Kunnskap for en bedre verden

Department of Computer Science

IT3708 - Bio-Inspired Artificial Intelligence

Project 2 : Optimizing Home Care Service Routing Using a Genetic Algorithm

Author:
Tanguy Guyot

March 9th, 2025

Table of Contents

List of Figures

List of Tables

1 Problem description

1.1 The Home-Care Vehicle Routing Problem

The project 2 problem is a Capacitated Vehicle Routing Problem with Time Windows (CVRPTW), named The Home-Care Vehicle Routing Problem Variant.

The Home-Care Vehicle Routing Problem (HCVRP) involves optimizing the routes of home-care nurses who visit patients at different locations. Each nurse starts and ends their route at a central depot and must visit a specific set of patients while adhering to several constraints. These include a maximum workload per nurse, strict time windows for each patient visit, and a required return time to the depot. The main goal is to minimize the total travel time across all routes while ensuring that every patient receives care within the given time constraints.

1.2 Benchmarks

We were given 10 train instances with benchmark to aim in terms of travel time, in order to test our genetic algorithm and our best solutions. We could also try our solution's travel time and validity on Visma Resolve x Bio-AI website (https://it3708.resolve.visma.com/) For the grading, we were given 3 additional test instances, and we had to get as close as possible to the benchmarks, such as:

- If our value is within 5% of the benchmark value or better, we get 8.33 points (full score)
- If our value is within 10% of the benchmark value or better, we get 6.5 points (full score)
- If our value is within 20% of the benchmark value, we will get 4 points.
- If our value is within 30% of the benchmark value, we will get 2 points.
- Otherwise, we get 0 points.

For information, here are the benchmarks per test instance:

Instances	Benchmark	5% Benchmark	10% Benchmark	20% Benchmark	30% Benchmark
Test instance 0	826	867	909	991	1074
Test instance 1	1514	1590	1665	1817	1968
Test instance 2	900	945	990	1080	1170

Table 1: Benchmark values for test instances

2 Implementation

2.1 Hardware and System Configuration

The experiments were conducted on a computer with the following specifications:

Component	**Specification**
Operating System	Windows 11 Family
Processor (CPU)	11th Gen Intel(R) Core(TM) i5-1135G7, 2.40 GHz, Quad-core
RAM	8GB
GPU	Iris Xe Graphics

Table 2: System configuration used for the experiment.

2.2 Software and Implementation Details

The genetic algorithm was implemented using the following software tools and libraries:

Software	**Version / Details**
Programming Language	Julia 1.11
Development Environment	VS Code
Libraries	JSON, Random, Plots, Clustering, Base.Threads

Table 3: Software and libraries used in the experiment.

3 Experimental setup

3.1 Structure of the genetic algorithm

The code is structured into several modules, each handling a specific aspect of the genetic algorithm:

- **Core Structures (structures.jl)**: Defines the main data structures, ProblemInstance (to store problem parameters) and Individual (representing a solution)
- **Utility Functions (utilitaries.jl)**: Provides helper functions for input handling, constraint validation, and solution formatting.
- **Initialization (initialization.jl)**: Generates the initial population of solutions using heuristic-based or random strategies.
- **Crossover Operators (crossover.jl)**: Implements crossover mechanisms to combine parent solutions and produce new offspring.
- **Mutation Operators (mutations.jl)**: Defines various mutation strategies to introduce diversity and avoid premature convergence.
- **Selection Methods (selections.jl)**: Implements selection techniques such as tournament selection to choose individuals for reproduction.
- **Local Search (local_search.jl)**: Periodically refines the best-found solutions using local search heuristics to improve performance.

3.2 Structures

Two structures were made and used for this implementation.

- **ProblemInstance**: to access to the json data easily.
- **Individual**: to define an Individual. An Individual is made of several routes which is a list of lists of integers (*VectorVectorInt*). Each list of integer is a route taken by a nurse, each integer representing a patient. It also has a total_travel_time, a score, a penalties and a feasability fields, initialized to 0.0 (and false for feasability). When going through fitness function, an individual will have those fields updated.

