations less within its predetermined runtime than with the first two settings. However, note that the crossover operator in general runs fast and the GA is able to iterate between 1.8 million and almost 10 million times when only performing the crossover operator. This number of iterations is not needed in this case, as the GA gets stuck in a local optimum earlier in its runtime.

The GA gets stuck in a local optimum even earlier when implementing a threshold during the crossover step as it was done in Experiment 4. Here, the crossover operator was only allowed to be performed when the offspring giant tours that were created in this step outperformed their parent giant tours (concerning the combined number of served customers) by 1%. Otherwise, no crossover would take place. Different threshold parameters were tested but all led to the same result of the GA getting stuck in a local optimum very early on in its run and therefore returning considerably worse results than

without an implemented crossover threshold. Thus, the implementation of such a threshold was no longer pursued in further experiments.

The most important takeaway from this set of experiments is the fact that the proposed GA can find good solutions when only performing the crossover operator (for example, the GA with the settings of Experiment 3 was able to come up with a solution that is only around 25% worse than the optimal solution in less than 5 minutes of runtime). There are not any targeted improvement operators needed for the GA to return significantly better solutions than those that were found after initialization; the crossover operator alone serves as a good improvement heuristic.

Mutation Experiments

Experiments 5-7

Experiment	Initialization	Selection	Destruct and Repair Mutation	Swap Mutation	Repeated Initialization Mutation	Best Tour Original Population	Best Found Tour	Compared to Optimal Solution	Found in Iteration	Total Iterations
5	Random	Random	0%	100%	100% (random)	92 (60 on average)	110	71.43%	32,365	174,341
6	Random	Random	0%	100%	100% (worst)	92 (60 on average)	125	81.17%	45,563	186,698
7	Random	Random	100%	100%	100% (worst)	92 (60 on average)	138	89.61%	73,789	179,883

Table 4: GA Results for Experiments 5-7 after 60 minutes runtime

In the next set of experiments, the mutation operators are added (see Table 4). This is done by setting up the GA in the same way as in the first set of experiments with random initialization and random selection for the crossover operator. Now, however, the different mutation operators are added to the settings, and different combinations of them are tested. To ensure compatibility, all mutations are performed with a probability of 100% in each iteration of the GA.

At first, only the two shake-up mutation operators (the swap mutation and the repeated initialization mutation) are tested. There are two variations of the repeated initialization mutation operator: One, where the newly initialized giant tour is replacing a random giant tour of the population, and one, where the newly initialized giant tour is replacing the worst tour of the population. The first variant, which was used on Experiment 5, did not show any promising results and was dismissed for further experiments for this reason. The second variant, which was used on Experiment 6, however, was able to drastically improve the results of the GA in comparison to the crossover-only experiments.

An important fact to note is that the promising results of Experiment 6 originate from a combination of the shake-up operators and the crossover operator. There has been an additional experiment performed where the crossover operator was turned off and only the

shake-up operators were performed during the GA run. The returned giant tour in this experiment was only able to serve 107 customers and thus performed far worse than the one that was returned in Experiment 6, where said combination of shake-up operators and crossover operators was used. This indicates that the GA is able to make use of its synergies of different operators to find good solutions for the MLLP.

In Experiment 7, the destruct and repair mutation is added. With the addition of this local search operator, even better results were found than in the crossover-only and the crossover + shake-up operator experiments. This, again, speaks for the synergy effect that is created by the combination of all operators of the proposed GA.

However, it needs to be noted that the inclusion of the mutation operators leads to a drastic decrease of iterations performed. The GA was only able to perform around 1.8% as many iterations within an hour in Experiment 7 in comparison to Experiment 1.

Further Mutation Experiments

Experiments 8 & 9

Experiment	Initialization	Selection	Destruct and Repair Mutation	Swap Mutation	Repeated Initialization Mutation	Best Tour Original Population	Best Found Tour	Compared to Optimal Solution	Found in Iteration	Total Iterations
8	Random	Similarity	100%	100%	100% (worst)	92 (60 on average)	135	87.66%	24,195	153,655
9	Random	Fitness	100%	100%	100% (worst)	92 (60 on average)	140	90.91 %	33,566	183,875

Table 5: GA Results for Experiments 8 & 9 after 60 minutes runtime

In Experiments 8 and 9, the same settings as in Experiment 7 are used with the exception of the selection method (see Table 5). The similar customer selection and the fitness value-based selection method are each combined with the mutation operators. Whereas the usage of the fitness-value based selection method led to an increase in solution quality, the similar customers selection method produced worse results than the random selection method. The number of total performed iterations of Experiments 7 and 9 are roughly the same. The similar customer selection method still slows the GA down, but not to such a drastic degree as in Experiment 3. This can be explained by the fact that the total number of iterations is drastically smaller than in the first set of experiments, and the selection step does not happen that often anymore. It still slows the GA down but not to the same degree as the mutation operators do, thus its effect on the GA speed is less noticeable.

