UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Metaheuristic-based approach to waste collection optimization

Author:
Maksym PROTSYK

Supervisor:
Oleksii MOLCHANOVSKYI

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences and Information Technologies
Faculty of Applied Sciences

Lviv 2023

Declaration of Authorship

I, Maksym PROTSYK, declare that this thesis titled, "Metaheuristic-based approach to waste collection optimization" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Reject common sense to make the impossible possible."

Anahori Shimon

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Metaheuristic-based approach to waste collection optimization

by Maksym PROTSYK

Abstract

The main topic of the thesis is the development of a program that will allow waste companies to reduce their time on route planning by automatizing this task. This work includes the implementation of the framework for experiments and the comparison of different metaheuristic algorithms (Tabu search, Simulated annealing, Genetics algorithms) on synthetic and real data, which was required due to the non-polynomial complexity of the given problem.

All the code can be found in my GitHub repository

Acknowledgements

I want to express my gratitude to professor Oleksii Molchanovskyi for helping me throughout the development process of this project, to my parents and friends for everyday support, and to the community of the Applied Science Faculty of Ukrainian Catholic University for the amazing four years.

Contents

List of Figures

List of Tables

List of Abbreviations

SA	**S**imulated **A**nnealing
GA	**G**enetic **A**lgorithm
PSO	**P**article **S**warm **O**ptimization

Dedicated to my parents

Chapter 1

Introduction

1.1 Motivation

Today with the infinite possibilities provided by modern computers, it is strange that many waste management companies still use manual route planning for their vehicles. Firstly, this approach is very time-consuming, and secondly, it is prone to human error. In this thesis, I would like to present an automated approach to this task, which will help companies to reduce the amount of spent resources and time.

1.2 Problem statement

Before getting any further, let's formulate the statement of the problem. We are given n garbage containers, m garbage trucks, and l landfills. For each object, we know such parameters:

1. Garbage container
    - Capacity
    - Garbage type
    - Processing time (time needed by a truck to collect garbage from the container)
    - Location (i.e., distances and paths to other locations are known)
    - Time window (the earliest time when the container processing may start and the latest time when the container processing may finish)
2. Garbage truck
    - Maximum capacities of each garbage type that the truck can transport
    - Unloading time (time needed by a truck to unload the garbage at the landfill)
    - Speed
    - Fuel capacity
    - Fuel consumption per kilometer
    - Location
    - Time window (the earliest time when the truck may start the waste collecting process and the latest time that the truck may finish waste collecting and return to its starting location)
3. Landfill

- Location
- Time window (the earliest time when the truck may start the unloading process at the landfill and the latest time when the truck may finish it)

We need to plan routes for trucks in such a way that:

- All time windows are satisfied
- All trucks haven't exceeded their fuel capacities
- All trucks never exceed their garbage capacity
- All containers should be processed no more than once
- At the end, all trucks must be located at their starting locations with no garbage in them

Taking into account these conditions, we want to minimize the number of orphans (containers that were not processed), the total amount of fuel spent, and the maximum route duration.

It's actually fairly hard to minimize these three values at the same time because, in most cases, more processed containers mean more fuel and time spent. However, the number of orphans affects the well-being of people living near the containers, and the other two values affect the money spent by the waste collecting company. That's why, in my opinion, orphan minimization should be prioritized.

1.3 Goals

In fact, for the amount of data that we will be working with, the route planning problem couldn't be completely solved in an adequate amount of time (the reason for that will be discussed in later chapters). Thus my goals here are :

1. Implement different algorithms that return approximate solutions
2. Compare their efficiency on synthetic and real data (container locations in several cities)
3. Evaluate the results and choose the best algorithms for real-life cases

Chapter 2

Related Works

2.1 Metaheuristic

The problem we are facing is an extended version of the well-known traveling salesman problem. The problem statement is as follows: we are given n cities and distances between each two of them and need to find the shortest path which will visit every city and return us to the first one. If we think a little bit about the solution, the first thing that comes to mind is to check every possible path, but this algorithm requires $\mathcal{O}(n!)$ time and is not suitable even for cases with 20 cities. Actually, this problem belongs to the class of NP-hard problems, which are believed to not have a solution in polynomial time. So what can we do if even such a simple version of our problem couldn't be completely solved? Here metaheuristics can help us.

To use metaheuristic, we, in most cases, need to define three things:

1. Cost function, which needs to be minimized. In our case, an example of a cost function can be: $Cost(solution) = TotalFuelSpent$
2. Solution space - a space that consists of every valid solution to our problem, which will be searched. For our problem, this space is "all possible ways to construct routes for garbage trucks so that no solution requirement is violated."
3. Method of generating neighbor solutions. For example, swapping containers in the route.

Metaheuristic defines a strategy of solution space traversing, which starts the search from some initial solution (or multiple initial solutions) and leads to a solution with a lower cost. It is not guaranteed that the result of the search will be the global minimum; however, in most cases, the algorithm will still return a fairly good solution.

2.2 Metaheuristic algorithm examples

Now, let's look at some algorithms presented in the "Essentials of Metaheuristics" (Luke, 2013)

2.2.1 Tabu search

The algorithm is very simple: on each iteration of the search, we generate the list of neighbor solutions for the current one and change it to the neighbor with the lowest cost. To avoid trapping in the local minimum, we also keep track of the visited solutions in the so-called "Tabu list" (they won't be visited again for a fixed amount of iterations).

Algorithm 1 Tabu search

Algorithm parameters: *iterations*, *MaxQueueSize*
Input: initial solution
Output: improved solution

currentSolution = initial solution
bestSolution = initial solution
tabuQueue.Enqueue(*solution*);
for *i* in [0, *iterations* − 1] **do**
  neighbours = *GenerateNeighbors*(*currentSolution*)
  bestNeighbor = *None*
  for *sol* in *neighbours* **do**
    if *sol* in *tabuQueue* **then**
      continue
    end if
    if *bestNeighbour* is *None*) or (*Cost*(*sol*) < *Cost*(*bestNeighbour*) **then**
      bestNeighbour = *sol*
    end if
  end for
  if *bestNeighbour* is *None* **then**
    break
  end if
  if *Cost*(*bestNeighbour*) < *Cost*(*bestSolution*) **then**
    bestSolution = *bestNeighbour*
  end if
  tabuQueue.Enqueue(*bestNeighbour*)
  if *tabuQueue.Size*() > *MaxQueueSize* **then**
    tabuQueue.Dequeue()
  end if
  currentSolution = *bestNeighbour*
end for
return *bestSolution*

2.2.2 Simulated Annealing

This algorithm is similar to the cooling process of a heated metal object. At first, we can change the object's shape very easily, but as temperature decreases, this becomes harder and harder. Simulated annealing consists of several iterations. During every iteration, we calculate the temperature:

$$t = 1 - \frac{IterationNumber}{TotalIterations}$$

After that, we generate neighbor solutions and take a random one of them. If the solution is better, we take it and continue to the next iteration; in another case, we take the worse solution with probability, which is a function of the cost of the current solution, the cost of the neighbor solution, and the temperature. The most known probability function is:

$$p(CostCurrent, CostNew, t) = \exp(-\frac{CostNew - CostCurrent}{t})$$

2.2.3 Genetic algorithm

A genetic algorithm is an example of a metaheuristic, which works with multiple solutions at once. The current set of solutions is called a population. On every iteration, the algorithm randomly chooses two solutions (the probability of solution is usually related to its cost) and somehow combines them to create two child solutions. This operation is called a "crossover". After that, we perform some other random operations on each child, which is called "mutation". In such a way, we create a new population of the same size as a starting one.

Algorithm 2 Genetic algorithm

Input: initial set of solutions
Output: improved solution

currentPopulation = initial set of solutions
bestSolution = solution with the lowest cost from initial set
for i in [0, iterations - 1] **do**
  $newPopulation = []$
  for j in $[0, len(currentPopulation)/2 - 1]$ **do**
    $parent1 = ChooseRandom(currentPopultaion)$
    $parent2 = ChooseRandom(currentPopulation)$
    $child1 = Crossover(parent1, parent2)$
    $child2 = Crossover(parent2, parent1)$
    $child1 = Mutation(child1)$
    $child2 = Mutation(child2)$
    if $Cost(child1) < Cost(bestSolution)$ **then**
      $bestSolution = child1$
    end if
    if $Cost(child2) < Cost(bestSolution)$ **then**
      $bestSolution = child2$
    end if
    $newPopulation.Add(child1)$
    $newPopulation.Add(child2)$
  end for
  $currentPopulation = newPopulation$
end for
return *bestSolution*

2.2.4 Particle swarm optimization

Similar to GA, Particle swarm optimization works with a set of solutions instead of just one. However, in this algorithm, we don't replace old solutions with new ones but just move them through the solution space in the direction of better ones. For example, if our solution can be described as a vector of n-th dimension $x = (x_0, x_1, ..., x_{n-1})$ and we want to move it in the direction of some better solution $y = (y_0, y_1, ..., y_{n-1})$ we can define speed as $v = y - x$ and replace x with $x + kv$. In the case of the PSO algorithm, the calculations of speed are a little bit more complicated because we try to move towards several solutions at once. Here is the pseudo-code of the algorithm:

Algorithm 3 Particle swarm optimization algorithm

Input: initial set of solutions (vectors of n-th dimension) (its size is m)

Parameters α, β, γ, δ, $jumpSize$ (real-valued numbers), $informatnsNum$ - number of informants solutions for each solution, $iterationsNum$