3.3 Fitness function

Through the *evaluate* function is calculated the fitness. The evaluate function calculates the total travel time of an individual, and also a score which is the total travel time added to a penalty score. This penalty score depends on a penalty factor defined as a parameter, and adds up to the score based on the constraints. Thus, if the capacity of a nurse or if the deadline for the return to the depot time are exceeded, a capacity cost and/or a late depot cost are added to the score. And if a time window is not respected, a delay cost is applied, proportionally to the late time between the elapsed time and the end time of the patient.

It is also considered that if a nurse is early compared to a time window, instead of penalizing the nurse, it is considered that the nurse waits until the start time.

Finally, if there is any penalty, we consider the Individual to be unfeasible (it does not respect at least one constraint). This function returns an evaluated Individual.

3.4 Initialization

The initialization is done through creating an initial population made of different types of heuristics.

- A position-clustered based population: using kmeans, I clustered patients based on their coordinates on the map. Different amounts of clusters were chosen to help diversify populations, and also since there is, by experience, usually less than the maximum amount of nurses involved, so I changed the amounts according to my experience of the results of an instance.
- A time window-clustered based population: Also using kmeans, I clustered patients based on their time window, using two parameters : the median time : $(start\ time + end\ time/2)$; and the length of their time window $(end\ time - start\ time)$. Then, each route is constructed sequentially by taking one individual of each cluster so that they have different time windows, and sorted by start time and end time.
- A totally random based population: a random permutation, also with a random amount of empty routes.

This was motivated by a will of having several diverse yet relevant sources of individuals that could be relevant to crossover afterwards.

3.5 Parent selection

Parent selection is done through a simple Tournament selection.

3.6 Crossover

The crossover method chosen is the one suggested by Visma on their slides of the project 2 kickoff.

- This crossover takes two parents, parent1 and parent2, copies each of them into a child1 and a child2 and chooses one random non-empty route of each one of them.
- **Suppression phase**: The patients of route chosen from child1 are completely deleted from child2, and vice-versa
- **Repair phase**: For each missing patient of a child, we find the best replacement in the child's routes. To do so, we try to insert the patient in every position of each route, and compare the score of the new potential individual to the current one. If it scores lower, then we continue the algorithm by comparing the new best individual and other candidates individuals.

3.7 Mutations

Several mutations were implemented in this genetic algorithm. They occur if the mutation rate is higher than a random float between 0 and 1. The mutations are:

- Swap mutation: swaps two patients of two routes.
- Shuffle mutation: shuffles a subset of a route.
- Insert mutation: inserts a patient in an other route.
- Split mutation: splits a route into two routes (from the middle).
- Merge mutation: concatenates two routes.

This amount of different mutations allows to create diversity in the individual pool using different methods.

3.8 Survivor selection

The survivor selection is a mix of elite selection (taking the best individuals of a generation) and tournament selection, in order to keep diversity and not converge too fast to a local maxima, but still keeping promising individuals. I keep 30% of elite individuals, and 70% are chosen through tournament selection (rule of thumb).

3.9 Local search

A simple local search has also been added, because I noticed that, on some unfeasible solutions which score was close to the travel time, a simple swap could make it into a feasible solution. So for every 5 generations, a local search is done on the best individual to check if better solutions can be done through swapping two neighbouring patients, by evaluating every possibility.

3.10 Island niching

A simple island niching is done when searching for a solution. First, the algorithm is run three times using different initializing parameters. Then we collect the final populations of each algorithm and concatenate them to run the algorithm on them as the new initial population. It allows to get a diverse yet adapted initial population for the problem, and allows some slight improvements in the score.

3.11 Parameters and overall strategy

To find a solution, the genetic algorithm is used 4 times with different initializations:

- 30% position-cluster, 70% random solutions
- 70% position-cluster, 30% random solutions
- 100% time-window cluster solutions
- One last time using a population based on merging the three previous final population obtained through the three first algorithms (Island niching, see 3.10).

Most parameters were chosen based on rule-of-thumb and by experiencing based on the trade-off of improving the solution (more generations and more diversity to search) and computational time. The proportion is [position cluster, random solution, time window].