Fitness Value-Based Initialization Experiments

Experiments 10 & 11

Experiment	Initialization	Selection	Destruct and Repair Mutation	Swap Mutation	Repeated Initialization Mutation	Best Tour Original Population	Best Found Tour	Compared to Optimal Solution	Found in Iteration	Total Iterations
10	Fitness	Random	0%	0%	0%	106 (84 on average)	131	85.06%	933,285	1,009,621
11	Fitness	Fitness	100%	100%	100% (worst)	106 (84 on average)	136	88.31%	59,676	184,005

Table 6: GA Results for Experiments 10 & 11 after 60 minutes runtime

In the final set of the initial experiments, fitness value-based initialization is tested (see Table 6). One reason for this kind of initialization is to check whether the process of purely random crossovering still can come up with good results when it is operating on an improved population. The other reason was to explore the outcome of a better initial population on the solution quality.

To find answers to the first question, whether the crossover operator still improves the solution quality when working with a better initial population, the same settings that were used in Experiment 1 are used again for Experiment 10 with the one exception that the fitness value-based initialization method which was discussed in the methodology part of this thesis is used instead of the random initialization method. This resulted in a significantly better best giant tour of the initial population (106 in Experiments 10 and 11 compared to 92 in earlier experiments) as well as a significant increase of the average number of served customers per giant tour of the initial population (84 in Experiments 10 and 11 compared to 60 in earlier experiments). Although the GA was working with a significantly better initial population in Experiment 6, it was still able to find drastic improvements by only performing the crossover operator, without any targeted improvement operators.

However, to answer the second question, the fitness value-based initialization was not able to improve the overall solution quality when all the operators of the GA were activated. In fact, when the settings from Experiment 8, which are the most promising of the initial experiments, are applied again on an improved initial solution, the GA returns worse results than when the same settings were applied on the GA run that was working on a completely randomly initialized population as in Experiment 7.

Performance Testing

As the GA does not seem to be able to find optimal solutions with the instances that were operated on in the initial experiments, there arises the question of whether the GA can find optimal solutions at all. And, if this is not the case, how close to the optimal solution the GA can get.

Experiments with Smaller Instances

To answer the first question, the GA was performed on a considerably smaller instance with 20 customers and 5 possible stopover locations. For these sets of experiments, the settings of experiment 10 were used.

Two experiments were conducted on this smaller instance. In the first one, an MPL count of 2 and an MPL capacity of 10 were used, in the second one, an MPL count of 4 and an MPL capacity of 5 were used.

In both experiments, the GA was able to come up with the optimal solution within tenths of a second. This proves that the GA in general can produce the optimal solution for the MLLP but its performance is limited by instance size.

Experiments on 20 customers, 5 stopover locations instance

MPL Count	MPL Capacity	Optimal Solution	Solution Found by GA	GA time elapsed
2	10	13	13	0.1 sec.
4	5	14	14	0.02 sec.

Table 7: GA Results for smaller instances

Advanced Experiments

The findings of the initial experiments were used to find good settings for advanced experiments that go more in-depth. In particular, the settings of experiments 7 and 9 were applied for the first set of advanced experiments as they showed the most promising results. Each advanced experiment was performed ten times with ten different fixed random seeds.

It is important to note that in the advanced experiments, there is another mutation operator performed which thus far has not been discussed: The zero-customer stopover removal operator. This operator is only used once during the GA run, namely at the very end. The returned best-found giant tour of the whole GA run is checked for stopovers that serve zero

customers. These stopovers then are removed, and, in their place, stopovers are inserted that, similarly to the workings of the destruct and repair mutation operator, improve the total number of served customers of the entire giant tour the most.

The idea behind this mutation operator is that stopovers that serve zero customers do not serve any purpose in reality. Because of the dynamic nature of stopovers in this interpretation of the MLLP, the occurrence of stopovers with zero customers served during the GA run is quite common. However, as stopovers that do not serve any customers can become stopovers that serve a multitude of customers by performing a simple crossover operation on the giant tour, they are part of, the removal of zero customer stopovers does not make sense during the GA run; if "bad" stopovers would be replaced by "better" stopovers during a GA run, the danger of getting stuck in local optima increases drastically.

There have also been some experiments performed with this operator that include removing stopovers that serve either zero or one customer. However, these experiments did not show promising results. As one can see from the optimal solution to the instance these experiments are performed on (see page 10), stopovers that only serve one customer are very common and are needed for a good solution.

