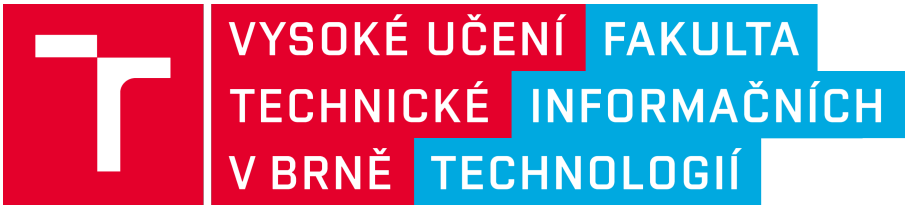


GAL Project 2023

Comparison - Vehicle routing problem

Vojtěch Fiala (**xfiala61**)
Peter Močáry (**xmocar00**)

1 Introduction

This document addresses the comparison of 2 chosen algorithms which are used to calculate the most optimal route in the *Vehicle Routing Problem (VRP)*.

Specifically, it addresses algorithms used for solving one of the variants of the VRP – the *Capacitated VRP (CVRP)*, which is more specific (in contrast to the general VRP) in that it takes into account the capacity of each vehicle.

The algorithms chosen for comparison are the *genetic algorithm (GA)* and the *Improved Clarke-Wright Savings algorithm.*

The CVRP graph problem is inherently associated with complete graphs — every pair of nodes has an edge that connects them. The input of both algorithms presented in this work consists of nodes representing a complete graph. Every node has associated coordinates and goods demand. The edges are implicitly generated and weighted by the algorithms. Calculated weights which represent the actual distances between nodes are stored in an adjacency matrix. Both algorithms then search for strongly connected sub-graphs which represent routes on the output.

This documentation will also include comparison of the theoretical and practical time & space complexities of both algorithms. The time complexity of the algorithms were measured using `std::chrono` library that made timestamps in microseconds which were then subtracted to provide estimated run-time of the algorithm. This simple profiling technique was used instead of utilization of dedicated profiler programs since the task is pretty simple and profilers tend to add more overhead. In order to measure the space complexity we used the `valgrind` tool to track heap allocations.

2 Genetic algorithm

Genetic algorithm is a heuristic algorithm commonly used to solve combinatorial problems. It's based on the idea that generating new solutions from the previous solutions improves their quality. The algorithm work best with small or medium problems because it often scales exponentially with the size of the search space [7].

The algorithm has 5 distinct phases, which will all be briefly explained based on our implementation. Aside from the initial population generation, each phase is repeated given number of times until the iteration limit is reached – only then the algorithm terminates. In our implementation, this limit is 50 000, which proved to

be enough to obtain satisfactory results while keeping the run time relatively low. The implementation itself is based on a slightly edited version of the algorithm explained in [5]. More profound explanation of the GA can be found in [1]. The algorithm itself is visually described in the figure 1.

