FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Boška

Nature inspired algorithms for demand-responsive transport

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Fink, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to thank my supervisor, RNDr. Jiří Fink, Ph.D., for his guidance, patience, and helpful insights.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Title: Nature inspired algorithms for demand-responsive transport

Author: Ondřej Boška

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Fink, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis explores demand-responsive transport, where vehicles pick up and drop off passengers based on individual requests. We present a model of the Dial-A-Ride Problem (DARP), which uses real road networks from OpenStreetMaps. Customers ask for rides between two locations, providing their preferred departure time. The goal is to minimize both the operating cost and the customer waiting time. We implement three different encodings of an individual for genetic algorithms and three Ant Colony Optimization frameworks. We compare the results of these algorithms on our custom generated datasets.

Keywords: genetic algorithm, ant colony optimization, demand responsive transport, metaheuristics

Název práce: Přírodou inspirované algoritmy pro poptávkovou dopravu

Autor: Ondřej Boška

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Fink, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Práce se věnuje dopravě reagující na poptávku, kde vozidla vyzvedávají a vysazují pasažéry na základě individuálních požadavků. Představujeme model problému Dial-A-Ride (DARP), který využívá reálné silniční sítě z OpenStreetMaps. Cestující žádají dopravu mezi dvěma zastávkami a specifikují svůj požadovaný čas odjezdu. Cílem je minimalizovat jak provozní náklady, tak čekací dobu zákazníků. Implementujeme 3 různé kódování jedince v evolučním algoritmu a 3 frameworky Ant Colony Optimization. Výsledky těchto algoritmů porovnáváme na našich vlastních generovaných datech.

Klíčová slova: evoluční algoritmy, ant colony optimization, poptávková doprava, metaheuristiky

Contents

Introduction

Optimization of transportation-related problems has become a popular topic for many researchers [1]. The probably most famous *Travelling Salesman Problem (TSP)* focuses on finding the shortest path to visit all given cities and return back to the origin [2]. Over time, more sophisticated models were developed, like the *Vehicle Routing Problem (VRP).* The *VRP* was first proposed by Dantzig and Ramser for finding the optimal routing of multiple delivery trucks [3]. The goal is to minimize the travel cost, which can consist of distance or time traveled.

Many other models were derived from the *VRP* [4]. For instance, in a *Capacitated VRP*, the vehicle used has a maximum capacity, which cannot be exceeded [5]. In a *VRP with Time Windows*, each customer has a specific time interval in which they must be served [6]. The interval can be *soft*, in which case the vehicle can arrive outside the time interval, but incurs a penalty. The *Periodic VRP* can be used to optimize services like waste collection or public transport, where the vehicle routes repeat over a period of time [7].

The Dial-A-Ride Problem (*DARP*) was first mentioned by Psaraftis [8]. It is the most general form of the *Vehicle Routing Problem* [9]. In this model, passengers request transport between two stops, giving a desired departure time from *departure location* or a desired arrival time from *destination location.* The goal is to find a set of routes for a fleet of buses so that all the customer requests are satisfied while the cost of operation is minimized. All buses need to start and end at a given depot. In a more general form, there can be multiple depots, and each vehicle can start and end its route at a different one. The bus fleet can be *homogeneous*, meaning that all buses have the same capacity, or *heterogeneous*, where different vehicles can have different capacities or other parameters.

Several algorithms for solving the *DARP* have been proposed in recent years. Popular techniques include *branch-and-cut* [10], *tabu search* [11], *insertion based heuristics* [12] or *neighborhood search* [13]. In this thesis, we will focus on solving the problem using *nature-inspired metaheuristics*, namely *genetic algorithms* and the *Ant Colony Optimization.*

Genetic algorithms [14] are an optimization technique based on the idea of natural selection. They are based on an iterative improvement of solutions called *individuals.* Each individual has a *fitness value* assigned to it, representing the quality of the solution. Using genetic operators like *crossover* and *mutation*, new individuals are created from the individuals in previous generations called *parents.* The next generation is selected from these new individuals using a *selection* mechanism.

Ant Colony Optimization [15] is a metaheuristic inspired by the behavior of ants. Like ants laying pheromones while searching for food, the algorithm uses a *pheromone matrix* to guide the search. New solutions are constructed by traversing through the *search space*, where paths with higher pheromone values are more likely to be chosen. The more the ants traverse a path, the higher the pheromone value on the path is.

The goal of this thesis is to implement and compare various individual encodings for the genetic algorithm to address our *DARP* model. We seek to determine how different encodings influence the genetic algorithm's performance. We then implement a variant of the Ant Colony Optimization to solve the DARP and further compare it to the genetic algorithms. We again try using different *ACO* frameworks to find out how great a difference they have on optimization performance.

Our *DARP* model uses real-world coordinates, with distances and durations representing actual shortest paths in road networks. The generated routes then correspond to real roads and can be visualized using *geographic information systems* like *QGIS* [16].

In Chapter 1, we describe in detail our model of the *DARP*. In Chapters 2 and 3, we present several implementations of nature-inspired metaheuristics to find the optimal solution for the *DARP* model described earlier. Finally, in Chapter 4, we perform a series of experiments to compare these implementations. In Appendix A, we describe how to run all scripts and algorithms implemented and how to replicate the results of our experiments. In Appendix B, we briefly describe the details of the implementations.

1 Problem description

In this chapter, we present a specific model of the *DARP*. In Section 1.1, we describe the characteristics of the input data for our model. In Section 1.2, we describe the solution to the problem and its validity. In Section 1.3, the objective function is defined. In Section 1.4, we then talk about our types of datasets and how they are generated.

1.1 Input description

The problem instance is defined by the following data. Let R be the set of customer requests, and S the set of all the origin and departure locations. Each request $i \in R$ contains the desired departure time t_i (measured in seconds from the start of the simulation), the size of the group s_i, the coordinates of the origin location $o_i \in S$, and the coordinates of the destination location $a_i \in S$. The coordinates of the *depot* are denoted as D. All buses are homogeneous with capacity B_c, operational cost per kilometer B_o, and one-time fee for using the bus B_f. The size of any group cannot exceed the capacity of the bus. The distances between each location are given by the function $d : S \times S \to \mathbb{R}$ in kilometers. The durations between each location are given by the function $d_t : S \times S \to \mathbb{R}$ in seconds.

1.2 Solution

The solution is a set Z of routes, where each route is an ordered list of group indices in the order the bus handles their pick-ups and drop-offs.

For a solution to be valid, the following constraints must hold.

(1) Each group is handled by exactly one bus.

(2) Each group must be picked up and dropped off by a bus exactly once and as a whole.

(3) No bus carries more passengers than its capacity allows it to.

This means that each group can appear in only one route. Each group's index must be included in its corresponding route *exactly twice* - the first occurrence marks the pick-up, and the second occurrence marks the drop-off.

Every route needs to start and end at the depot. Since only one is available, it is implicitly added to every route's beginning and end.

1.3 Objective function

Since the goal is to minimize the costs, they form the foundation of the function. With r being a route within the solution Z, we transform it into an ordered list of locations the bus visits, including the depot. Then, with $\mathrm{r}_i \in S$ being the ith stop on the route, the total operational cost of the buses is given by the equation

$$\text{operational cost} \equiv \sum_{\mathrm{r} \in Z} (B_f + B_o \cdot \sum_{i=1}^{|r|} d(\mathrm{r}_{i-1}, \mathrm{r}_i)) \tag{1.1}$$

We must also account for the group's *delays* when evaluating a solution. The delay for each group is calculated as the difference between the time we drop the group off and the *expected arrival time*, which for group i is defined as $t_i + d_t(o_i, a_i)$.

We represent the "satisfaction" of the customers by the equation

$$\text{satisfaction cost} \equiv p \cdot \sum_{g \in R} d_g^2 \tag{1.2}$$

where p is the penalty constant for late arrival and d_g is the group's delay. The group's delay is squared to penalize larger delays more. The penalty constant is a hyperparameter and depends on the priority of handling all the customers as soon as possible at the expense of higher operational costs.

The objective function is then the sum of the *operational* and *satisfaction* cost.

$$\sum_{r \in Z} (B_f + B_o \cdot \sum_{i=1}^{|\mathrm{r}|} d(\mathrm{r}_{i-1}, \mathrm{r}_i)) + p \cdot \sum_{g \in R} d_g^2 \tag{1.3}$$

We can see that the problem of minimizing this objective function is NP-hard, because for $p = 0$, we are solving the *Capacitated Vehicle Routing Problem (CVRP)*, which is known to be NP-hard [17].

1.4 Input data generation

There are some already existing datasets, for example the one created and used by Ropke et. al. [18]. This dataset generates pick-up and drop-off locations in a $[0, 200] \times [0, 200]$ square, with a single depot in the center of the square. Each request has a *time window* assigned to it, defining the earliest possible departure time and the latest possible arrival time. Our model, however, uses real-world coordinates and has only the departure time defined in the customer requests. Therefore, we create our own benchmark data to compare the algorithms presented in the thesis.

We have two basic dataset types, *uniformly distributed* and *commute*.

The *uniformly distributed* data generator takes a geographical area represented in *OpenStreetMaps*[1], the number of requests to generate, the maximum size of a group in a request, and the latest possible departure time of a group. Based on these parameters, customer requests are generated, where departure and destination coordinates are randomly chosen platforms in the given area. The departure times and group sizes are chosen randomly. The number of platforms to sample from can be limited to force multiple customers to use the same platform for departure/destination. The depot is also chosen randomly from the available platforms. The bus type included in the depot can be changed in the generated file.

[1]`https://wiki.openstreetmap.org/wiki/Area`

The *commute* generator works analogously to the *uniformly distributed* generator but takes three areas instead: an *origin area*, *destination area*, and a *depot area*. The departure coordinates are randomly chosen platforms from the *origin area*, and the destination coordinates are randomly chosen platforms from the *destination area*. The depot coordinates are randomly chosen from the platforms in the *depot area*.

Customer requests are stored in *GeoJSON* file. The use of *GeoJSON* allows for simple visualization of the datasets in common geographical information systems.

The *distance* and *duration* matrices are generated using the Open Source Routing Machine (*OSRM*) API [19], that generates both matrices from a list of coordinates. The matrices are stored in separate *CSV* files.

2 Genetic algorithm

The concept of genetic algorithms to solve computationally hard problems was first presented by John Holland [14]. Starting with a *population* formed with initial (in most cases random) solutions called *individuals*, *crossover* and *mutation* operators are applied to gradually improve the fitness score of the population and thus find the best individual overall.

The most essential thing when designing a specific genetic algorithm is the encoding of an individual. Based on the encoding, we need to choose the genetic operators - *crossover* and *mutation.* The *crossover* operator takes two random individuals (*parents*) from the population and uses them to create two new individuals, which combine the properties of their parents. The *mutation* operator takes only one individual and makes a change within him. The change should usually be small and can be either completely random or try to improve the individual using some heuristic (called a *smart mutation*). The operators are performed on an individual or pair of individuals with a given probability (so in each generation, only some individuals participate in crossover or are mutated).

After the operators are applied, the new population is created using the *selection.* In our cases, the selection can either be a *roulette wheel selection* or a *tournament selection.* In the *roulette wheel selection*, we randomly sample individuals from the current population, but the individuals are weighted using their fitness value. When using the *tournament selection*, we take t (usually 2 or 3) random individuals from the population and compare their fitness values. The best one is chosen and added to the new population. This process is repeated until the new population is fully populated.

Apart from using the *roulette wheel* or *tournament* selection, we employ a strategy called *elitism.* After the new population is sampled using the *selection*, we choose n individuals randomly and replace them with the first n best individuals from the old population. This ensures that the best individual in each population survives into the next generation.

We present three different encodings of an individual:

- Individual as routes made of stops (EVO-STOPS)
- Individual as separate clustering and routing (EVO-CR)
- Individual as only clustering, with routing solved with greedy heuristic (EVO-H)

Each encoding has its separate population initialization and genetic operators, described in detail in sections 2.1 to 2.3. They share the fitness function, which is evaluated by transforming the individual into a solution and evaluating the objective function defined in 1.3.

2.1 Individual as routes of stops

2.1.1 Coding of an individual

The individual is encoded as a list of routes, where each route is a **list of stops** in the order the bus visits them.

Transforming the individual into a solution is done by simulation. We simulate every route - at each stop, we pick up any group waiting for the bus at that stop and drop off every group present on the bus that can be dropped off. If multiple groups are waiting at the stop, but not all can fit into the bus, we prioritize the groups that are waiting longer. At the end of the route, if any groups are left on the bus, we drop them off in the order of their departure time and return to the depot. After all route simulations are completed, we create another route by taking all the groups not picked up by any bus, picking them up, and immediately dropping them off in the order of their departure time. This ensures that the simulation returns a valid solution.