These are the results for the instance with 200 customers, 50 possible stopover locations, and 10 MPLs:

Experiments on 200 customer, 50 stopover locations instance

(MPL count 10, MPL capacity 20)

Settings of experiment	Optimal Solution	Solver runtime	Avg. GA solution	Best GA solution	Iteration found in	Total GA iterations
7	154	~37 sec.	137.6	140	86,658	187,081
9	154	~37 sec.	137.9	143	95,769	173,767

Table 8: Collected Results of 10 GA runs (60 minutes runtime per run) for Experiment settings 7 (Random Initialization, Rand Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation (worst) all activated) and 9 (Random Initialization, Customer Similarity Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation (worst) all activated) on 200 customers, 50 stopover location instance. The solver runtime indicates the amount of time needed for an exact solver to find the optimal solution. For detailed results see Appendix.

As the settings of Experiment 9 proved to be the most promising once again, they were used for the following advanced experiments which are performed on bigger instances:

Experiments on 400 customers, 100/400 stopover locations instance

(MPL count 10, MPL capacity 40)

Settings of experiment	Stopover locations	Optimal Solution	Solver runtime	Avg. GA solution	Best GA solution	Iteration found in	Total GA iterations
9	100	276	~152 sec.	248.5	254	26,407	49,922
9*	400	282	~103 min.	230.5	241	43,021	53,810

Table 9: Collected Results of 10 GA runs (60 minutes runtime per run) for Experiment setting 9 (Random Initialization, Customer Similarity Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation (worst) all activated) on 400 customer, 400/100 stopover location instance. The solver runtime indicates the amount of time needed for an exact solver to find the optimal solution. For detailed results see Appendix.

To ensure comparability between the advanced experiments, it has been decided that the mutation operators of the GA runs on the biggest instance (400 customers, 400 stopover locations, and a vehicle capacity of 40) are each only triggered with a probability of 10%. Otherwise, the GA would have only been able to perform around one-tenth of its current iterations for this setting and thus produce distinctly worse results.

For the instance consisting of 400 customers, 100 stopover locations, and a vehicle capacity of 40, a runtime deceleration can be observed. Within an hour, the GA was only able to complete 30% of iterations with the settings of Experiment 9 compared to the smaller instance. An even more drastic deterioration in runtime occurred when the GA was run on the instance consisting of 400 customers, 400 stopover locations, and a vehicle capacity of 40. Here, the GA was only able to run for around 6,000 iterations with the settings of Experiment 9 within an hour which (for comparison, the GA iterated circa 150,000 times within an hour with the same settings when it was performed on the instance with 200 customers, 50 stopover locations and a vehicle capacity of 20).

This effect can be attributed to the increased complexity the GA has to face when running on bigger instances.

Overview of Advanced Experiments

There have been four advanced experiments performed. Two of them were performed on the smaller instance with 200 customers and 50 possible stopover locations; the other two on the larger instances with 400 customers and 100 and 400 possible stopover locations respectively. Two different experimental settings were used in these advanced experiments, which proved to be promising in the initial experiment. Each advanced experiment has been performed ten times with ten different random seeds on each instance.

The four advanced experiments in detail:

- **Advanced Experiment 1:** 200 customers, 50 stopover locations, 10 MPLs, 20 MPL capacity, random initialization, random selection, 100% Destruct and Repair Mutation, 100% Swap Mutation, 100% Repeated Initialization Mutation (worst giant tour removed)
- **Advanced Experiment 2:** 200 customers, 50 stopover locations, 10 MPLs, 20 MPL capacity, random initialization, customer similarity selection, 100% Destruct and Repair Mutation, 100% Swap Mutation, 100% Repeated Initialization Mutation (worst giant tour removed)
- **Advanced Experiment 3:** 400 customers, 100 stopover locations, 10 MPLs, 20 MPL capacity, random initialization, customer similarity selection, 100% Destruct and Repair Mutation, 100% Swap Mutation, 100% Repeated Initialization Mutation (worst giant tour removed)
- **Advanced Experiment 4:** 400 customers, 400 stopover locations, 10 MPLs, 20 MPL capacity, random initialization, customer similarity selection, 10% Destruct and Repair Mutation, 10% Swap Mutation, 10% Repeated Initialization Mutation (worst giant tour removed)

The best solution of these ten runs as well as the average solution are compared to the optimal solution of each instance and are illustrated in the following bar charts. Furthermore, the iteration history of each of the GA runs where a best solution was found is printed below:

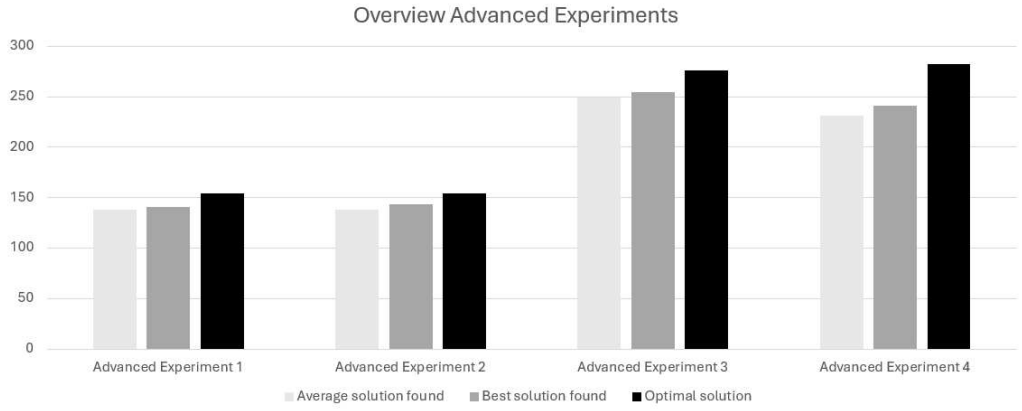


Figure 8: Overview of results of all advanced experiments (in terms of served customers)