Output: improved solution

$speeds$ = array of size m containing random speed of m-th solution
$informants$ = array with randomly chosen $informantsNum$ indices from 0 to m-1 for each solution (its shape is $(m, informantsNum)$)
$solutions$ = initial set of solutions
$bestFoundSolutions$ = initial set of solutions;
for i in $[0, iterationsNum - 1]$ **do**
  for j in $[0, m - 1]$ **do**
    $bestInformant$ = solution with the minimal cost from $bestFoundSolutions$ with an index equal to the index of some informant from $informants[j]$
    $bestSolution$ = solution with the minimal cost from $bestFoundSolutions$
    $currentBestSolution = bestFoundSolutions[j]$
    $informantSpeed = bestInformant - solutions[j]$
    $bestSpeed = bestSolution - solutions[j]$
    $currentBestSpeed = currentBestSolution - solutions[j]$
    for dim in [0, n-1] **do**
      $b = Random(0, \beta)$
      $c = Random(0, \gamma)$
      $d = Random(0, \delta)$
      $speeds[j][dim] = \alpha * speeds[j][dim] + b * informantSpeed[dim] + c * bestSpeed[dim] + d * currentBestSpeed[dim]$
    end for
    $solutions[j] = solution[j] + jumpSize * speeds[j]$
    if $Cost(solutions[j]) < Cost(bestFoundSolutions[j])$ **then**
      $bestFoundSolutions[j] = solutions[j]$
    end if
  end for
end for
return the best solution from $bestFoundSolutions$

It is obvious that this algorithm in its initial format won't work on discrete data like in our problem. However, it is possible to adapt it for the discrete case, which was done, for example, in **A discrete version of particle swarm optimization for flowshop scheduling problems**(Liao, Chao-Tang Tseng, and Luarn, 2007)

2.3 Related publications

Here are some publications related to using metaheuristics for the waste collection problem that I found interesting.

"Metaheuristics for a bi-objective location-routing problem in waste collection management" (Farrokhi-Asl et al., 2016) and **"A hybrid genetic algorithm for waste collection problem by heterogeneous fleet of vehicles with multiple separated compartments" (Rabbani, Farrokhi-Asl, and Rafiei, 2016)** are very similar due to the same problem statement and used algorithms. Here each truck visits several customers with garbage containers and then visits several disposal facilities to get rid of the collected garbage. That means each truck performs only one trip and

then returns to the starting location. Both papers present fairly complex algorithms (non-dominated sorting Genetic algorithm and multi-objective particle swarm optimization in the first one and Genetic algorithm in the second). The main difference is that the first paper is focused not on the route minimization for the given set of locations but on the way to distribute disposal facilities in the best way possible.

In **"Metaheuristics for the waste collection vehicle routing problem in the urban areas" (Stanković et al., 2020)**, the authors implemented four metaheuristic algorithms: Simulated annealing, Genetic algorithm, Particle swarm optimization, and Ant colony optimization. However, the problem they are working on still isn't the same as mine because the truck, as in the previous publication, performs only one trip. Moreover, they test the algorithm on a very small number of locations.

In **"The applications of multiple route optimization heuristics and meta-heuristic algorithms to solid waste transportation: A case study in Turkey" (Derecia and Karabekmez, 2022)**, the problem is close to my case because the authors work on larger real-life data. However, they still make only one trip for each truck. The implemented metaheuristic algorithms here are less complex than in the previous two publications, but their amount is greater: Simulated annealing, Greedy descent, Guided local search, and Tabu search algorithms.

Chapter 3

Implementation details

3.1 Framework architecture

Due to my goal of comparing different initial algorithms, search algorithms, and neighbor generation methods, I needed to implement an easily extendable framework, which could allow me to swap parts of the complete algorithm easily. I decided to use C++ as the main language, but due to a lack of easy-to-use and modern plotting libraries, I have chosen Python for the visualization task.

3.1.1 Main classes

Firstly, I'll describe the main classes used in my program

- **Container** - a class that contains garbage container parameters mentioned in the problem statement
- **GarbageTruck** - a class that contains garbage truck parameters
- **Landfill** - a class that contains landfill parameters
- **Problem** - a class that reads a config and input data from respective files and contains all the **Container**, **GarbageTruck**, and **Landfill** objects. This class allows to access objects by their ids (indices of objects in respective lists). It is implemented as a singleton because the data contained in it is required in almost every part of the framework.
- **Exportation** - every route consists of several exportations. **Exportation** is a trip where a truck collects waste from some containers and unloads it at some landfill. **Exportation** contains ids of the processed containers, truck, and landfill. It also can check if the requirements of the problem statement are satisfied for this trip.
- **Route** - class that contains several exportations. This class also can check if the problem requirements are satisfied for the given finish fuel and time.
- **Solution** - this class contains information about a particular solution: routes, orphan containers, and unused trucks. It also can be used to calculate the total time, fuel, and distance of the solution.
- **BaseSolution** - a base class for all classes used to create an initial solution, contains only one virtual method: "CreateSolution"
- **Action** - a base class for all classes used to perform actions (swapping containers in routes, moving them, etc.) on a solution. This class contains three virtual methods:

1. "PerformAction" - performs the action on the given solution if possible (satisfies all the problem requirements) and does nothing if it is not. Returns a boolean value, which tells if the action was successfully performed.
2. "GetAffectedIds" - returns the ids of containers that will be affected by this action
3. "ExpectedDiff" - the expected difference in total fuel spent after performing the action

- **Heuristic** - base class for classes that generate actions that lead to neighbor solutions.
- **Search** - base class for classes that perform metaheuristic searches. Contains used **Heuristic** and only one virtual method, "search", which takes an initial solution as an argument and returns the newly found one.

3.1.2 General pipeline

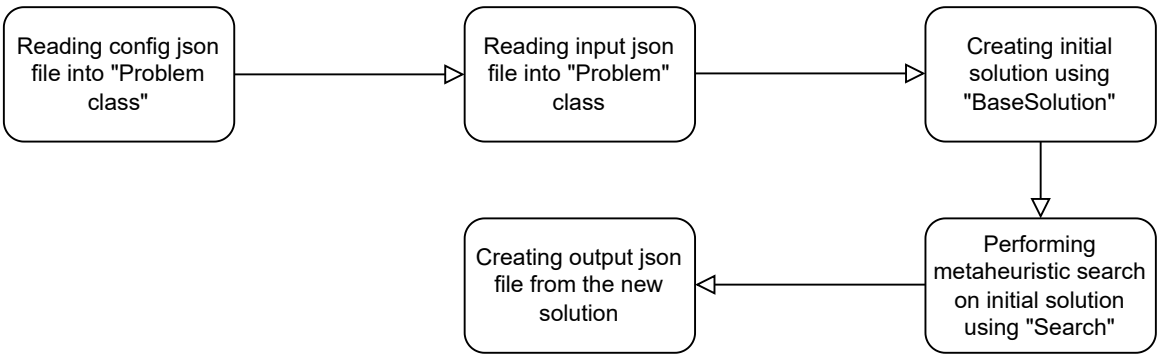

FIGURE 3.1: Pipeline of the program

3.1.3 Input of the program

There are two JSON files required to run the program: config file and input file.

In the config file, you need to specify possible types of containers and trucks.

```
{
  "container_types": [
    {
    "capacity": 1,
    "processing_time": 10,
    "garbage_type": "mixed",
    "type": "MixedRegular"
    },
  ],
  "truck_types": [
    {
      "type": "BigTruck",
      "speed": 10,
      "fuel_consumption": 0.5,
      "fuel_capacity": 500,
```

```
      "capacities": {"mixed": 20},
      "unloading_time": 15
    }
  ]
}
```

In the input file, you need to specify location and time windows for each container, garbage truck and landfill.

```
{
    "containers": [
        {
            "latitude": 682,
            "longitude": 237,
            "type": "MixedRegular",
            "start": 0,
            "finish": 100000000
        }
    ],
    "trucks": [
        {
            "type": "BigTruck",
            "latitude": 0,
            "longitude": 499,
            "start": 0,
            "finish": 100000000
        }
    ],
    "landfills": [
        {
            "latitude": 1000,
            "longitude": 500,
            "start": 0,
            "finish": 100000000,
            "max_trucks": 100
        }
    ]
}
```

Paths to these files and other parameters should be passed as the program arguments, this includes:

1. "config" - path to the config file
2. "input" - path to the input file
3. "output" - path to the output file
4. "map_file" - path to the osrm folder, which contains data about the city that we want to build routes for

5. "real" - boolean value, which tells if we need to calculate real-life distances and paths using "map_file"
6. "base_solution" - base solution to use
7. "heuristic" - heuristic to use to generate neighbor actions
8. "search" - metaheuristic search to use

There are also other parameters, but they are used only for particular algorithms and will be mentioned later.

3.2 Implemented algorithms

3.2.1 Base solutions

I decided to try out two variants of the initial solution creation: regular greedy search and greedy search based on clustered locations.

Algorithm 4 Greedy search

Sort trucks by the total garbage capacity in descending order
Create an empty route for each truck
Add index of each route to *goodRoutes* list
while *goodRoutes* is not empty **do**
    for *routeIndex* in *goodRoutes* **do**
        Add empty exportation to the route
        while there is a container, which won't break any of the problem requirements **do**
            Add the container that can be processed the earliest, which won't break any of the problem requirements
            Add landfill at which the truck can unload the earliest to the exportation, which won't break any of the problem requirements
        end while
        if exportation is empty **then**
            Delete the last exportation
            Delete *routeIndex* from *goodRoutes*
        end if
    end for
end while

Here, trucks are sorted by the total garbage capacity in descending order so that big trucks can choose from bigger amounts of containers, and the time spent on each route would be more balanced.

