Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Statistical Racing Crossover Based Genetic Algorithm for Vehicle Routing Problem

Scientific Students' Association Report

Author:

Ákos Holló-Szabó
Második Szerző

Advisor:

István Albert
dr. János Botzheim

2021

Contents

Kivonat

A genetikus algoritmusok moduláris metaheurisztikák, amik az evolúciós folyamatot szimulálják egy megoldás halmaz felett. Az optimalizálás nagyon rugalmas, de lassú, ami jelentősen nehezíti a kutatást a területen. A dolgozat terméke egy új variáns, ami más variánsokat versenyeztet, miközben statisztikai adatot gyűjt. Eredményeink azt mutatják, hogy az új variáns egy hatékony, önálló és még adaptívabb megoldás. Azok a variánsok, amik gyorsabban konvergálnak vezetik a versenyt, de elakadnak lokális minimumokban. Ezekben az esetekben a rugalmasabb kombinációk lassabb konvergenciával nagyobb futási valószínűségre tesznek szert és jobb megoldásokat találnak távolabb a lokális minimumtól. A hibrid gyorsabb konvergenciára képes minimális futási idő többlettel.

Abstract

Genetic algorithms are modular metaheuristics simulating the evolutionary process over a solution set. The optimization is very adaptive but slow, making statistical research difficult. An algorithm was designed where different variants are racing against each other while statistics are gathered. Our results show that this algorithm is an efficient, standalone, and even more adaptive solution. Those variants that result in faster convergence lead the race, but get stuck in local minima. In these cases, the more agile combinations with slower convergence gain higher probability and find better solutions farther from the local minimum. The hybrid is capable of faster convergence with minimal additional runtime.

Chapter 1

Introduction

Transport costs are a constant load on the economy, and present in the sales of all physical products. Goods are transported between producers, distributors, and consumers. Reducing the cost leaves more profit at the producers and distributors and leaves more money at the consumers. More money also means customers can buy more products, and their money is worth more. Therefore reducing transport costs produces wealth from the economy as a whole.

These days nonphysical products are common, but the need for transportation increases. Physical products have a shorter lifespan, and people can buy more as technology becomes cheaper. The current pandemic almost doubled the traffic of online shopping, which kept increasing in the last decades. Since people prefer comfortable solutions and its easier to collect data through the internet, this tendency probably remains.

Consumer satisfaction might be another goal. Quality management became customer focused and smooth delivery increases the experience of quality of shopping as a service. Customers may return for the higher quality even if prices are higher. It is easier to schedule delivery and arrive on time if transports are shorter. Constraints given by customers are easier to build into iterative optimizations, and the efficiency decrease caused by said constraints can be measured. Transport can be optimized to satisfy customer constraints with minimal profit loss.

Optimal transportation has environmentally friendly effects too. Transportation is responsible for a significant share of the CO2 emissions, which can be reduced by curbing fuel consumption. Electronic vehicles are slowly gaining popularity, but it makes the production of accumulators necessary. In-warehouse delivery is mostly electric already and usage of accumulators can be avoided. That's why we focus on cross-storage transportation, where fossil fuels are dominant still.

A real-life transport has several agents and parameters like the drivers, vehicles, locations, products, and packaging. Our goal is to plan the optimal transport from a center position for multiple transport units, packages, and goal positions. The Vehicle Routing Problem (VRP) is the most accurate model of transports with the right extensions. This is an NP-hard problem because it is Karp reducible to the Travelling Salesman Problem (TSP), which is NP-complete. We have to rely on heuristics and iterative optimizations to find near-optimal solutions.

Genetic algorithms are a very popular and well-documented group of metaheuristics [9]. They are proven to be efficient for TSP and can be adapted for VRP too. The main advantage of genetic algorithms is modularity. The phases of the iteration of the opti-

mization are fully separable. Every phase has several variants with significantly different properties. This grows the field of possible combinations and makes selection very difficult. Automation of this selection process is necessary.

During the research, our biggest challenge was the runtime of the genetic algorithm. Even if it is still smaller with orders of magnitude than the runtime of a CNN (Convolutional Neural Network), it would have taken years to test each combination. In this paper, an algorithm is proposed where the variants of a phase are racing against each other. In each cycle, a variant is chosen randomly, and statistics are gathered to measure its efficiency. During later cycles, a higher probability is insured for more efficient combinations.

In the following chapters, I will introduce the algorithm, alternatives to it, and discuss even some implementation details. In chapter 2 I will introduce the basic terms and math that are necessary to understand the algorithm, design decisions, and context. In chapter 3 I will introduce the mathematical problem and its variants. In chapter 4 I will introduce the most known heuristics to the problem as alternatives and modular extensions of the algorithm. The algorithm will be introduced in chapter 5 with all modules and design decisions. The crossover itself will be introduced in a separate chapter 6. I will detail the implementation and used technologies in chapter 7. Finally, I will present some measurements in chapter 8 and draw conclusions in chapter 9.

Chapter 2

Terms and Background

2.1 Graphs

Any set can be interpreted as vertices. Edges are nothing else but the pairing of these vertices. Every graph consists of a set of vertices and a set of edges interpreted on them. We mark this as G(V, E), where G is the graph, V is the set of vertices and E is the set of edges. Parallel edges are edges pairing the same two vertices and self-loops are edges pairing a vertex with itself. Usually if not mentioned, parallel edges and self-loops are not allowed. The order of vertices in a pairing does not matter by default, otherwise, we call the graph directed. In this case, edges are parallel only if the order of vertices is the same in the pairing too. A graph is called complete if all possible, allowed pairing exists in E. When we store data in the edges or vertices, we represent it with a function that assigns this data to the edges or vertices. In most cases all vertex stores data with the same data structure, same for edges.

Graphs are widely used in science to represent networks, hierarchical structures, associations, workflow, etc. We are going to use it to store the properties of customers and routes between them.

A walk is a sequence of vertices and edges of a graph. An open walk is a walk that's first and last vertices are different. A trail is an open walk without any edges repeated. A Cycle is a trail where the first and last vertices are the same. For example, a walk can be any random movement on a road network from conjunction to conjunction, while a minimal length route between two locations on the road network is a trail. We use cycles to represent a transport starting and ending in the transport center. An Euler cycle of G(V, E) graph is a cycle on G that contains all $e \in E$ exactly once. A Hamiltonian cycle of G is a cycle containing all $v \in V$ exactly once except its endpoint that is included twice. Both are used in heuristics (see chapter 4) for TSP. Euler cycles are more important for the Christofides algorithm.

2.2 Big O notation

When we implement a program one of its most important attributes is its resource requirement. The requirement of storage space, CPU runtime limits the practical usage of said algorithm. The problem is that the requirements depend a lot on the computer the algorithm is running on, the exact implementation, and the input to process. That makes more abstract measurements necessary. Ordo defines an upper limit of an algo-

rithm depending on the length of its parameters. If f(n) is O(g(n)), that means for every environment a c constant exists that $f(n) \leq c \cdot g(n)$.

If an algorithm with a parameter N with n length requires $O(n^2)$ runtime, that means for every environment the best implementation's CPU runtime requirement will be below $c \cdot n^2$, where c is a constant specific to that environment. So, the set of algorithms with $O(n^3)$ runtime includes the set of algorithms with $O(n^2)$ runtime. We usually use the multiplication of the next function types: 1, $log(n)$, n^x, x^n, $n!$, n^n, where $x \in \mathbb{Z}$. These function families are also increasing in said order. So $O(n^{x_1}) < (x_2^n)$ is true for any $x_1, x_2 \in \mathbb{Z}$. For better clarity we also mean by O(g(n)) that there is no better O constructed from the previously mentioned functions multiplication. So if an algorithm has $O(n^3)$ runtime then it does not have an $O(n^2)$ runtime. Of course, if we specify the environment and the parameter better then the algorithm can have a better O specific to that environment and parameter. For example, some algorithms might have better O on a quantum computer.

The most used algorithms are O(n), sorting an array takes $O(n \cdot log(n))$ (Merge sort). Since the complete graph has $O(|V|^2)$ edges, most graph-based algorithms take $O(n^2)$ runtime. If an algorithm has $O(n^3)$ runtime requirement, we already find it problematic. It means on twice the input length it might take eight times more time. Still more complex algorithms taking vertices as input and using graphs will exceed this limit. We have to restrict these algorithms' input length.

In practice, there is an inconsistency between the assumptions of storage space and the runtime requirements of an algorithm. Let there be an n long array of v_i values between zero and m maximal value, where $v_i, m \in \mathbb{N}$. Storing this array will take $O(n \cdot log(m))$ storage space, still we usually say that reading this array takes O(n) time. This is because we use primitive numerical types on computers, and we must rebuild the code to change these primitive types. This means that every implementation has a maximal value that it can handle, and a length of numerical variables are constants. The numbers length becomes relevant again only when the array is archived or loaded into memory. In the case of permutations, the size of the array and the maximal value is the same. That means we don't handle permutations longer than 2^{64}.

2.3 Permutations

Permutation and order are similar concepts, but not the same. A permutation is an order relative to another order. So if we have houses with different colors in this order: red, blue, green, yellow, then this is an order, but a permutation of them is an array of numbers like 3,1,4,2. In itself, 3,1,4,2 does not define the order but if we take red, blue, green, yellow as the base order then it means green, red, yellow, blue (where the numbers referred to the indexes of the houses in the original order). Still, we say "the permutation of something" interchangeably by order, but in that case, we always assume some basic order. A set with a size of n will always have n! possible arrangements, that's why most algorithms searching for some optimal order have unacceptable runtimes. 100! is more than 10^{100}, it is impossible to check this many variants one by one. A cycle on a graph can be defined by the set of vertices in the cycle in some order. That's why brute force TSP solutions have an O(c!) runtime requirement, where c is the number of customers.

2.3.1 Representations

The representation of a permutation is not trivial. Different representations can have different benefits. A goal can be to store the permutation in minimal storage space or to run the algorithm in minimal time or some balanced solution. On graphical cards, the usage of matrices could be beneficial. We present a simple example of a transport group for three transport units. The last transport unit has no customer to serve. All examples are representations of this transport group (Figs. 2.1, 2.2).

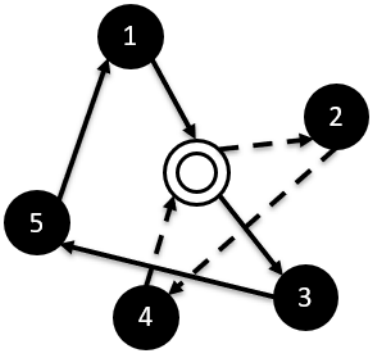

Figure 2.1: Example for an abstracted tour group

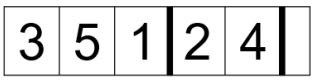

Figure 2.2: Example for an abstracted tour group as segmented permutation

- **Path representation [6] (Fig. 2.3):** The most natural representation is an array of the values in the right order. In the case of the n value, this means $O(n \cdot \log(n))$ storage space. A position can be found in $O(1)$ time but a value in $O(n)$ time. This may make the implementation of some operators difficult.

Figure 2.3: Example for path representation

- **Inverted representation (Fig. 2.4):** If it is beneficial to find the values fast an inverse representation can be used. Here the positions of the array correspond to the values and the stored value corresponds to the position in the order. This means $O(n \cdot \log(n))$ storage space. A position can be found in $O(n)$ time and a value in $O(n)$ time. In common cases, path representation is given but inverse permutation is calculated from it. This calculation takes O(n) time, so it does not affect runtime significantly.

Figure 2.4: Example for an inverted representation

- **Sequential representation [6] (Fig. 2.5):** In the case of transports, the position in order is not the direct cause of cost. The cost comes from the edges of the distance graph. Meaning the following value of a value is more important than its position. Some operations building on this fact can benefit from a sequential representation. In this representation, the position corresponds to the first value and the value corresponds to the second value of neighboring value pairs. The main disadvantage of this permutation is that it is hard to manipulate without breaking the cycle into multiple small ones. Also, n + 1 numbers are needed since one of them must mark the start of the permutation. Most operators are implemented for path representation and refactored later for runtime improvement. Conversion still takes O(n) time.