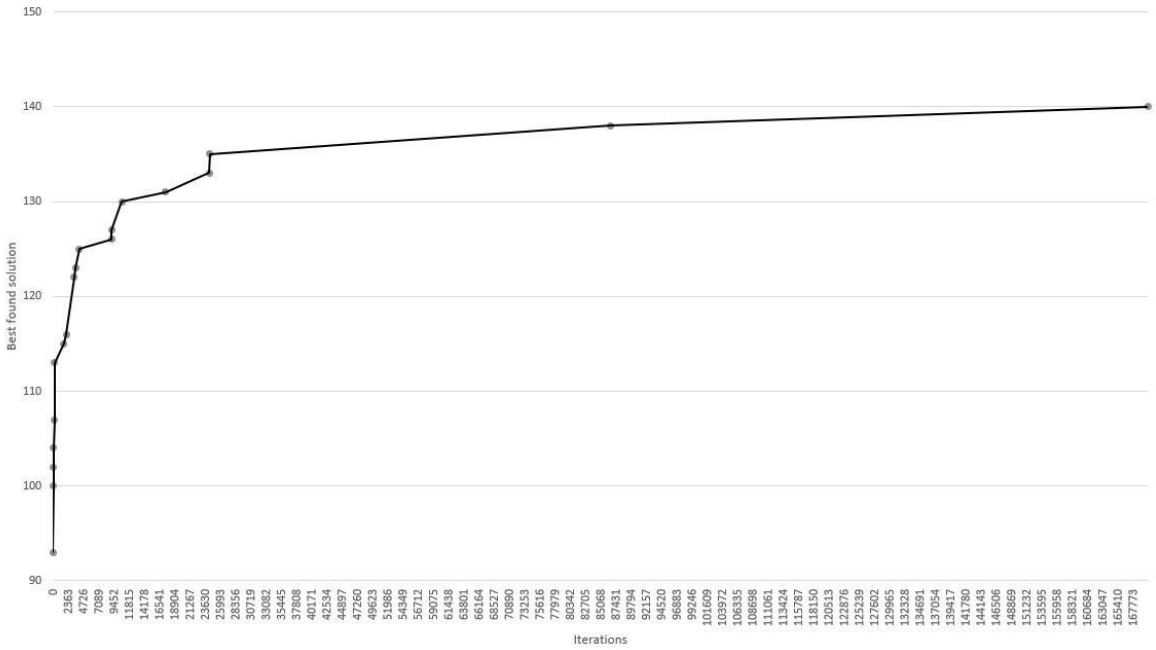


Figure 9: Iteration history of advanced experiment 1

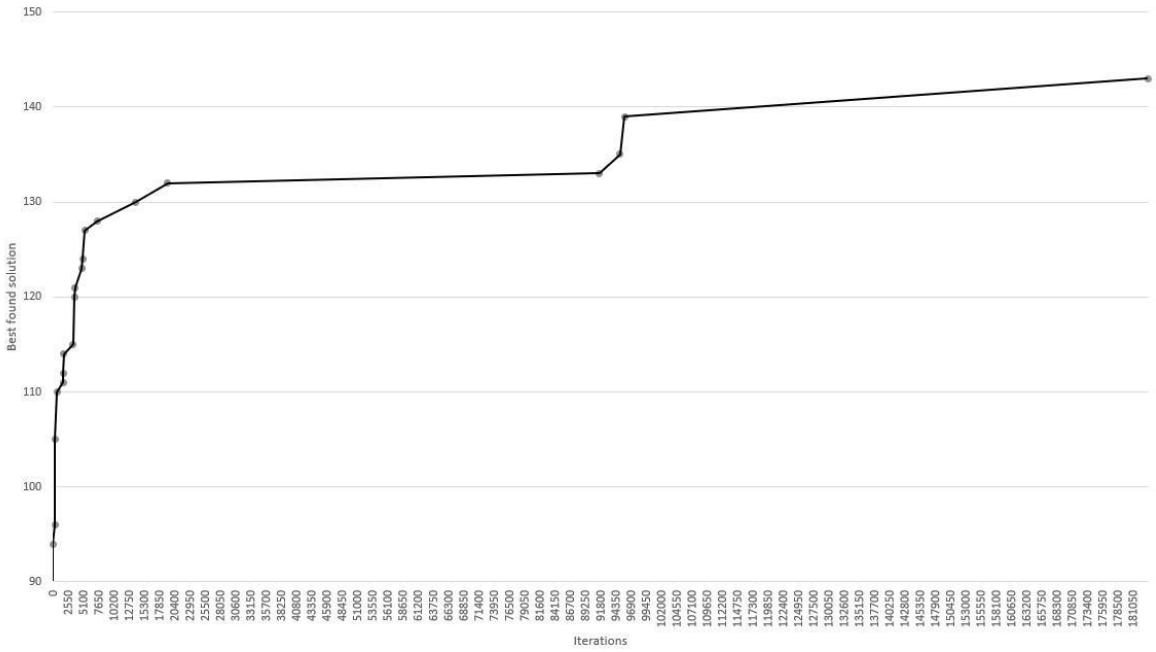


Figure 10: Iteration history of advanced experiment 2

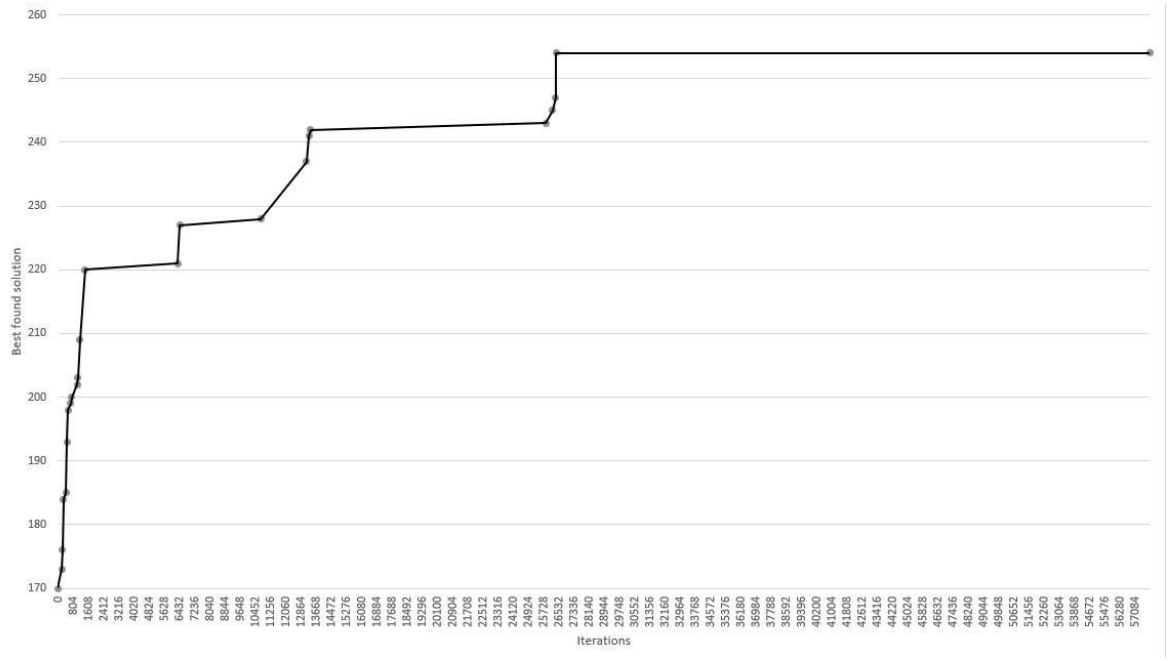


Figure 11: Iteration history of advanced experiment 3

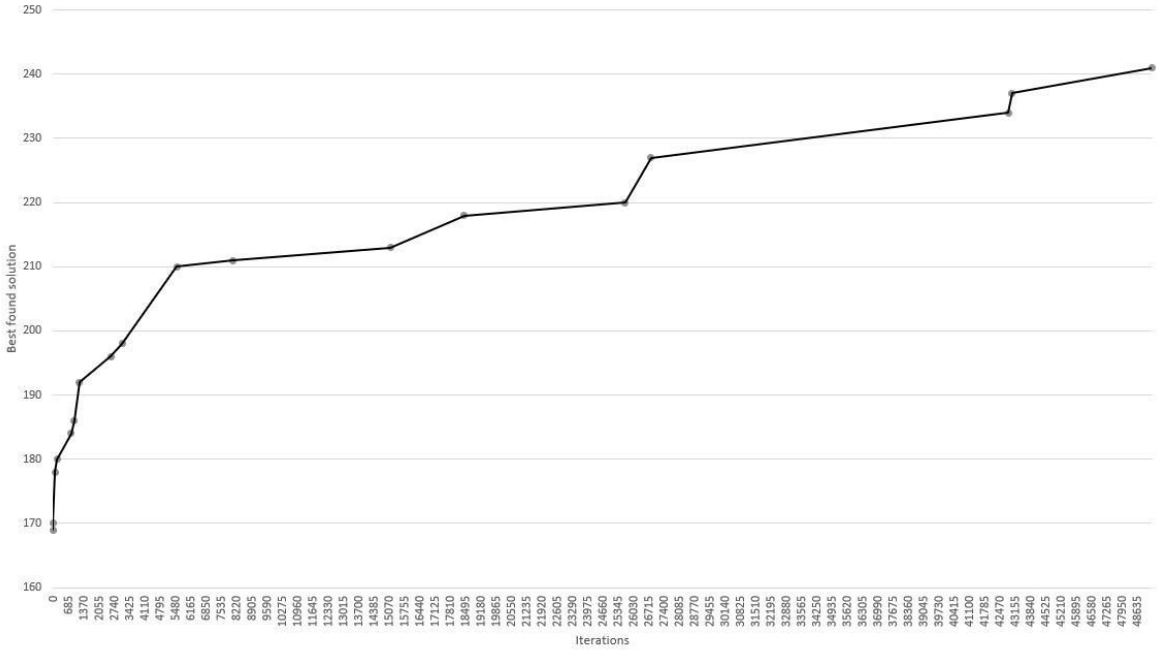


Figure 12: Iteration history of advanced experiment 4

These iteration histories paint an accurate picture of the behavior of the GA when faced with different extents of complexity. When the complexity is low, the GA finds a new best solution more quickly but gets stuck in a local optimum early. In the more complex experiments, it does not seem to get stuck in a local optimum at all, but its solution quality seems to be limited by the runtime of the GA. Also, note the sudden jump in solution quality at the very end of iteration histories 1,2 and 4. This can be attributed to the

removal of zero customer stopovers which happens once the ordinary GA run has been finished. This added operator seems to improve the solution quality in most cases.

Conclusion

The goal of this thesis was to introduce a novel Genetic Algorithm (GA) approach to the Mobile Locker Location Problem (MLLP). The idea behind developing this alternative approach to this rather recent approach has been explained in the Theoretical Background section of this thesis. Next, in the Methodology part, the structure of the GA and the functioning of its operators have been detailed. Different variants of the proposed GA have been tested in the Experimental Design part of this thesis, and the insights gained from this were used in the performance testing section to thoroughly examine the capabilities of the proposed GA.

Strengths and Weaknesses

First and foremost, the most important strength of the GA proposed in this thesis is the fact that it is working. This GA was constructed from scratch and without any external help. It is able to produce optimal solutions for the MLLP for small instances within a short time frame and still manages to produce good results within a reasonable time frame for larger and therefore more complex instances.