For the second algorithm, we clusterize every given location using Agglomerative Hierarchical Clustering and try to add containers from the same cluster as the route's last visited location (if it is not possible, we take container as in the first algorithm). This clusterization algorithm was chosen because it only depends on distances between the locations and not their coordinates. Algorithms such as k-means won't work here because the path between two points in real life is almost always not a straight line.

Algorithm 5 Agglomerative Hierarchical Clustering

Algorithm parameters: clustering coefficient
Input: locations
Output: clusters

$maxClusterDistance = clusterCoef * AverageDistance(locations)$
$clusters = Range([0, len(locations) - 1])$ ▷ Initially every location is in separate cluster
$clustersCount = len(locations)$
$clusterDistances = CreateDistanceMatrix(locations, clusters)$ ▷ Matrix which contains distances between each two clusters (maximal distance between two location from this clusters)
while $clustersCount > 1$ **do**
  $cluster1, cluster2, dist = FindClosestClusters(clusterDistances)$
  if $dist > maxClusterDistance$ **then**
    break
    $clusters = CombineClusters(clusters, cluster1, cluster2)$ ▷ Changes cluster of all locations from cluster2 to cluster1
    $clusterDistances = CreateDistanceMatrix(clusters)$
    $clustersCount$ -= 1
  end if
end while
return *clusters*;

3.2.2 Actions

Now, lets list implemented actions which could be performed on solutions during metaheuristic search:

- **InterChangeAction** - swaps any two containers in the constructed routes.
- **MoveAction** - moves container to some another place in constructed routes
- **ReverseAction** - reverses the order of containers of some part of exportation

3.2.3 Neigbourhood generation

Based on the implemented actions, I created several base methods on neighbourhood generation:

- **SwapInExportation** - generates all actions that swap containers from one exportation
- **MoveInExportation** - generates all action that move container to another place in its exportation
- **RemoveLoops** - generate reverse actions which get rid of loop in some exportation

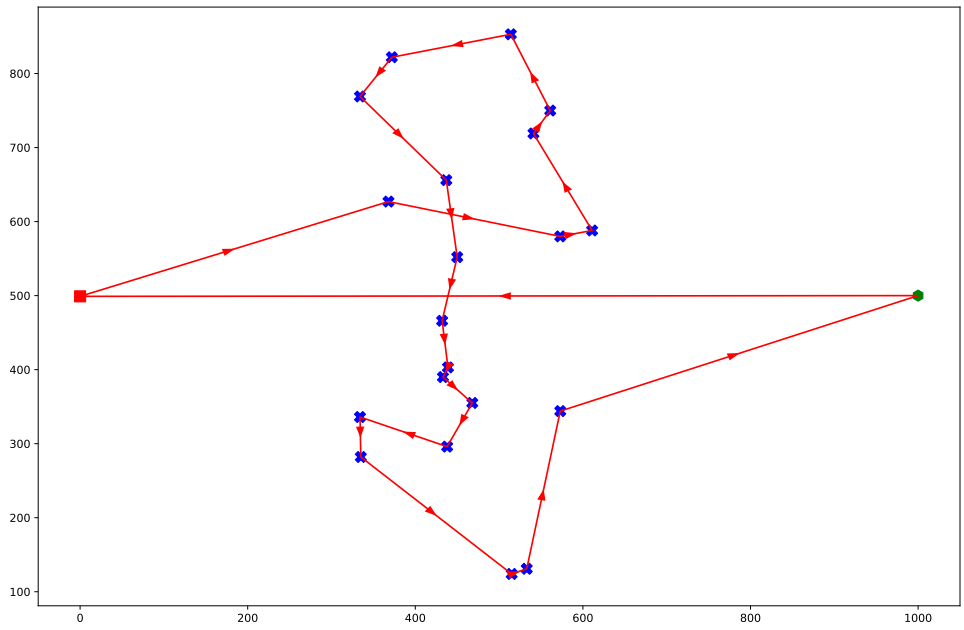

FIGURE 3.2: Example of route with loops

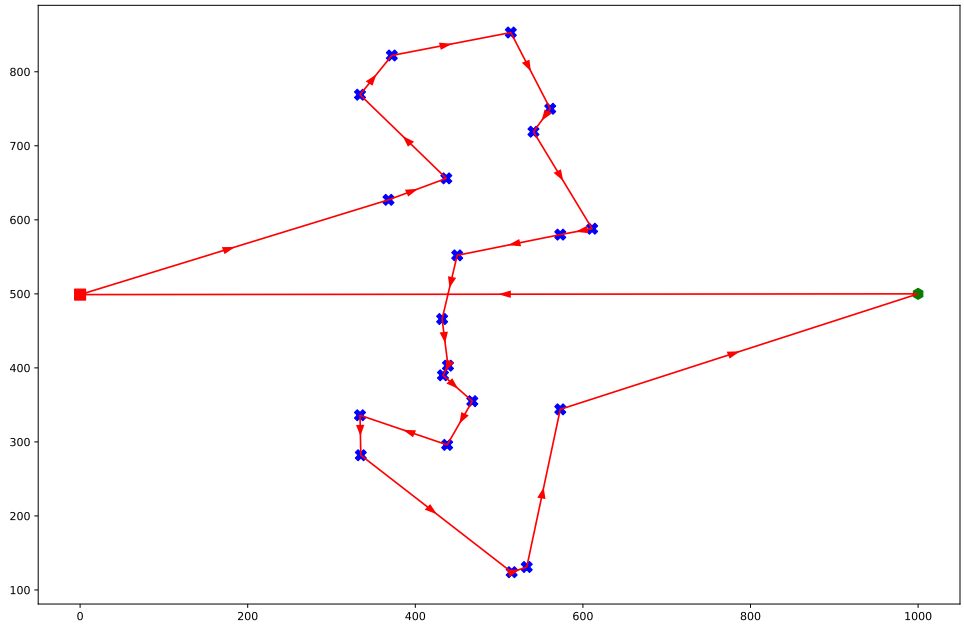

FIGURE 3.3: Example of route with removed loops

- **SwapAny** - randomly selects a fixed amount of containers and generates all swap actions for them (includes swaps in different exportations)
- **SwapNeigbors** - swap containers in the same exportation, which are placed close enough to each other in containers order

In addition to this base classes, I also implemented methods to create more complex neighborhoods:

- **MultipleSelector** - generates all actions for a given list of neighborhoods
- **ProbabiliticSelector** - randomly selects a neighborhood from the given list based on specified probabilities
- **FixedOrderSelector** - given a list of neighborhoods, uses them for a specified amount of times (for example, **SwapAny** is used 2 times, **SwapInExportation** is used 3 times and then again **SwapAny** is used 2 times ...)
- **TimeDependenceSelector** - given a list of neighborhoods, uses all except the last one for a fixed amount of times and then uses the last one for all other iterations

Using these classes, I implemented several complex neighborhoods:

- "ANY_EXPORTATION_PROB1" - with probability 0.5 uses **SwapAny** and with probability 0.5 uses **SwapInExportation**
- "ANY_EXPORTATION_PROB2" - with probability 0.75 uses **SwapAny** and with probability 0.25 uses **SwapInExportation**
- "ANY_EXPORTATION_FIXED1" - uses **SwapAny** once, then **SwapInExportation** once and so on
- "ANY_EXPORTATION_FIXED2" - uses **SwapAny** 5 times, then **SwapInExportation** 3 times and so on
- "ANY_MOVE_EXPORTATION_FIXED2" - uses **SwapAny** 5 times, then **MoveInExportation** 3 times and so on
- "ANY_EXPORTATION_TIME1" - uses **SwapAny** for half of the iterations and **MoveInExportation** for the rest
- "ANY_EXPORTATION_TIME2" - uses **SwapAny** for $\frac{2}{3}$ of the iterations and **MoveInExportation** for the rest
- "ANY_EXPORTATION" - uses all actions from **SwapAny** and **SwapInExportation**
- "ANY_MOVE_EXPORTATION" - uses all actions from **SwapAny** and **MoveInExportation**
- "ANY_MOVE_SWAP" - uses **SwapAny** 5 times and 3 times randomly chooses between **MoveInExportation** and **SwapInExportation** with equal probability
- "REMOVE_LOOPS_SWAP_EXPORTATION" - uses all actions from **RemoveLoops** and **SwapInExportation**

- "REMOVE_LOOPS_SWAP_ANY_EXPORTATION" - with probability 0.6 uses "REMOVE_LOOPS_SWAP_EXPORTATION" and with probability 0.4 uses **SwapAny**
- "REMOVE_LOOPS_MOVE_EXPORTATION" - uses all actions from **RemoveLoops** and **MoveInExportation**
- "MOVE_SWAP_EXPORTATION" - uses **SwapInExportation** 15 times, **MoveInExportation** 20 times and so on

3.2.4 Metaheuristic algorithms

I have implemented 3 well-known metaheuristic algorithms but slightly adjusted them. These algorithms try to maximize the score function, which I defined as

$$Score(solution) = -100 * OrphansCount(solution) - TotalFuelSpent(solution)$$

Tabu search

The main idea is the same as in the original Tabu search (1). However, I've still made some adjustments.

Firstly, here instead of adding solutions to the Tabu list, I add containers on which the actions were performed and don't perform any actions on them in the next few iterations. This modification was required due to the huge size of solution space for real-life examples (the Tabu list was too small to help achieve better solutions). It also sped up the algorithm because, with this modification, we don't need to compare solutions with the solutions from the Tabu list.