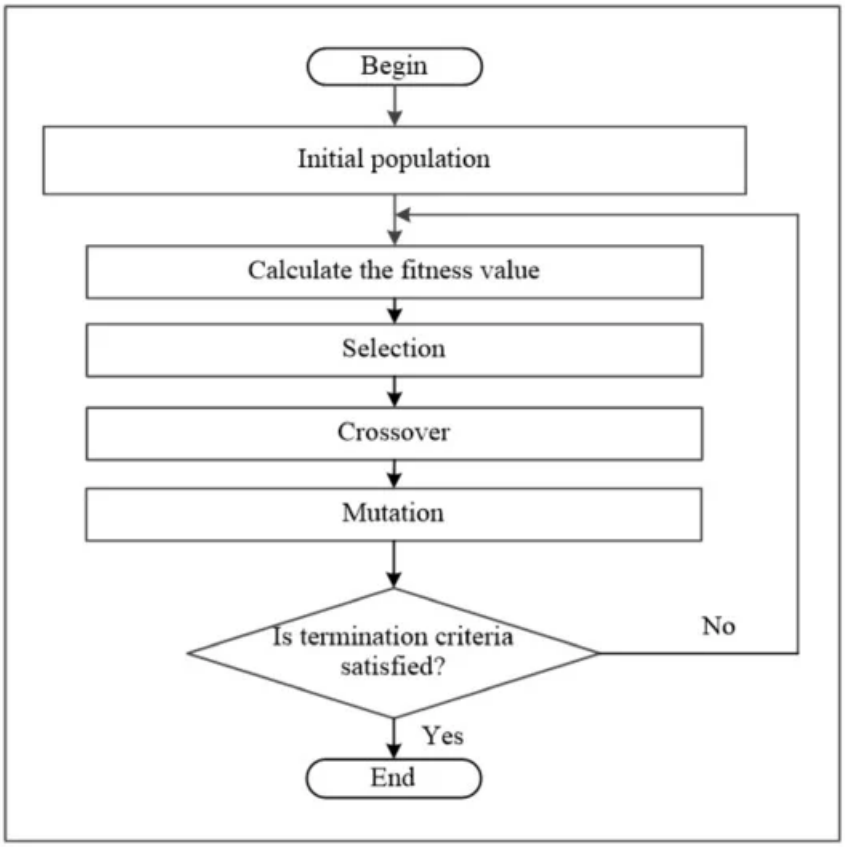


Figure 1: Phases compounding the genetic algorithm, taken from [1]

2.1 Initial population

The first step of the genetic algorithm is generating the initial population of chromosomes. Each chromosome represents a feasible distinct solution. But before that, it is necessary to use a correct encoding for the solutions. In this case, each number in the chromosome represents a customer and because of that, each permutation of the list of all customers represents a feasible solution. These per-

mutations are randomly generated so that each chromosome is unique. In this implementation we utilized a population of 50 chromosomes.

2.2 Fitness

Because the chromosomes need to be comparable to each other, it's necessary to calculate their "fitness". Fitness is a value used to describe how good (or bad) a solution is. It is calculated using an objective function.

The way the objective function works is that it calculates an euclidean distance between the depot and the first customer, distance between consecutive customers in the subroute, and the distance back to the depot. This is calculated for each subroute. A new subroute is made each time the vehicle capacity is exceeded.

The distance value is added together with a penalty in the form of number of returns back to the depot (which is increased each time a new subroute is made).

2.3 Selection

The selection process consists of selecting 2 parents from which a new offspring will be created. These parents are chosen using the tournament method which involves selecting a certain number of chromosomes and then choosing the best one. This chromosome will then be used as one of the parents of the new solution. This implementation uses a binary tournament [3], which means that 2 chromosomes are chosen randomly and the better one becomes one of the parents.

2.4 Crossover

The crossover operation is used to produce a new chromosome from the 2 parents. This implementation uses the ordered crossover method [4], which chooses a substring from one parent, copies this substring into the new child-chromosome and deletes the customers which are part of this substring from the other parent. The rest of the child-chromosome is generated by filling the missing places with customer numbers from the other parent. This method generates 2 children (one substring from each parent), but only the better one is used, the other child is cast aside. The process is illustrated in the image 2.

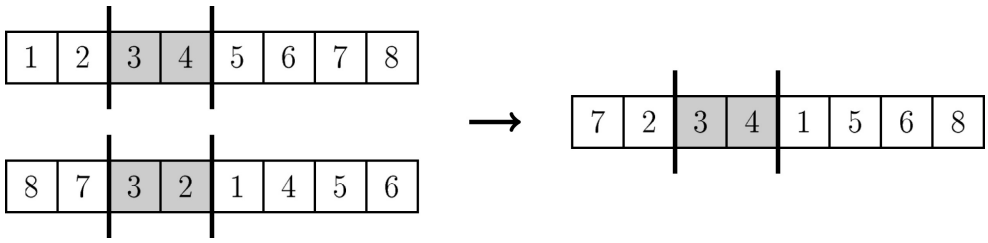


Figure 2: Ordered Crossover algorithm, taken from [4]