The GA profits from the interaction of its different operators. The more operators are added, the better the results of the GA become. There can be observed synergies between different operators which indicates that the proposed GA is efficiently designed and might build the basis for further research into GA approaches to the MLLP.

One of the main strengths of the proposed GA is its ability to come up with reasonable solutions even for very large instances within minutes, whereas the exact solver struggles to find any solution until it hits a certain runtime. For the largest instance that has been experimented with (400 customers, 400 possible stopover locations), the exact solver did not come up with any solution for the first 103 minutes of its runtime.

However, the proposed GA also struggles with the increased complexity of larger instances. The number of completed GA iterations within a predetermined runtime decreases exponentially with an increase in instance size. With the same settings, the GA manages around three times as many iterations within one hour with the instance consisting of 200 customers and 50 possible stopover locations as an input compared to the instance consisting of 400 customers and 100 possible stopovers and around 25 times as many iterations compared to the instance consisting of 400 customers and 400 possible stopovers (for this reason, the mutation probability is reduced to 10% in this set of experiments).

This can be attributed mostly to the increase in possible operations the GA has to take into account at every step of its run. For example, the destruct-and-repair mutation operator is one of the biggest contributors to an increase in solution quality but also the operator that

slows the GA down the most when instance sizes increase because of the increased number of repair options that have to be tested out.

Conclusion and Outlook

In this thesis, a new Genetic Algorithm approach to the MLLP has been introduced. The proposed GA was tested on different instances and the results of the GA were compared to those of an exact solver both in terms of solution quality as well as runtime efficiency. While the GA was not able to compete with the solution quality of the exact solver, it was still able to return tour plans with satisfactory results.