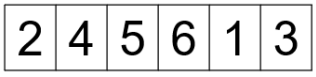

Figure 2.5: Example for sequential representation

- **ordinal representation [6] (Fig. 2.6):** The most efficient compression uses the natural representation. Fills the positions of the array in order and decreases all values by the number of lower values preceding it. The last value can be removed since it is always one. This uses an $O(\sum_{m=n}^{0} log(m))$ storage space and can be used on inverse and sequential representations too. Compression and depression take $O(n^2)$ time.

Figure 2.6: Example for ordinal representation

- **Permutation matrix [6] (Fig. 2.7):** The permutation matrices use $\mathbb{R}^{n \times n}$ binary values. In the case of an M matrix, $m_{i,j}$ is 1 if and only if the j^{th} position holds the i value. Multiplying a vector $v \in \mathbb{N}^n$ representing the values in order by a permutation matrix orders the values the right way. This can be used on graphical cards that are stronger in matrix operations. Just like the compression, inverse and sequential representations have matrix formats. In the case of the inverse representation, the permutation matrix just has to be transposed. Conversion between permutation matrix and path representation takes $O(n^2)$ time. This means the two representations are rarely used in the same algorithm.

0	0	1	0	0
0	0	0	1	0
1	0	0	0	0
0	0	0	0	1
0	1	0	0	0

Figure 2.7: Example for representation by permutation matrix

- **Precedition representation (figure 2.8):** The precedition matrix M[i,j] is 1 if i precedes j in the order. This can be used to minimize data loss with minimal storage space sacrifice. Also, conversion takes O(n) on GPU-s.

0	1	0	1	0
0	0	0	1	0
1	1	0	1	1
0	0	0	0	0
1	1	0	1	0

Figure 2.8: Example for representation by precedition matrix.

Distribution can be represented in several ways that can be combined with All order representations.

- **Multi-chromosome representation [1] (Fig. 2.9):** It uses t separate order representations for t transport units. It still has to be marked which client belongs into which transport. In the case of natural representation, using the indices of the customers serves the task. In other cases, a separate n long binary array is needed to mark the information for each customer. It makes implementation of new operations and management of transports transparent. It requires $O(n \cdot t)$ additional storage space and $O(t)$ pointer calculations when iterated. It is mostly used to transport data between representations.

3	5	1
2	4	

Figure 2.9: Example for multi-chromosome representation

- **Two chromosome representation (Fig. 2.10):** It consists of an order representation and an n long array. The array contains the index of the transport unit the

customer on the same index belongs to. This means $O(n \cdot log(t))$ additional storage space requirement. It's less transparent since the same transport groups can be represented in multiple ways. It is easy to handle distribution but sorting by transport unit index is necessary to handle order efficiently. Sorting takes $O(n \cdot log(n))$ time.

3	5	1	2	4
1	2	1	1	2

Figure 2.10: Example for two chromosome representation

- **Two-part chromosome representation [1] (Fig. 2.11):** It consists of an order representation and a t long array. Each value in the array represents the number of customers belonging to the transport unit on the same index. This representation requires $O(t \cdot log(n))$ additional storage space. It takes one pointer calculation to iterate making it faster. It is harder to implement some operations and it is hard to adapt operators used for TSP.

3	5	1	2	4	3	2	0

Figure 2.11: Example for two-part chromosome representation

- **One-chromosome representation [2] (Fig. 2.12):** It uses breakpoints inserted into the representation as it was one permutation with an $n+t$ length. Each breakpoint represents a return to the transport center and the transport unit the transport belongs to. It requires $O(t \cdot log(n+t))$ additional space, but it varies depending on the representation of the order. Most operations used for TSP can be easily adapted by using this representation.

-1	3	5	1	-2	2	4

Figure 2.12: Example for one-chromosome representation

Representations are most relevant in the crossover and mutation phases of genetic algorithms, where the transport groups are not just read but altered. Our focus was the crossover operations since they are more complex and take more runtime. A different representation should be preferred for each variant.

2.4 Logistiscs

Logistics is the branch of engineering working on the efficient storage, packaging, transportation, and scheduling of physical goods. There are sites containing storage facilities. Transportation is done between said facilities not touching factories, shops, etc. directly, only their storage. The shops and factories supply themselves from their nearby storages. This reduces packing time at the goal locations since the packaging is done in the storage system instead of during packing.

In the storage system, storage locations are organized into a regular grid. Racks are organized into lines where the space between two uprights of the rack is called a bay. One

bay is separated into levels and each level is organized into storage locations. One storage location holds multiple stocks of products. Vehicles in the storage move from storage location to storage location grouping and separating units of products. Every stock can be identified by a five number vector and finding the optimal rout between two locations takes no effort. If multiple locations must be touched in one tour usually the vehicle takes a snake like rout through the roads between the rack lines. Each road is one directional no time can be gained by playing with the order of locations in the transport. The only parameter to optimise is the distribution of tasks between vehicles. That's why we focus on inter storage transports instead.

When a transport vehicle arrives, the transport is already collected into the packing position packaged and ready to transport. This means the time of packing mostly depends on the volume of transport instead of its weight or type. Even if the transport is fragile it is packaged in ways making flawless stacking possible. The speed of packing is specific to the storage and measured in cubic meters per second. Storage systems have also a capacity for vehicles that can be loaded at the same time. If capacity is exceeded, every new vehicles have to wait until a slot is freed.

The cost of transportation comes from the fuel consumption and maintenance of vehicles and the salary of the driver. The driver's payment mostly depends on the time of the transportation while fuel consumption depends more on the length of the rout. The drivers also need to take breaks from time to time and vehicles need to be refueled.

Usually the delivery date of orders are set beforehand, so the transport group can be calculated for the exact they for the central storage. There are always some last minute orders so the transport plans must be adjusted manually. The vehicles start at the same time waiting for the packing of the first packages, and leaving together. After they return, they can take new tours until the work time of the driver is over. The goal is to minimize the length and time of the sum of transports while the drivers must work during their whole work time.

Chapter 3

Problem Statement

The Travelling Salesman Problem (TSP) is an NP-complete mathematical problem with a lot of literature. A salesman plans to visit its clients in minimal time and return to its original location. The distances (in time) between the clients are represented by an undirected, weighted, complete graph where the clients are the vertices. The algorithm must find a minimal weight Hamiltonian cycle on the given graph. The optimum will be represented by the optimal order of the clients.

c is the number of clients and G(E, V) is the graph where $G \in \mathbb{R}^{c \times c}$. There is also $C : e \mapsto t$ where $e \in E$ and t is in R, the weight. Weight calculation of a route takes O(c) time. The only exact solution is to iterate through all possible permutations, meaning c! iterations, each taking O(c) time.

There are multiple heuristics (see 4). Their runtime ranges from $O(c^2)$ to $O(c^6)$ where slower algorithms are advanced metaheuristics like Genetic algorithm and Ant Colony Optimization. Genetic algorithms are a subset of Evolutionary Algorithms and Ant Colony Optimization is a subset of swarm intelligence-based algorithms. Multiple other algorithms were published for the problem from both sets. Neural networks can be also used. For example, an Elastic Network was published in 2004 [5].

The Multiagent Travelling Salesman Problem (MTSP) is an extension of the TSP, which is Karp reducible to TSP making it NP-hard. The goal is to plan multiple trips from the same starting location, touching each client. There are multiple variants regarding the optimum. The optimum is either defined as the minimum sum of weights of trips or as the minimum sum of weights of the heaviest trip. The optimum will be represented as the distribution of customers between trips and their optimal order in each trip.

Let there be c as the number of clients and G(E, V) is the graph where $G \in \mathbb{R}^{c \times c}$. There is also $c : e \mapsto t$ where e is in E and t is in R, the weight. S is the set of salesmen and ci is the length of the trip of si where i is in N. Weight calculation of a route takes O(c) time. Brute force calculation would take O((c+s)!) time.

All algorithms for TSP can be adapted to MTSP by simply adding extra copies of the start location to the graph, signifying a return to the location, pretending they are customer vertices. Of course, the weight calculation function must be changed if the second variant of the optimum definition is used. Also, all algorithms for MTS can be used for TSP if |S| is one.

The Vehicle Routing Problem (VRP) is a group of mathematical problems with a lot of variances. The goal is to plan a minimal cost trip visiting each client with multiple vehicles, but the cost is not necessarily built into the distance graph. The complete graph

representing time, distance, and all properties of the routes between clients is directed and has parallel edges. Each edge from one customer to another represents the properties for each transport unit separately. For example, bigger vehicles might be unable to take some routes that a small, more agile vehicle can, and vehicles can have different maximal velocities. The cost might come from consumed fuel, the payment of the driver, the rent of equipment, etc. . Some vehicles might take multiple trips. Some customers might not be served if it is too costly. The goal is not to minimize the cost but to maximize the profits.

In our variant of the VRP, the orders are permanent, meaning that each customer must be served. The income is constant, so the goal is to minimize the cost. The cost is calculated from a complex simulation based on the order and distribution of the customers. The simulation has the properties of the transport units, packages, goal locations, routes redefined. The vehicles have capacity limits, and the drivers have a work time maximizing the length of trips. Invalid trips are allowed but not accepted as results and punished with additional costs for overload and overwork.

The optimum will be represented as the distribution of customers between trips and their optimal order in each trip. The cost is calculated by a complex simulation adjusted to the exact current task. Other attributes of the transport plan like the optimal start time, the optimal gate, necessary breaks, etc. are calculated during the simulation. Resulting in a long simulation runtime but still shorter optimization. This reduces VRP to an MTSP in representation, meaning multiple solutions used for MTSP can be applied.

Chapter 4

Heuristic Approaches

There are two groups of heuristics for trip planning: trip construction and trip optimization algorithms. Trip optimizations take an initial trip and use iterative methods to improve it. The initial trip might be random, a result of a trip construction algorithm or some other methods specific to that type of optimization.

4.1 Trip Construction Algorithms

- **Nearest Neighbour:** The Nearest Neighbour heuristic is a greedy trip construction algorithm. In the case of TSP, only one trip is created. Starts with the nearest location to the transport center and always selects the nearest to the last location that is not used yet. The result is usually far from optimal since the optimal trip might require the use of more costly edges near o the beginning of the trip. The algorithm is very fast, having an $O(c^2)$ runtime. In the case of TSP, multiple routes are generated. Still, the nearest location is added but there are many methods to determine which trip is the location added to.
  - The algorithm iterates through the trips, always adding the location to the next one. Generation takes $O(c^2)$ runtime.
  - The salesman with the shortest yet trip selects the not selected nearest location to its last location. Generation takes $O(c^2 \cdot t)$ runtime.
  - like b, but trips without any locations have length equal to the minimum of a trip to a location and back. This stops the algorithm from using all transport units, improving on b. Generation takes $O(c^3 \cdot t)$ runtime.
  - The location for each trip is only marked first. If the same location were marked for multiple locations the location belongs to the trip which has the closer last selected location. The locations not getting the location marks the next nearest location. Repeat until each location marks a separate location or there are no free locations. Add locations to trips. Generation takes $O(c^2 \cdot t)$
  - Like d but locations not getting the location on the first attempt does not get a location. Generation takes $O(c^2)$
  - like e but the distance of location is also multiplied by the length of the trip when determining which trip the marked location will belong to. Generation takes $O(c^2)$
  - One trip is generated every time. The transport center is always suggested as an unselected location. If the center is closer than any other location, the

trip is closed. If all salesmen had a trip but not selected locations to remain, the salesman with minimal cost trip continued selecting new locations, not the center.