The difference between the encoding of a solution and our individual is that the individual uses stops instead of groups to encode a route. The main reason for introducing this change was that maintaining the correctness of the solution after performing the genetic operators on it would be problematic. Making a (semi-)random change in the individual could easily end up in the violation of constraints (2) or (3). Using stops instead of groups eliminates the problem of invalidating an individual since there is no invalid individual.

2.1.2 Initial population

We present two ways to create the initial population - *random* and *greedy*.

The *random* individual generator generates random sequences of stops and takes as a parameter the number of buses to use and the maximum allowed length of a route.

The *greedy* generator tries to generate already feasible routes. At each stop, we generate a set of reasonable next actions - picking up a new group or dropping off a group present on the bus. When considering groups to pick up, we only consider those whose departure time is close to the arrival time to their pick-up stop. For this, the generator takes a parameter defining how large this *time window* should be. If no more options exist, the bus ends the route by traveling to the depot. The generator can also be limited by parameters that define the maximum number of buses that can be used or the maximum number of groups that can be picked up within one route.

2.1.3 Genetic operators

Crossover

We use a simple *one-point crossover*. We take **one** random route from each individual, select a *crossover point*, and swap the right parts of the routes between the individuals.

We decided to use the crossover on only one of the routes for each individual. This is because even a change in one route is usually quite a big change - it can easily result in not picking up or dropping off some groups and, therefore, making the new individual largely penalized. However, it also makes the genetic algorithm much more exploratory.

Mutation

Unlike crossover, we have many options on what mutation operators to choose. We decided to implement multiple and randomly choose one at each mutation. The probability of each *operator* is determined by its weight, and the weights are set before an experiment as hyper-parameters.

The possible *operators* are:

- Delete a random stop from a randomly chosen route.
- Add a random stop to a randomly chosen route.
- Change a stop to a random one in a randomly chosen route.
- Reverse a random sub-route of a randomly chosen route. This is the main source of changing the stops' order.
- Shuffle the routes within the solution. This can greatly impact the simulation when evaluating the fitness function since each group is picked up by the first bus (in the simulation order) that arrives at their location at the right time. If more buses travel through the same stop, groups on that stop can be picked up by a completely different bus.
- "Smart mutation" - depending on the quality of the individual, it either tries to add an "unhandled" group to a random route, or it tries to perform a swap of a group between 2 routes. In both cases, we need to insert to a route both group's pick-up and drop-off. The maximum distance between these in the route is given as a parameter.

2.2 Individual as separate clustering and routing

The individual encoding as routes of stops is not ideal for several reasons. The crossover operator might break the solutions in datasets similar to the *commute*, where the bus should first pick up a number of groups and drop them off afterward. The encoding also does only say, which stops should the routes pass, but not which groups to pick up/drop off when visiting them. If more groups are waiting at the stop at the moment the bus arrives, all of them must be picked up. We will try to fix these issues in the second encoding.

2.2.1 Coding of an individual

The individual consists of two parts: the assignment of the groups to routes, which *clusters* the groups, and the order in which the groups are picked up and dropped off.

The groups to routes assignment is defined by an array, where the array indices represent the *group identifiers* and the values represent the *identifier of the route that handles the group*. The maximum number of routes is given as a parameter to limit the number of routes used.

The order of pick-ups and drop-offs is defined by an array, where every group occurs twice; the first occurrence marks the pick-up, and the second marks the drop-off.

To transform an individual into a solution, for each route, we first determine which groups are handled by that route, and then we construct the route by picking the group's indices in the order they are present in the individual's order-defining array. When constructing a route, if picking up the next group would violate the capacity constraint, we drop off groups with the earliest departure times until the bus has enough capacity to pick up the group.

2.2.2 Initial population

We generate the initial solutions randomly - given the maximum number of routes, we assign each group to a random route, and the order of the groups is created by a random shuffle.

2.2.3 Genetic operators

Crossover

In the crossover, we only cross the order of handling the groups. We first transform the order array into a permutation by adding $|R|$ to every second occurrence of each group. On this permutation, we use the *Partially Mapped Crossover (PMX)* [20]. We then transform the permutation back to the order array by subtracting $|R|$ from every second occurrence of each group.

Mutation

Mutation has multiple options for changing the individual. It can change the route assignment by either swapping two random groups between routes or assigning a random route to a random group, or it can change the order by either reversing a part of the order array or swapping two values in the order array.

2.3 Individual as only clustering with heuristic routing

The previous encoding of an individual comprised two parts: the clustering of the groups and the ordering of the groups within the routes. Therefore, the genetic algorithm has to optimize two objectives at once. We try to make the evolution simpler by only optimizing the group clustering, leaving the ordering to a greedy heuristic.

2.3.1 Coding of an individual

The individual comprises only the assignment of the groups to routes, similarly to 2.2.

To convert an individual into a solution, we first divide the groups into routes based on the individual. The order in which we handle the groups within a route is then calculated based on a greedy heuristic described by Baugh et al. [21]

We can think of the problem as a graph, where nodes are either group pick-ups or drop-offs. Each group corresponds to two different nodes. The edge between nodes i and j then marks the action of either picking up/dropping off the group corresponding to node j, when departing from the location corresponding to the pick-up/drop-off of group i.

The heuristic uses *breadth-first search (BFS)* with limited breadth and depth. When evaluating the route, we first find d "cheapest" possible options to travel to, where d is a heuristic depth parameter. We only choose those options that would not violate any constraints defined in Section 1.2. For each of the options, we then try to determine the cost of the resulting route if we choose this option. We do this by evaluating the cost of a route where, for d stops, we would always choose the "cheapest" option available. Based on the costs of these routes, we then select the best option overall. We then repeat this process until no options are available and all groups are picked up and dropped off.

The movement cost between two nodes is calculated as a weighted sum of the travel time between the two locations and the *time violation*. For pick-up nodes, the *time violation* is the amount of time either the bus had to wait for the group or the group had to wait for the bus. For drop-off nodes, it is equal to the delay of the dropped-off group. For a more detailed description, see Algorithm 1.

Algorithm 1 Move cost between nodes i and j

```
Require: $i, j, current\_time$
  $travel\_time \leftarrow durations[NodeToCoords(i), NodeToCoords(j)]$
  $arrival\_time \leftarrow current\_time + travel\_time$
  $group\_j \leftarrow NodeToGroup(j)$
  if $j$ is a pick-up node then
      $expected\_time \leftarrow group\_j.departure\_time$
  else
      $group\_duration \leftarrow durations[group\_j.start, group\_j.end]$
      $expected\_time \leftarrow group\_j.departure\_time + group\_duration$
  end if
  $tw\_viol \leftarrow |arrival\_time - expected\_time|$
  return $w_{travel\_time} \cdot travel\_time + w_{time\_window} \cdot tw\_viol$
```

2.3.2 Initial population

We generate the initial solutions randomly - given the maximum number of routes to use, we assign each group to a random route.

2.3.3 Genetic operators

Crossover

We use a uniform crossover - for each of the two individuals, we go through the group-routes assignment, and with a given probability, we change the current assignment to the one in the second individual.

Mutation

The mutation swaps two random groups between their routes or assigns a random route to a random group.

2.4 Hyperparameters

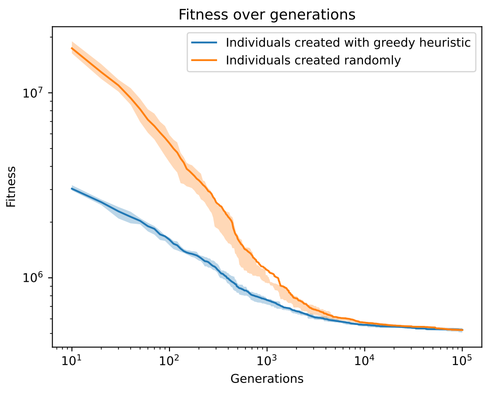
All of the algorithms above depend on the setting of multiple hyperparameters. We use a from of a *greedy grid search* to find the best settings. For example, we start by running a smaller grid search containing the probability of mutation, the probability of crossover, and the selection method. Other parameters are, at the moment, set heuristically. After finding the best values for parameters included in the grid search, we run another small grid search, usually containing one to three other hyperparameters. This is repeated for all hyperparameters.

The values tried in the grid search are chosen from a fixed interval. For example, for the crossover probability, we try values $[0.1, 0.2, 0.3, 0.4, 0.5]$. If the best value found by the grid search lies on the edge of the interval, we repeat the search with that interval shifted.

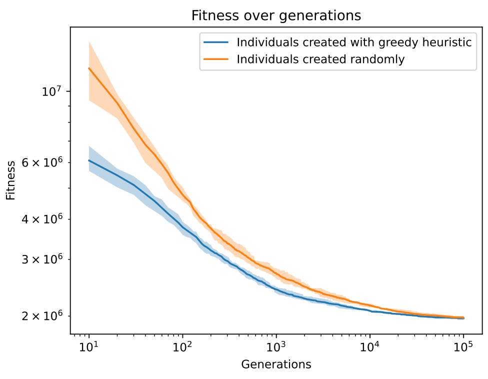
2.4.1 Individual as routes of stops

We describe the experiments in detail on the population initialization function. For both dataset types, we conduct two experiments, one with *greedy* initialization and one with *random* initialization. Both datasets are made of 50 customer requests, with group sizes between 1 and 10 persons and departure times in a 2-hour window. Each experiment runs the genetic algorithm 10 times. In figure 2.1, for each experiment, the mean fitness of the 10 runs is depicted as the primary line, accompanied by the first and third quartiles represented as a translucent region. Both x and y axes are logarithmic.

From figure 2.1, we see that for both datasets, the *greedy* initialization resulted in faster convergence at the beginning. We therefore decided to use the *greedy* initialization.



(a) Uniformly distributed dataset



(b) Commute dataset

Figure 2.1 Individual as stops - population initialization

The rest of the hyperparameter values were chosen in a similar way and are shown in the table 2.1.

Mutation probability	0.8
Crossover probability	0.2
Selection	tournament ($t = 2$)
Population size	40
Create individual function	greedy
Smart mutation weight	12
Reverse a route weight	3
Add/Delete/Change a stop randomly weight	9
Shuffle individual weight	2
Smart mutation maximum pick-up-drop-off distance	3

Table 2.1 Individual as stops - hyper-parameter settings

2.4.2 Individual as separate clustering and routing

The hyper-parameter settings for both dataset types were chosen experimentally and are shown in the table 2.2.

Mutation probability	0.8
Crossover probability	0.4
Selection	tournament ($t = 2$)
Population size	30
Mutate route assignments probability	0.5
Swap two groups between routes mutation probability	0.5
Reverse a part of the order array mutation probability	0.7

Table 2.2 Individual as cluster and route - hyper-parameter settings

2.4.3 Individual as only clustering with heuristic routing

The hyper-parameter settings for both dataset types were chosen experimentally and are shown in the table 2.3.

Mutation probability	0.8
Crossover probability	0.3
Selection	tournament ($t = 3$)
Population size	20
Travel time weight in heuristic cost	1
Time window violation weight in heuristic cost	2
Uniform crossover switch assignment probability	0.2
Swap two groups between routes mutation probability	0.5

Table 2.3 Individual as only clustering with heuristic routing - hyper-parameter settings

3 Ant Colony Optimization

The *Ant Colony Optimization (ACO)* algorithm is a probabilistic optimization metaheuristic first presented by Dorigo and Stützle [15]. The original algorithm was improved by Stützle and Hoos, who created the $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS)[22]. Blum and Dorigo then refined the $\mathcal{MM}$AS framework and created the *hyper-cube framework (HCF)* for *ACO*[23]. In this chapter, we compare these 3 frameworks on the *DARP*.

3.1 *ACO* frameworks

In our study, we implement three different *ACO* frameworks to evaluate their effectiveness in solving the *DARP*. All the frameworks need two functions specific to the problem: a function for creating new solutions and an attractiveness function to guide the solution construction process. The difference between each framework is described in the following sections.

3.1.1 Ant system

Ant System (AS) was the first *Ant Colony Optimization* algorithm. The algorithm is based on the behavior of ants. When searching for food, multiple ants scatter around their anthill. They lay down small amounts of pheromone to remember the way back. If their search is successful, they return to the anthill while laying down a much stronger layer of pheromone to mark the path to the finding. Other ants can then follow this pheromone trail instead of searching randomly. The pheromone, however, slowly evaporates, so the trail gets thinner for longer paths. This allows the ants to find the shortest paths to nearby food.

The *AS* builds on this metaphor. Our *ants* generate possible solutions by moving through the *search space*. When they create a solution, they update the *pheromone* stored in the *pheromone matrix* based on the fitness value of their solution. When new ants then move through the search space, they follow paths with higher pheromone levels, increasing the chance of finding a better solution. Apart from the pheromone levels, the ant also decides on the heuristic value of the path - *attractiveness*.