In terms of runtime efficiency, the GA was able to outperform the exact solver when larger instances were operated on, possibly making it a valuable approach for logistics providers facing the challenges of last-mile distribution. Furthermore, the interaction of various operators within the GA suggests that the algorithm is well-designed, with synergies that could be interesting for future researchers in this field.

However, as the size of the problem instance grows, the GA's performance tends to worsen. The number of completed iterations decreases exponentially with larger instances due to the increased complexity of mutation operators, which require more computational resources and time to evaluate potential solutions. While the GA can produce reasonable solutions quickly, it often cannot compete with exact solvers in terms of solution quality. This discrepancy becomes more obvious with larger instances, where the GA may return solutions that are significantly worse than those of the exact solver.

This thesis could be the basis for further research into GA approaches to the MLLP. At its core, this GA would be able to come up with optimal solutions for the MLLP, as it was proven in the experiments on smaller instances, but its performance is limited by the complexity of the MLLP and the increasing number of needed computational activity in each step when it is working on bigger instances. The operators of the proposed GA seem to be worth exploring further as they showed promising synergies within each other.

Appendix

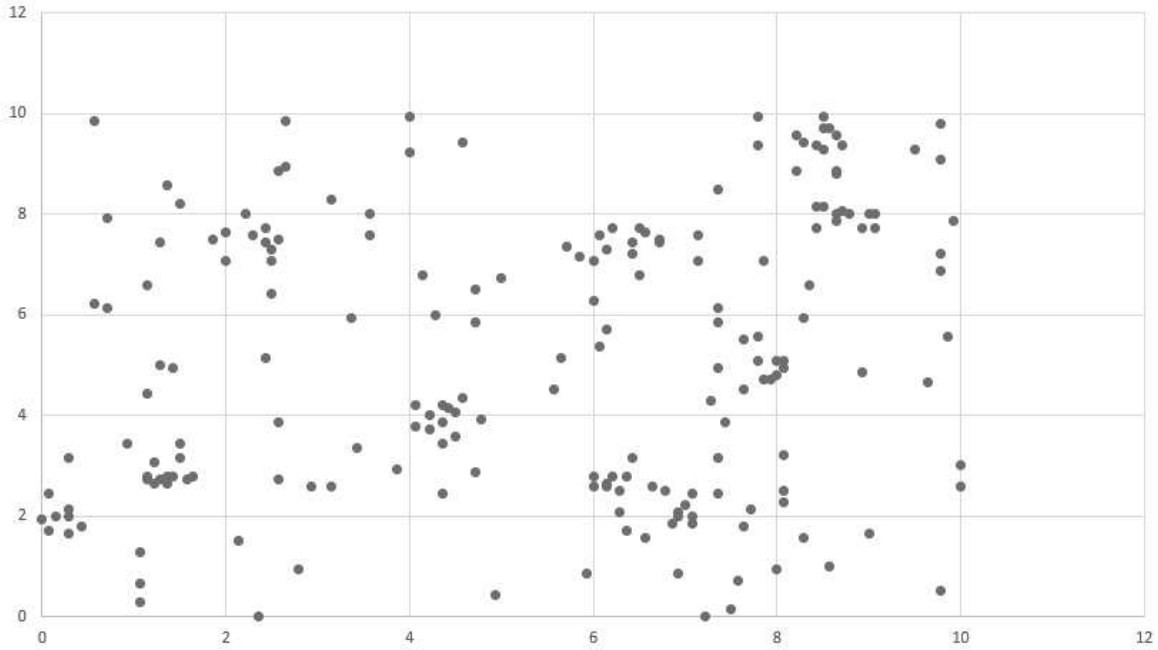


Figure 13: Customer location distribution for 200 customer locations following Gehring & Homberger (1999)

Accompanying table to table 7

Random Seed	Experiment 7 (200 customers, 50 stopover locations)	Experiment 9 (200 customers, 50 stopover locations)
777	138	141
239	139	138
726	137	135
120	139	137
956	139	140
610	135	143
141	134	136
948	139	136
127	140	137
601	136	136

Table 10: Detailed m Results of 10 GA runs (60 minutes runtime per run) for experiment settings 7 (Random Initialization, Rand Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation (worst) all activated) and 9 (Random Initialization, Customer Similarity Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation all activated) on 200 customers, 50 stopover location instance.

Accompanying table to table 8

Random Seed	Experiment 9 (400 customers, 100 stopover locations)	Experiment 9 (400 customers, 400 stopover locations)
777	245	221
239	254	241
726	246	227
120	248	231
956	250	231
610	247	230
141	254	234
948	249	229
127	248	234
601	244	227

Table 11: Detailed Results of 10 GA runs (60 minutes runtime per run) for experiment setting 9 (Random Initialization, Customer Similarity Selection and Destruct and Repair Mutation, Swap Mutation and Repeated Initialization Mutation (worst) all activated) on 400 customers, 400/100 stopover location instance.