  - Trips might be built from both ends always taking location nearest to both ends adding it to the nearest end. In the case of VRP, a simulation is run. Since the vehicles have the capacity and the goal is to use the minimal number of vehicles, one trip can be generated at a time. The actual trip is generated with all vehicles. Each vehicle continues until one of its capacities exceeds its limit. The version with the least cost per completed order is selected for the trip. The next trip is generated with the left vehicles. Its generation takes $O(c^2 \cdot t)$

- **Cost based probability:** The same as the nearest neighbor but the chosen location is not always the nearest. First, a weight is calculated from the reciprocal of the distance of the location. The choice is random and each location gains probability proportional to its weight. The heuristic may run multiple times taking the best result. The heuristic may run also with different metrics with convergence to the nearest neighbor heuristic, meaning 100

- **Greedy:** Sort the edges by cost. Always take the edge with the minimal cost unless it creates a not Hamiltonian cycle, remove it otherwise. This generation for TSP takes $O(c^2 \cdot log(c))$ because of the sorting. In the convergence remove first the transport center. Then find a Hamiltonian cycle on the rest of the graph. Remove the t most costly edges. Connect each subtour to the transport center with two edges. Generation still takes $O(c^2 \cdot log(c))$. In a more advanced approach: Let there be a and b customer locations in the Hamiltonian cycle. The cost of a return to the center instead of a->b is the cost of a to center plus center to b minus a to b. Collect edges with negative remove cost. Break the cycle by removing the t edges of the minimal and negative removal costs. If there isn't t then fewer trips will be generated. Generation still takes $O(c^2 \cdot log(c))$. In the case of VRP, the Hamiltonian cycle is generated the same. The center is added back, and all vehicles start the route with each location. The vehicle with the lowest cost per customer starts with the trip resulting in the trip with the best ratio. The vehicle returns if capacity is exceeded, and the next vehicle continues with the next location on the Hamiltonian cycle. Generation takes $O(t \cdot c^3)$ runtime at a minimum.

- **Insertion heuristic:** Starts with the shortest edge as a cycle of two locations. Add a not included location with the closest location to any of the included locations. Add the selected location into a location that causes minimal cost increase. Continue until all locations are added. Generation takes only $O(c^2)$ runtime. In the case of TSP and VRP, the same method can be used as in the case of the Greedy heuristic.

- **Advanced insertion heuristic:** Starts with the convex hull of the locations as a tour. Find the optimal insertion location of each location and choose the location with the least insertion cost and insert it. Continue until all locations are inserted. Generation takes $O(c^2 \cdot log(c))$ runtime. In the case of TSP and VRP, the same method can be used as in the case of the Greedy heuristic.

- **Christofides:** Find a minimal spanning tree on the graph. Duplicate all its edges. Find an Euler cycle, then remove duplicates from the cycle. Found reduced cycle will be a Hamiltonian cycle. Generation takes $O(c^2 \cdot log(c))$ runtime. Reducing the Euler to the Hamiltonian cycle takes O(c), but reduction can be started from any position. A better result can be found by trying all positions and taking the best

Hamiltonian cycle. In the case of TSP and VRP, the same method can be used as in the case of the Greedy heuristic.

- **Extended Christofides:** Find a minimal spanning tree on the graph. Take all vertices with odd degrees, and find minimum weight matching on the subgraph of the taken vertices. Merge two graphs and find the Euler cycle. Reduce the Euler like in Christofides. In the case of TSP and VRP, the same method can be used as in the case of the Greedy heuristic.

4.2 Trip OptimisationAlgorithms

- **2-Opt:** Take one Hamiltonian cycle. Iterate through all two edges of the cycle. Invert the order of vertices between the two edges. If the new cycle is better, keep it, if not, revert the last modification. Continue iteration and repeat until no two edges result in improvement. One iteration takes $O(c^2)$ runtime since all two edges are checked. Cost recalculation is not necessary since the cost difference comes only from the two removed and two added edges. Required runtime is more if the direction of the edges matter, the wrong data structure is used, or cost calculation is more complex. In the case of TSP and VRP, the same method can be used as in the case of the Greedy heuristic.

- **simplified 2-Opt:** Same as 2-Opt but every two vertices are checked, and they are switched. This variant requires less runtime if edge direction matters or array data structure is used, but more complex cost calculations still slow it significantly.

- **3-Opt:** Same as 2-Opt but three edges are used and the inversion of all three cycle segments are tested. Takes at least $O(c^3)$ runtime. A simplified version can be also used, like simplified 2-Opt.

- **k-Opt:** Same as 2-Opt but k edges are used and the inversion of all k cycle segments are tested. Takes at least $O(c \cdot k)$ runtime. Multiple k-Opt algorithms could be used together. For example, starts with k=2, k-Opt. If no more improvements are found then continue with an incremented k. If any improvement is found by higher than 2 k, decrements k. A simplified version can be also used, like simplified 2-Opt.

- **Lin-Kernighan [7]:** Same as adaptive k-Opt but a complex calculation is added to determine the correct k necessary to improve in each iteration. The calculation is quite complex, but in the case of TSP the found local optimum will be only two percent worse than the optimum itself. The necessary iteration count is unknown however and one iteration takes $O(c^{2.2})$ runtime. The algorithm also has worse results on VRP since the cost is graph independent and time dependent.

- **Tabu-Search:** 2-Opt is a neighborhood search meaning it takes elementary modifications of a permutation and checks permutations it can get by applying the modification on all possible positions. However, 2-Opt stops when it does not find any more possibilities for improvement. Tabu-search improves on this by letting changes with negative gain in these situations. The changes continue randomly until a better position is found. Since some modifications invert each other, the algorithm lists the modifications that invert earlier ones. For example, since switching two values twice results in their original order, the algorithm bans switching of already switched values. The usage of list could increase the time requirement to $O(c^3)$ per iteration

for simplified Opt-2 moves, but using clever implementation with $O(c^2)$ memory requirement this can be reduced to $O(c^2)$.

- **Simulated Annealing:** Simulated annealing is an improvement on neighborhood search like Tabu-search. However instead of using a list it uses probability. It allows moves with negative gain with a low probability and by adjusting this probability over time it can avoid local minimums. Since it does not have a stop condition, it can run for any time. The results of more advanced versions are comparable to those of Lin-Kernighan.

- **Banch & Bound:** It is a tour construction algorithm. It iterates all possible permutations searching for the optimum, but in an almost optimal order. As it builds the tour it considers each customer as the next one and associates cost with them and also a graph from the original one with reducing weights and removing used vertices. Choose the customer with the lower cost, then that customer's graph will be the new graph. If the route is constructed the cost of the associated cost of the next customer will be exactly the cost of the tour. It checks for considered subtours of the past eliminating subtours with already higher cost. If a subtour with lower cost is found then the search continues with that subtour. If there is no subtour left in consideration the algorithm stops and the optimum is guaranteed. The choice of one subtour should be interpreted as one iteration of the algorithm. This iteration takes $O(c^3)$ time because of the graph manipulation. The construction of the first tour takes $O(c^4)$ time however time requirement decreases since their subtour is already considered. The method is well adaptable for MTSP, but not so well for VRP.

- **Ant Colony Optimization:** It applies swarm intelligence to TSP. There are multiple agents walking the cost graph after each other. Each leaves a pheromone in each visited node meaning a constant value marking the quality of the node. Later agents choose vertices with higher pheromone value with a bigger probability, while increasing the values themselves. The pheromones are also decreased in each iteration, aging old assumptions. This process continues and ants are finding better and better solutions in time. One iteration takes $O(c^3)$, meaning it is comparable with genetic algorithms. It can be applied to MTSP and VRP by starting multiple agents in one iteration, always moving the one with the lowest subtour. These take $O(c^4)$ runtime per iteration since the ant with the lowest time must be found.

Chapter 5

Proposed Algorithm

5.1 Genetic Algorithm

Genetic algorithms are metaheuristics that belong to the group of evolutionary algorithms. Heuristics are simple algorithms, strategies to find suboptimal solutions for optimization problems. Metaheuristics are a subset of heuristics lacking a clear separation from other heuristics. Usually, we call a heuristic metaheuristic if it implements a more abstract strategy that can be adapted to most optimization problems. Evolutionary algorithms belong to this set since they simulate a simplified model of the evolutionary algorithm over any suboptimal solution set. They use controlled selection and mutation to breed a more optimal solution set over the iterations.

In a typical case, the problem to solve is NP-hard, call it HP (hard problem). HP must meet several requirements. A problem must exist for HP that is P-hard, call it LP (light problem). HP's solution set should be a subset of LP's. There should be a function that defines for each solution of LP a distance from the set of HP's solutions, call it distance function. In classical cost minimization problems, the goal value is unknown making distance calculation impossible. Instead of a distance, a cost function is defined. The goal of the optimization process is to minimize this cost. In most versions, we take the reciprocal of the cost and call it a fitness function. This way it is easier to handle problems where HP's solution set is finite and LP's solution set is infinite.

In our case, LP is the task to find a possible transport group on the given parameters and HP is the problem to find a transport group with optimal cost. HP's solution set is finite but LP's solution set is finite too. There are finite different permutations and distributions even if their number increases factorially with the number of clients and transport units. Therefore a cost function can be more efficient than a fitness function: $cost : (Slp) \mapsto C$, where Slp is a Solution of LP and C is the cost, $C \in \mathbb{R}^+$.

During the implementation of a Genetic Algorithm (Fig. 5.1) LP's solution set is treated as a species. The algorithm starts with the generation of the initial population (P0) that consists of a polynomial number of solutions of LP (Slp). Every Slp is treated like a specimen of the species and an evolutionary process is simulated. In each iteration, survivors are selected, and others are removed. A higher probability of survival is insured for the specimen that has a lower cost, driving the convergence. New solutions are breaded from the survivors and the newborn specimens are mutated. Usually, two survivors are merged to generate two new specimens keeping the original ones too and the survivors are mutated based on pure randomness.

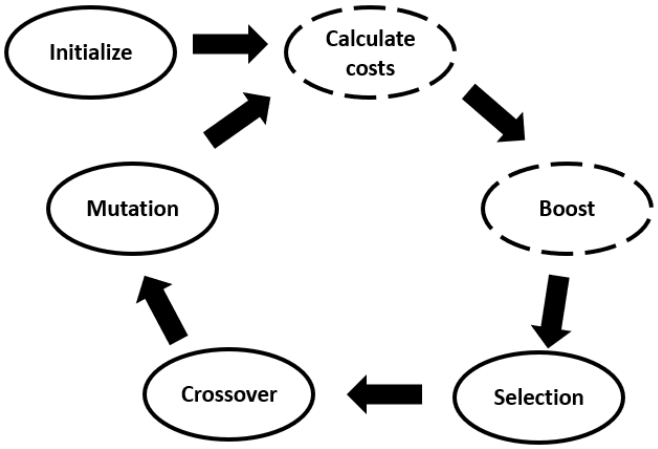

Figure 5.1: Flowchart of the genetic algorithm

The main disadvantage of genetic algorithms is their runtime. In the case of c customers and a population of p solutions, an $O(p \cdot c)$ runtime is required for one iteration. This approximation still assumes O(c) runtime requirement per operation. Any more complex operations and cost calculations affect the runtime significantly. If p was O(c), then one iteration took $O(c^2)$. Since at least O(c) iterations are required the whole optimization takes $O(c^3)$ runtime. In our original version, the p was $O(c^2)$ and the optimization took $O(c^2)$ iteration meaning an $O(c^5)$ runtime. We had to scale back population and operation runtime significantly before applying it to one thousand customers.

There are two other variants of evolutionary algorithms worth mentioning because of their similar mechanics and the promise of lower runtime. One of them is virus evolutionary algorithms [11] that adds two steps to the genetic algorithm's iteration. One step is infection and the other is destruction. The two steps take place after crossover (breeding) but before mutation. Infection inserts advantageous elementary properties (genes) into specimens. Destruction chooses disadvantageous elementary properties already in specimen replacing them. These operations are usually applied to the newly generated specimens. These steps can be implemented to O(n) runtime but still slow down iteration. They are beneficial because they speed up convergence. The other is bacterial evolutionary algorithms [12], where the selection phase is removed, and the crossover phase is replaced with gene transfer and use mutation for different purposes. The mutation step accurses before gene transfer. Each specimen is cloned multiple times and the mutation is applied to each except one. One of the clones is replacing the original specimen and the step is repeated a few times. During gene transfer, elementary properties are injected from one of the best specimens to one of the worst specimens. This way the smaller population can be used compensating with the cloning mechanics. p can be O(c) and most operations take O(1) time. The only exceptions are the cloning mechanic and the ordering of the population by cost to choose the best and worst specimen. Creating and mutating l clones take $O(c \cdot l)$ runtime, and the sorting population takes $O(p \cdot log(p))$. However, there are techniques where each specimen is cloned in l examples only once and copies are kept in later iterations reducing runtime requirement to O(l). Also, a tournement mechanic can be used during gene transfer eliminating the need of sorting the population. This means an $O(c^3)$ runtime in case of an O(c) p, O(c) l, and O(c) iteration, but l can be reduced to O(1) resulting in an $O(c^2)$ runtime.