Secondly, I calculate the expected total fuel spent difference caused by the action before checking if it is possible to perform it; this means that bad actions won't be performed and won't increase the execution time.

Finally, I parallelized the algorithm. On every iteration, actions are split between several threads, and in the end, the best action of each thread is compared.

The parameters of this algorithm are iterations count and the Tabu queue coefficient, which defines the Tabu queue size as

$$TabuSize = TabuQueueCoef * ContainersCount(Problem)$$

Simulated annealing

For the SA algorithm, I used a modified temperature function described in "Metastrategy simulated annealing and Tabu search algorithms for the vehicle routing problem" article (Osman, 1993). To calculate max and min temperatures, firstly, we need to calculate the max and min score difference after performing some neighbor action on the initial solution.

$$T_{max} = max(|Score(newSolution) - Score(initialSolution)| \ \forall \ newSolution \in NeighborSolutions)$$

$$T_{min} = min(|Score(newSolution) - Score(initialSolution)| \ \forall \ newSolution \in NeighborSolutions)$$

We also calculate parameters:

$$\alpha = ContainersCount(Problem) * PossibleActionsCount$$

$$\gamma = ContainersCount(Problem)$$

Where $PossibleActionsCount$ is the number of actions that don't break any requirements of the problem. The temperature on iteration i is then calculated as

$$T_i = \frac{T_i}{1 + \beta_i T_i}$$

$$\beta_i = \frac{T_{max} T_{min}}{(\alpha + \gamma\sqrt{i}) T_{max} T_{min}}$$

T_0 here is equal to T_{max}. There is also a possibility that the algorithm won't choose any action from all the neighbor actions; in this case, we perform a reset and change the temperature to

$$T_{reset} = max(\frac{T_{reset}}{2}, T_{best})$$

Where initial T_{reset} is equal to T_{max} and T_{best} is the temperature at which we found the best solution so far.

The parameters of this algorithm are:

1. Iterations count
2. The maximal number of resets before ending the search
3. Temperature coefficient - we multiply α and γ by this value to speed up or slow down the temperature decrease rate

Genetic algorithm

For this algorithm, I also made several adjustments to the original one (3) to increase its effectiveness. Instead of just creating new population of the same size as previous one, I create a population with size equal to $2 * PopulationSize + 1$, which contains the best solution found yet and $2 * PopulationSize$ children. After that I choose $\frac{PopulationSize}{2}$ best solutions from this list and $\frac{PopulationSize}{2}$ randomly selected solutions (which are not contained in the best half). I also made the algorithm parallel (each thread separately creates several children and mutates them). Now let's talk about mutation and crossover operations. Mutation operation is fairly simple, with some fixed probability we either perform a random good action on solution (action which increases the score of solution) or a completely random action. Used crossover type is called an ordered crossover. To perform it, we firstly need function that generate containers order for the given solution.

Algorithm 6 Creation of containers order

Input: solution
Output: containers order

Add index of each route to *goodRoutes* list
$exportationIndex = 0$
$containersOrder = []$
while *goodRoutes* is not empty **do**
  for *routeIndex* in *goodRoutes* **do**
    if The number of exportations in route == $exportationIndex$ **then**
      Delete *routeIndex* from *goodRoutes*
    end if
    Add containers from the exportation with index equal to $exportationIndex$ to the order
  end for
  $exportationIndex$ += 1
end while
Add orphan containers to the order
return *containersOrder*

Now, let's look how the new container order is created from the given orders for parents.

Algorithm 7 Child order creation

Input: *order*1, *order*2
Output: *newOrder*

$start = Random([0, len(order1) - 1]$
$end = Random([0, len(order1) - 1]$
if $end < start$ **then**
  $swap(start, end)$
end if
$newOrder = []$
Add containers from *order*1 from index 0 to $start - 1$ to *newOrder*
Add containers from *order*1 from index $end + 1$ to $len(order1) - 1$ to *newOrder*
Insert containers that are not present in *newOrder* at index start in the same order as they are present in *order*2
return *newOrder*;

Creation of a solution based on the order of the containers is very similar to the initial greedy algorithm (4), but instead of adding the container that we can process the earliest, we try to add the next container from the order.

The parameters of GA are:

1. Iterations count
2. Population size
3. Mutation probability - the probability of performing random action instead of a good one while mutating

Additional adjustments

I have also defined a parameter called epsilon. If the score difference after performing some action has an absolute value less than this parameter, we don't perform it. This helps to prevent cases when we swap containers very close to each other, etc, and makes minimal temperature for SA algorithm adequate.

Chapter 4

Experiments

4.1 Initial experiments on synthetic data

As the initial data, I had 11 synthetic input files with a number of containers from 5 to 300 (all of the same type). All time windows were set in such a way that each action could be performed anytime, and the time window problem requirement wouldn't be broken. This synthetic data helped to sort out neighborhood structures that showed worse results. For example, "SWAP_NEIGBOURS" neighborhood structure almost couldn't improve the base solution, which you can see on the results plots for the input with 100 containers (squares are trucks, hexagons - are landfills, and crosses are containers).

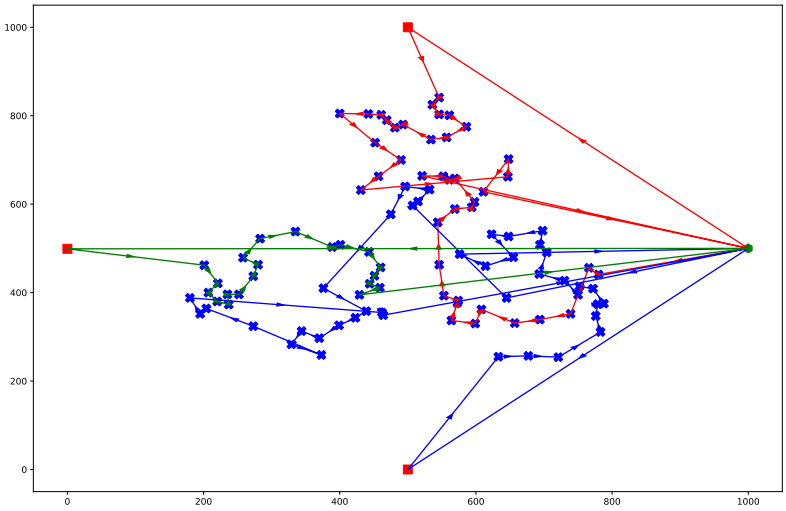

FIGURE 4.1: Initial solution based on clusters

Total distance: 10568

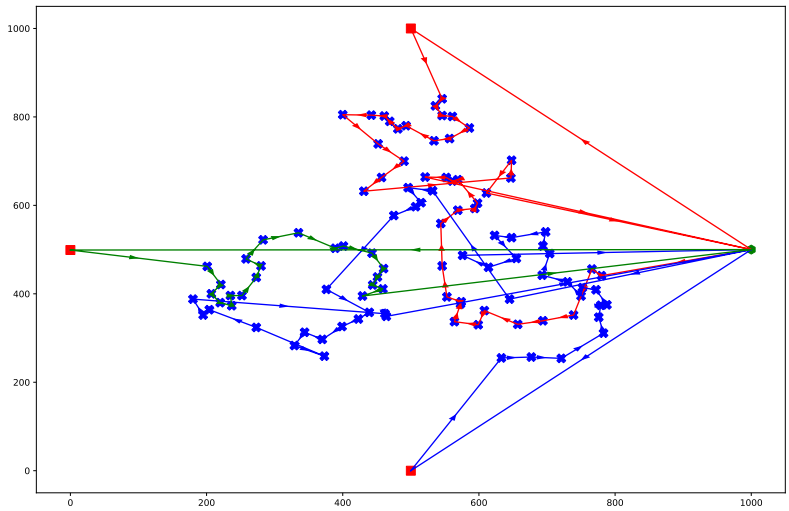

FIGURE 4.2: Tabu search with 500 iterations, "SWAP_NEIGBOURS"
Total distance: 10566

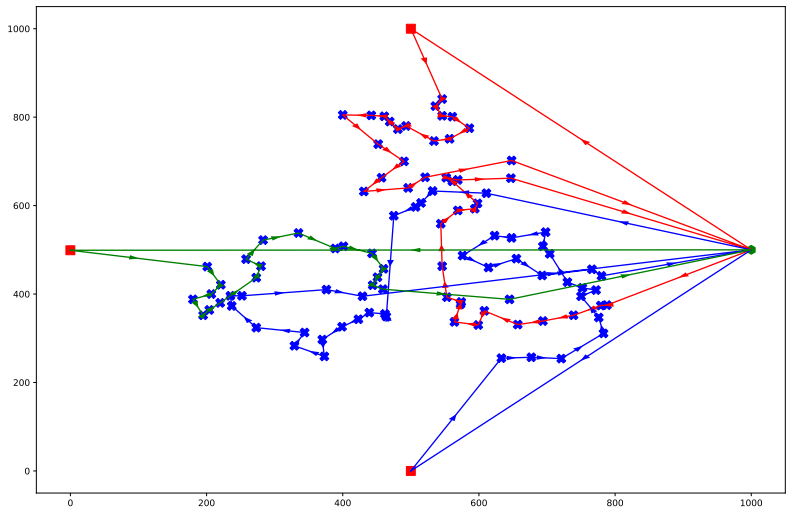

FIGURE 4.3: Tabu search with 500 iterations, "ANY_MOVE_SWAP"
Total distance: 9876

Based on the synthetic data, I filtered out 9 good neighborhood structures:

- "MOVE_IN_EXPORTATION"
- "REMOVE_LOOPS_SWAP_ANY_EXPORTATION"
- "REMOVE_LOOPS_MOVE_EXPORTATION"
- "ANY_EXPORTATION_PROB1"
- "ANY_MOVE_EXPORTATION_FIXED2"
- "ANY_MOVE_EXPORTATION"
- "ANY_EXPORTATION_TIME1"
- "ANY_EXPORTATION"
- "ANY_MOVE_SWAP"

I also decided to test only the greedy base solution because it worked better than the solution based on clusters in almost every situation.