Order	Proportion	Generations	Population	Tournament	Child Factor	Mutation Rate	Penalty
1	[0.3, 0.7, 0]	60	100	3	4	0.15	300.0
2	[0.7, 0.3, 0]	60	100	3	4	0.15	300.0
3	[0.0, 0.0, 1.0]	60	100	3	4	0.15	300.0
4	Concatenate 1, 2 and 3	30	300	3	4	0.15	500.0

Table 4: Genetic Algorithm Parameters

3.12 Feasability considerations and adaptive penalty

Depending on the amount of generated feasible solutions, the penalty cost applied on time windows will vary so that it penalizes less the individual if there is already a huge amount of feasible solutions, and vice versa. Therefore, it keeps a certain amount of feasible solutions while not penalizing too much which might result in converging to a value too fast. At the end of each generation, I keep at least one feasible solution in order not to lose feasability, and also to get, at the end, at least one feasible solution if the best solution is unfeasible even with the penalty cost.

3.13 Threading

To improve the speed of the algorithm, the crossover and mutation step of each generation is parallelized with 4 threads, because those task are independant to each other. I used the library Base.Threads to do so. A lock had to be set in order to avoid conflicts of access, when pushing the childs to the next population list.

3.14 Adding diversity

I implemented a simple mechanism to add diversity during my genetic algorithm. If the same best solution is obtained through more than 5 generations in a row, then the next population will be injected of additional random solutions, in order to create more diversity for crossovers. As long as the solution remains the same (it converged to one best solution), it will reinject random solutions every 5 generations. This helps creating diversity in the algorithm and avoid abrupt convergence to a solution.

4 Results

4.1 Test0

Nurse capacity : 200
Depot return time : 1236
Objective value (total duration): 828.9368669428338 (< 5% benchmark)

Nurse	Duration	Capacity	Route
Nurse 1 (N1)	50.80	170	D(0) -> 20(10.0-100.0)[0-1226] -> ... -> D(1040.80)
Nurse 2 (N2)	76.07	170	D(0) -> 90(20.62-110.62)[0-1215] -> ... -> D(976.07)
Nurse 3 (N3)	59.62	180	D(0) -> 5(15.13-105.13)[0-1220] -> ... -> D(1139.62)
Nurse 4 (N4)	64.81	160	D(0) -> 43(16.55-106.55)[0-1219] -> ... -> D(1234.81)
Nurse 5 (N5)	101.88	200	D(0) -> 57(35.0-125.0)[0-1201] -> ... -> D(821.88)
Nurse 7 (N7)	97.23	200	D(0) -> 32(31.62-121.62)[0-1204] -> ... -> D(907.23)
Nurse 8 (N8)	95.94	190	D(0) -> 98(30.81-120.81)[0-1205] -> ... -> D(905.94)
Nurse 10 (N10)	127.30	150	D(0) -> 81(47.43-137.43)[0-1188] -> ... -> D(937.30)
Nurse 12 (N12)	59.40	200	D(0) -> 67(12.21-102.21)[12-167] -> ... -> D(1049.40)
Nurse 14 (N14)	95.88	190	D(0) -> 13(30.81-120.81)[30-182] -> ... -> D(815.88)

Table 5: Routes output for test0 (full output file in .zip deliverable)

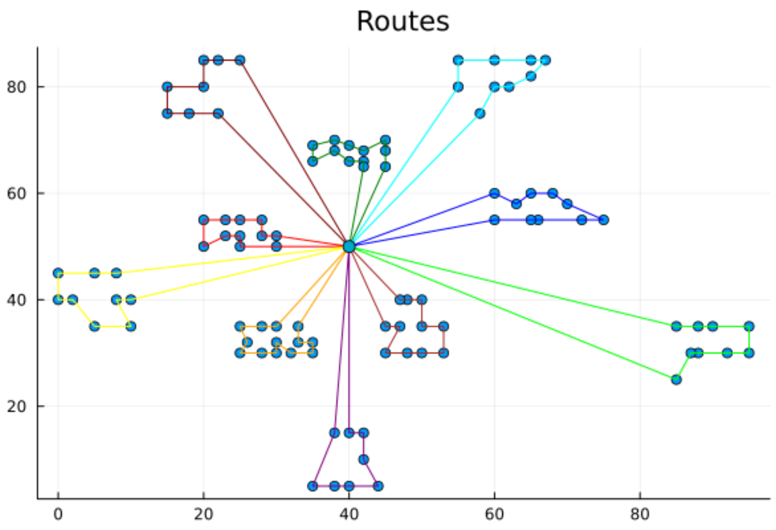

Figure 1: Route planning for test 0

4.2 Test1

Nurse capacity : 200
Depot return time : 240
Objective value (total duration): 1659.1448976431811 (< 10% benchmark)