2.5 Mutation

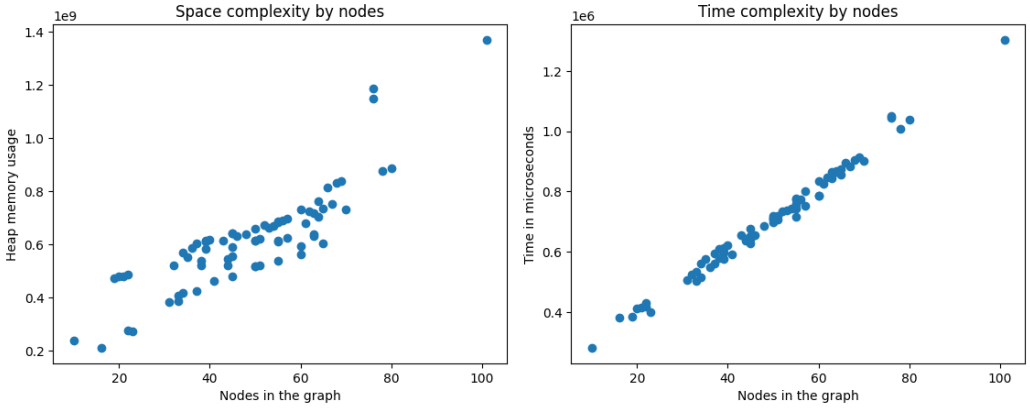
The point of the mutation operation is to keep variability in the population and to not get stuck in a local extreme. Each of the generated offspring goes through the mutation which consists of randomly selecting 2 customers from different subroutes. These customers are consequently swapped [3].

2.6 Time & Space complexity

Because a genetic algorithm can be implemented in many ways, both time & space complexities depend on the specific implementation.

This implementation's time complexity was determined to be $O(n^2)$, but the omitted coefficients play a vital part here and when including them, we receive a more informative time estimation – $O(i \cdot (n \cdot (10 + p) + 4n^2))$, where i is the number of iterations and p is the population size. With our numbers it becomes $O(3e6n + 2e5n^2)$. The high coefficients are present because of the nature of the genetic algorithm – it repeats all its phases until iteration count is reached.

Space complexity was determined to be $O(n)$. Below are graphs with actual measured complexities. Measurements were taken on graphs containing fewer than 100 nodes.



(a) Space complexity of the GA

(b) Time complexity of the GA

The measured time complexity does not correspond with theoretical $O(n^2)$ but as was explained above, when taking into account the lower polynomial and mainly their coefficients, it may explain why the graph has more of a linear upward trend. The by far most often performed operation is the fitness calculation, which has time complexity $O(n)$ and the $O(n^2)$ comes from the ordered crossover, which is performed far less times.

Space complexity metric might not be as accurate as time complexity, because the measurements were done using the valgrind heap allocation results which doesn't really measure the things we would like it to. Despite that, the space complexity seems to follow the linear trend but with some inconsistencies – the reason for them might be that the genetic algorithm creates the initial population randomly and all operations are influenced by the initial random choice. For example, replacing the population with the new chromosomes – some chromosomes will be unfit to add, while others will be fine – the number of chromosomes that will be added is dependent on the initial population and so is the memory usage.

3 Improved Clarke and Wright's Savings Method

The *Clarke and Wright's savings method* (CWSM) is one of the widely known heuristic methods for solutions of CVRP. The savings concept was originally proposed by Clarke and Wright who based it on the computation of savings for combining two customers into the same route. It is an effective method when it comes to small and medium size problems where it achieves good solutions. However,

the method is still a heuristic and won't achieve the optimal solutions. The savings principle is outlined in the following Section 3.1.

The CWSM has several modifications and improvements (many of which are described in [6]). Some update its whole algorithm leaving only the core principle [8], others update mainly its heuristic [2] and many modify it to comply with additional constraints that need to be satisfied. In this work, we chose to implement the *Improved Clarke and Wright's savings method* (ICWSM) from [8] since it changes the whole algorithm while preserving the savings method and claims to provide better overall distance traveled than the regular CWSM. The Section 3.2 describes inner workings of ICWSM algorithm.

3.1 Savings Principle

The savings principle [8, 6] is rooted in a single value for every pair of customers — *savings* value. The algorithm decides, based on the savings value, whether the pair of customers should be part of a route — the higher the savings value is, the better it is to connect the two customers in one route.