4.2 Experiments on the Lviv data

For experiments on Lviv, I found google maps layer containing locations and types of 1475 garbage containers. I have parsed the respective KML file using Python to create an input JSON file (all trucks are located at some random container location, the landfill location was found on the internet). I have also changed the type of all containers and trucks to the same one for my experiments. The configuration file was made to be close to reality.

TABLE 4.1: Garbage container configuration

Capacity	1100 liters
Processing time	$\approx$ 2 minutes

TABLE 4.2: Garbage truck configuration

Capacity	35000 liters
Speed	45 KM/H
Fuel consumption per KM	0.25 liters
Fuel capacity	300 liters
Unloading time	3 minutes

I tested three implemented metaheuristic algorithms using such parameters:

TABLE 4.3: Tabu search parameters

Iterations	200; 500;
Tabu queue coefficient	0.3; 0.5

TABLE 4.4: SA parameters

Iterations	3000; 5000;
Temperature coefficient	0.0001; 0.00001

TABLE 4.5: GA parameters

Iterations	200; 400;
Population size	10; 20
Mutation probability	0.1; 0.3

I have also decided to test algorithms on various numbers of trucks (10, 15, 20) to be sure they work fine regardless of this value.

It is worth mentioning that all further experiments were performed on the computer with **macOS 12.4** operating system, **Intel Core i5-8257U** processor (4 cores, 1.6 GHz base frequency) and **16GB** of RAM. Now, let's look at the results.

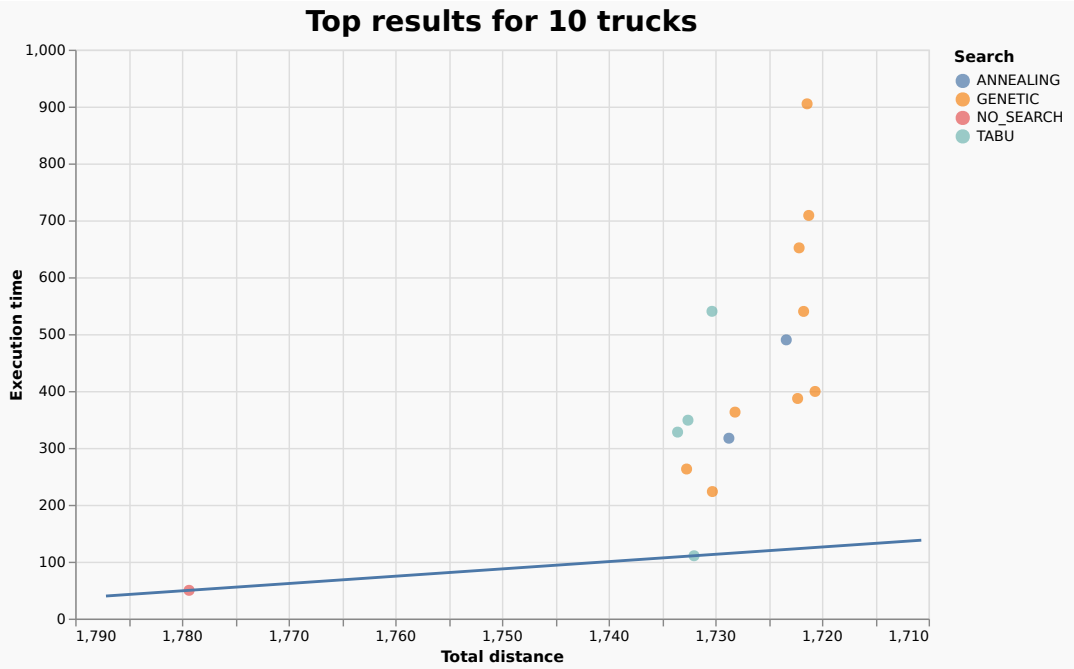

FIGURE 4.4: Best variants of algorithms (10 trucks)

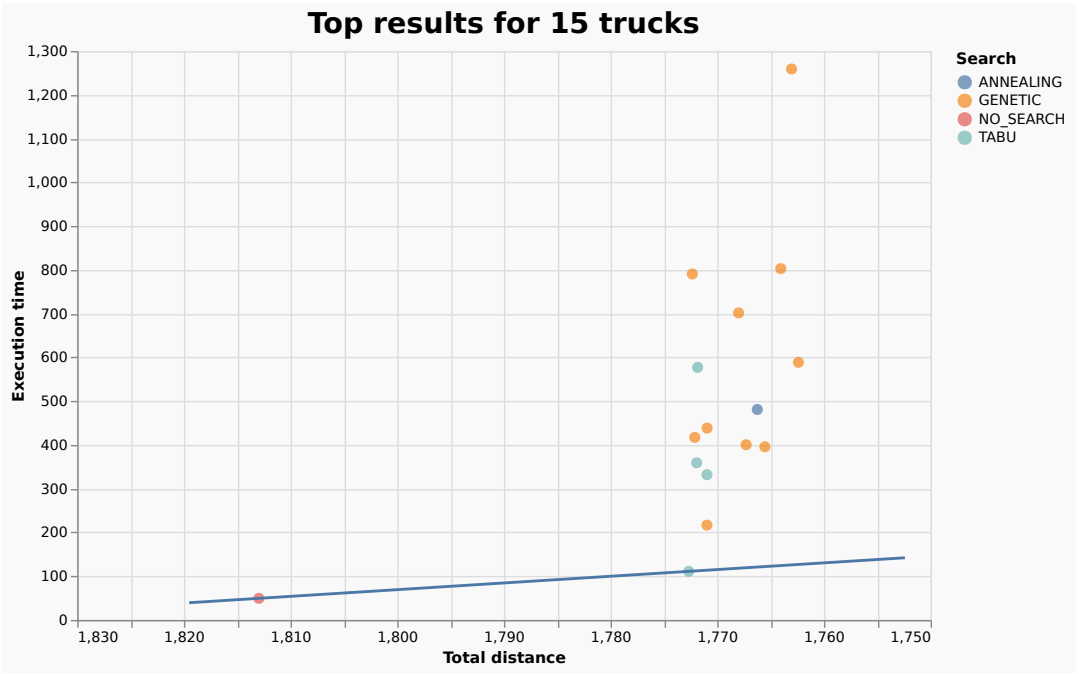

FIGURE 4.5: Best variants of algorithms (15 trucks)

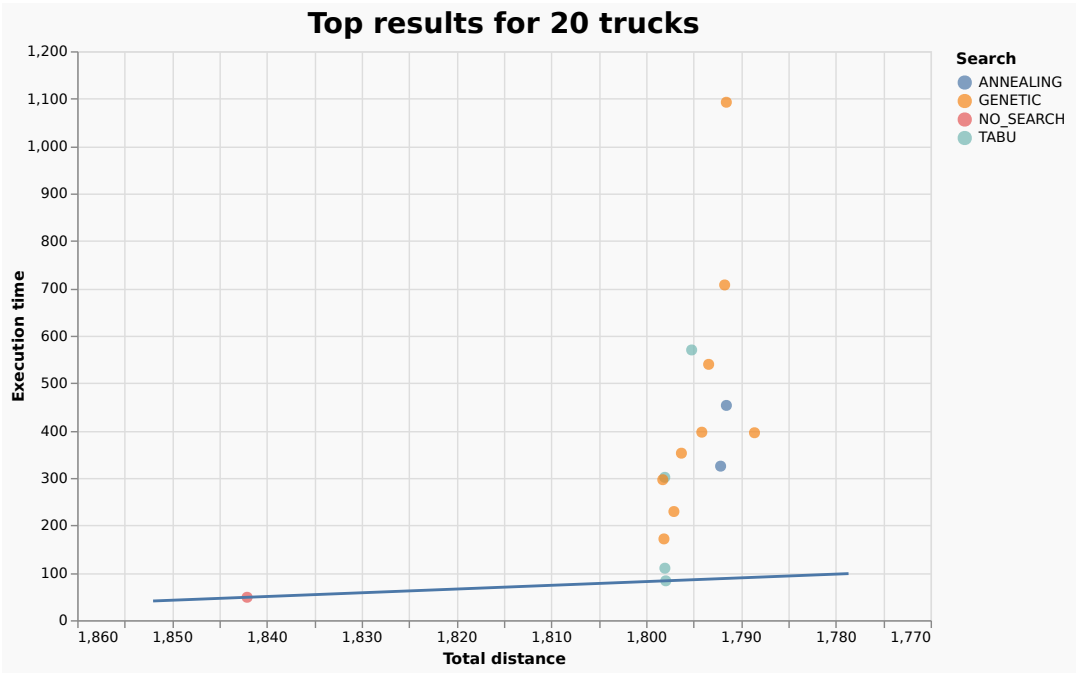

FIGURE 4.6: Best variants of algorithms (20 trucks)

Here I displayed the best 15 results for each amount of trucks. The line shows the algorithm with the minimal value of $\frac{TotalDistance(BaseSolution)-TotalDistance(NewSolution)}{ExecutionTime(NewSolution)-ExecutionTime(BaseSolution)}$ (I measure total distance instead of fuel because all trucks are of the same type and have the same fuel consumption rate). As we can see, for every case, the total distance is minimized by around 50-60km compared to the base solution, which is around 3 percent (for 10 trucks best algorithm decreases the distance by 3.3%, for 15 - by 2.7% and for 20 - by 2.9%).