For pheromone matrix $\mathcal{T}$, the pheromone update is defined as $\mathcal{T} = (1-\rho)\mathcal{T} + \Delta\mathcal{T}$. Matrix $\Delta\mathcal{T}$ is calculated using the solutions generated in the current iteration as $\Delta\mathcal{T} = \sum_{k=1}^{ants} \Delta\mathcal{T}^k$, where

$$\Delta\tau_{ij}^k = \begin{cases} Q/L_k & \text{if edge } ij \text{ was used in the } k\text{th solution} \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

Q and ρ are hyperparameters, L_k is the fitness value of the kth solution. Therefore, for the pheromone update, all generated solutions are used.

3.1.2 $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System

Stützle and Hoos improved the original *Ant Systems* by introducing 3 differences from the original algorithm:

- Only the ant that found the best solution lays down the pheromone.
- The amount of pheromone on each trail is limited to an interval $[\tau_{min}, \tau_{max}]$, which helps to avoid stagnation of the algorithm. These pheromone bounds are updated during the optimization based on the fitness value of the current best solution.
- The pheromone matrix is initialized with the value τ_{max}. This makes the algorithm prefer exploration at the beginning of the run.

3.1.3 Hyper-Cube Framework

The *HCF* aims for a more robust behavior than $\mathcal{MMAS}$. Instead of changing the pheromone bound dynamically, in this framework, all pheromones are limited to an interval of $[0, 1]$. This is possible because of changes in the pheromone update rule. To update the pheromone, 3 solutions are used - *global-best* solution, *restart-best* solution, and *iteration-best* solution. Each solution has its own weight, κ_{gb}, κ_{rb}. and κ_{ib}. These weights must sum up to 1. The pheromone update $\Delta\mathcal{T}$ is then calculated as

$$\Delta\mathcal{T} = \kappa_{ib} \cdot \bar{s_{ib}} + \kappa_{rb} \cdot \bar{s_{rb}} + \kappa_{gb} \cdot \bar{s_{gb}} \tag{3.2}$$

$$\bar{s_*} = \begin{cases} 1 & \text{for edges } ij \text{ used in the solution} \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

The framework also introduces restarting the algorithm when it has converged. To measure how far the algorithm is from convergence, a convergence factor is calculated as

$$cf = 2 \cdot \left(\left(\frac{\sum_{\tau_{ij} \in \mathcal{T}} max(\tau_{max} - \tau_{ij}, \tau_{ij} - \tau_{min})}{|\mathcal{T}| \cdot (\tau_{max} - \tau_{min})} \right) - 0.5 \right) \tag{3.4}$$

When the pheromone is initialized with 0.5, the convergence factor is 0. The closer cf is to 1, the more the algorithm converges to a single solution. The κ values for the pheromone update are set based on the convergence factor value. This allows for changing the relative influence of the *iteration-best* and *restart-best* solutions based on how far the algorithm is from convergence. In addition, a Boolean variable *bs_update* is defined and becomes true when the algorithm reaches convergence. The kappa settings are defined in table 3.1.

	bs_update = False				*bs_update*
	$cf < 0.4$	$cf \in [0.4, 0.6)$	$cf \in [0.6, 0.8)$	$cf \geq 0.8$	$= True$
κ_{ib}	1	2/3	1/3	0	0
κ_{rb}	0	1/3	2/3	1	0
κ_{gb}	0	0	0	0	1

Table 3.1 Kappa values for pheromone update [23]

The entire framework is described in Algorithm 2.

Algorithm 2 Hyper-Cube Framework ACO [23]

```
Require: $\alpha > 0, \beta > 0, \rho \in (0,1), ants > 0, iterations > 0$
  $s_{gb} \leftarrow null, s_{rb} \leftarrow null$                ▷ global-best, restart-best solutions
  $cf \leftarrow 0.0, bs\_update \leftarrow False$
  for all $\tau_{ij} \in \mathcal{T}$ do $\tau_{ij} \leftarrow 0.5$
  for $iterations$ do
      $solutions \leftarrow CreateSolutions(ants, \alpha, \beta, \mathcal{T})$
      $s_{ib} \leftarrow argmin(Fitness(solutions))$
      $Update(s_{ib}, s_{gb}, s_{rb})$
      $cf \leftarrow ComputeConvergenceFactor(\mathcal{T})$
      if $bs\_update$ and $cf > 0.9999$ then
          $bs\_update \leftarrow False$, $s_{rb} \leftarrow null$
          for all $\tau_{ij} \in \mathcal{T}$ do $\tau_{ij} \leftarrow 0.5$
      else
          if $cf > 0.9999$ do $bs\_update \leftarrow True$
          $\Delta\mathcal{T} \leftarrow PheromoneUpdate(\mathcal{T}, s_{gb}, s_{rb}, s_{ib}, cf, bs\_update)$
          $\mathcal{T} \leftarrow (1-\rho)\mathcal{T} + \Delta\mathcal{T}$        ▷ Evaporate pheromone, apply the update
          for all $\tau_{ij} \in \mathcal{T}$ do $\tau_{ij} \leftarrow min(max(\tau_{min}, \tau_{ij}), \tau_{max})$
      end if
  end for
```

3.2 Pheromone matrix

We represent our *search space* as an oriented graph. Vertices are the group's pick-ups and drop-offs. The edges represent the traveling, e.g., the edge between the group's i drop-off and the group's j pick-up represents the path between the group's i destination location and the group's j departure location. We must also add a node for the depot. The solution is then a set of paths through the graph, where each path satisfies the constraints defined in 1.2.

The pheromone matrix is of size $(2|R|+1) \times (2|R|+1)$, where R is the set of all customer requests. For each group with id i, the pheromone at index i represents the group's pick-up, and the pheromone at index $i+|R|$ represents the group's drop-off. The last index represents the depot.

3.3 Creating solutions

An *ant* generates new routes until all the customer requests are handled. Each route starts by picking up the first unhandled group with the lowest departure time. We then create a set of all possible options: pick up a new group, drop off a group sitting in the bus, and, only if the bus is empty, return to the depot. The probability of choosing each option is based on the amount of pheromone between the two nodes τ and the attractiveness value of the transition ν. The probability of transition from vertex i to vertex j is then proportional to $\tau_{ij}^{\alpha} \cdot \nu_{ij}^{\beta}$, where α and β are hyperparameters specifying the weights of τ and ν. If the option to return to the depot is selected, the route ends.

Attractiveness

We use a simplified heuristic used in section 2.3. The attractiveness is equal to a weighted sum of the travel time between the stops and the *time violation* - how soon or late would the bus arrive.

When considering dropping off a group, we can adjust the attractiveness with a *drop-off bonus coefficient* hyperparameter. This coefficient is multiplied by the length of the current route. The whole bonus is then multiplied by the attractiveness of the drop-off nodes. This helps to drop the groups faster and, therefore, reduces delay penalties. We tried multiple variants of how this bonus could work (fixed value, first picking up multiple groups and then dropping them all off, or having different strategies for different ants). The variant used in Algorithm 3 worked the best with both dataset types.

When calculating attractiveness for returning to the depot, we have no *time violation* to consider in the weighted sum. Instead, we only multiply the travel time to the depot with the *depot attractiveness coefficient* hyperparameter to favor or disfavor the depot and end the current route.

Algorithm 3 Attractiveness between nodes i and j

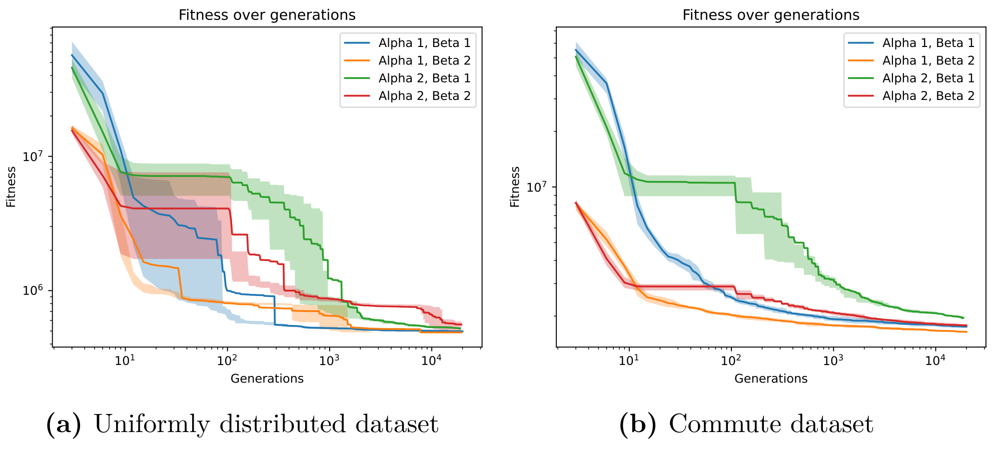
```
Require: $i, j, current\_route, current\_time$
  if $nnode$ is depot then
      $attractiveness \leftarrow travel\_time\_weight/duration[i, j]$
      return $attractiveness \cdot depot\_coef$
  end if
  $attractiveness \leftarrow 1/MoveCost(i, j, current\_time)$ ▷ MoveCost from EVO-H 1
  if $nnode$ is a drop-off node then
      $attractiveness \leftarrow attractiveness \cdot length(current\_route) \cdot drop\_off\_coef$
  end if
  return $attractiveness$
```

3.4 Hyperparameters

The hyper-parameter settings for both dataset types were chosen experimentally and are shown in the table 3.2.

The experiments were done the same way as in the section 2.4. For example, in figure 3.1, we show experiments for setting the α and β parameters.

For the commute dataset, the value of the *depot attractiveness coefficient* does not have much impact on the results. After dropping off all the groups in the destination area, the attractiveness of picking up any group in the source area is too low because the group would be picked up too late, while the depot attractiveness stays high. However, for the random dataset, a lower value, such as 0.1, lowers the number of buses used. This may be because we only consider returning to the depot when the bus is empty, but we do not penalize the waiting of an empty bus. However, the attractiveness function still considers the *time violation*, so if the bus were to wait for a long time, it would rather return to the depot. While we still need to account for the time violations since picking up groups with earlier departure times might be more favorable, we do not want their attractiveness to be significantly lower than the depot's attractiveness. Therefore, lowering the depot attractiveness by multiplying it with a small fixed constant works well.



(a) Uniformly distributed dataset **(b)** Commute dataset

Figure 3.1 Ant Colony Optimization - Alpha and Beta setting

α	1
β	2
ρ	0.1
Number of ants	20
Travel time weight in heuristic cost	1
Time violation weight in heuristic cost	2
Drop-off bonus coefficient	10
Depot attractiveness coefficient	0.1
Convergence factor to restart	0.9999

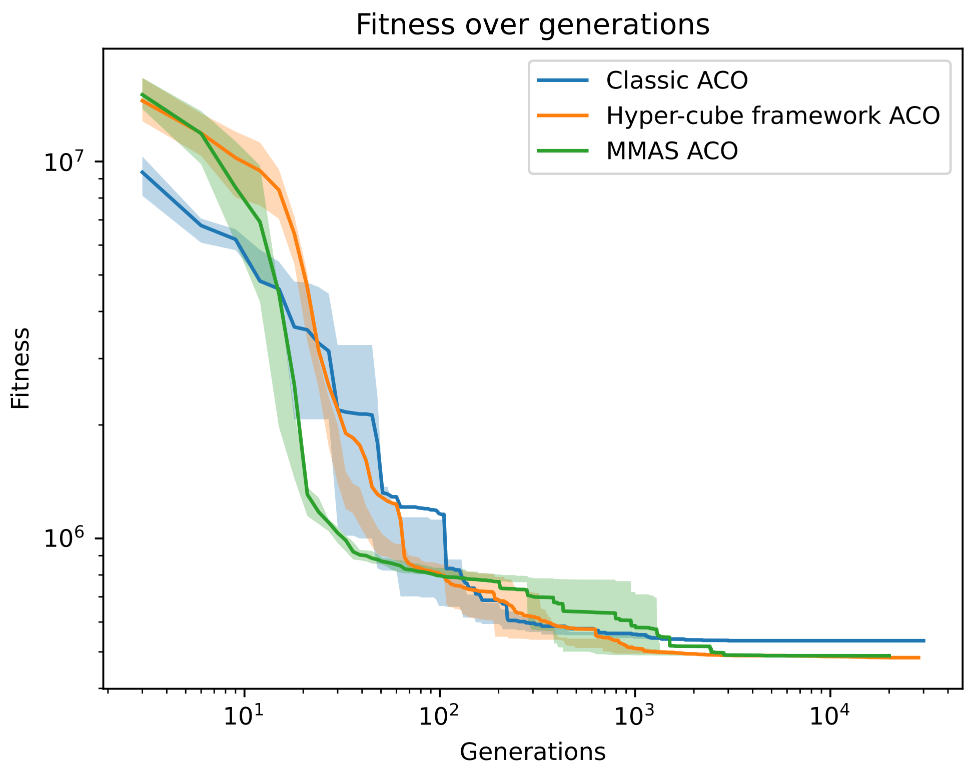
Table 3.2 Ant Colony Optimization - hyper-parameter settings