Nurse	Duration	Capacity	Route
Nurse 1 (N1)	98.03	153	D(0) -> 2(30.81-40.81)[30-160] -> ... -> D(188.03)
Nurse 2 (N2)	56.75	81	D(0) -> 90(4.24-14.24)[74-137] -> ... -> D(116.75)
Nurse 3 (N3)	117.65	66	D(0) -> 63(39.05-49.05)[39-118] -> ... -> D(167.65)
Nurse 4 (N4)	89.77	115	D(0) -> 69(9.22-19.22)[51-71] -> ... -> D(149.77)
Nurse 5 (N5)	104.89	150	D(0) -> 19(40.05-50.05)[58-125] -> ... -> D(164.89)
Nurse 6 (N6)	90.41	101	D(0) -> 61(15.81-25.81)[55-116] -> ... -> D(160.41)
Nurse 7 (N7)	97.29	87	D(0) -> 79(38.47-48.47)[99-119] -> ... -> D(137.29)
Nurse 8 (N8)	138.03	90	D(0) -> 59(42.43-52.43)[42-172] -> ... -> D(188.03)
Nurse 9 (N9)	94.77	118	D(0) -> 39(36.06-46.06)[36-84] -> ... -> D(154.77)
Nurse 10 (N10)	131.44	123	D(0) -> 64(20.62-30.62)[53-92] -> ... -> D(201.44)
Nurse 11 (N11)	121.15	120	D(0) -> 28(55.71-65.71)[55-130] -> ... -> D(181.15)
Nurse 12 (N12)	105.76	71	D(0) -> 92(14.76-24.76)[15-35] -> ... -> D(175.76)
Nurse 14 (N14)	111.41	97	D(0) -> 99(20.52-30.52)[87-107] -> ... -> D(171.41)
Nurse 17 (N17)	118.74	140	D(0) -> 14(35.36-45.36)[35-127] -> ... -> D(198.74)
Nurse 19 (N19)	77.43	99	D(0) -> 65(11.18-21.18)[11-53] -> ... -> D(137.43)
Nurse 23 (N23)	105.63	113	D(0) -> 42(33.54-43.54)[33-76] -> ... -> D(165.63)

Table 6: Routes output for test1 (full output file in .zip deliverable)

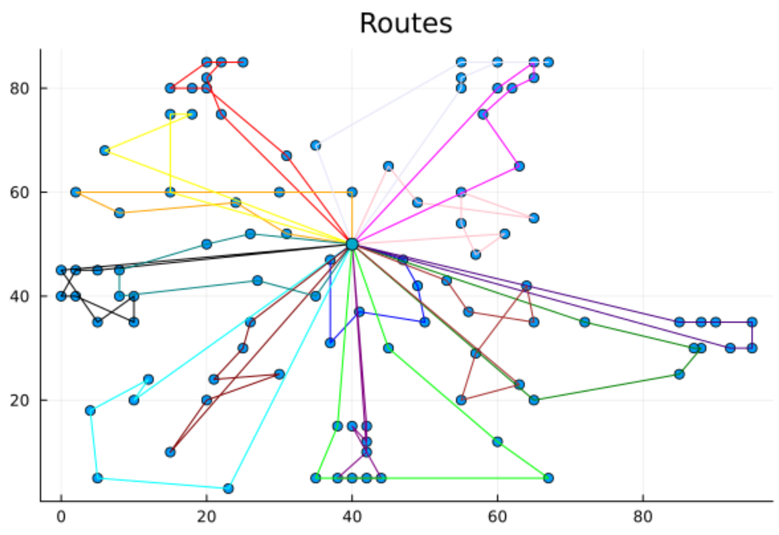

Figure 2: Route planning for test 1

4.3 Test2

Nurse capacity : 200
Depot return time : 1236
Objective value (total duration): 953.4100789339775 (< 10% benchmark)