Still, we chose to experiment with genetic algorithms since they are more commonly used. This way our research affects more people's work. We are also more familiar with genetic

algorithms. The introduced new crossover operator promises to reduce runtime significantly. Our results can be easily applied to virus and bacteria algorithms too and we are going to experiment with those variants too.

5.2 Modified Genetic Algorithm

There are two reasons a Genetic Algorithm can get stuck in local minima. One is too slow convergence and the other is the extinction of elementary properties, genes. If a specimen is represented by numbers, each number is usually treated as a separate gene. In our case, this could be an index of a customer occurring in a position of the permutation. If there was no more specimen with a value in a position necessary to construct the permutation of the optimal tour then the chances of such a specimen being born before the iteration limit is exceeded is negligible.

There are two common treatments for the two phenomena. A well-adjusted boost step can be applied to some of the population, to speed up convergence. Usually, it consists of one iteration of local search on the specimen with the lowest cost. Also, extinct genes can be revived by replacing some specimens with specimens consisting mostly of extinct genes or by inserting those genes into newly born specimens. The two methods are contraproductive. Boosting the best specimen is reducing the chance for another specimen to replace it. If the same specimen stays in the first position it will have more and more descendants that have similar genomes. The more similar the specimens are to one another, the more uneven the distribution of genesis. We call this the incest effect since the best specimen will become closer relatives. If some of the best specimens get replaced or less advantageous genes stay in the population, the convergence slows down. By preferring advantageous genes during injection local searches may be stuck in local minima more easily. In the original setup bringing back, extinct genes are done by mutation, but survival chances of those genes are very slow. As convergence is approaching local minima survival chances to drop by an exponential rate.

Still, both steps might be necessary since both phenomena must be treated. The virus algorithm solves this problem by merging the methods into one. By injecting back genes into the population all genes have some chance and by eliminating costly edges the convergence does not slow significantly and new edges are born. The problem with this approach is that the two steps serve both purposes and adjusting the two processes affects each other directly. In our solution, the steps are separated. Boosting becomes a separate step and gene distribution treatment replaces mutation.

Another new step we added is the simulation phase. In VRP the calculation of cost for the specimen is costly. In older versions, it is done during the selection phase. Separating it to a new phase makes other steps easier, makes the process more ready for parallelization, and lets us run it on way stronger computers.

So, in the modified version the optimization starts with the generation of the population and the simulation phase. The iteration starts with the selection, followed by crossover, then mutation (gene revival), simulation, and finally boost. Our architecture makes it simple to switch between the different variants even turning off boost or replacing edge revival with other mutation methods.

For local search, the OPT2 algorithms variant was implemented. This algorithm is used on permutations. Every two positions of the permutation are selected, and the values are swapped. After each change, the simulation runs, and the change is kept only in the case of a better cost. The iteration is repeated until no better solution is found. In our Boost

phase, the local search is stopped after the first better solution to minimize the negative side effects.

5.3 Boost

Boost as previously mentioned is local search applied to some of the population. The target can be the best of the population to affect the convergence directly or the worst of the population to find new unique local minima. If the local search is running too many iterations the incest effect might take place. By improving multiple specimens it can be reduced. So, there are three questions. Who should be boosted? Which local search should be applied? How many iterations should the local search run?

As previously mentioned (reference background and heuristics) there are multiple ways to specify the distance of two permutations. The distance is always specified by the amount of some elementary modification that is necessary to transform one of the permutations into the other. This modification might be the writing of positions, switching two values, reversing a segment of the permutation, etc. One iteration of the local search is done by applying these modifications in every possible way. If any better permutation is found, the old permutation should be replaced by the new one. In some variants, it is done immediately, in others all permutations of the iteration are tested and the best permutation is chosen at the end of the iteration. Immediate overwrite is beneficial because the iteration can be stopped after every test of permutation. In this case, we usually refer to this test as a step of iteration. At the start of the optimization immediate overwrite might result in faster convergence too.

The most common choice is the Opt-n algorithm family. In Opt-n the Hamiltonian cycle is cut into n parts and all permutation of parts is tested trying also to reverse the permutation of values of parts. In simplified versions, n positions are selected and all possible permutations are tested. Since these optimizations take $O(c^(n+1))$ runtime, Opt-2 is the optimal choice for boost. The runtime would have been reduced to $O(c^n)$ if cost calculation of permutations took O(1), but VRP simulation is more complex taking at least O(n) time. This means Opt-2 has an $O(c^3)$ runtime per iteration per specimen.

Since simplified Opt(2) selects two positions in the permutation per step all two positions are tested in an iteration. We could stop optimization if it got stuck in local minima or after one iteration or after one step. We can also wait until an amount of improvement is achived or choose a hybrid of the two perspectives. We choose to stop the boost after the first improvement, but before the second iteration. This seemed to be the fastest we could run it without the whole population getting stuck in local minima.

At boost the survivors can be classified into the following groups: survivors because of their low cost (elite), survivors because of randomness (lucky), survivors because newly born (child), survivors because of gene revival (mutant). Of course, one specimen can belong to multiple groups and different classifications might be beneficial for the different selection processes. If we wanted the fastest convergence, the elitist solution might be chosen: the whole elite, elite of the elite, some of the elite, best of the elite, worst of the elite. Newly revived genes start in mutants, then usually move to lucky, then children and have a small chance to get into the elite. So local search might be used for the mutants removing unnecessary edges and bringing in new competitors to the elite. Or might be used for the lucky, so they have a better chance to pass their genes. Or might be applied to the best child so it has a better chance to replace the best permutation. Even a more complex approach could be implemented where a function is defined too as $u : Slp \mapsto y$

where y measures the uniqueness of the given specimen. The specimen with the maximal y per cost should be chosen.

For the sake of simplicity, we choose the elitist approach. Since our goal was to measure the effectiveness of crossover operators and other approaches could deform the results complicating the interpretation of statistics. The best specimen is affected, but the small boost those not increase the incest effect.

So, we use simplified Opt-2 on the best specimen until the first improvement or the end of the first iteration.

5.4 Maintaining Gene Diversity

As previously mentioned there are two ways of reviving genes. The genes can be generated by replacing old specimens or by inserting extinct genes into the old specimens. Which should be used? Which child should be affected? Which genes should be revived? What should be perceived as genes?

We observed that the cost of genes does not come directly from the position of a value in the permutation. It comes instead from the neighboring of the values. This means that a gene is an edge. First, the extinct edges must be identified. We create a whole graph with the same vertices but different edges that have the amount of said neighboring in the population as weight. Let's call this graph edge counter graph. They initialize by zero then we iterate through each specimen of the population. After counting we can simply take the edges with zero weights. This takes $O(p \cdot c)$ runtime, so $O(c^2)$ in our case. Let e be the number of extinct edges. e is also $O(c^2)$. Another solution could be to skip calculation and simply revive all edges. This removes the cost of counting but maximizes the cost of insertion.

We can take an edge from ca to cb and insert it by moving the cb right after ca or by moving ca right before cb. If the original edges ca to cc, cd to cb, and cb to ce existed then moving cb means the destruction of these edges and the creation of ca to cb and cb to cc and cd to ce. This is unwanted behavior because by the insertion of one edge other edges might go extinct. If ca exceeds cb in permutation, we could simply invert the section from cc to cb, where cc is the element after ca. This means two edges are destroyed and two are created. This is exactly an Opt-2 step. In this case, we assume that the edges are undirected. If there is a significant difference between the edges between two locations, this means the destruction and creation of $2 \cdot (d + 1)$ edges. If a permutation is stored as a list, finding the position of ca and cb takes longer because O(c) pointer calculations are necessary, but moving ca and cb to the right position takes O(1) time. If a permutation is stored as an array, finding the position of ca and cb is fast but inserting means other values must be moved resulting in O(c) runtime. Insertion takes O(c) time anyway. Inserting e edges takes $O(c^3)$ time. This is a big deal because it means $O(c^3)$ iteration time. A good strategy might be to run insertion for every n iteration or inserting only O(c) edges per iteration.

Construction of new edges might be faster. It should be started from a vertex of the counter graph with zero edges. Generate cycle by always taking the edge with the lowest weight not leading to already visited vertices. We increase the weight of all edges. O(c) new specimen is enough, but generation still takes $O(c^2)$ time because edges must be checked. Time can be reduced to $O(c^2 \cdot log(c))$ by sorting the edges of each vertex beforehand. If there are no zero edges the lowest cost edge might be selected or the edge with the lowest $cost \cdot weight$. Cost can be further reduced to $O(c^2)$ by choosing a completely random edge.

If we choose new specimen construction and revival of all edges modulo cycling permutation generation might be used. It will be introduced in a later section (reference section).

We choose modulo cycling permutation generation. We reset exactly c-1 specimen instead of mutation of every second child. We chose it for its minimal runtime, and because it can be applied to the multiplicative of c. Since these genes' survival rate drops exponentially by iterations the reset is running in each iteration.

5.5 Selection Process

In the standard genetical algorithms, the selection process means the deletion of half of the population. Since the same amount of children are generated, p / 2 instances are deleted and initialized. We chose to flip a flag in the specimen and use them as child specimens. Our goal during selection is to keep the best specimen and give better chances of selection for the better specimen. The elitist solution is to sort the population by cost and select the better half of the population. This leads to fast convergence but local minimums and incest effect. The other solution is to select specimens by pure chance. This can eliminate any convergence leading to aimless search in the solution set.

We chose a hybrid of the two where a quarter of the population is selected elitist, and one quarter is selected randomly. This means ordering them is necessary, and it takes $O(p \cdot log(p))$ runtime. Convergence is still good and can be adjusted in many ways. This is also the most standard approach. Sorting the population had other advantages in other steps and the collection of statistics.

There are several advanced approaches.

One of them is tournement selection [8]. The population is shuffled and sliced into pairs. The specimen with the best cost is kept from each pair. This means at least a quarter of the best specimen is kept and survival chance is directly proportional to the backward order of the specimens. It is because if a specimen is at the back quarter line of the population then one fourth of the population leads to its selection if chosen as a pair. A less elitist solution would be to slice the population into groups of four and select the best and random. A more elitist solution would be to slice the population into groups of four and select the better two of the four. Each version takes only O(p) time since ordering is not required.

Another method is the roulette wheel selection method [8]. In this version, the sum of the reciprocal cost of specimen in population is calculated. A random number is selected between zero and the sum. We iterate through the population reducing the value by the reciprocal of the specimen's cost. If the value drops below zero, the specimen is selected. The selected specimen is removed from the further process. Selection is repeated, taking $O(p^2)$ runtime. It can be reduced by preparing n random numbers between zero and one. These numbers should be sorted. We take the first number, make a copy, and reduce it with cost per sum. If it drops below zero, the specimen is selected. The sum is reduced by the reciprocal of its cost, the second number is reduced by copy and value, and scaled by sum / old sum. The first value is removed. If the new first value is negative, we step backward adding a reciprocal of the cost of the specimen until it raises over zero. We repeat iteration until the number array is empty. In another version, one specimen can be selected multiple times. The random numbers are generated between zero and sum and sorted. Value is reduced by reciprocal of cost. In case of selection only the value is removed and the second value is reduced by the copy and other value. If it is negative then the same specimen is selected again. Both versions take $O(p \cdot log(p))$ runtime.

During selection, the reciprocal of the cost was used. This is dangerous because of the accuracy of numerical types. Values may drop to zero or exceed maximal value. We could use more complex numerical types instead of values, but it would multiply runtime by a higher polynomial. The solution is to select specimens for deletion instead of survival. This way the weights could be proportional to cost.

Since we overwrite some of the specimens, to deal with gene extinction, a new selection task should be added. The specimen to overwrite should be collected into a separate group, next to the survivors and children. This means that not p / 2 but (p - r) / 2 specimen should be selected for survival, where r is the amount of the reseted specimen. This way the mutation step can be removed entirely. If one-third of the population is selected as parents and one-third is selected for reset then the tournement selection can be used. Shuffle population and sort it into groups of three. The best of the three is a survivor and the worst of the three is reset. The middle one might be overwritten by the best's child.

5.6 Initialization

The standard way of initialization of the population is pure randomness. There are multiple ways we found in implementations and publications to generate purly random permutations. Therefore the most trivial solution is to use one-chromosome representation and build c+s long permutations. A common mistake is to choose random numbers between 1 and c+t adding them to a list. If the list already contains the number, skip it and continue until the list is c+t long. The biggest problem with this method is that it takes multiple generations to generate the last values, because the more value is already present the harder it is to choose randomly a number not present yet.