Given D_0 as the depot from which are the goods delivered, and customers as C_1 and C_2. The distance between customers is l_{12} and distances between depot and customers are l_{01} and l_{02}. The separate distribution of goods to both customers is $L = 2(l_{01} + l_{02})$. While combined distribution of goods to both customers is $L' = l_{01} + l_{02} + l_{12}$. The savings value is then defined as difference between these distributions $S = L - L'$ which when simplified is $S = l_{01} + l_{02} - l_{12}$.

The main goal of savings is to minimize total distribution distance under ideal conditions. However, it does not take into account time and other constraints.

3.2 Improved Clarke and Wright's Savings Method

The ICWSM aims to reduce the gap to the best solution that CWSM has. This is done through modification of the algorithm in a significant way. It builds route one by one based on the savings value, however, it doesn't merge routes as the original algorithm. This way provides more routes (more vehicles are required) and results in shorter overall distance traveled thanks to selection of the best saving more often than the original method.

The algorithm can be broken down into the following steps:

1. **Step one**: Calculate the distance between all pairs of customers, as well as

between each customer and the depot.

$$d_{pq} = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$$

2. **Step two**: Calculate the savings values between all pairs of customers. Rank the savings values and omit those below zero.

$$s_{pq} = d_{p0} + d_{q0} - d_{pq}$$

3. **Step three**: Choose two customers with maximum savings value and make sure they satisfy the truck load limit. Initialize new route using these two customers.

$$R_1 = \{p, q\}$$

4. **Step four**:

   (a) Choose a candidate customer that could be added to the end or start of the route. This customer is chosen by finding savings that is the largest one and contains the same customer of either the start or the end of the route from previous iteration R_{i-1}.

   (b) If this savings is the largest one for this customer (there is no better savings for this customer) add the customer to the from previous iteration R_{i-1}. If not, skip this customer and move on to the second best candidate customer in *Step four (a)*

5. **Step five**: Until there is no customer that can be added to the route R keep repeating the *Step four*.

6. **Step six**: Until all customers have been added to routes, keep repeating the *Steps three*, *Step four* and *Step five*.

It is apparent from the description of the algorithm that it is a graph algorithm. A *savings* is an edge from the graph that that is weighted by the *savings value*. Therefore when the route is being constructed all edges that can be connected to it are considered. Based on their weight is decided which is the best to chose (the higher the better). The node (customer) that is connected by this edge is then added to the route on its start or end based on the edge that was selected.

When implementing this algorithm few problems with its specification were discovered. The main problem, for example, is that it does not solve the case

where a new route can't be initialized with a pair of customers. This was solved simply by adding a check after the *Step five* that would be triggered if an empty route has been formed. In this case every customer that was not served is added as a separate route. This, however, is not the only problematic part. The *Step four* has a check in its part (b) that allows a candidate customer to be added to a route only if the savings found is the largest for this customer. In the only implementation we found on GitHub[1] this part was omitted and it seems to work better – it produces shorter overall distances. However, this part was preserved in our implementation to test the algorithm defined by the paper [8] with as little modifications as possible.

3.3 Time & Space complexity

The *time complexity* of the algorithm implementation was determined step by step in the source code. Each loop and significant operation has its complexity assigned in the source code as a comment above it (see `savings.cpp` for reference). The notation contains m which represents number of edges in the graph and n as the number of nodes in the graph. Following are simplification steps after determining the complexity of each part of the algorithm:

$$O(m + m * log(n) + (m * n + m * (n + m) + n + n) * n)$$

The algorithm always considers complete graph. Therefore, we can base the number of edges on the number of nodes: $m = \binom{n}{2}$ which can be simplified to n^2 after a few steps. The

$$O(m + m * log(n) + (m + m * m + n) * n)$$

$$O(n^2 + n^2 * log(n) + (n^2 + n^4 + n) * n)$$

$$O(n^2 + n^2 * log(n) + n^3 + n^5 + n^2)$$

$$O(n^5)$$

The resulting time complexity is therefore polynomial – $O(n^5)$

The *Space complexity* for this algorithm is straight forward. The algorithm saves all the edges of connected graph which are weighted — $O(m)$. Then it saves a boolean flag for each of the nodes that determines if it was already served

[1] `https://github.com/shlok57/VehicleRoutingProblem/blob/master/Savings/cvrpImprovedImpl.py`

— $O(n)$. And finally every route is saved during the algorithm as well, which in the worst case should be half of the nodes count — $O(n)$.