3.5 Comparing different frameworks

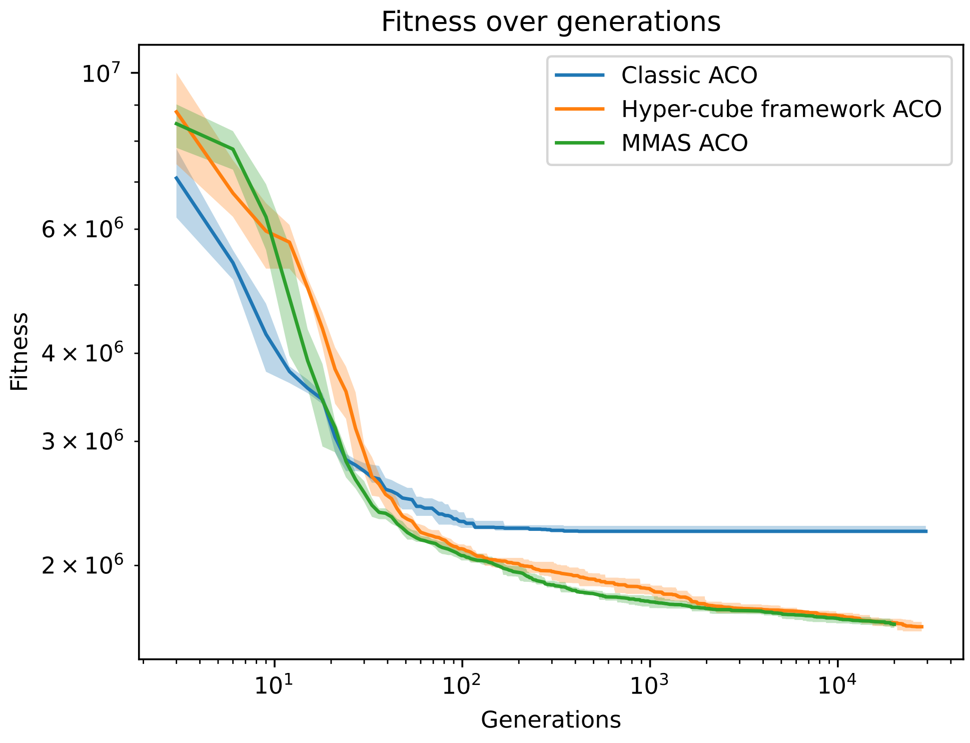
During development, we tried different versions of *ACO* frameworks - *Ant System (AS)*[15], $\mathcal{MM}$AS[22] and *HCF*[23]. A comparison of these frameworks on datasets with 50 customer requests can be seen in Figure 3.2. All experiments ran for 20 minutes.

An important thing to note is that *HCF* and $\mathcal{MM}AS$ use the same attractiveness function as described in 3.3. However, for the *AS*, we failed to find a single *drop-off bonus coefficient* or different attractiveness function for which both datasets would converge to a reasonable solution. We therefore set the coefficient to 5 for the commute dataset and 100 for the random dataset. This is a major problem for the *AS*, as it is less universal.

Based on the results, we decided to use only the *HCF* in the final experiments, as it consistently outperformed the *AS* and showed better characteristics during longer runs than the $\mathcal{MM}AS$.



(a) Uniformly distributed dataset



(b) Commute dataset

Figure 3.2 Ant Colony Optimization - Different Frameworks

4 Experimental results

We have four different optimization algorithms to choose from:

- *EVO-STOPS* - an evolutionary algorithm that encodes a solution as a list of routes of stops and tries to improve the solution by modifying these routes.
- *EVO-CR* - an evolutionary algorithm that encodes a solution using a mapping of groups to buses and an array defining the order of pick-ups and drop-offs.
- *EVO-H* - an evolutionary algorithm that uses an assignment of groups to buses and creates routes using greedy heuristics.
- *ACO-HCF* - iteratively creates new solutions using the *(Hyper-Cube Framework) Ant Colony Optimization* metaheuristic.

To compare the approaches, we generated benchmark datasets and ran all the algorithms on them. In this chapter, we present the results.

4.1 General parameters

Some dataset parameters are the same for all used datasets. Each group has between 1 and 10 people. The depot is always set in Prague. It is important to note that the first group in each route gets always picked up on time. This means that the location of the depot has no effect on the group's delays.

All the algorithms use the same fitness function, with the delay penalty constant equaling 0.01. All genetic algorithms have 10% of the population's best individuals (*elites*) passed from the old to the new population. Other parameters are set as described in Sections 2.4 and 3.4.

All the genetic algorithms can limit the number of buses used by a hyperparameter. However, the *ACO* does not have this option, as it generates the routes until all customer requests are satisfied. Therefore, all the limits on the number of buses set for each experiment apply only to the generic algorithms.

All experiments were performed on an AMD EPYC 7532 processor. Each algorithm ran for a fixed amount of time before being terminated, and the time of each experiment can be seen in the fitness plots.

For all experiments, we include a plot showing the progression of the best fitness value found by each approach. The value shown is the mean best fitness of 10 different runs, with the first and third quartiles depicted by the translucent region. The figure shows progression in time and has both the x and y axis on a logarithmic scale. The statistics about the best solutions returned by each algorithm are described in two tables. First shows the basic information: total costs (in thousands), kilometers traveled in total, size of the delay penalties in the fitness function (in thousands) and how much of the fitness value they take, and the number of buses used. The second table shows information about group delays - the maximum delay of a group and the average and median delays of all groups, respectively.

4.2 Commute dataset

In the commute dataset, 100 different customer groups depart from a random location in Pilsen and travel to a random location in Prague. The distance between the two cities is approximately 90 km. The departure times are randomly distributed in a four-hour time window. The available bus type can carry up to 80 people, its operating cost per kilometer is 80, and its fixed rental cost is 50000. The algorithms ran for exactly 90 minutes, after which they were terminated.

30 buses

When setting the bus limit (for generic algorithms) to 30, we see that the *ACO* performed the worst, since it failed to minimize the number of buses used. This resulted in the highest total costs and the greatest number of kilometers traveled. All genetic algorithms returned similar results. *EVO-CR* minimized the total costs the most while *EVO-H* handled the group delays slightly better.

We can compare the results to a situation where every group would use its own vehicle. The total distance traveled by all groups from their departure locations to their destinations is 10,112 km. All the genetic algorithms found solutions that reduced the total distance traveled. However, the *ACO* used more buses, resulting in a higher total distance traveled. It is important to note that the buses start from a depot in one city, pick up the groups, drive to another city, and then return to the original depot. Therefore, each bus travels between the cities twice. If we had a two-way commute scenario, where we would use the same buses to return the customers back to their original city, the savings would be much more significant.

20 buses

We can try to reduce total costs by limiting the number of buses. The results of the *ACO* algorithm stay the same, as the limit does not apply to it. *EVO-CR* performed the best, with the lowest total costs and distance traveled overall, and the lowest average delay of all genetic algorithms. The *EVO-H* solution is very similar. Both the maximum and average delay are nearly double the amount when the bus limit was set to 30. Even though the fitness values are higher for the lower number of buses limit, the solution is not strictly worse.

We were able to find a solution that travels 6900 km in total. In this solution, the buses used the highway between Pilsen and Prague 60 times, therefore spending ca. 4900 km on the highway and 2000 km in the cities distributing the passengers.

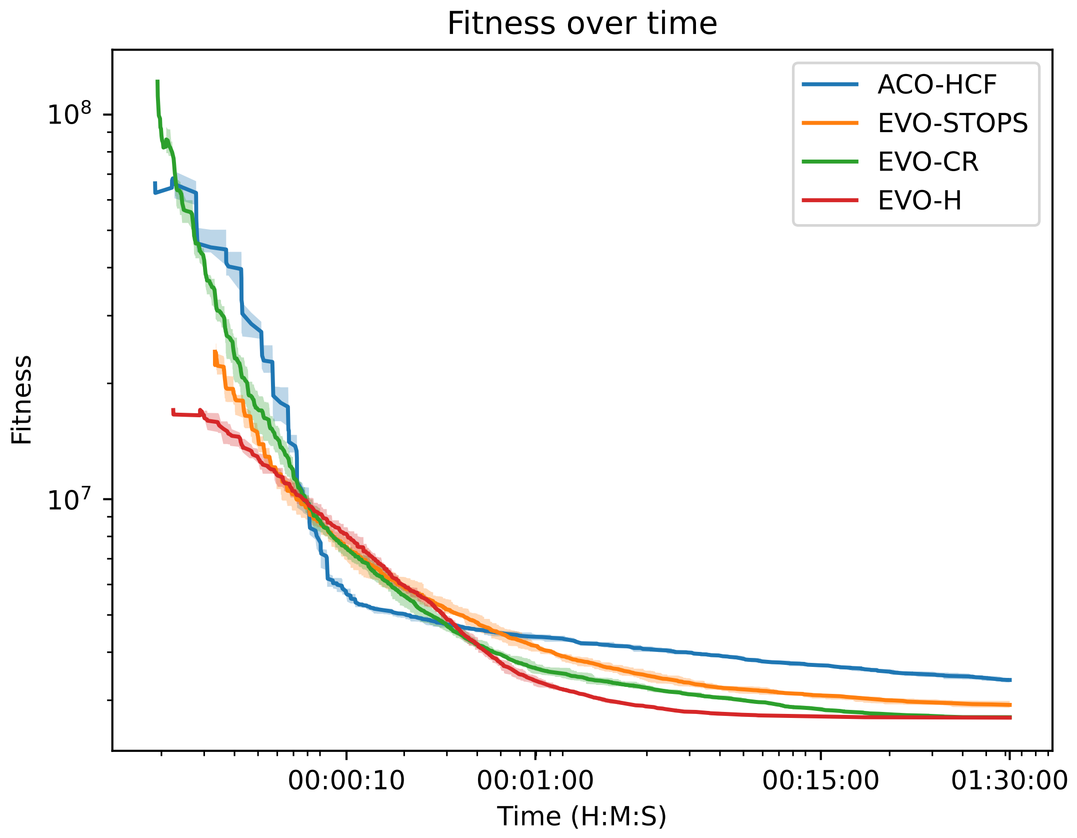


Figure 4.1 Commute dataset, 30 buses limit - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	2149	8732 km	612 (22%)	**29**
EVO-CR	**2140**	**8626 km**	524 (20%)	**29**
EVO-H	2240	9255 km	**429 (16%)**	30
ACO-HCF	2822	12147 km	472 (14%)	37

Table 4.1 Commute dataset, 30 buses limit - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	30m25s	11m26s	11m21s
EVO-CR	28m2s	10m10s	10m26s
EVO-H	**23m3s**	9m5s	8m56s
ACO-HCF	37m2s	**7m36s**	**5m1s**

Table 4.2 Commute dataset, 30 buses limit - Delay statistics

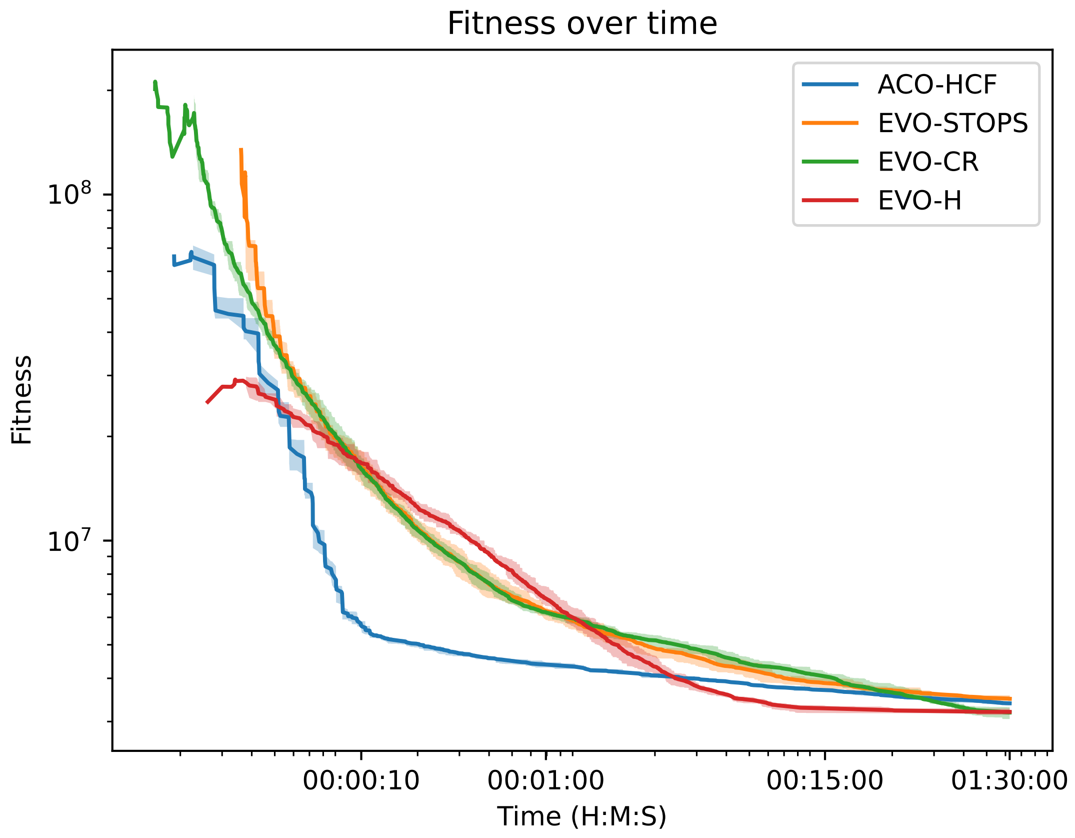


Figure 4.2 Commute dataset, 20 buses limit - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	1629	7233 km	1604 (50%)	21
EVO-CR	**1552**	**6900 km**	1464 (49%)	**20**
EVO-H	1554	6927 km	1472 (49%)	**20**
ACO-HCF	2822	12147 km	**472 (14%)**	37