References

Bouly, H., Dang, D. C., & Moukrim, A. (2008). *A memetic algorithm for the team orienteering problem.* A. M. Mora, P. P. Chen, P. J. Fleming, & G. Nicosia (Eds.), Applications of Evolutionary Computing (pp. 649-658). Springer Berlin Heidelberg.

Chao, I., Golden, B. L., & Wasil, E. A. (1996). *The team orienteering problem.* European Journal of Operational Research, 88, pp. 464–474.

Deutsch, Y., & Golany, B. (2018). *A parcel locker network as a solution to the logistics last mile problem*. International Journal of Production Research, 56. pp. 251-261

Gehring, H., & Homberger, J. (1999). *A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows.* Proceedings of EUROGEN99, K. Miettinen, M. Mäkelä, and J. Toivanen (eds.), University of Jyväskylä, Jyväskylä, Finland, pp. 57–64.

Gevaers, R., Van de Voorde, E., Vanelslander, T. (2011). *Characteristics and typology of last-mile logistics from an innovation perspective in an urban context. City distribution and urban freight transport, multiple perspectives.* Edward Elgar Publishing, Cheltenham, pp. 56–71

Gevaers, R., Van de Voorde, E. & Vanelslander, T. (2009). *Technical and process innovations in green logistics: opportunities, barriers and best practices by using case studies.* C. Macharis (Ed.), Proceedings of the BIVEC-GIBET Transport Research Day, Brussels: VUBPress., pp. 227-243

Jiang, L., Dhiaf, M., Dong, J., Liang, C., Zhao, S. (2019). *A traveling salesman problem with time windows for the last mile delivery in online shopping.* International Journal of Production Research.

Karbowska-Chilinska, J. & Zabielski, P. (2013). *Genetic Algorithm Solving the Orienteering Problem with Time Windows.* Journal of Telecommunications and Information Technology, (3), pp. 39-44.

Kötschau, R., Soeffker, N., & Ehmke, J. F. (2023). *Mobile parcel lockers with individual customer service*. Networks, 4. DOI: 10.1002/net.22173.

Lang, M. A. K., Cleophas, C., Ehmke, J. F. (2021). *Multi-criteria decision making in dynamic slotting for attended home deliveries.* Omega, 102. DOI: 10.1016/j.omega.2020.102305

Li, J., Ensafian, H., Bell, M.G.H., Geers, D.G., (2021). *Deploying autonomous mobile lockers in a two-echelon parcel operation.* Transp. Res. Part C: Em. Technol. 128, 103155. DOI: 10.1016/j.trc.2021.103155.

Liu, Y., Ye, Q., Escribano-Macias, J., Feng, Y., Candela, E., & Angeloudis, P. (2023). *Hybrid Quantum Metaheuristic for Last Mile Parcel Locker Configuration.* Transportation Research Part E. DOI: 10.1016/j.tre.2023.103234.

Manerba, D., Accorsi, R., Ferrari, E., & Zanoni, S. (2018). *Attended home delivery: reducing last-mile environmental impact by changing customer habits.* Procedia Manufacturing, 22, 558-565. DOI: 10.1016/j.promfg.2018.03.111.

Peppel, M., Spinler, S., & Winkenbach, M. (2024). *Integrating mobile parcel lockers into last-mile delivery networks: an operational design for home delivery, stationary, and mobile parcel lockers.* International Journal of Physical Distribution & Logistics Management, 54(4), 418–447. DOI: 10.1108/IJPDLM-01-2023-0055.

Schwerdfeger, S., & Boysen, N. (2020). *Optimizing the changing locations of mobile parcel lockers in last-mile distribution.* European Journal of Operational Research, 285(3), pp. 1077–1094.

Song, Z. (2021). *The geography of online shopping in China and its key drivers.* Geographical Analysis, 53(2), 231-252.

Statista. (2024a). *Worldwide retail e-commerce sales.* Retrieved from [https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/] (Accessed November 11, 2024).

Statista. (2024b). *E-commerce as percentage of total retail sales worldwide from 2015 to 2026.* Retrieved from [https://www.statista.com/statistics/534123/worldwide-retail-e-commerce-sales/] (Accessed November 11, 2024)

Vansteenwegen, P., & Gunawan, A. (2014). *Orienteering problems: Models and algorithms for vehicle routing problems with profits.* Society for Industrial and Applied Mathematics, pp. 56-66.

Wang, Y., Bi, M., Chen, Y. (2020). *A scheduling strategy of mobile parcel lockers for the last mile delivery problem.* Promet - Traffic&Transp. 32, 875–885. DOI: 10.7307/ptt.v32i6.3531.