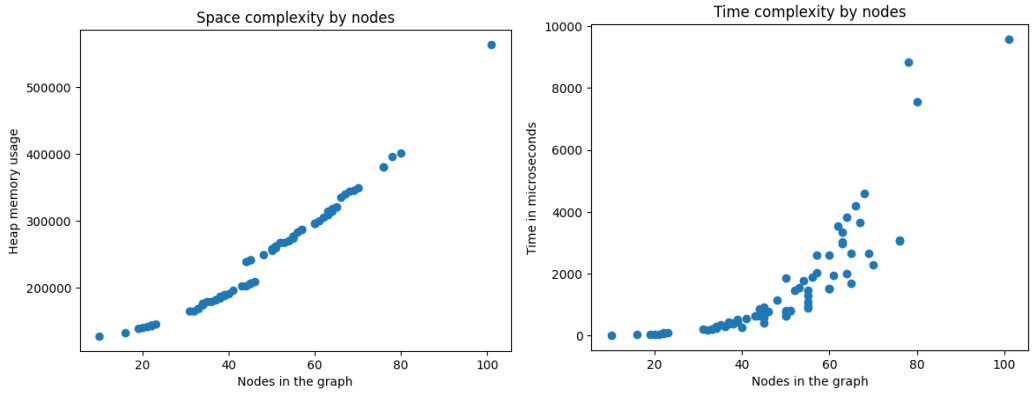
$$O(m + n + n)$$

$$O(n * (n - 1) + n + n)$$

$$O(n^2)$$

After similar simplification as in the time complexity part, the space complexity is linear — $O(n^2)$.

The measured time and space complexity in the Figure 4 seems to be similar to expected results. The space complexity looks a bit linear but when put to scale it is actually quadratic.



(a) Space complexity of the Savings Algorithm

(b) Time complexity of the Savings Algorithm

Figure 4: Measured time and space complexity.

4 Program Implementation

We implemented a simple application that enables a user to select one of the above algorithms and specify the set of nodes with requirements and vehicle capacity in special `XML` format based on the datasets[2] related to CVRP problems we found on-line. To parse the XML data, `pugiXML`[3] library was used.

[2] `http://www.vrp-rep.org/datasets/item/2014-0001.html`
[3] `https://github.com/zeux/pugixml`

In order to build the program run `make` in the root folder. After the program is built there will be a `gal` executable present in the root folder. This executable can be run with option `--algorithm` (or `-a` for short) that takes as an argument one of the implemented algorithms specified by a keyword `savings` for the Heuristic algorithm from the Section 3 and keyword `genetic` for the Genetic algorithm form the Section 2. As the last argument it expects valid path to a `XML` representation of the CVRP problem to solve. It should be one of the `*.xml` files located in `data` folder. An example of execution:

```
./gal --algorithm savings ./data/B-n31-k05.xml
```

We also made a simple visualization tool in Python that runs the gal executable with provided arguments and plots its result. It expects 2 arguments that are passed to the `gal` executable — the keyword specifying the algorithm and the data file that represents the input graph. It requires `matplotlib`, so it is advised to use a virtual environment such as `venv` before running the script.

```
python ./plot-path.py savings ./data/B-n31-k05.xml
```

We also implemented several python scripts to use for experiments with the implementation. They launch the executable for both algorithms and compare the output. Further information is included in the README file, but the syntax for the basic comparison script launch can be found below.

```
python ./compare.py
```