Table 4.3 Commute dataset, 20 buses limit - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	51m54s	17m3s	16m0s
EVO-CR	42m22s	17m19s	16m58s
EVO-H	44m41s	17m32s	16m27s
ACO-HCF	**37m2s**	**7m36s**	**5m1s**

Table 4.4 Commute dataset, 20 buses limit - Delay statistics

4.3 Uniformly distributed dataset

4.3.1 100 customers

The uniformly distributed data set has both departure and destination locations scattered randomly throughout Prague, and departure times are also randomly distributed within a 5-hour time window. The available bus type can carry up to 40 people, its operating cost per kilometer is 60, and its fixed rental cost is 30000. The wall time was set to 90 minutes.

25 buses

When the upper limit of buses for genetic algorithms is set to 25, the *ACO* clearly outperforms all other approaches, as the limit is set too high. *EVO-CR* struggled the most with lowering the buses used, resulting in the highest overall costs. We can also see that *EVO-STOPS* fails the most in dealing with delay penalties, which are higher than for the *ACO* while using 5 more buses. *EVO-H* found a solution with minimal group delays, with the maximum delay being under 4 minutes.

16 buses

Lowering the upper limit for buses to 16 greatly improves the performance of all genetic algorithms. However, the *ACO* still managed to minimize the total costs the most. *EVO-STOPS* performed the worst in all aspects. *EVO-CR* and *EVO-H* performed similarly, with *EVO-CR* using a bus more, but having slightly shorter delays. The *ACO* run is the same as in the previous experiment.

All algorithms found solutions that have a median delay equal to 0 seconds. This means that more than half of the customers get picked up exactly on time and are immediately dropped off. For example, the solution returned by *ACO* never has 2 different customer groups on a bus at once. *EVO-CR* decided to pick up a group while another one was on the bus only 3 times.

To further analyze the solutions, we compare them with a simple greedy heuristic. With a fixed maximum delay for any group, we try to form routes by picking up the closest group and immediately dropping it off. To find the closest group, we use a weighted sum of travel time to the group's departure place and time violation, that is, how long either the bus waited for the group or the group waited for the bus. We choose only from the groups whose delay would not exceed the maximum delay set.

When setting the maximum delay to 15 minutes and the equal weight of travel time and time violation, the heuristic finds a solution that uses 27 buses, travels 2685 km, and costs 971k. The average delay is then 1 minute and 44 seconds. The best solution found by *ACO-HCF* uses only 14 buses and costs 566k, nearly half of what the greedy approach found. The average delay is also slightly shorter, only 41 seconds.

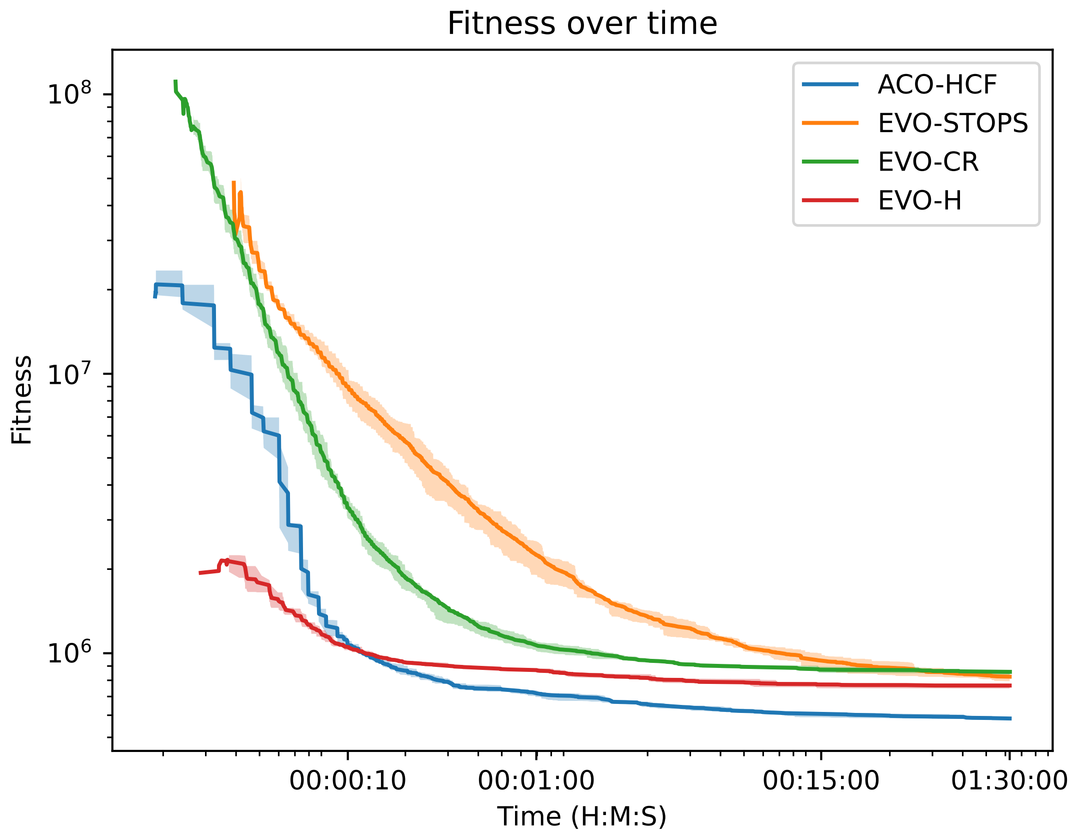


Figure 4.3 Uniformly distributed dataset, 25 buses limit - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	732	2698 km	18.9 (2%)	19
EVO-CR	807	2458 km	1.3 (0%)	22
EVO-H	718	2459 km	**0.9 (0%)**	19
ACO-HCF	**566**	**2431 km**	14.6 (3%)	**14**

Table 4.5 Uniformly distributed dataset, 25 buses limit - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	13m26s	35s	**0s**
EVO-CR	4m18s	7s	**0s**
EVO-H	**3m52s**	**5s**	**0s**
ACO-HCF	11m54s	41s	**0s**

Table 4.6 Uniformly distributed dataset, 25 buses limit - Delay statistics

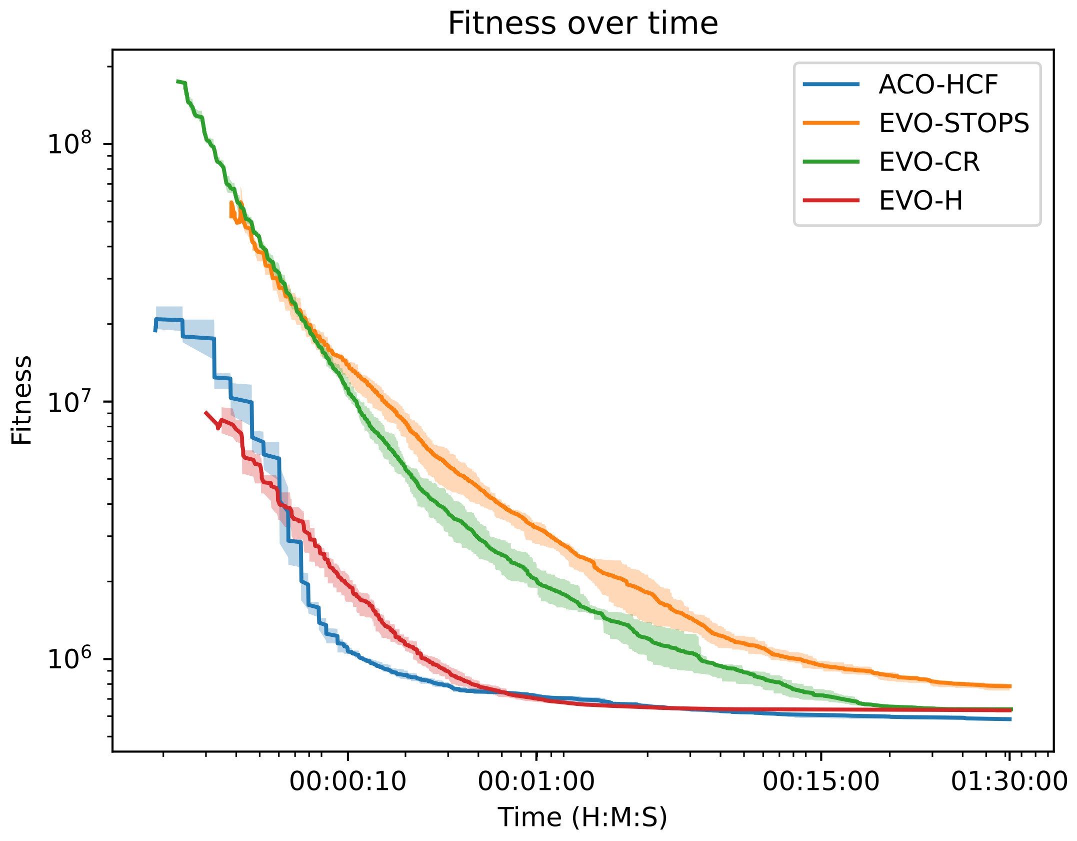


Figure 4.4 Uniformly distributed dataset, 16 buses limit - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	635	2577 km	108 (15%)	16
EVO-CR	626	2440 km	**6 (1%)**	16
EVO-H	597	2443 km	9 (1%)	15
ACO-HCF	**566**	**2431 km**	15 (3%)	**14**

Table 4.7 Uniformly distributed dataset, 16 buses limit - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	28m15s	2m12s	**0s**
EVO-CR	**8m19s**	**23s**	**0s**
EVO-H	8m33s	30s	**0s**
ACO-HCF	11m54s	41s	**0s**

Table 4.8 Uniformly distributed dataset, 16 buses limit - Delay statistics

4.3.2 150 customers

For 100 customer requests over a 5-hour time window, the solutions ended up looking more like a taxi service instead of utilizing the advantages of bus transport. To avoid this, we created a larger and denser dataset, with 150 different requests during a 2-hour period. The bus type used is the same as in the previous dataset. We also doubled the computation time to 3 hours.

When using the *ACO* approach, there is no direct way to limit the number of buses used, only by changing the delay penalty in the fitness function. In all *GA* approaches, we limited the number of buses to 25 to force them to utilize the buses. Therefore, the solution returned by the *ACO* is not directly comparable to the other returned solutions.

The *ACO* algorithm found a solution that uses 40 buses. The characteristics of the solution remain the same as in the previous dataset. There are never 2 different customer groups on a bus at once, and the median delay is still 0. Both the total costs and distance traveled are, therefore, the highest. However, it minimized the fitness function the most.

The best solution with a limited number of available buses was found by *EVO-H*, with both total costs and customer delays better than in solutions found by *EVO-STOPS* and *EVO-CR*. *EVO-STOPS* converged to a strictly worse solution, while also using a bus more (as to how see 2.1.1). *EVO-CR* failed to converge in 3 hours and probably could have found a better solution if it ran longer. This makes *EVO-H* a better choice for larger datasets.

We can also compare the results with the greedy heuristic approach described in 4.3.1. When we run it with a maximum delay of 30 minutes, the travel time weight equal to 100 a time violation weight equal to 1, we get a solution that uses 42 buses, costs almost 1500k and travels for 3750km. The average delay is 8 minutes and 17 seconds. So *ACO* found a solution with similar total costs, but smaller customer delays, while *EVO-H* found a much cheaper solution with similar customer delays.