A better solution would be to use perturbation. Take the values in any order and switch two randomly selected values $2 \cdot (c + s)$ times. It is proven that all possible permutations have the probability using this method. It takes $4 \cdot (c + s)$ random number generations and writing of numbers to use this method. Another solution can be to create a c + s long list of all the values and choose a random one every iteration, removing it from the list and adding it to the permutation. This takes only c + s random number generations and writing but slower because of using lists. It can be further improved by using a c + s long array and in the i^{th} iteration choosing a random position between i and c + s.Than the value in position is switched with the one in the i^{th} position.

The first intuition to speed up the optimisation would be to start with a better population. Specimens of the population could be created using some kind of tour constructing heuristics or randomly but improved by some kind of iterative optimisations. In the first case it is a big challenge to construct multiple kinds of specimens. Usually some kind of random construction is used with cost based probability, like the nearest neighbour variant. The resulting transports are diverse and have better cost, but still some genes will be extinct from the initial population. Other versions improve by giving lover probability for some edges if they mean other good edges get excluded and building the rout from multiple locations. The other versions with iterative optimisation use Opt-2 or other method keeping the time requirements under $O((c + s)^3)$. These versions still require more time than most construction methods. These kinds of optimisations tend to drive the optimisations into local minimums. If only a part of the population is optimized or created using some heuristics the effects are worse becouse of the incest effect.

We chose to generate diverse populations instead optimising for even distribution of genes. This can be done multiple ways. We could use a nearest neighbour with a cost based probability variant. If we initialize a complete graph's edges with 1 as cost and double the cost of all edges selected, that means edges never used gain higher probability over time. This means $O((c+s)^2)$ generation for each specimen. We found a better solution called modulo stepper initialization. In this initialization a completely random population is chosen as base permutation. Other specimen's permutations are generated by walking this permutation with different step sizes equal to the specimen's index relative to the base specimen. All values touched during walk are copied to the new permutation, if already existent values are touched they are skipped. If more specimens can not be generated from the base permutation, then a new base permutation is generated. This way generation takes O(c+s) per permutation and the distribution will be almost even. We could assume that by forming permutations from the base permutation, we can get a population that is too regular. This could cause stucking in local minima, but we didn't experience any signs of it. By always using a random permutation as base all permutations will be random.

A similar state to populations generated by Opt-2 can be achieved in the first cycle of the optimization (c iterations). The population can be generated in $O(c \cdot p)$ time using minimal memory space.

5.7 Crossover Operators

During the crossover phase, the survivors are paired as parents and two new transport plans are generated, referred to as the children of said parents. Usually, a child is based more on one parent than the other so the parents are referred to as primary and secondary parents. The goal is to mix the properties of the transports in a way that keeps the essence of both parents, so the children have a higher chance to inherit the advantageous ones. The operation is called crossover and the variant used to generate the children is called crossover operator. Usually some values are set based on the primary parents and then the empty positions are filled with the missing values in the same order the values are in the secondary parent. That's what we are going to refer to as "fill based on the other parent." or " filled in the order of the other parent". We used fourteen different crossover operators that were published for the TSP problem. Therefore one-chromosome representation was used in most cases. All operators are listed in this section.

- **Order Crossover [6] (OX1, Fig. 5.2):** Two children are generated and the primary parent of one is the other child's secondary parent. First, two positions are randomly selected and the sequence between the positions is copied to the children from their primary parent to the same positions. The missing values are filled based on the secondary parent. This is fast and most edges are inherited by the children. If the segment length is long then the child is based most on its primary parent, while if the segment length is short then the secondary parent becomes dominant.

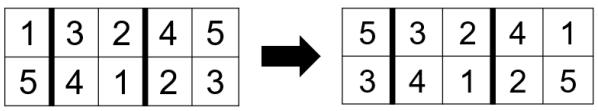

Figure 5.2: Example for order crossover

- **Sorted Match Crossover [6] (SMX, Fig. 5.3):** One child is generated from two parents. The parents are searched for sequences consisting of the same values.

The sequence must have the same start and end value, length and values present in it. The only difference will be the order of the values in the middle. The cost of the sequences is measured. The child is filled with the higher cost parent's genes replacing the sequence with the other parent's. The operator requires $O(c2)$ CPU time for finding the sequence, but the simulation can increase it further. We removed the simulation from the operator and made it generate two children. It has slower but still good convergence and runs significantly faster. We used a simplified, faster variant, where two children are created by switching the sequences. This operator has the highest possible inheritance rate since the child will contain only genes present in parents. The generation takes O((c+s)2) runtime, but provides the most stable convergence, that's why we kept the operator.

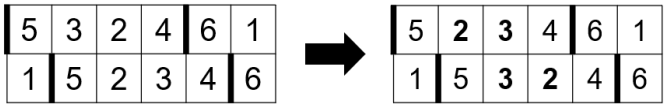

Figure 5.3: Example for sorted match crossover

- **Maximal preservation crossover [6] (MPX, Fig. 5.4):** One child is generated from two parents. The length and the starting position of the sequence are randomly selected. The sequence is copied from the primary parent to the child's beginning. The missing values are filled in the other parent's order. Since the length of the sequence is restricted, most genes are inherited from the secondary parent.

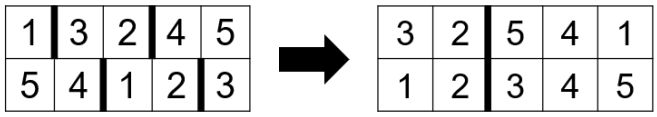

Figure 5.4: Example for maximal preservation crossover

- **Partially Matched Crossover [4] (PMX, Fig. 5.5):** It is similar to the OX. The difference is that the elements not in the sequence are copied. The positions of the sequence are filled in the order of the other parent.

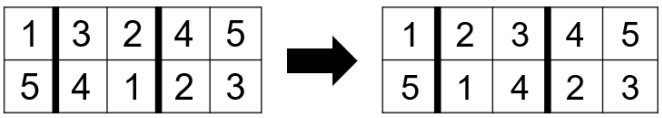

Figure 5.5: Example for partially Matched crossover

- **Cycle Crossover [6] (CX, Fig. 5.6):** Two children are generated from two parents. Optimisation starts by copying a random value from primary parent to child into position and follows the next iteration. Finds the last copied value's position in the secondary parent. Copies the value of the position from the primary parent to the position in the child. If the child contains the value, the iteration ends. The missing values are filled in the secondary parent's order. This results in less inherited edges since the cycle can be very small or the copied values may not neighbour each other.

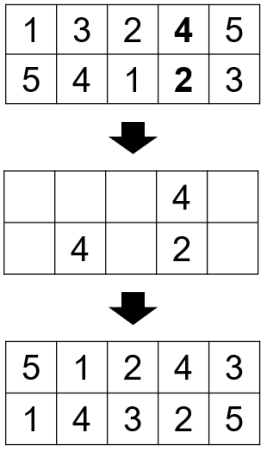

Figure 5.6: Example for cycle crossover

- **Voting Recombination Crossover [6] (VR, Fig. 5.7):** One child is generated from multiple parents. l (smaller than parent count) limit is chosen. If more than l parents have the same value in a position then the value is copied to the child into said position. The empty positions are filled with missing values in random order. In our variant, two parents were used and l was two. The runtime was high because of the randomization. Since more than two parents would be needed for an efficient user, the convergence is very low. In the case of a local minimum, many permutations become copies of each other. A too high p with too low l would stop the algorithm from getting out of the local minimum. In the current form, the algorithm is a useful noise in the case of local minima. The runtime is $O(p \cdot n)$, so it is optimal in the case of a given p.

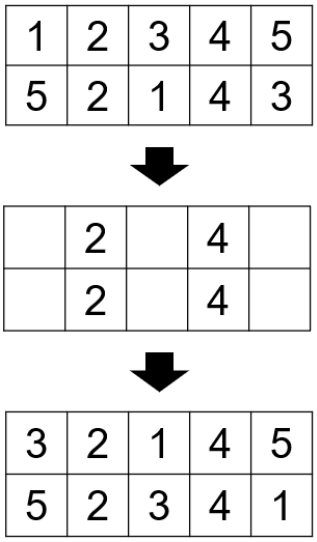

Figure 5.7: Example for voting recombination crossover

- **Genetic Edge Recombination Crossover [6] (EX, Fig. 5.8):** We live with the presumption that the direction of the edge is of secondary importance. One child is generated from two parents. An n times n matrix is generated, where each line corresponds to a value, and the line is filled with the values neighboring them in either parent. For each value, neighboring values are collected from parents into a

table. We choose a random value to be the first element and remove it from the table. In some variants, the value must be from any end of any parent. We choose one of the values from its line randomly from the ones with the least values in their lines. The value is copied and removed from the table, the line is cleared. We repeat until the child is filled and every time a chosen value line is empty, a random missing value is chosen. In a modified variant the edges in both parents have priority. This crossover is efficient because of its optimal O(n) runtime and great emphasis on edges instead of values.

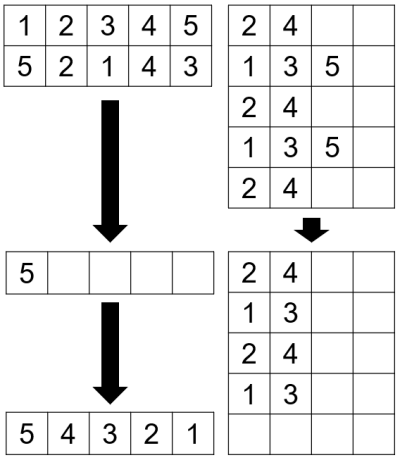

Figure 5.8: Example for edge recombination crossover

- **Heuristic Crossover [6] (HX, Fig. 5.9):** One child is generated from two parents and the first value of the child is chosen randomly. In each iteration the neighboring values of the last inserted value are collected from parents. The neighbours already in the child are deselected. The next value is randomly chosen from the neighbours with probabilities proportional to the cost of the route between the last inserted and the neighbour. If all neighbours are already in the child, then a random missing value is selected. The iteration is repeated until the child is filled. According to statistics, on average sixty percent of neighboring values are inherited from parents. The weight calculation can be time-consuming, but usually a simplified calculation is used. With simplified calculation HX has an optimal runtime, and a fast convergence, however the local minimums are not avoided. The crossover may be more adaptive if it chooses random values with a small probability even if neighbours are missing from the child.

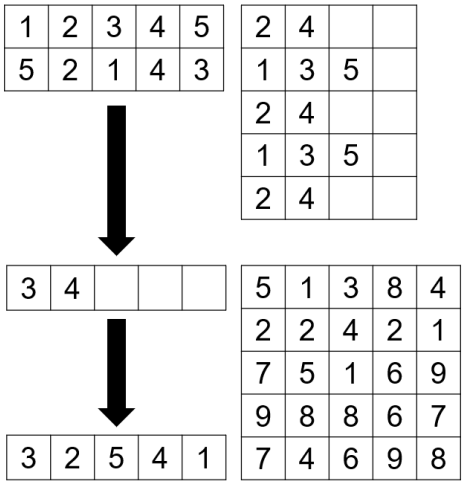

Figure 5.9: Example for heuristic crossover

- **Alternating-Position Crossover [6] (AP, Fig. 5.10):** One child is generated from two parents, and the primary parent's first element is copied to the child. During iteration it keeps switching between the parents, always taking the next not touched value of the parent. If the value is already present in children, skip the value. Continues until all positions of children are filled. This generation breaks almost all edges making inheritance low, however edges present in both parents have higher chances.

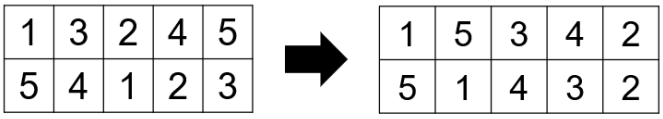

Figure 5.10: Example for alternating position crossover

- **Alternating Edge Crossover [6] (AEX, Fig. 5.11):** The sequential representation is used and a random value is chosen as the first element of the child. The iteration keeps switching between the parents. Always takes the value following the last inserted value in the actual parent. In case of values already present in the child a random value is selected instead. AEX is a flexible operator that rarely gets stuck in local minima. It has a slow convergence but a high inheritance rate.