Nurse	Duration	Capacity	Route
Nurse 1 (N1)	784.88	220	D(0) -> 40(20.62-110.62)[264-411] -> ... -> D(2314.88)
Nurse 2 (N2)	611.62	250	D(0) -> 27(17.12-107.12)[261-406] -> ... -> D(2141.62)
Nurse 4 (N4)	723.95	390	D(0) -> 95(37.20-127.20)[0-1198] -> ... -> D(2343.95)
Nurse 7 (N7)	536.00	270	D(0) -> 28(17.72-107.72)[0-1218] -> ... -> D(1796.00)
Nurse 9 (N9)	665.42	340	D(0) -> 2(20.62-110.62)[0-1215] -> ... -> D(2105.42)
Nurse 11 (N11)	890.57	340	D(0) -> 18(35.36-125.36)[0-1200] -> ... -> D(2510.57)

Table 7: Routes output for test2 (full output file in .zip deliverable)

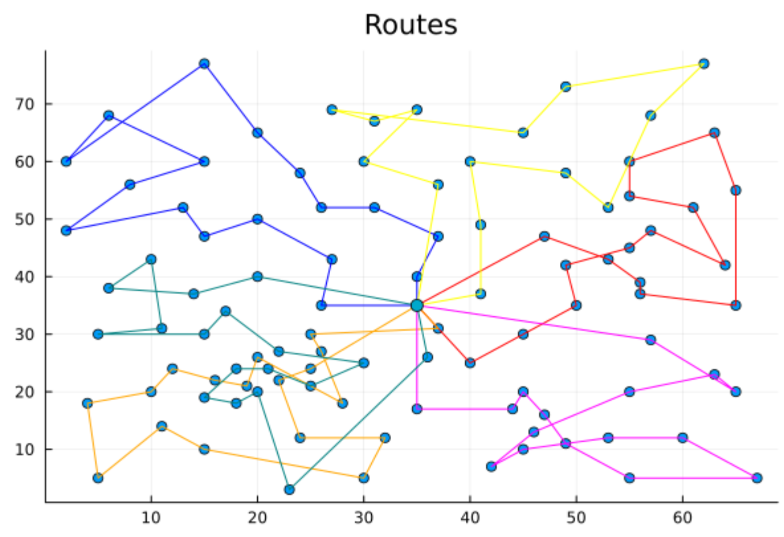

Figure 3: Route planning for test 2

4.4 Discussion about the results

The route outputs taking too much space, the complete file outputs are available in the .zip deliverable. We observe that the results got close to the benchmarks while respecting the constraints. It worked particularly well on test0, which benchmark is only 2 units away from my result.

4.5 Impact of the island niching

With the simple island niching, I could get slightly better results than without. For example, on test 0, I got these scores on the three different algorithms :

- Algorithm 1 : 984.7357200345855 (feasible)
- Algorithm 2 : 960.6845814248707 (feasible)
- Algorithm 3 : 1062.2724384903313 (feasible)

And after concatenating the three final populations into the genetic algorithm as an initial population, I got the score of 953.4100789339775 for a feasible solution, so it got slightly better.

5 Conclusion

Thanks to the genetic algorithm, I could reach the extended benchmarks with feasible solutions that respect all of the constraints. I think that it can definitely be improved, both on optimizing its speed and its results, with more time and ideas. For instance, the local search I am using looks to be used quite often and is not very computationally demanding, but it is still very simple, so maybe I could deep more into it in the future. Being able to tackle such a computationally hard problem and with many constraints and applied to a real-world context shows the relevance of using genetic algorithm in order to deal with real-world optimization problems.

Bibliography

[1] *A Python Implementation of a Genetic Algorithm-based Solution to Vehicle Routing Problem with Time Windows (VRPTW)*. URL: https://github.com/iRB-Lab/py-ga-VRPTW?tab=readme-ov-file (visited on 8th Mar. 2025).

[2] Visma Resolve Camilla Dybdal Espen Grødem. 'Project 2 Kick-Off and Guest Lecture'. In: 2025.

[3] Xuewen Xi Jiani Liu Lei Tong. 'A genetic algorithm for vehicle routing problems with time windows based on cluster of geographic positions and time windows'. In: (2024). URL: https://www.sciencedirect.com/science/article/pii/S156849462401367X.

[4] S.R. Thangiah; K.E. Nygard; P.L. Juell. 'GIDEON: a genetic algorithm system for vehicle routing with time windows'. In: (1991). URL: https://ieeexplore.ieee.org/document/120888.