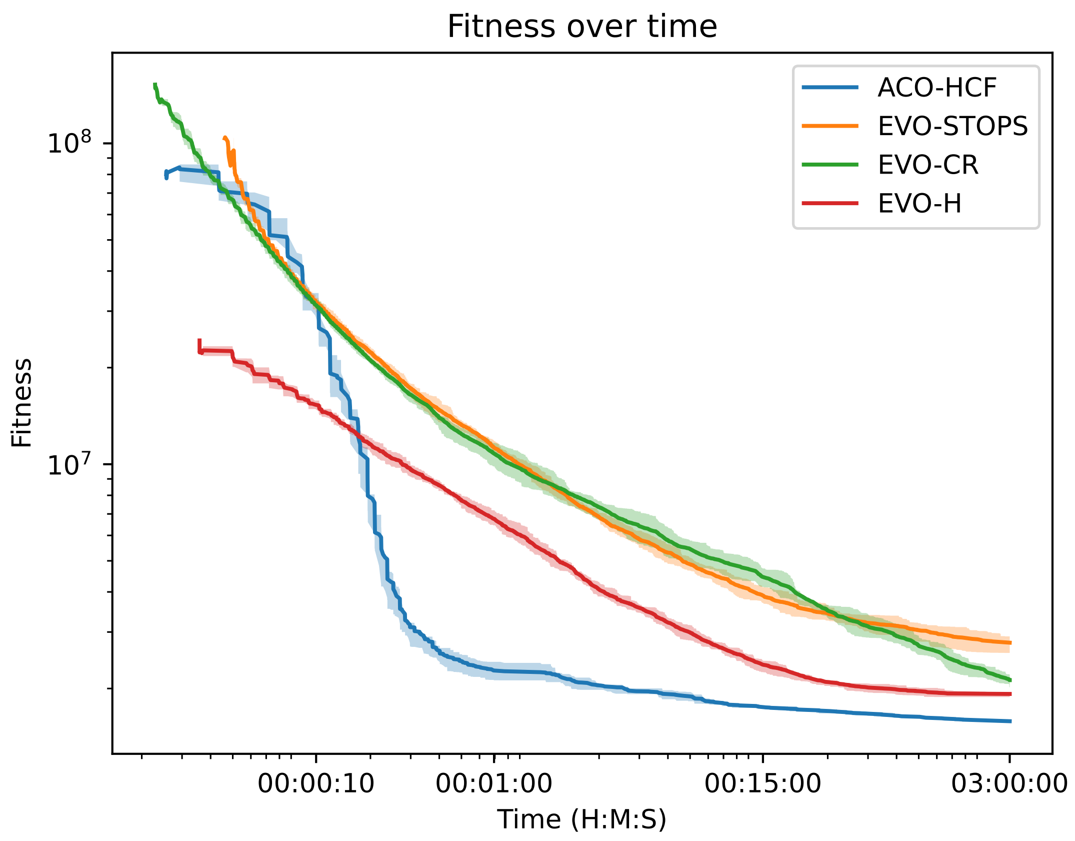


Figure 4.5 Uniformly distributed dataset, dense data - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	963	3049 km	1585 (62%)	26
EVO-CR	921	2850 km	971 (51%)	**25**
EVO-H	**919**	**2822 km**	**905 (50%)**	**25**
ACO-HCF	1427	3779 km	128 (8%)	40

Table 4.9 Uniformly distributed dataset, dense data - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	48m36s	13m4s	9m21s
EVO-CR	38m38s	10m11s	9m22s
EVO-H	**36m56s**	**9m30s**	7m25s
ACO-HCF	23m7s	2m16s	**0s**

Table 4.10 Uniformly distributed dataset, dense data - Delay statistics

4.4 Combined dataset

Finally, we combined both types of requests in a *combined dataset*. Approximately half of the 100 customers travel between Prague and Pilsen, while the other half travels either within Pilsen or Prague. Departure times are distributed randomly within a 4-hour time window. The available type of bus is the same as for the commute dataset. The upper limit for buses used for genetic algorithms was set to 20. The wall time was set to 120 minutes.

The performance of each algorithm reflects its performance in both previous datasets. The *ACO* performed the worst. When inspecting the best solution returned, it can be seen that the algorithm had the most trouble with the commute requests, most of the time failing to pick up multiple customers before traveling between cities. This resulted in the highest number of buses being used. However, it performed well with the requests within each of the cities, with the median delay being zero seconds. The genetic algorithms gave all surprisingly similar solutions. *EVO-CR* managed to reduce total costs the most, while *EVO-H* dealt the best with minimizing group delays. Both of the algorithms have the same maximum delay, which belongs to the same group.

For illustration purposes, in figure 4.6, we see a visualization of one of the routes found by *EVO-H*. The blue pin marks the depot, the green markers show group pick-ups, and the red markers show group drop-offs. The bus starts by picking up and dropping off one group in Prague, then travels with one group in the bus to Pilsen, where it handles three other groups and returns to Prague with two groups in the bus. In total, there are two pick-ups and three drop-offs in Prague and five pick-ups and four drop-offs in Pilsen.

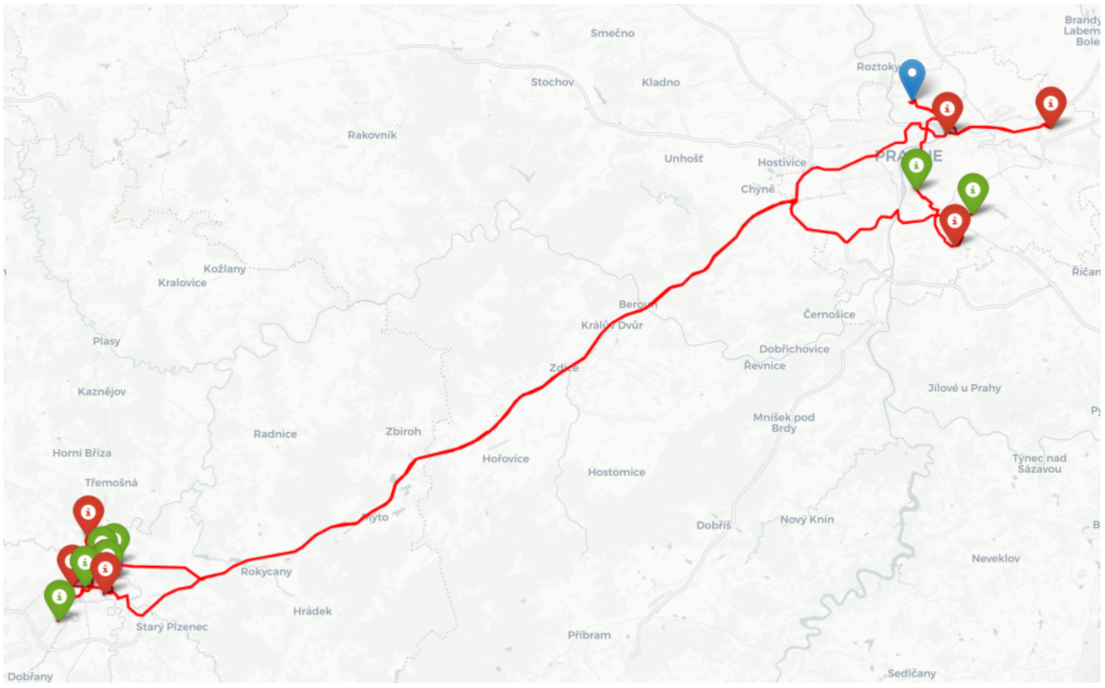


Figure 4.6 Visualization of an example route found by *EVO-H*

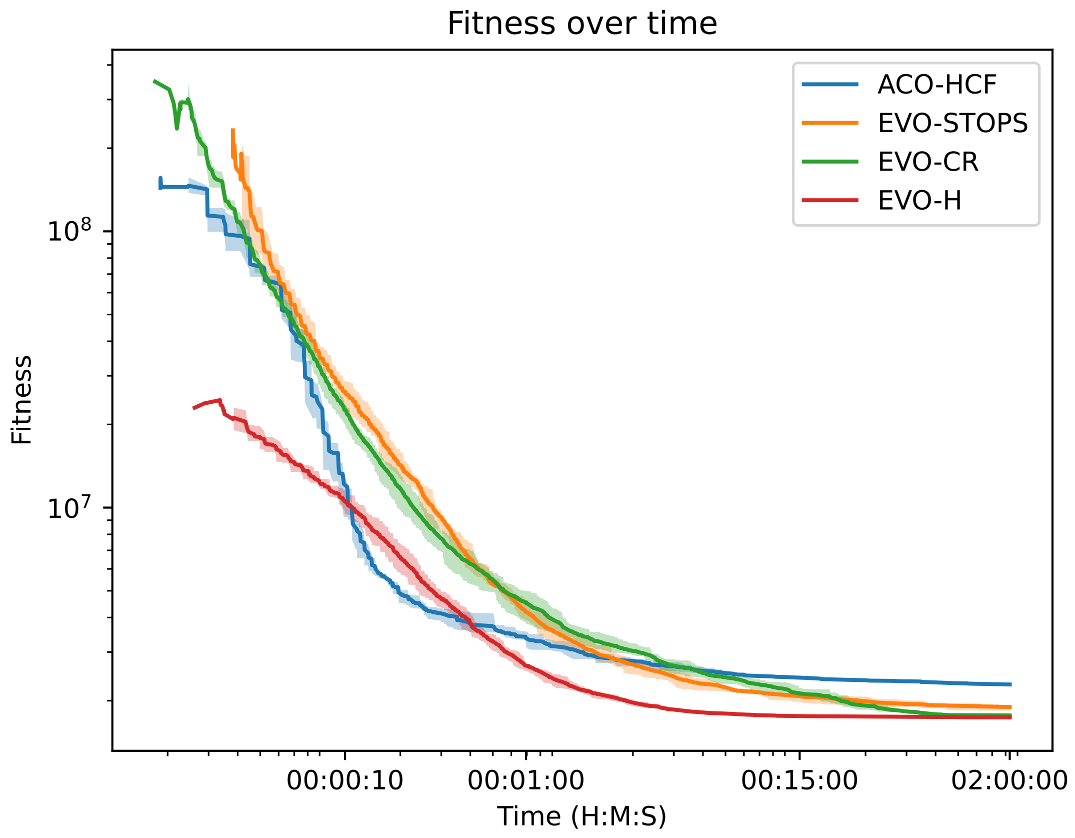


Figure 4.7 Mixed dataset - Fitness values in time

	Total costs	Distance traveled	Delay penalties	Buses used
EVO-STOPS	1597	6833 km	219 (12%)	21
EVO-CR	**1522**	**6528 km**	194 (11%)	20
EVO-H	1529	6613 km	**159 (9%)**	**20**
ACO-HCF	2060	9501 km	174 (8%)	26

Table 4.11 Mixed dataset - Route statistics

	Maximum delay	Average delay	Median delay
EVO-STOPS	**21m23s**	5m15s	3m18s
EVO-CR	21m47s	4m43s	2m6s
EVO-H	21m47s	**2m5s**	15s
ACO-HCF	34m11s	2m50s	**0s**

Table 4.12 Mixed dataset - Delay statistics

Conclusion

In this thesis, we implemented multiple approaches how to solve our model of the *Dial-A-Ride Problem*. We introduced three different encodings of an individual representing a *DARP* solution for a *genetic algorithm*, each with its own crossover and mutation operators. We also designed a function for creating solutions and an attractiveness function for the *Ant Colony Optimization* technique. We tried using these functions in three different known *ACO* frameworks and compared the results. We then compared all the presented approaches of the *DARP* optimization using our own generated datasets.

Our *ACO* algorithm performs the best for datasets, where dropping groups right after their pick-up resulted in the lowest cost. This was the case with less dense *uniformly distributed* datasets. It is, however, a suboptimal choice for datasets, where picking up multiple groups without dropping them off immediately is essential, like in the *commute* datasets. Overall, the algorithm appears to be the least universal of all the approaches presented.

The three *genetic algorithms* seem to be much more balanced, giving good results for all types of datasets. All the approaches strongly depend on the hyperparameter for setting the maximum number of buses used. When set too high, the algorithms have limited ways to lower the number of buses themselves. However, setting the limit lower can result in further minimization of operating costs, of course at the expense of customer satisfaction. In our experiments, the *EVO-STOPS* generally returned the worst results of all three. Using lists of stops proved to be ineffective, as the search space is much larger because buses can visit the same stop multiple times. The *EVO-H* approach returned very good results for all types of datasets. It also managed to converge faster than the other two and performed better on larger datasets. The *EVO-CR*, however, returned results of very similar quality, sometimes even surpassing the *EVO-H*. We are pleasantly surprised by its performance since the encoding does not use complicated crossover or mutation operators. Its transformation from an individual to a solution is also the simplest, without the need for any additional heuristics.