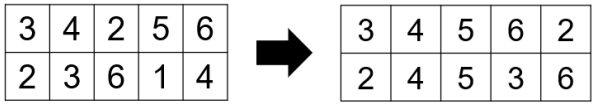

Figure 5.11: Example for alternating edge crossover

- **Order Based Crossover [6] (Ox2, Fig. 5.12):** Two children are generated from two parents and multiple positions are selected randomly. Values from selected positions are copied from primary parents to children. The unselected positions are filled with the missing values in the secondary parent's order. OX2 breaks most neighboring pairs resulting in slow convergence. OX has better convergence, but OX2 has a lower chance to run into local minima.

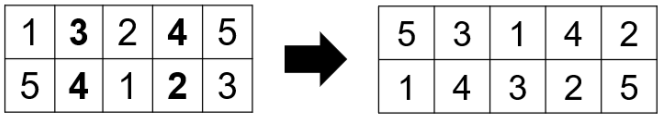

Figure 5.12: Example for order based crossover

- **Distance Preserving Crossover [3] (DPX, Fig. 5.13):** Two children are generated from two parents. Values that are in the same positions in both parents are copied to the child into position. If a value is in the j^{th} position of the primary parent and i^{th} position of the secondary parent, then copy the value from the secondary parent's i^{th} position to the child's j^{th} position. DPX has a similar convergence speed to CX but is a bit more flexible.

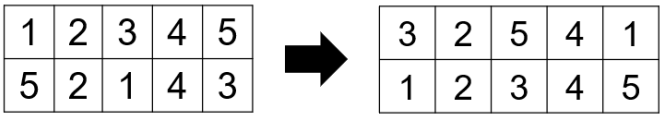

Figure 5.13: Example for distance preserving crossover.

- **Position-Based Crossover [6] (POX, figure 5.14):** It's the same as OX2 but only a few positions are selected and the parent's roles are switched.

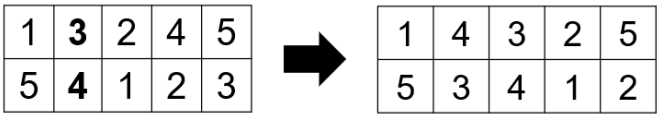

Figure 5.14: Example for position based crossover

- **Subtour chunks crossover [6] (SCX, Fig. 5.15):** One child is generated from two parents using sequential representation. We are taking a random sequence from the first permutation and follow up with a sequence from the other parent switching until the child is filled. In the case of an already used position, a random other unused position is chosen. SCX is resulting in one of the best convergence with a lot of flexibility.

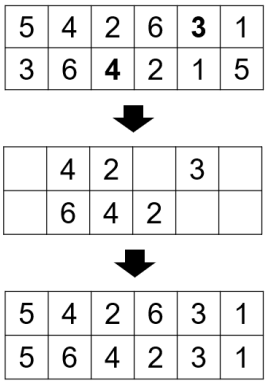

Figure 5.15: Example for subtour chunks crossover

5.8 Mutation

Mutation and crossover have opposite goals and a similar dynamic to boost and edge distribution correction. Crossover's goal is to gain similar specimens to already existing ones with minimal amount of new genes, while mutation's goal is to create new genes. High mutation slows down convergence while weak mutation increases the risk of getting stuck in local minima. Not all children are mutated, only a specific ratio, usually 50There are multiple ways to mutate a specimen, even if edge distribution corrections are not included.

- Relocation of gene of specimen. In this case two random positions are chosen. One to select the value of that position and a second as the new position. The disadvantage of this method is that the values between the two positions must be shifted. In the case of arrays shifting the segment requires O(c+s) time.
- Swapping two genes in the permutation. This requires two random index generation and O(1) runtime.
- Shuffling multiple genes in the permutation. This requires O(c+t) random index generation and O(c+t) runtime.
- Reversing a segment of the permutation. This requires two random index generation and O(c+s) runtime. The fastest implementation could be
- Shuffling a segment of the permutation. This requires two random index generation and O(c+s) runtime.
- Shuffling segments of the permutation. This requires two random index generation and random numbers between 1 and six. The two indexes are the breakpoints and the random number will determine the new order of the three segments. Process still takes O(c+s) runtime.
- Cost proportional probability can be applied to these operators randomization too. Each operator destroys and creates new edges. We should choose to break costly edges with higher probability. Adds an O(c+s) timecost for the weight calculation and makes the return of costly edges harder. Another solution could be to give higher probability for modifications that create lower cost edges. This can significantly increase the time requirement of the modification.

Since mutation and edge distribution correction have similar purposes, we chose to merge them into one step. We used the reset of multiple children to complete the phase in $O((c+s)^2)$ using the modulo stepper method. The new specimens are rarely optimal and have an exponentially increasing chance of death, but have a good chance to crossover with better specimens inserting rare edges back into the population. We also run setups with the value switching and segment reversing variants. We found sinergies competing with SRX, but SRX worked best with the resetting method.

Chapter 6

Statistical Racing Crossover

In most cases, the population is squarely proportional to the number of customers. Since a crossover operator has to fill the genes of a specimen it has at least an $O(n)$ runtime. The crossover phase takes at least an $O(n^3)$ runtime. In the case of a thousand customers, the operator must run a billion times. The operator must be simple and as fast as possible. Speed is primary compared to the speed of convergence and avoidance of local minima. The currently used operators are fast enough, but the simplicity stops them from performing either at the speed of convergence or at the avoidance of local minima. More complex operators exist but they require at least $O(n \cdot log(n))$ runtime.

Our goal in designing the new operator was to combine the advantages of the previously mentioned operators with a minimal additional time cost. This is achieved by running one of the operators at a time. The only additional cost is the gathering of statistics and the selection process. The selection process runs only once per iteration minimizing the costs further. The operator had multiple variants, we are presenting the best-performing one.

The Statistical Racing Crossover (SRX, Fig. 6.1): The first time the operator runs it initializes the statistical data. This way SRX doesn't require resources before it is in use. Two counters are stored for each operator, the run counter and the improvement counter. The run counter is incremented every time the SRX runs with the operator selected. The improvement counter is incremented every time the child's cost is better than a parent's. It is increased by two for the primary parent and by one for the secondary parent. The run counter is initialized as 2 and the improvement counter is initialized as 1.

On the first run of each iteration, the previous operator's statistics are aged if there were any, and a new one is selected. On aging, the run and improvement counters are multiplied by 0.9 and incremented. The incrementation is necessary because the counters can hold only whole values, and the aging could reduce them to 0 otherwise. The selection is random, but a probability is insured for each operator proportional to their improvement run ratio. The goal is to favor operators with faster convergence, but leave an opportunity for operators with lower convergence to improve their statistics. Aging causes operators stuck in local minima fade faster while leaving the probability of the aged operator the same.

The selection process resembles the roulette wheel selection method [10], but used to select only one element. Also there are multiple other ways to measure the success of an operator.

- **SRX with equal parents**: This variant assumes that the secondary and primary parents have the same importance in the child's creation. In this case, the counter should be increased by 1 for each. We observed other tendencies. Most crossover

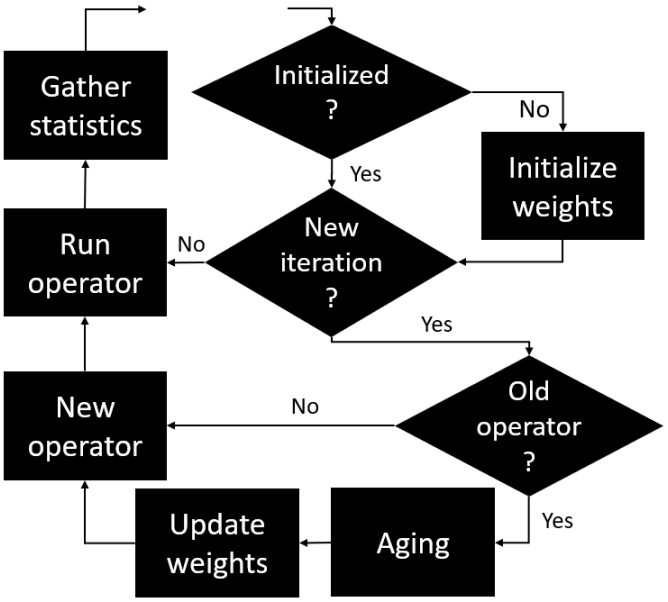

Figure 6.1: SRX operator flowchart

operators copy values from the primary parent and fill empty positions in the secondary parent's order breaking most neighboring of it resulting in less similarity with the secondary parent.

- **SRX with preset ratio**: The importance ratio of the two parents depends on the operator. We measured the weight of the parents in separated runs. We found that the correct ratio varies within optimizations. We observed an even higher variance between data sets making it unreliable. It must be adjusted to the input data based on statistical assumptions.

- **SRX with dynamic ratio**: The importance of parents can be measured by counting the edges inherited. We observed better results, but additional runtime increased from $O(1)$ to $O(n)$. Conversion to sequential representation is necessary, and two comparisons are iterating the new arrays. Overhead overwhelmed the time won by needing less iteration during our tests.

- **SRX with improvement rate**: Another observation we made was the importance of the improvement rate. If the increase was the ratio of the parent's and child's cost, the convergence had been faster at the start of the optimization. However, later the improvement slowed down quickly. Some operators were better at improving specimens with high cost while being bad at improving the leading ones. Since mutation produces new specimens with a high cost every iteration, these operators led the race instead of others, who were still capable improve optimum at a slower rate. Further experiments are necessary to identify operators necessary to remove.

- **SRX with bounty**: There should be an extra increase called bounty for improving specimens with the best cost. The lucky ones getting the first bounties led the race. Leading operators collect more bounty, and they fall back very slow. We experimented with different bounty amounts, but it seemed impossible to pinpoint the correct one. We tried a higher aging rate, but it meant a new parameter with an optimum dependent on data.

- **SRX with parent cost penalty**: The lower the cost of the improved specimen is the better. That means the increase should be divided by the parent's cost. This meant aging like effect since the specimen had lover costs in later iterations. However, the aging was faster on a specimen that was run rarely. This made racing impossible for some operators losing in the first iterations.
- **SRX based on parent position**: In a population sorted by cost, the further forward a specimen is, the harder it is to improve it. The key to long-term convergence is to reward improvement on the specimens ahead. The increase could be the index from the back of the population or polynomially proportional to it. This version was promising but made sorting the population a necessity. We prefer the proposed version for its wide range usability.

SRX is useful to collect statistical data about operators without running a different optimization for each. It combines the strength of the built-in operators. When operators with fast convergence lead into local minima, others got higher weight and find better transport groups. In the long term, the position of each operator is stable within the race. The positions are changed regularly marking a local minimum but the contestants are realigned to the original order within a few iterations.

Chapter 7

Implementation and Technologies

7.1 Kotlin

The algorithm was implemented in Kotlin. Kotlin is a programming language developed by JetBrains in 2011. It replaced Java on Android as the official language of the platform in 2019, because of disagreements between Google and Oracle. Since then it has gained bigger popularity than Java and Dart (programming language of Flutter, a multiplatform solution by Google) (according to stack overflow statistics). It runs on JVM like java making it multiplatform but can be translated to JavaScript and C++ too. On JVM all java libraries and frameworks are accessible and work well with the already present Java codes making the switch from Java to Kotlin easy. Kotlin has multiple features that were announced as the future for Java years later. A multiplatform solution was also added in 2020 as a competitor for Flutter.

Kotlin brings a typescript-like syntax to JVM where the name of variables comes before their type. The language is strongly typed but marking the type of the variables is usually optional. Kotlin also introduces properties and lambdas with a cleaner syntax. For example, if the lambda parameter is the last parameter of a function then it can be written into curvy brackets right after the call brackets instead of into the call brackets. This results in a statement-like syntax (like cycles, if-else, switch) making the introduction of new structures more transparent. Attributes are removed from class bodies forcing the best practice of properties. A field variable replaces it that can be manipulated from the get and set of the property. Semicolons are optional in Kotlin if expressions are in separate lines removing thousands of unnecessary characters from bigger code bases. These features can be misinterpreted as syntactic sugar, but they improve code readability, coding speed, and extendibility significantly.