The contents of the provided archive contain the implementation in the `src` folder and a subset of datasets used for the experiments (Section 5) together with other things needed to run the experiments in the `data` folder:

```
xfiala61.zip
├── documentation.pdf - this file
├── presentation.pdf - the presentation
├── data
│   └── *.xml - files with VRP problem definitions
├── *.py - python files to run the experiments
├── requirements.txt - text file with required python libraries
├── results
│   └── *.txt,csv - various performance measures
└── libs - libraries used in the implementation
```

```
├── Makefile
├── src
│   ├── gal.cpp
│   ├── genetic.cpp - Genetic algorithm from Section 2
│   ├── genetic.hpp
│   ├── savings.cpp - Heuristic algorithm from Section 3
│   ├── savings.hpp
│   ├── util.cpp
│   └── util.hpp
└── structures
    └── *.cpp,hpp - shared structures
```

5 Experiments

In order to compare the algorithms we decided to gather some statistical data about the created routes and compare it based on the simple metrics. These algorithms are generally compared based on the *overall distance* that needs to be covered to satisfy all the customers. This distance is calculated from the planned routes by the give algorithm. This will be, therefore, our main metric for the comparison. Other than that, we decided to include some additional metrics such as:

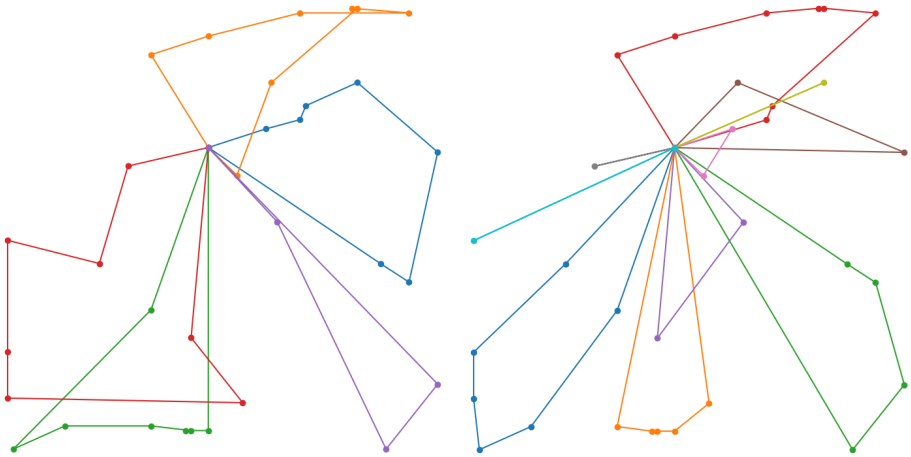
- number of vehicles needed (in other words, the number of routes generated by the algorithm)
- average number of customers per route
- number of routes linking only one or two customers
- unused capacity of vehicles

Before the experiments the program was compiled with the -O2 optimization `gcc` flag. The experiments (as well as practical time & space complexities) were performed on a Ryzen 5 5600X CPU coupled with DDR4-3000 memory. The data were gathered and aggregated by custom python scripts created for this purpose.

Our experiments included running the algorithms on the graphs from the datasets mentioned in the section 4. Across all datasets, we measured the above mentioned metrics in total on 200 instances divided into three sets ranging from roughly 10 to 100 nodes, 100 to 500 nodes and a set of larger graphs from 500 to 1000 nodes. The last set of graphs that contained up to 1000 nodes took too long

to execute as expected, since these algorithms are not designed for large CVRP problems. Therefore, we estimated the usable range to be up to 500 nodes with run-time that is acceptable (up to 90s), however, the algorithms perform best in the lower ranges of nodes, therefore, we will include only the datasets with 10-200 nodes and their results.

Firstly, lets compare two generated routes in the Figure 5. It is apparent that the Genetic algorithm generated less routes than the Heuristic one. The genetic algorithm also groups more nodes into a route, while the Heuristic algorithm is more keen to create single node routes. It is also apparent that the routing is not optimal for either algorithm.



(a) Genetic algorithm routes. (b) Heuristic algorithm routes.

Figure 5: Comparison of routes generated by both algorithms on the same 33 node graph. The Depot is the node where all the routes intersect and the other nodes are customers.