For future work, the algorithms could be tested and used on more datasets. One scenario could include commuting from densely populated areas to a single place, for example, a school or a workplace. The effect of some hyperparameters, such as the delay penalty constant in the fitness function, could be further tested. The algorithms could also be further extended to more complex models, being able to support multiple depots or heterogeneous fleets.

The implemented genetic algorithms have a great disadvantage as they need a good estimate of the number of buses used beforehand to return good solutions. Ideally, the algorithms should be universal for every dataset, and therefore, this number should be set automatically. This could be done perhaps by using a heuristic approach to get an estimate. We could also find a reasonably small range for this parameter and run the algorithms multiple times, each time with a different bus limit.

As the *EVO-CR* approach seems promising, some more complex genetic operators could be thought of and tested to help the algorithm converge faster.

For the *ACO*, other and more complex variants of the *attractiveness* function could be tested, to help the algorithm with datasets where the drop-off should not be immediate.

Bibliography

1. Díaz-Parra, Ocotlán; Ruiz-Vanoye, Jorge; Bernábe Loranca, María Beatríz; Fuentes-Penna, Alejandro; Barrera-Cámara, Ricardo A. A Survey of Transportation Problems. *Journal of Applied Mathematics*. 2014, vol. 2014. Available from DOI: 10.1155/2014/848129.

2. Laporte, Gilbert. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*. 1992, vol. 59, no. 2, pp. 231–247. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/0377-2217(92)90138-Y.

3. Dantzig, George B.; Ramser, John Hubert. The Truck Dispatching Problem. *Management Science*. 1959, vol. 6, pp. 80–91. Available also from: https://api.semanticscholar.org/CorpusID:154381552.

4. Adewumi, Aderemi; Adeleke, Olawale. A survey of recent advances in vehicle routing problems. *International Journal of System Assurance Engineering and Management*. 2016, vol. 9. Available from DOI: 10.1007/s13198-016-0493-4.

5. Laporte, Gilbert. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*. 1992, vol. 59, no. 3, pp. 345–358. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/0377-2217(92)90192-C.

6. Toth, Paolo; Vigo, Daniele. Models, relaxations and exact approaches for the capacitated vehicle routing problem. *Discrete Applied Mathematics*. 2002, vol. 123, no. 1, pp. 487–512. ISSN 0166-218X. Available from DOI: https://doi.org/10.1016/S0166-218X(01)00351-1.

7. Francis, Peter M.; Smilowitz, Karen R.; Tzur, Michal. The Period Vehicle Routing Problem and its Extensions. In: *The Vehicle Routing Problem: Latest Advances and New Challenges*. Ed. by Golden, Bruce; Raghavan, S.; Wasil, Edward. Boston, MA: Springer US, 2008, pp. 73–102. ISBN 978-0-387-77778-8. Available from DOI: 10.1007/978-0-387-77778-8_4.

8. Psaraftis, Harilaos N. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science*. 1980, vol. 14, no. 2, pp. 130–154.

9. Jørgensen, Rene Munk. *Dial-a-Ride*. 2003. ISBN 87-91137-10-1. PhD thesis. Technical University of Denmark.

10. Cordeau, Jean-François. A Branch-and-Cut Algorithm for the Dial-a-Ride Problem. *Operations Research*. 2006, vol. 54, no. 3, pp. 573–586. Available from DOI: 10.1287/opre.1060.0283.

11. Cordeau, Jean-François; Laporte, Gilbert. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*. 2003, vol. 37, no. 6, pp. 579–594. ISSN 0191-2615. Available from DOI: https://doi.org/10.1016/S0191-2615(02)00045-0.

12. Jaw, Jang-Jei; Odoni, Amedeo R.; Psaraftis, Harilaos N.; Wilson, Nigel H.M. A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows. *Transportation Research Part B: Methodological.* 1986, vol. 20, no. 3, pp. 243–257. ISSN 0191-2615. Available from DOI: `https://doi.org/10.1016/0191-2615(86)90020-2`.

13. Parragh, Sophie N.; Doerner, Karl F.; Hartl, Richard F. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research.* 2010, vol. 37, no. 6, pp. 1129–1138. ISSN 0305-0548. Available from DOI: `https://doi.org/10.1016/j.cor.2009.10.003`.

14. Holland, John H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992. ISBN 0262082136.

15. Dorigo, Marco; Birattari, Mauro. Ant Colony Optimization. In: *Encyclopedia of Machine Learning.* Ed. by Sammut, Claude; Webb, Geoffrey I. Boston, MA: Springer US, 2010, pp. 36–39. ISBN 978-0-387-30164-8. Available from DOI: `10.1007/978-0-387-30164-8_22`.

16. Team, QGIS Development. *QGIS Geographic Information System.* QGIS Association, [n.d.]. Available also from: `https://www.qgis.org`.

17. Toth, Paolo; Vigo, Daniele. *The Vehicle Routing Problem.* 2002. Available from DOI: `10.1137/1.9780898718515`.

18. Røpke, Stefan; Cordeau, Jean-François; Laporte, Gilbert. Models and a Branch-and-Cut Algorithm for Pickup and Delivery Problems with Time Windows. *Networks.* 2007, vol. 49, no. 4, pp. 258–272. ISSN 1097-0037. Available from DOI: `10.1002/net.20177`.

19. Luxen, Dennis; Vetter, Christian. Real-time routing with OpenStreetMap data. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* Chicago, Illinois: ACM, 2011, pp. 513–516. GIS '11. ISBN 978-1-4503-1031-4. Available from DOI: `10.1145/2093973.2094062`.

20. Goldberg, D. E.; Lingle, R. Alleles, loci and the traveling salesman problem. *Proceedings of an international conference on genetic algorithms and their applications.* 1985, vol. 154, pp. 154–159. Available also from: `https://api.semanticscholar.org/CorpusID:204211074`.

21. Baugh, John W.; Kakivaya, Gopala Krishna Reddy; Stone, John R. Intractability of the Dial-A-Ride Problem and a Multiobjective Solution Using Simulated Annealing. *Engineering Optimization.* 1998, vol. 30, pp. 91–123. Available also from: `https://api.semanticscholar.org/CorpusID:122010006`.

22. Stützle, Thomas; Hoos, Holger H. MAX–MIN Ant System. *Future Generation Computer Systems.* 2000, vol. 16, no. 8, pp. 889–914. ISSN 0167-739X. Available from DOI: `https://doi.org/10.1016/S0167-739X(00)00043-1`.

23. Blum, Christian; Dorigo, Marco. The Hyper-Cube Framework for Ant Colony Optimization. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*. 2004, vol. 34, pp. 1161–72. Available from DOI: `10.1109/TSMCB.2003.821450`.

List of Abbreviations

- **DARP** - Dial-A-Ride Problem
- **OSRM** - Open Source Routing Machine
- **EVO-STOPS** - Genetic algorithm implementation where the individual is encoded as a list of routes of stops
- **EVO-CR** - Genetic algorithm implementation where the individual is encoded as separate clustering and routing
- **EVO-H** - Genetic algorithm implementation where the individual represents only clustering and routing is done heuristically
- **ACO** - Ant Colony Optimization
- **HCF** - Hyper-Cube Framework, a metaheuristic algorithm derived from Ant Colony Optimization

A User Guide

This section contains the information necessary to run the optimization. The software is divided into 3 separate parts:

- Data generation
- Optimization algorithms
- Results evaluation

A.1 Prerequisites

For the *data generation* and *results evaluation* scripts, the user needs

- a Python interpreter in version 3.11.9. Compatibility with different versions is probable but not guaranteed. The libraries needed and their versions are listed in the *requirements.txt* file.
- a running instance of Open Source Routing Machine (OSRM) [19] service. The authors provide a demo server at `https://router.project-osrm.org`. To host the service, a *Makefile* with `prepare-server` and `run-server` targets is prepared in the *osrm* folder in the project files.

The *optimization modules* requires a Julia programming language compiler. The program was tested on Julia versions 1.7, 1.8 and 1.9. The packages needed are listed in the *Project.toml* file. The correct environment should be installed by running Julia with the `--project` argument.

A.2 Generating New Instance Data

The datasets used for running this thesis's experiments are available in the attachments in the *test_data* folder. If you want to generate new datasets, two scripts are available: *dataGeneratorUniform.py* and *dataGeneratorCommute.py*. Both scripts can be configured using command line arguments, described in detail when running the script with *--help*. Running the script generates a new folder with all the needed files: the *GeoJSON* file with customer requests, *CSV* file with the distance matrix, *CSV* file with the duration matrix and a *parameters.txt* file, where the settings of the generator are stored.

The script assumes that the *OSRM* service runs on your local machine. If your instance runs elsewhere, you can override this with the `--osrm_url` option, e.g.

```
python dataGeneratorCommute.py --osrm_url https://router.
    project-osrm.org
```

The coordinates used are sampled from public transport platforms in the given area. Note that when generating larger datasets, there must be enough unique platforms to use in that area. If the number of available platforms is lower than what is needed to sample, the generators exit with an error.

The bus type(s) available from the depot are hardcoded in the script. Either change them in the code or afterward in the generated *GeoJSON* file.

A.3 Running the optimization algorithms

All the Julia source files are stored in the *src* folder. Running the algorithms is done using the *EvoDARP.jl* runner script. All hyperparameters and other algorithm configurations are stored in the *config.yaml* file together with their descriptions. The program loads the configuration file from the *src* folder by default, but this can be overridden with the `-config` option.

The easiest way to run the program is with the following command:

```
julia EvoDARP.jl
```

The command compiles the program, loads the configuration from *config.yaml*, and runs the optimization. If you want to override some configuration variables directly from the command line, you can enter key-value pairs *variable=value* in the command. Some of the examples include:

```
julia EvoDARP.jl algorithm=evolution_stops cross_prob=0.5
   mut_prob=0.8 num_generations=20000

julia EvoDARP.jl input_data_folder=my_data algorithm=aco
   number_of_runs=10 num_ants=10 num_iterations=10000
```

The configuration file is divided into several sections. When running a certain optimization algorithm, all parameters from its section must be present. If some are missing, the program stops and prints an error message describing what is missing.

When the optimization finishes (by reaching maximum iterations or by exceeding the wall time), the runner stores in the output folder the following files:

- *best_solution.csv* with the overall best solution found.
- *config.yaml* with all the hyperparameter settings for the current run. This file can be used to reproduce the experiment.
- *run_i.fits*, where *i* is the run's id for each run, logging the current iteration, time, best fitness, and mean fitness of the population, respectively. Used for creating plots and analyzing the algorithm's convergence. The frequency of the logging can be set in the configuration.

The most important configuration values include the following.

- `algorithm` - which of the algorithms implemented is used for the optimization. Possible values are *evolution_stops*, *evolution_cr*, *evolution_heuristic* and *aco*.
- `input_data_folder` - the folder with the instance data. Example datasets are prepared in a `available_datasets` dictionary and can be selected by the *chosen_dataset* parameter. To create valid instance data, use one of the data generators attached. If you want to use your own dataset, it must be stored in a folder containing three files: *data.geojson* file with the customer requests, *distances.csv* file with the distances in meters for all coordinate pairs and *distances.csv* file with the durations in seconds for all the coordinate pairs. The *GeoJSON* file must conform to the JSON schema in Figure A.1.