Kotlin also has multiple functionalities present in C#, but with cleaner or more Kotlin-like syntax: properties, operator overload, extension functions, lambdas like LINQ, data classes instead of structs, async-await, as, yield, var, when instead of switch expression, etc.

Kotlin's most famous feature is null safety. In Java, null pointer-based errors are the most common. In Kotlin all types are non-nullable by default meaning variables can't store the null value. Of course, all type has a nullable variant, but they are used only if necessary, so the risk of null values is eliminated. When nullable variants are used Kotlin forces null check on every use and introduces new structures for it. Data can be stored in values or variables marked by val or var making the reference read-only or writeable further improving code safety. Only var properties have a set method. Set and get can have

different visibility. Properties must have an initial value or must be marked by lateinit signaling the risk of undefined value.

Kotlin also introduces coroutines. Coroutines are an alternative to thread pooling. In this solution, a few threads are used for thousands of asynchronous tasks. A thread takes the first task but does not stop on completion, it takes the next task instead and keeps running. This makes the program compatible with every computer's thread capacity and reduces the cost of the asynchronous behavior. We do not use anync await in Kotlin since it makes debugging difficult. A run blocking lambda function is used instead, that waits until all asynchronous tasks are completed, and coroutine scope is used to launch new coroutines.

Kotlin has a big focus on fluent chaining of functions. They introduced special extension functions with lambdas available for any type that make modifications easier. The four functions are let, run, apply and also. Let and run returns the value returned in the lambda, while apply and also returns the value the function was called on. The value is seen in let and also as a parameter supporting embedding better while run and apply makes it available as this making syntax clearer.

Kotlin keeps improving every year. They are taking features from other languages, making improvements on syntax and coming up with new features. Their biggest focus is multiplatform and efficiency now. The language was chosen for implementation for its elegant, clear design, the promise of fast coding, and safe handling of data structures. Most features of the language were used during development proven to be usefull.

We used Hibernate and MySQL Connector for database communication. Hibernate is a flexible ORM layer for java. It can be configured from a hibernate.cfg.xml file where all data classes must be listed. The data classes can be further configured by annotations handling relations between schemas, keys, and mapping. The schemas are automatically updated on configuration based on the data classes. Columns are not removed.

Ktor was applied for web communication. Ktor is a library for Kotlin supporting REST based web communication. It can be configured from application.conf and the Application.main() extension function, where install lambdas can be called. It maps URL-s to functions by adding said functions to the Rout object in the install(Routing). Apis are defined as extension functions on said Rout object. Listeners for application events can be added by calling the environment.monitor.subscribe lambda.

We used apache fluent for calling apis and Open Trip Planner for routing.

7.2 React

A React client was created for the project. React is a technology developed by Facebook making web development easier. The HTML of the webpage is rendered for every significant state change. The programmer can create new HTML components by defining functions that return HTML based on the state of the component. The state can be handled by Typescript code while HTML generation is done by embedding HTML code into the function. The component can react to state changes by hooks.

We can define properties for the component by adding properties to the function. The components can be embedded into each other's HTML where properties can be set like other HTML components attributes. The inner state of the component is stored in the properties and state variables. We can define state variables by the useState hook that returns a reference to the active state and to a set method of said state. Other hooks

can subscribe to those hooks by useEffect hooks that contain a reference to the state variables and a lambda as the reaction to modification of said variable. The lambda is triggered if any referenced state variable gains new value. Important to note that reference modification is necessary to trigger.

In a usual use case buttons and all inputs manipulate state in the component or in its parents. Prents are reached with callbacks injected by parents as lambdas. The modification is captured by a useEffect hook making modifications, interacting with the logic, and setting other state variables. State variables are embedded into the HTML code resulting in other layouts.

The main advantage of React is its simplicity. It is easy to structure the user interface and produce reusable codes. More hooks can be introduced to cover more complex situations. React is modular, flexible, and scalable making it optimal for web development. The main disadvantage of react is the render mechanism. The HTML is rendered for every state change of the component meaning if the screen displays a loading line in the root component, the whole screen was rendered every second. This can mean flashing UI in case of more complex views. The only solution is to minimize components that need render and store state variables directly in the effected components.

React's simplicity also has a payoff. Multiple supporting systems should be added for larger applications. Event processor pattern can be used to replace the callback system, so they do not have to be passed as parameters. A central data storage can make it easier to handle data relevant for multiple components. Listeners should be added for most central data. A separate framework is necessary to handle communication with the server. We used portable-fetch and url for this purpose.

The web interface was generated from swagger script in swagger editor. The script supports all forms of REST communication and data structures. The main advantage of this method is the well formatted documentation that can be also generated. The resulting code is rich in comments and well structured. The data classes are also generated making the generated code suitable starting points of the client. Custom generators can be written in java to support frameworks not listed on the editor webpage.

Framer library was used to avoid css usage simplifying development. It introduces frames as an easy weight base type for components. Also adds Stack as a base layout and scroll for displaying lists. Stack is like Android's linear layout and Scroll makes implementation of scrollable views easy. Only the input HTML component is necessary to make it a fully capable UI system. Framer also has a desktop editing software where drag and drop and scripting can be mixed for faster production. We switched to pure scripting because of a lack of npm package install option in the framework and windows support. The functionality was added this year making the tool optimal for production.

For displaying maps with the routes we used mapbox. In mapbox lets you customize the map display and load it through its online api. We used the mapbox-gl package to display and load the map into the application. The map can be moved, zoomed and tilted. A reference must be made to a div in our HTML to set it as a container for the map in the constructor. We can set data sources for features of the map and connect them to layers that display them. Changes can be made by setting the data for the source or removing and adding layers. Event listeners can be added by the on lambda function.

7.3 Client

The client functions similar to a wizard menu always moving to the next screen with a button at the bottom. The process is segmented into five steps. All screens consist of a left menu and a map.

The first screen is for setting up the task. Transport locations and transport units can be added, removed or edited here. The whole task can be saved and all previously saved tasks can be loaded. Here the map is interactable and can be adjusted to fit the added locations.

After the task is set the server calculates the distance between every two transport locations. The user must wait until it completes.The calculated routes appear continuously on the map which is not interactable anymore. The continue button and an estimation for the best and worst cost appears on completion.

After routing the user can select the algorithm type to run and the stop conditions. The setup can be also saved or loaded from the server. On the next screen the algorithm is started but not running. It can be stepped by iteration or as many iterations as many goal locations are defined. The results are regularly refreshed on the map. The algorithm can be also set to run. In this case it runs until the time limit or iteration limit is exceeded.

Functionality was the main focus of the development of the client, but a pleasant look was our secondary objective. This was achieved by the rounded corners, use of icons instead of text whenever possible and use of multiple shades of gray. This resulted also in a more clear, intuitive, easy to see through design, since the shades segment the screen into components and guide the eye to the inputs.

7.4 Server

The server code is divided into logic, model, network and utility. Logic is the core of the server containing the most complexity. More structuring was considered, but was dropped for efficiency's sake. The most patterns were applied to the inner logic to provide factorability, modularity, mobility and extendibility.

As previously mentioned, the database is handled by Hibernate. To separate transaction and session handling from the rest of the code, a singleton class was created with generic methods called OHibernateManager. O refers to the world object which is the keyword for singleton classes in Kotlin. Connection with the database is built lazily on the first call and it is removed by the closeFactory method that closes the session factory. Records can be loaded by id even supporting more complex ids list items that have to implement the IListItemKey interface. This makes saving and loading of collections simple. Records can be saved, updated, deleted or listed. For more complex queries separate functions must be defined. All records have string UUID, most records have names with better readability, all list items have orderInOwner column and reference to owner. The most complex shame is DGraph, where D refers to it being a data class. It contains an array of DObjectives referring to goal locations that are differentiated from central position. Two separate arrays store DEdges from the and to the central position. The edgesBetween field stores the edges between objectives as an array of DEdgeArrays. DEdgeArrays is a wrapper for an array of DEdges that is necessary to make it saveable into the database. Hibernate makes decisions about the foreign key structure automatically. Usually creates

a separate table for the connection. This is applied to DEdges too since DEdges are list items of DGraphs and DEdgeArrays too.

In the networking layer functionality is divided into three separate apis, SetupApi, LifeCycleApi and UpdateApi. Tasks and Settings can be configured, saved and loaded through SetupApi. LifeCycleApi is responsible for control over the complex optimization, while UpdateApi allows monitoring and routing. All apies interact with the OAlgorithmManager, a singleton class that is the interface of inner logic.

All algorithms are working with specimen representations. Each representation type implements the ISpecimenRepresentation interface. It provides multiple lambdas for uniform handling of specimens to avoid constant type checks in the code. Most algorithms implemented are generic and support all representations. As previously mentioned all representations consist of a permutation and a distribution. One slice is the segment of the permutation belonging to one transport unit. There are lambdas to iterate through the slices or map them as the representation was a Multi-chromosome representation. There is a factory object for each representation that can produce and also copy the instances. This object is passed to the algorithm in the constructor with the task data.

There are multiple types of heuristics implemented. All groups are in separate packages implementing a sealed class containing all common properties of the group. For example, all nearest neighbour heuristic variants extend the SNearestNeighbour class. The class provides cost measurement for edges, objectives and whole transport groups and an entry point.

The genetic algorithms have the most complex implementation in our source code, that supports step by step control, lifecycle management and full customizability. Each phase of the iteration, each lifecycle function and control function calls functors injected through the constructor. The exact implementations are specified by the GeneticalAlgorithmSetup object that contains instances of enumerations. Each enumeration is associated with one function of the genetic algorithm and each member is an exact implementation. In Java and Kotlin abstract functions can be specified in enumerations that each member implements differently and Kotlin has operator overload. Our elegant solution is to make each member a functor with a call operator and specify the interface of said funcion as an abstract operator of the enumeration. Each functor has the algorithm's state and parameters as an input parameter. This way the caller gives access to the data for the functor in each call, and not the functor takes it.

Other constructor parameters include the representationFactory, timeLimit, iterationLimit, costGraph, salesmen and setup. The class has an inner enumeration for lifecycle state that is stored in the state property. While state is modified, runtime is measured. The algorithm itself runs on a separate thread controlled by the lifecycle functions. The genetic algorithm also keeps track of the iteration count of the best and worst specimen and the whole population of course.

The phases are broken into eight enumerations. Six is for the phases themselves and two is for operators. One is for the crossover and the other is for the cost calculation. This way if the genetic algorithm is run for another problem with the same data representation, only the ECost file must be modified, where the E refers to enumeration.

ECrossoverOperator takes the pair of parent specimens, the child to overwrite and the algorithm data as parameters. It returns nothing since the result can be found in the child. There are several methods used to decrease runtime requirements of the operators. We found that some operators made a "contains" function necessary, however each check took O(c+s) runtime. Therefore we created a boolean array that's ith value is true if the

ith value is already in the child. Each time a value is written to the child, the value's flag is set in the boolean array. This reduces "contains" check's runtime to O(1). Sometimes we had to choose a random value missing from the child, this also took O(c+s) time adding up to O((c+s)2) during construction of the child. We reduced this time by generating one random permutation and a counter starting from zero. Every time a random missing value was needed, a loop started to walk the random permutation from the counter as an index until it found the first missing value. It saved the index of the last found value back to the counter. This way the time requirement was O(c+s) for the whole operator. Another function with problematic runtime was the "indexOf" where the position of a value was necessary in the permutation. This also took O(c+s) time per run and was improved by generating the inverse of the permutation. The generation took O(c+s) time, but was necessary only once and all "indexOf" took only O(1) time. Using these tricks it was possible to implement all operators in O(c+s) time.

Chapter 8

Statistics

8.1 Data sets

For most statistics, we used multiple datasets with less than one hundred real Hungarian locations. A run of a brute force implementation is still estimated to take millions of years. This means a good enough complexity for the experiments while having a low few milliseconds per iteration runtime requirement. To increase complexity the locations were chosen from the same city, so the roads are more complex and parallel edges have a bigger difference in cost. We used this data to test long-term tendencies of the optimization and to test on as many locations sets as possible. We also made a bigger dataset of almost one thousand locations. We had to scale down the population from squarely proportional to linear. This way one iteration took only a few seconds.