Our findings for problems with less than 100 nodes are summarized in the following list:

- The Savings algorithm finds on average **1.18**× longer total distance
- Savings algorithm needs on average **1.6**× more vehicles than GA to fulfill the requests
- Savings serves on average **0.64**× less customers on each route than GA
- GA had a total of **0** routes with only 2 customers, Savings had **75** of them

- Savings has on average **1.08**× more unused space in vehicles than GA
- GA is on average **2695**× slower than Savings (in practical terms, this means that GA might run for 0.5 seconds while the Savings algorithm may run for only 0.0001 seconds), but this is only true for problems where number of nodes is lesser than 100. With increasing number of nodes, the times become more equal.

Results for problems with 100-200 nodes are summarized in another list below:

- The Savings algorithm finds on average **1.14**× longer total distance
- Savings algorithm needs on average **3.4**× more vehicles than GA to fulfill the requests
- Savings serves on average **0.33**× less customers on each route than GA
- GA had a total of **79** routes with only 2 customers, Savings had **455** of them
- Savings has on average **1.54**× more unused space in vehicles than GA
- GA is on average **8.3**× slower than Savings.

The measurement that differs the most is the run-time. To put it into perspective, the table below provides specific measured times (in microseconds) for an increasingly larger graphs:

Nodes in graph	22	57	80	125	153	186
Genetic Algorithm	642350	1202256	1465665	2857609	3285406	2919332
Savings Algorithm	100	3736	15703	199074	298182	1200020

Table 1: Times in microseconds for each algorithm for graph of given node count

6 Conclusion

Our findings include that the GA gives better results overall (and could give even better), but takes much more time to run. When pushing the GA iteration limit & population size, we soon reached the point of diminishing returns and decided to stay at a hard limit of 50000 iterations with population size of 50, as was already mentioned in section 2.

When we lowered the ”resolution” of the GA by decreasing the number of iterations and population up until the point where we got similar results to the Savings algorithm, we found that the Savings algorithm has much better run time & uses much fewer memory.

Depending on the number of nodes, the run-time becomes more similar because Savings has much worse time complexity scaling and when comparing instances with 100-200 nodes, GA is only 8x slower than Savings algorithm. This scales even more with increasing number of nodes, but as was already mentioned in the previous section, neither of these algorithms are suitable for large problems, but GA performs better on them.

When comparing the algorithms on larger number of nodes, we also found that GA serves much more customers per route than on smaller problems. The same goes for the number of vehicles needed to fulfill the requests – Savings algorithm needs more of them than in smaller problems. Finally, the unused space difference also increases for the Savings algorithm, which wastes much more space than in smaller problems.

Depending on what we are aiming for, both algorithm are viable options. As was mentioned in the previous section, the main performance metric is usually the travelled distance. If we take only that into account, the difference is only 18%, which is not insignificant, but it’s also not groundbreaking. If we aim for better results at the cost of worse performance, GA is better. If we instead aim for performance, Savings is the superior choice.

Literature

[1] ALBADR, M. A. Genetic algorithm based on natural selection theory for optimization problems. *Symmetry* (2020).

[2] ALTINEL, I. K., AND ÖNCAN, T. A new enhancement of the clarke and wright savings heuristic for the capacitated vehicle routing problem. *The Journal of the Operational Research Society 56*, 8 (2005), 954–961.

[3] BAKER, B. M., AND AYECHEW, M. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research 30*, 5 (2003), 787–800.

[4] HVATTUM, L. M. Adjusting the order crossover operator for capacitated vehicle routing problems. *Computers & Operations Research 148* (2022), 105986.

[5] PRAVEEN, K. Capacitated vehicle routing problem. `https://github.com/krishna-praveen/Capacitated-Vehicle-Routing-Problem`, 2020.

[6] RAND, G. K. The life and times of the savings method for vehicle routing problems. *ORiON* (2009), 125–145.

[7] WIKIPEDIA CONTRIBUTORS. Genetic algorithm — Wikipedia, the free encyclopedia, 2023. [Online; accessed 6-December-2023].

[8] XING, W., SHU-ZHI, Z., XING, W., HAO, C., AND YAN, L. An improved savings method for vehicle routing problem. 1–4.