Based on these results, I have chosen the best parameters set for each algorithm. Here, I decided to check two versions of the Tabu search: one regular and another with more generated neighborhood actions on each iteration (because its execution time is the smallest and allows to perform additional actions in an adequate amount of time). Here I talk about **SwapAny**, which by default generates swaps for 10 containers (the second version generates swaps for 500 containers).

TABLE 4.6: Tabu search best parameters

Iterations	500;
Tabu queue coefficient	0.3
Neighborhood structure	"ANY_MOVE_EXPORTATION"
Base solution	greedy

TABLE 4.7: SA best parameters

Iterations	5000;
Temperature coefficient	0.00001
Neighborhood structure	"ANY_MOVE_EXPORTATION"
Base solution	greedy

TABLE 4.8: GA best parameters

Iterations	400;
Population size	10;
Mutation probability	0.1;
Neighborhood structure	"ANY_MOVE_EXPORTATION_FIXED2"
Base solution	greedy

TABLE 4.9: Tabu search best parameters (more generated actions)

Iterations	500;
Tabu queue coefficient	0.3
Neighborhood structure	"ANY_MOVE_EXPORTATION"
Base solution	greedy

4.3 Experiments on other cities

On the internet, I have found 4 more datasets with container locations in other cities. Those are:

1. Groningen (2062 containers)
2. Sant Boi de Llobregat (1723 containers)

3. New York (544 containers)

4. Pittsburg (1194 containers)

Firstly, let's look at the locations of the containers, trucks and landfill for these cities:

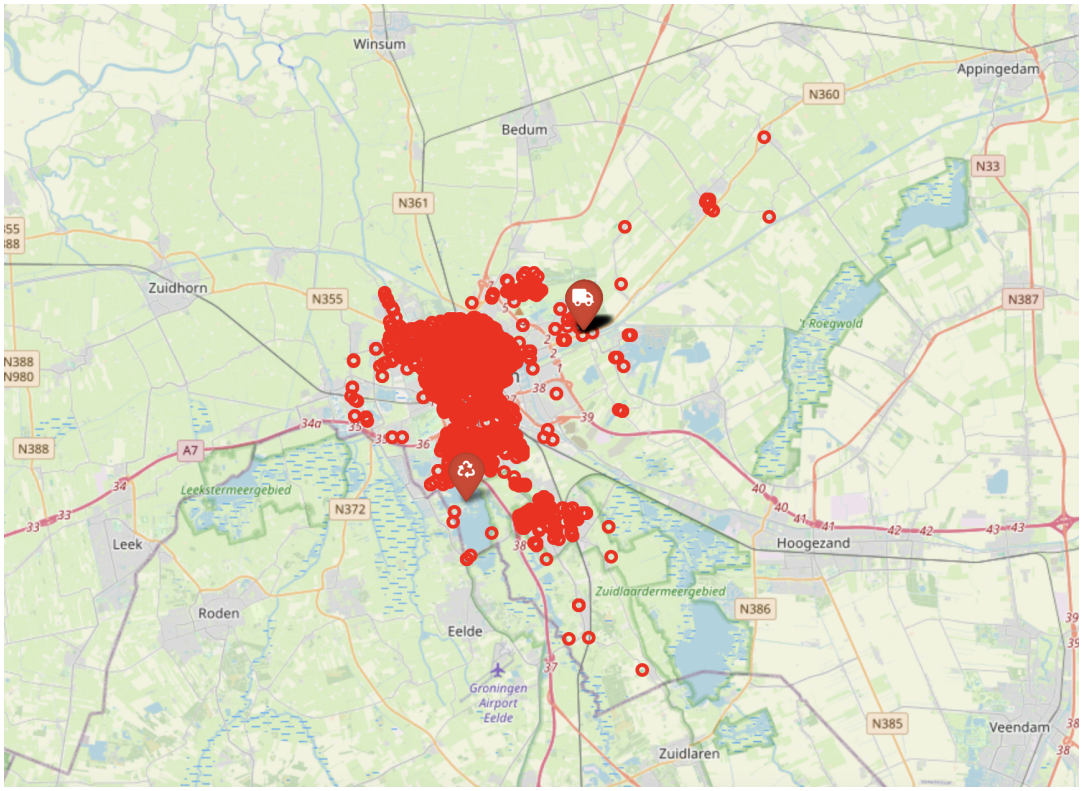

FIGURE 4.7: Groningen containers locations

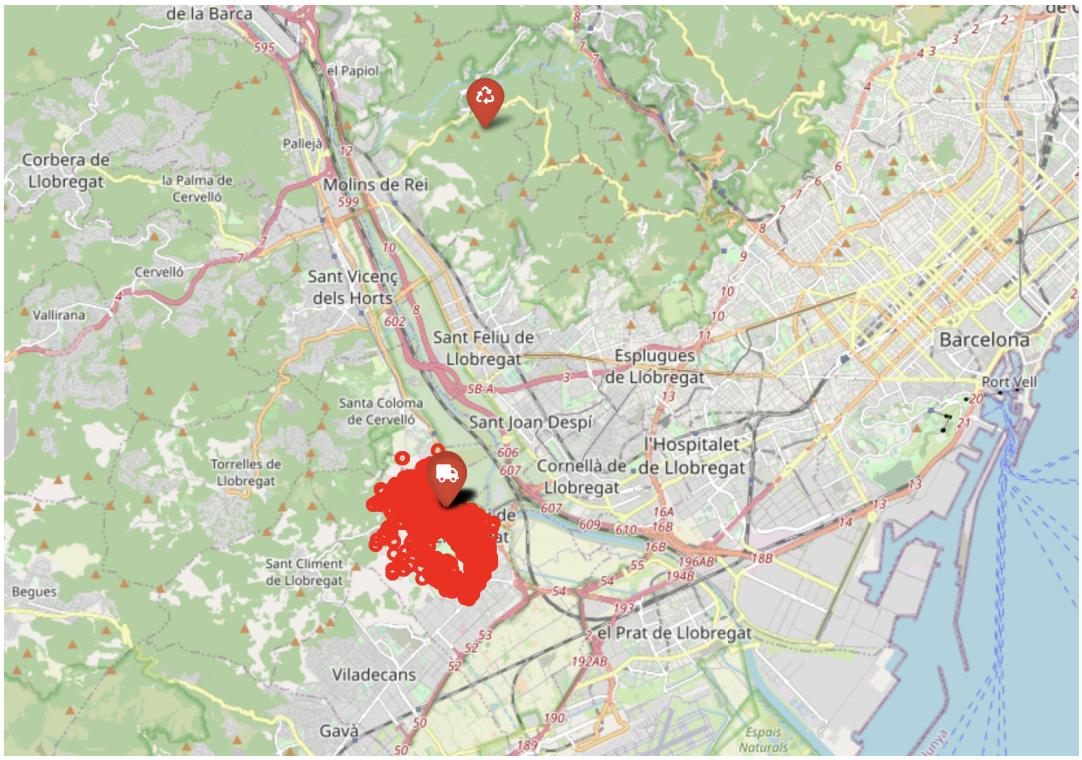

FIGURE 4.8: Sant Boi de Llobregat containers locations

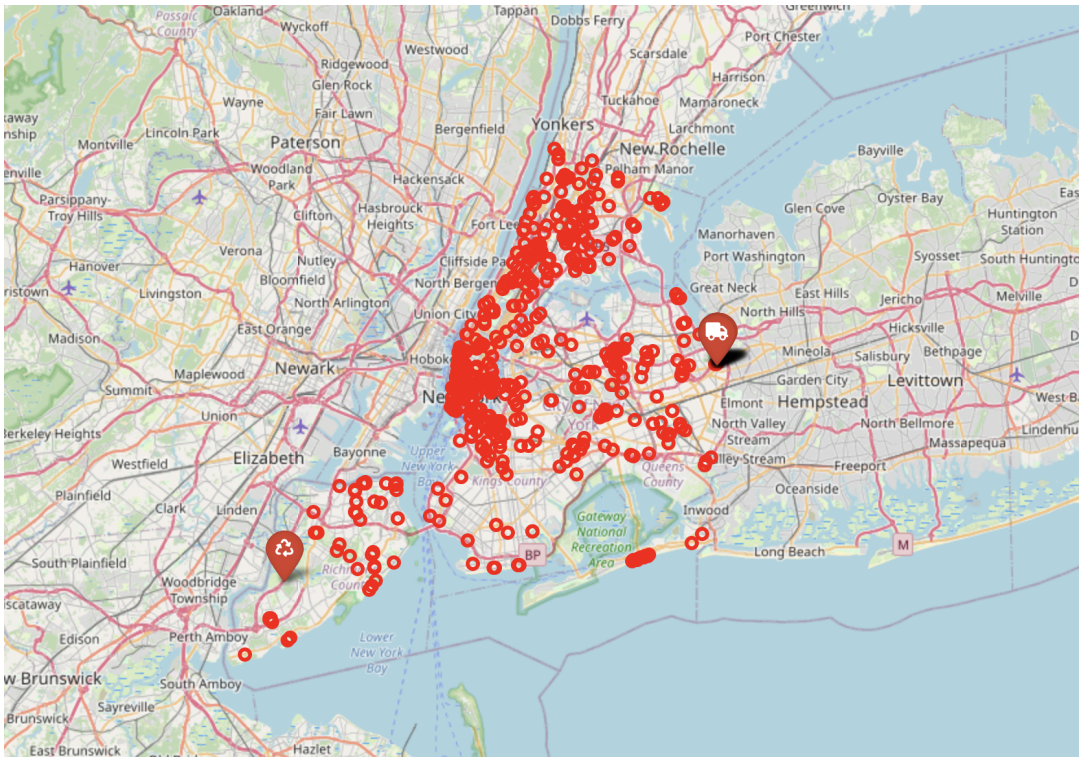

FIGURE 4.9: New York containers locations

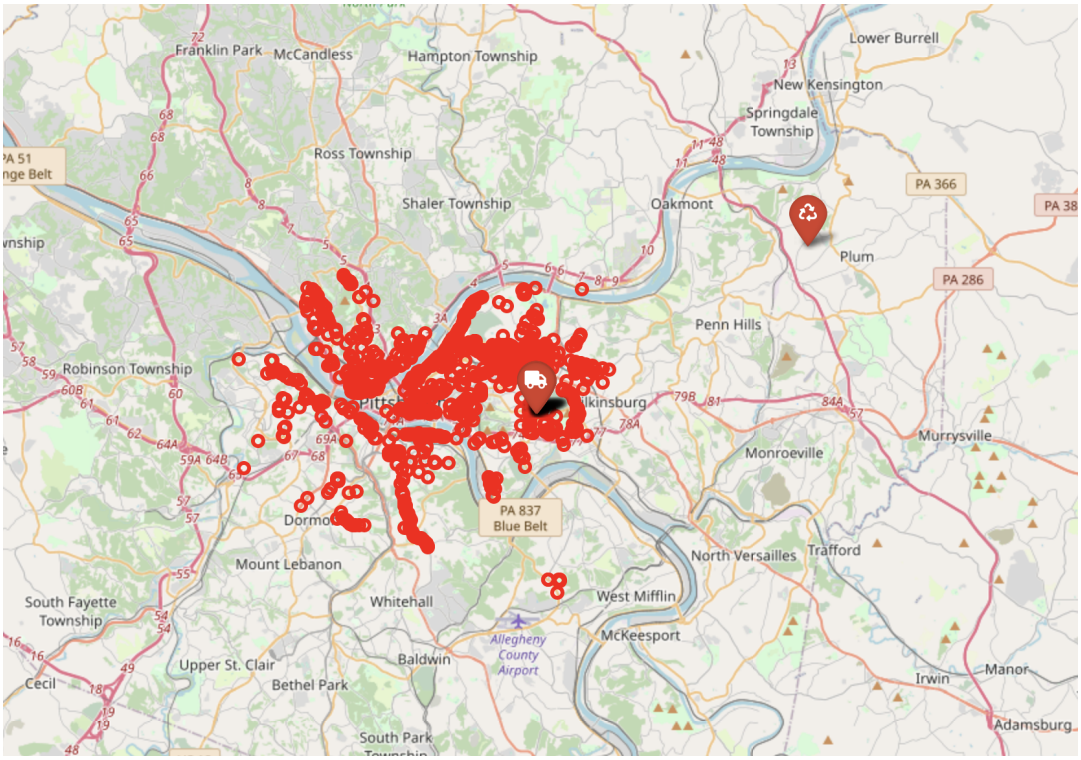

FIGURE 4.10: Pittsburg containers locations

I've measured the difference in execution time and total distance (in percentages) compared to the base solution of the best algorithms for each of these cities (for the case with 10 trucks). The results are as follows:

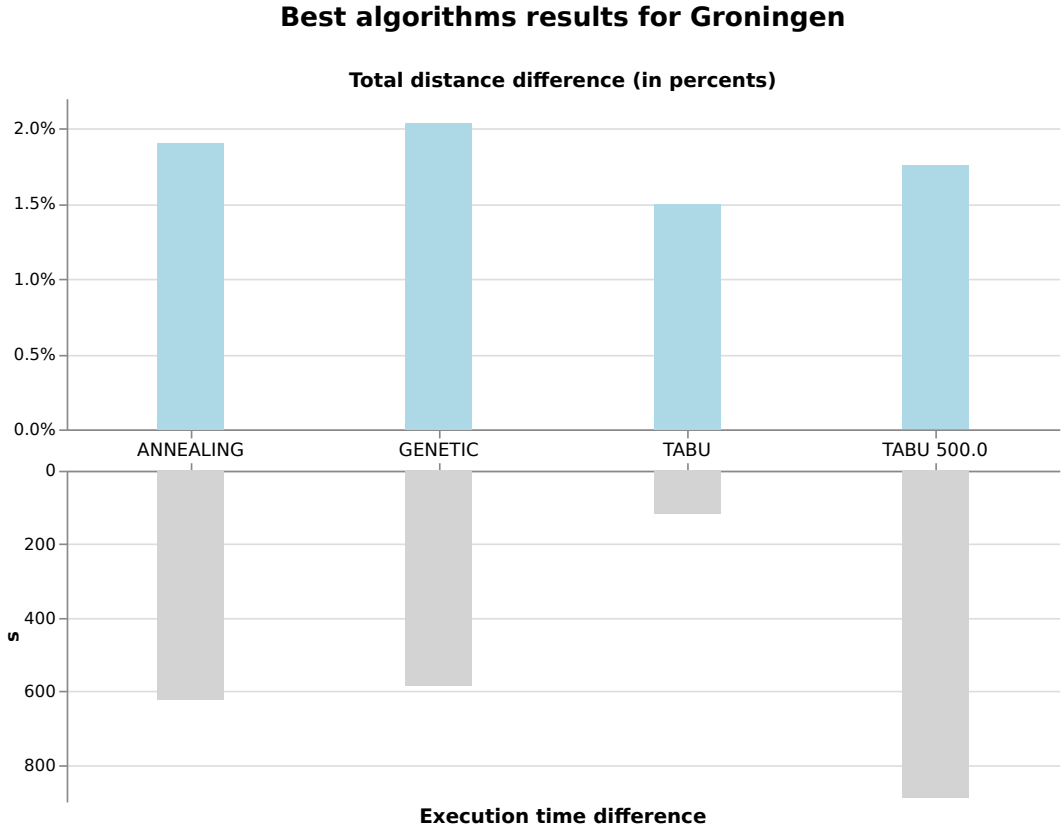

FIGURE 4.11: Results of best algorithms for Groningen

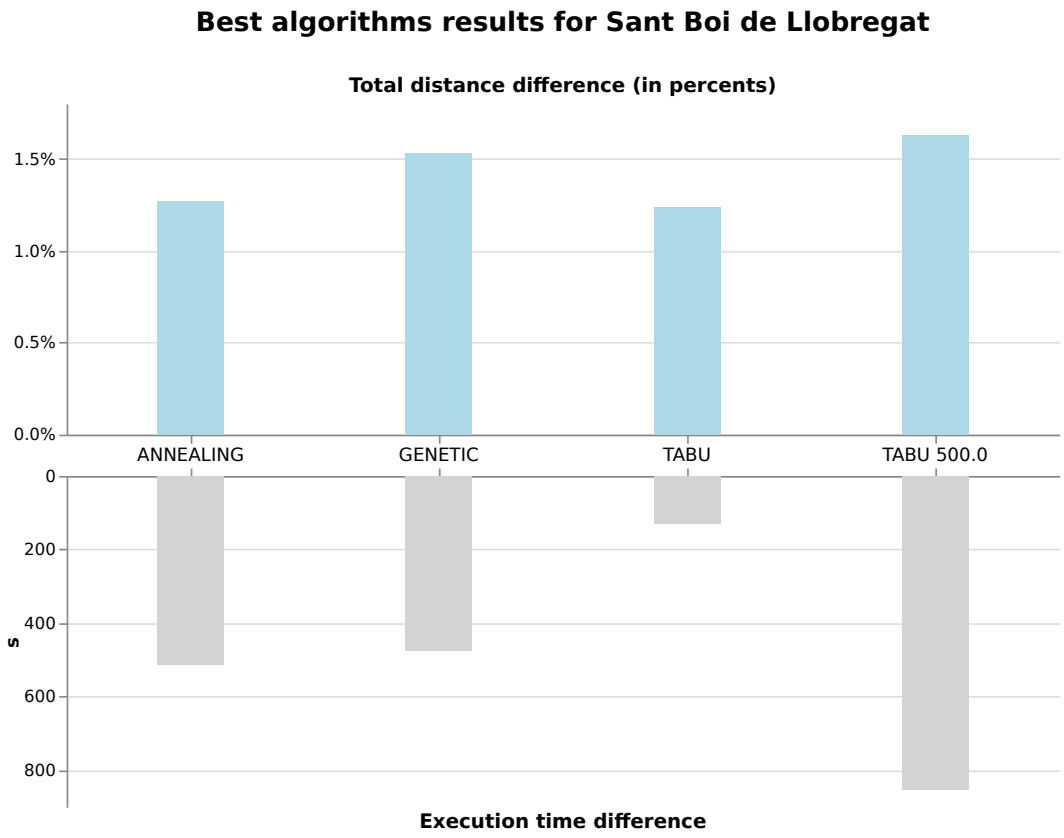

FIGURE 4.12: Results of best algorithms for Sant Boi de Llobregat

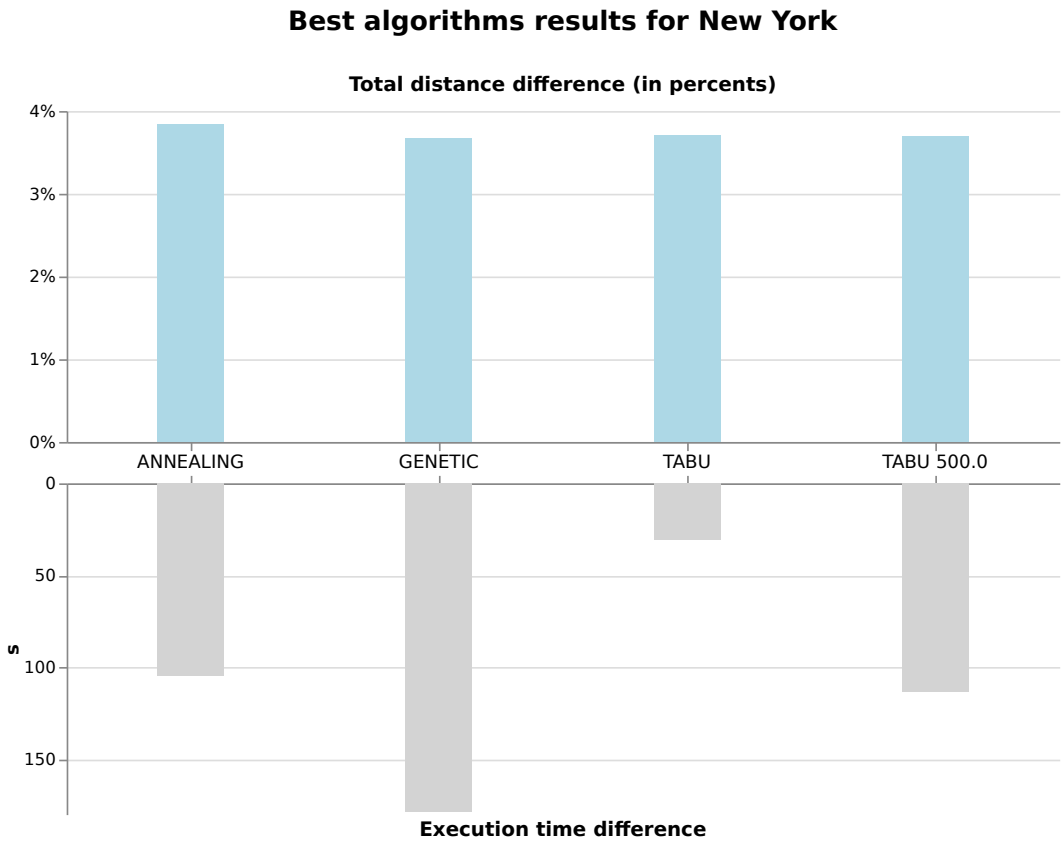

FIGURE 4.13: Results of best algorithms for New York

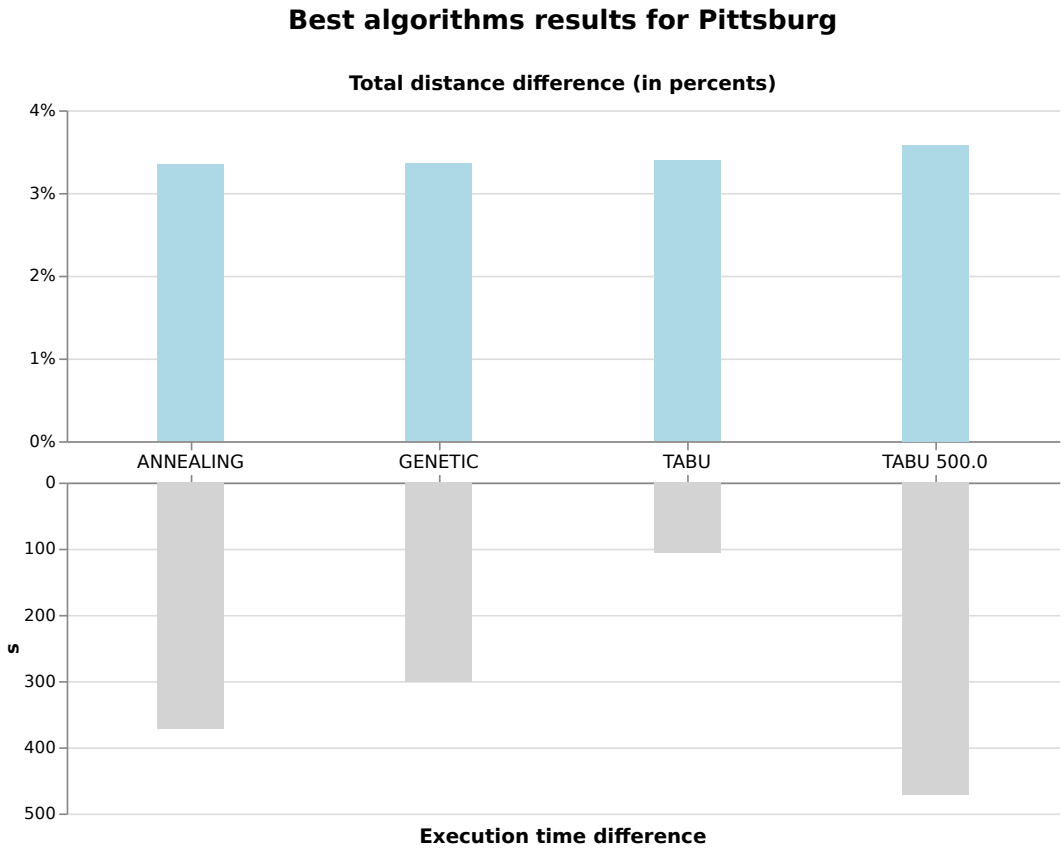

FIGURE 4.14: Results of best algorithms for Pittsburg

As we can see, the algorithms show better results for New York and Pittsburg (almost 4% and around 3.3% decrease of total distance) and worse results for Groningen and Sant Boi de Llobregat (around 2% and 1.5% decrease of total distance). I guess there are several reasons for that:

1. The number of containers - independently of the number of containers, the difference that some action performed on the solution can make to the total distance is almost the same. But the total distance traveled by trucks is actually related to the number of containers. This means cities with more containers (Sant Boi de Llobregat, Pittsburg) require more iterations to get as good results as cities with the smaller amount

2. Landfill location - some cities have landfills located very far from the majority of containers (Sant Boi de Llobregat, Pittsburg). This means that most part of the traveled distance is traveling to the landfill and returning back to the city, which couldn't be decreased

3. The density of containers distribution - in the case of more dense distributions, the base solution returns a pretty good result, which can be hard to improve (Sant Boi de Llobregat, Groningen)

We can see that the Tabu search with an increased amount of generated neighbors shows slightly better results than the other 3 algorithms. However, I don't think this algorithm would be very useful for real-life cases due to its huge execution time. In my opinion, the best two algorithms here are GA and regular Tabu search, but they should be used in different scenarios. If you have a lot of time, you can use GA

because, in most cases, it shows better results then Tabu search and SA, and if you are limited by time, you should use regular Tabu search because it shows fairly good results while spending less amount of time.

Chapter 5

Conclusions

In this thesis, I have compared a vast amount of different variations of metaheuristic algorithms for the waste collection problem. After all experiments, I can tell that the best of the implemented algorithms for this task is the genetic algorithm and the Tabu search. However, I can't say that the obtained results are ideal (3-4% improvement compared to the base solution). Nevertheless, with the framework that I have implemented, it is possible to easily extend the number of algorithms and types of actions performed on the solution. This can include more complex metaheuristics such as particle swarm optimization, actions that operate on multiple containers at once (from different exportations), etc. The room for improvement is simply limitless, due to the given problem being NP-hard.

Despite this, I still think that the solution is good enough to use in real-life cases. The program can greatly decrease the amount of time people in garbage companies spent on the route planning task and the only additional thing you need (besides configurations of containers and trucks and their locations) is OpenStreetMap data for your city.

Bibliography

Derecia, Ufuk and Muhammed Erkan Karabekmez (2022). "The applications of multiple route optimization heuristics and meta-heuristic algorithms to solid waste transportation: A case study in Turkey". In: *Decision Analytics Journal*.

Farrokhi-Asl, Hamed et al. (2016). "Metaheuristics for a bi-objective location-routing-problem in waste collection management". In: *Journal of Industrial and Production*. DOI: http://dx.doi.org/10.1080/21681015.2016.1253619.

Liao, Ching-Jong, Chao-Tang Tseng, and Pin Luarn (2007). "A discrete version of particle swarm optimization for flowshop scheduling problems". In: *Computers Operations Research* 34.10, pp. 3099–3111. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2005.11.017. URL: https://www.sciencedirect.com/science/article/pii/S0305054805003643.

Luke, Sean (2013). *Essentials of Metaheuristics*. second. Lulu.

Osman, Ibrahim Hassan (1993). "Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem". In: *Annals of operations research*, pp. 421–451.

Rabbani, Masoud, Hamed Farrokhi-Asl, and Hamed Rafiei (Mar. 2016). "A hybrid genetic algorithm for waste collection problem by heterogeneous fleet of vehicles with multiple separated compartments". In: *Journal of Intelligent Fuzzy Systems* 30, pp. 1817–1830. DOI: 10.3233/IFS-151893.

Stanković, Aleksandar et al. (2020). "Metaheuristics for the waste collection vehicle routing problem in the urban areas". In: *Working and Living Environmental Protection Vol. 17, No 1*, pp. 1–16. DOI: https://doi.org/10.22190/FUWLEP2001001S.