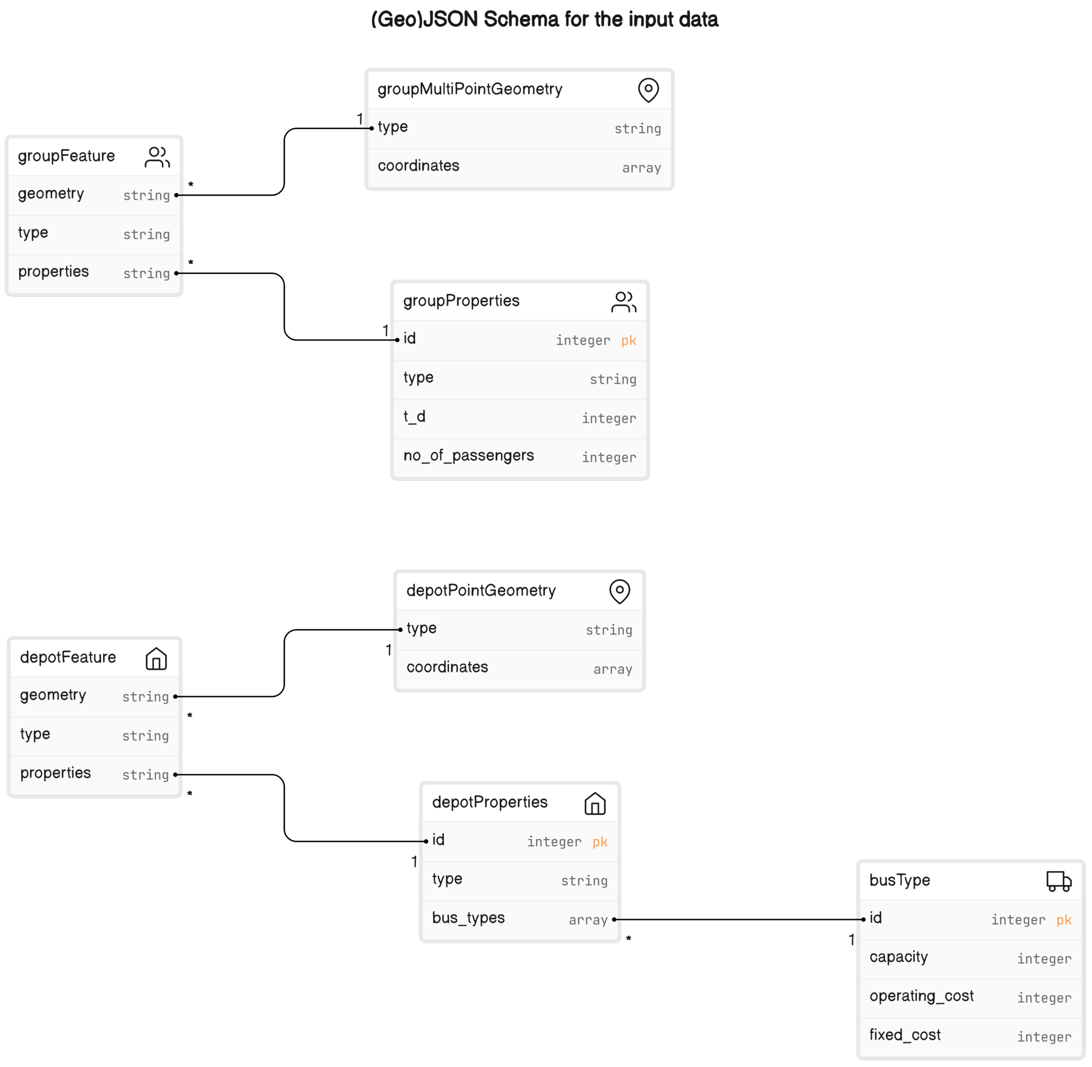


Figure A.1 Input GeoJSON schema visualization

- `output_data_folder` - folder where to store the results.
- `logs_frequency` - after how many iterations are the log files with best and mean fitnesses updated.
- `number_of_runs` - how many times will the algorithm run. All runs have a different random seed. Use the standard Julia `--threads` command line option to run the runs in parallel.
- `random_seed` - random seed to use. If not present or if its value is empty, the seed is chosen randomly. If `number_of_runs` is greater than one, each separate run has a seed of `number_of_runs + run_id`.
- `wall_time` - stops the algorithm if it runs for too long. Given in seconds.

A.4 Parsing the results

After the optimization finishes, the user can run the *resultsParser.py* script to analyze the results.

The standard way to run the script is with the following command:

```
python resultsParser.py [-f root_folder] [--visualize]
```

The script finds all folders with output from the optimization algorithm within the given folder and creates the following files:

- *report.txt* with basic statistics of the run's best solution, including the total costs, kilometers traveled, individual delays for each group, and the delay's mean and median.
- Plots visualizing the progress of the best run and the mean fitness of all the runs. The x-axis in the plots represents the number of generations, and the y-axis represents the fitness.

If the `-v` or `--visualize` option is used, the script also generates the following files. This option requires the *OSRM* service to be running.

- *routes.geojson* with the routes stored as *LineStrings*. This file can be used with software like *QGIS* to visualize the routes.
- *routes.html* with more basic visualization of the routes made with the *Folium* library.

To compare multiple experiments and plot them in one figure, use the script with the `-c` or `--compare` option:

```
python resultsParser.py -c [-f results_root_folder]
```

With this option, the script finds all the output folders within the root folder and creates a comparison plot. Again, the plots are generated in 2 variants: with the x-axis representing generations and the x-axis representing time elapsed. The labels in the plot's legend can be set by creating a *legend.txt* file in the experiment's results folder and storing the legend label in it.

There are also `--xlog` and `--ylog` options available to set the x-axis or y-axis scale as logarithmic in the plots. For larger experiments, to reduce the size of the plots, the user can use a `--compress` option. With this option, only points where fitness has changed are plotted.

Similarly to data generation scripts, the default *OSRM* host address can be overridden with the `--osrm_url` option.

The project also includes two small scripts to help analyze the results. The *separateCarsDistance.py* script takes 2 command line arguments, a path to a dataset and an output folder, and returns the distances between every group's departure and destination point, and the sum of all the distances. The *greedySpace-TimeHeuristic.py* takes the same 2 command line arguments, as well as additional `--max_delay`, `--travel_time_weight` and `--time_viol_weight` arguments. It uses a simple and naive greedy heuristic to find a solution for a given dataset. The heuristic is described in 4.3.1

B Developer Documentation

B.1 Data generators

Scripts for generating input data for the optimization algorithms are stored in the *data_generators* folder.

The *utils.py* file stores general functions used by all the generators. This includes:

- Sampling public transport platforms from *OpenStreetMaps* using the *Overpy* library and *Overpass* API. The query for the API is hard-coded in the file.
- Getting the distances and durations matrices using the *OSRM* API. This is done by simply using the *table service*[1] from the API.
- Writing the generated data to files.

The *dataGeneratorUniform.py* then uses the *utils* file to sample platforms from a given area, generates random attributes for the customer requests, and saves the generated data. The commute data generator in *dataGeneratorCommute.py* works similarly but takes 3 areas (from, to, depot) instead. There is also a *dataGeneratorSmall.py* script, which creates data with manually selected platforms and attributes. This dataset was used for testing purposes.

B.2 Optimization algorithms

All the algorithms are written in the Julia programming language. Each algorithm's source code is stored in a separate file. Other files include parsing the input data, common functions for all evolutionary algorithms (like selection or elitism), the fitness function, the runner script, configuration variables, and utility functions that did not fit elsewhere. Each of the following sections corresponds to one source file. Every function is described by a *docstring* and code comments where needed, so look directly at the source files for a detailed description.

B.2.1 Input data parsing

Input data parsing is done in the *input_parser.jl* file. The data is parsed by the `load_input` function. The function uses the *CSV* and *DataFrames* libraries to parse the distances and durations matrices, the *GeoJSON* library to load the input GeoJSON and *JSON3* and *JSONSchema* libraries to validate the input GeoJSON against a JSON Schema.

The group features are stored in a custom `Group` struct, and the depot is stored in a custom `Depot` struct holding the available bus types, also stored in a custom `BusType` struct.

The function then returns the parsed data in the form of a `InputData` struct, which holds a mapping between the group's IDs and their features (in a dictionary if the IDs in the input GeoJSON were not continuous), the depot, the distances and durations matrices and the coordinates to their id mapping.

[1]`https://project-osrm.org/docs/v5.24.0/api/#table-service`

B.2.2 Fitness function

The fitness function calculation is stored in the *fitness.jl* file. The fitness is evaluated using the `evaluate_fitness` function, which takes a valid solution and an input data instance. The solution must be stored as a `Vector{Vector{Int}}`, where each vector is a route consisting of pick-ups and drop-offs of groups (the first occurrence of the group's id marks its pick-up, the second occurrence marks the drop-off). The function separately calculates the bus costs (fixed cost per bus and operating cost per kilometer) and the delay penalties. These two values are then summed and returned. The boolean `multi_objective` parameter can be used to return these values separately in an array.

B.2.3 Genetic algorithms

Functions common for all the genetic algorithms are stored in the *evolution_base.jl* file. Implemented functions include the tournament selection, minimizing roulette wheel selection (by using the inverse of the fitness function), the method for creating an initial population from a *create individual* function, methods for applying crossover and mutation on the whole population based on the given probability, and the method for applying elitism. The `run_genetic_algorithm` function then implements the main loop of the genetic algorithm using these functions. This function also logs the algorithm's progress. When the `verbose` parameter is set to `true`, the progress is also printed to the standard output. The `map_fun` parameter can be used to replace the classic `map` function with e. g. the `parallel_map` from *utils.jl* to calculate the fitnesses of the population in parallel.

Each of the 3 implemented encodings for the genetic algorithm is implemented in its separate file. In all implementations, we first define the function for creating the initial individuals. The `cross` function then implements the crossover, and the `mutation` function implements the mutation. These functions are named the same for all encodings and when running the GA, the correct ones are chosen by Julia's multiple dispatch. Every encoding also implements a `individual_to_solution` method to convert the individual to an instance of the solution, for which the fitness can be calculated.

Individual as stops

The individual is stored in a `Vector{Vector{Int}}`, where each of the vectors is one route.

When converting an individual to a solution, we need a data structure that returns all the groups departing from a given place. We precompute this before the GA with the `get_departure_place_group_map` function and pass it to the fitness function as a parameter.

The submutations are implemented as nested functions within the `mutation` function. When the operator is called, one of these submutations is chosen using the `sample` function from the `StatsBase` package.

Individual as separate clustering and routing

For the *EVO-CR* encoding, the individual is stored in a `EvoCRIndividual` struct. The struct includes a `Vector{Int}` for mapping of groups to buses (routes) and a `Vector{Int}` for the pick up/drop off order (its length is twice the number of groups, values of the array are group's ids, each id is present exactly twice).

Individual as clustering with heuristic routing

The individual is encoded in a single `Vector{Int}` defining the mapping of groups to buses.

The greedy heuristic was inspired by the implementation by Baugh et al. [21]. While the original approach worked with *heuristic depth* fixed to 4, we define it as a parameter.

B.2.4 Ant colony optimization

The *ACO-HCF* implementation is stored in the *aco.jl* file. The file includes a `ant_colony_optimization` function with the main loop, `initialize_pheromone` and `update_pheromone` functions for working with the pheromone matrix, the `attractiveness` function and the `construct_solution` function for creating new solutions based on the pheromones. The generated solutions are stored as `Vector{Vector{Int}}` and are then directly passed to the fitness function.

The `attractiveness` function imports the greedy heuristic from *EVO-H* and namely uses the `move_cost` function for calculating the weighted sum between travel time and time violations.

B.2.5 Configuration

The configuration variables, stored in the *config.yaml* file, are parsed using the *config.jl* file. The file implements functions for loading the YAML file and storing it in a dictionary, validating it against a JSON Schema, extracting specific sections from the configuration (and putting all the variables in the top level for easier usage) or updating the configuration (which returns a new copy of the dictionary and is useful for e.g. grid searching). The configuration dictionary uses Julia's `Symbol` type as keys.

B.2.6 Runner

In the *EvoDARP.jl* file, a runner function is implemented for each optimization algorithm. Each function implements a `run_once` function, which prepares necessary data structures (if needed), stores the configuration used, sets the random seed, and runs the optimization algorithm once. The runner functions run this function in parallel (using `@threads`).

Two *main* functions are implemented; one is used when running the runner from the command line, and the other when running it in interactive mode (for example, in VS Code). Having the *interactive main* separate allowed for easier experimenting during development and algorithm fine-tuning (using interactive mode reduced the compile time needed during the development because only changed functions get recompiled).

B.3 Results parser

The script for creating experiment reports, plots, and visualization maps is stored in the *resultsParser.py* file in the folder *scripts* folder. It is divided into several regions for easier navigation. Important regions are *Experiment report*, *Map visualization*, and *Plotting*.

For creating the reports and maps, the script needs to load the input dataset. It does so by looking for the dataset folder in experiment's *config.yaml*, and from that folder, it loads the data into custom *dataclasses* `group`, `bus_type` and `depot`. It also reads the best solution from *best_solution.csv*. The data loading functions are imported from a separate *utils.py* file.

When creating the experiment reports, the script goes through the solution and runs a simulation for each route while storing kilometers traveled, operating costs, delays for each group, and the maximum number of people on the bus simultaneously. After simulating all the routes, the results are put together and written in a file.

There are two "main" functions for plotting. One creates plots for a single experiment, and the other creates comparison plots for all experiments in the root folder. They read the *.fits* files with fitness values in time and store them in *pandas* `Series`. The index of this series is either current generation or time (in milliseconds), depending on what should be visualized on the x-axis. The value is then the best fitness in said generation/time. The series are then compressed by removing adjacent entries with the same value if the `compress` argument is *true*. (this makes resulting plots significantly smaller in size). All the series are then put together into a `DataFrame`, from which `NaN` values are removed using `ffill()`. From this `DataFrame`, the final statistics (mean, first, and third quartile) are then calculated. Those statistics are then plotted using the *Matplotlib* library and stored in a *.pdf* file.

To visualize the routes, the script first uses the *OSRM* API to get the routes as GeoJSON LineStrings. The API has a convenient *route service*[2] for this. The *Folium* library is used to create a simple visualization in HTML. The routes are also stored in a GeoJSON together with the stops for more complex visualizations later.

[2] https://project-osrm.org/docs/v5.24.0/api/#route-service