Open Trip Planner was used for routing and trafficking. Each route was precalculated connecting all locations with each other. This was time-consuming since each route took almost a second for the OTP. One hundred routes took two and a half hours while the big dataset took eleven and a half days to connect with each other. We choose to cache the data into the database. Later we randomly selected a hundred locations for each test from the bigger dataset and loaded the routes from the database. We used this method to test the variance in behavior between different inputs.

A custom simulation was used for cost calculation. The vehicles were loaded with all the packages and the packages were removed as the vehicle progressed. The fuel consumption was simulated with only a distance-dependent constant and the speed on the roads was equal to the speed limit since OTP-s traffic data was unreliable. The cost of a route was calculated on arrival to the next location. These simplifications saves a lot of runtime during the simulation. Our estimations show that the simulations would take double the time if it was more realistic.

Our simulations were run on a Legion y530 with a 10^{th} generation CPU and 8GB memory. We experienced similar resources in the industry. Even if servers suitable for AI have better prices every year, most logistic companies want to avoid the maintenance of such hardware. They also want to avoid solutions as a service since it would mean that the other company can easily monitor their transports. Our goal is to support transports up to five thousand locations using stronger gaming pc-s. OTP is clearly not suitable for this size and precalculation is not a soulution eather since the storage of routs between all possible custemers of a company would take peta bytes to store.

8.2 Simulation Results

We had multiple goals during the test runs. We wanted to prove that SRX can improve the cost even in later iterations when other setups already got stuck in a local minimum. One of our goals was to show that SRX has a faster convergence than most other operators and achieves better solutions in a shorter time. We found the comparison of different SRX variants necessary to show the effects of different success metrics. We tested SRX on huge data sets to make approximations on resource requirements. We are going to show that SRX can adapt to more input data than other operators.

We used cycles in our statistics as a unit consisting of p iterations, where p is the size of the population. Our biggest restriction was time since longer tests required multiple weeks to complete. Even at the end of our first year, most test scenarios were tested less than ten times. That is why we are going to collect more statistical data before presenting our data tables. We made big progress however and we are going to present our detailed statistics next year.

As previously mentioned we had three types of data set. We had a small data set of less than one hundred customers at the start, later (late September) we created a data set of one thousand customers and later we used subsets of the big data set.

We used the first data set to test the long-term behavior of SRX. On average p was about 5000 and the test run for almost sixteen cycles (Fig. 8.1). After several runs, we found that the operators achieve the same rankings in the race on the data. The operators were sorted within the first few cycles. Operators performing similarly changed their positions regularly but realigned in a few iterations. The increase of change in order had a high correlation with the increase of drop rate of improvement, marking local minimums.

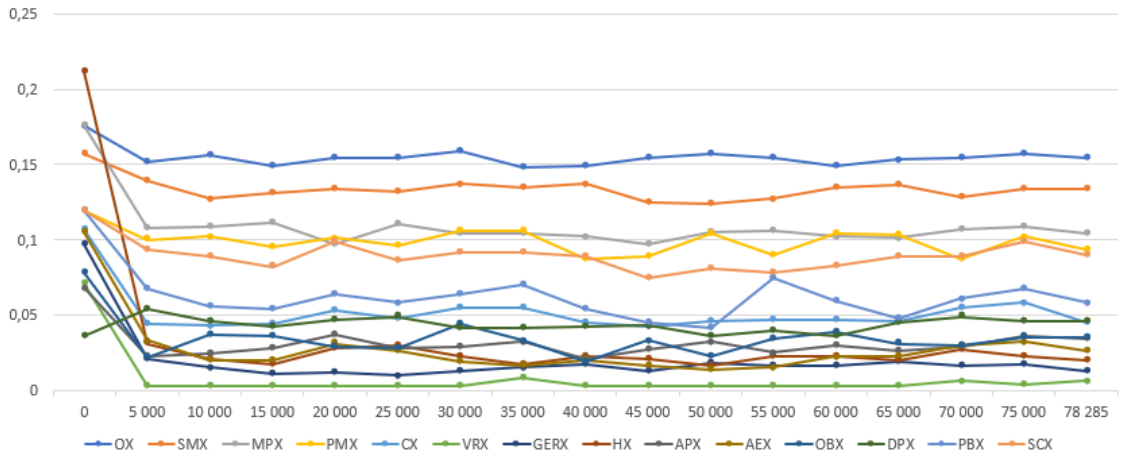

Figure 8.1: Long term behaviour of SRX

We were also comparing SRX with other operators running each operator for almost nine cycles (Fig. 8.2, Fig. 8.3). We were especially interested in operators leading the race over the test data. The statistics show that the SRX has the cheapest cost after the first cycle and leads right to the end. It also improves its cost even in later cycles. MPX improves faster in the first 20000 cycles but doesn't get close in cost to SRX having the second-worst. SCX, SMX, and OX have a close long-term result getting the closest to SRX, but SRX improves almost 20% even compared to them.

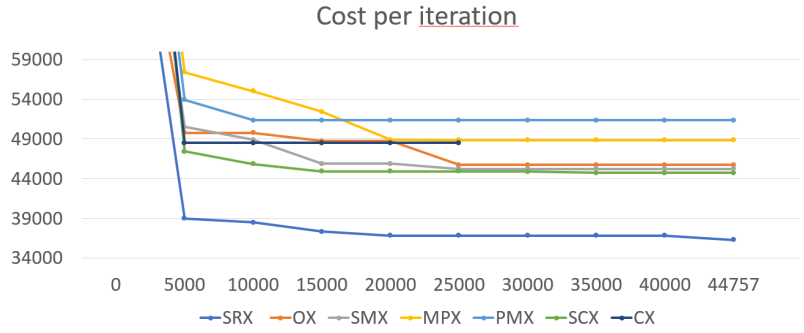

Figure 8.2: Comparison to SRX zoomed

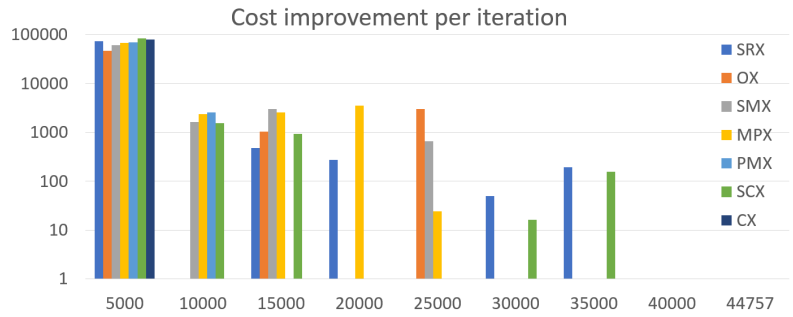

Figure 8.3: Comparison to SRX improvement logaritmic

On the bigger data set, SRX showed similar behavior, with the race stabilizing within the first cycle. However, HX tends to lead the race in the first cycle almost to the end when its results become comparable to OX's. After the first cycle, it slowly lost its position falling to the last position in multiple cycles. That's why we started to experiment with HX. Interestingly HX had a surprisingly good synergy with simplified Opt-2 step leader boost and reverse mutation. In the boost phase, the best specimens were searched for value pairs that are switching resulted in an improvement of cost and stopped after the first improvement. The mutation choose two positions randomly and reversed the order of the segment between the two positions. This version had 5% better results than SRX. However, HX based configuration reacted badly for edge distribution correction by resetting. This not just slowed but stopped convergence since children almost always inherited the genes of the parent with the better cost. Unique genes went extinct faster than usual and resetting couldn't bring back any of them.

This meant that edge distribution correction strongly depends on the crossover step and the fall of HX may have been caused by the incompatible mutation operator. This also shows that SRX is an optimal tool to find promising combinations and run tests saving a lot of time before experiments.

On the smaller subsets, the order of crossover operators varied strongly. However, some operators tended to lead the race while others tended to fall back. Ox, SMX, SPX, and HX were leading usually, while DPX, OX2 was always one of the last operators. The efficiency of the HX variant also varied. The less even the locations were distributed on the map the better was the performance of HX both in the race and alone. In some cases, it even got stuck in the first cycle without making any improvements later. SRX led the race in about 60% of tests in the long term, but HX still remains advantageous, because of its initial speed.

Chapter 9

Conclusions and Future Works

After our research in heuristics, we concluded that most accurate heuristics take too much time. They tend to be $O((c+s)^4)$ meaning they are strongly limited in time and customer count. Unfortunately, most neural networks fall into this category. There are a few metaheuristics relying on probabilistic methods, like genetic algorithms. Genetic algorithms are limited in population size, iteration count, and operator complexity since these are the main factors of their runtime requirement. Additional methods are necessary to unlock more control over algorithm convergence speed and gene diversity.

SRX is faster than other operators in the given environment. The data shows that local minima are not avoided but solved. It meant a 10-20 percent improvement in cost for $O((c+s)^2)$ additional time requirement. Further improvements are needed before genetic algorithms can be applied in the industry for larger combinatorial problems. We achieved a total runtime of $O((c+s)^3)$.

HX seems to be the strongest competitor of SRX since it has faster initial convergence. It could be further improved by the right gene distribution correction. Resetting seemed to be counterproductive, gene injection might be necessary. Statistical racing should be applied to other phases. It could be an efficient tool to discover new synergies competing with HX. SRX is also useful to identify data sets that HX is less efficient on compared to other operators.

We are going to apply statistical racing methods to other phases. We plan to add more variants to our operator pool. Higher level racing will be implemented between different evolutionary algorithm types. We are going to implement a hybrid pool where crossover operators are mixed with gene transfer operators. Boost should be mixed with bacterial mutation and infection. Mutation should be mixed by gene distribution correction and gene destruction. These phases and operators have similar purposes, therefor they efficiency can be compared.

More statistical data will be collected to further prove our statements. We are planning to collect data on gene distribution and its correlation with smooth convergence. We are going to design metrics for the balance of convergence and gene distribution. Some elements of the pools will be eliminated based on advanced statistics. We plan to develop methods for selecting the correct setup of the algorithm based on the statistics of the input data.

Acknowledgements

I would like to thank the help of all my colleges, friends, and family members for helping my work. Thanks to Dr. János Botzheim for the help in the scientific background, motivation, and correction of the hypothesis. Thanks to István Albert for the help with administration, management of progress and correction of phrasing and language. Thanks to my wife for help with difficult design decisions, correction of language, and moral support. Thanks to QLM, my workplace, for being flexible on work time and providing hardware. Finally but not least thanks to my friends and family for supporting me in these difficult times.

Bibliography

[1] M. Albayrak and N. Allahverdi. Development a new mutation operator to solve the traveling salesman problem by aid of genetic algorithms. *Expert Systems with Applications*, 38(3):1313–1320, 2011.

[2] E. C. Brown, C. T. Ragsdale, and A. E. Carter. A grouping genetic algorithm for the multiple traveling salesperson problem. *International Journal of Information Technology & Decision Making*, 6(2):333–347, 2007.

[3] B. Freisleben and P. Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *Proceedings of IEEE International Conference Evolutionary Computation*, pages 616–621. IEEE, 1996.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley PC., 1989.

[5] Zong-Ben Xu Kwong-Sak Leung, Hui-Dong Jin. An expanding self-organizing neural network for the traveling salesman problem. *Neuralcomputing*, (62):267–292, 2007.

[6] Pedro Laranga, Cindy Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[7] B. W. Lin. S. & Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21(2):498–516, 1973.

[8] John Geraghty Noraini Mohd Razali. Genetic algorithm performance with different selection strategies in solving tsp. *Proceedings of the World Congress on Engineering 2*, II, 2011.

[9] Adewole Philip, Akinwale Adio Taofiki, and Otunbanowo Kehinde. A genetic algorithm for solving travelling salesman problem. *International Journal of Advanced Computer Science and Applications*, 2(1):26–29, 2011.

[10] Sándor Szénási and Zoltán Vámossy. Implementation of a distributed genetic algorithm for parameter optimization in a cell nuclei detection project. *Acta Polytechnica Hungarica*, 10(4):59–86, 2013.

[11] K Shimojima T. Fukuda, N. Kubota. Virus-evolutionary genetic algorithm and its apllications totravelling salesman problem. *Evolutionary Computation: Theory And Applications*, pages 235–255, 1999.

[12] János Botzheim Tamás Bódis. Bacterial memetic algorithms for order picking routing problem with loading constraints. *Expert Systems with Applications*, 105:196–220, 2018.