Collaborative Intelligence

Joanny Tavernier, Benoit Leroux, Elise Françoise, Raphaël Roiné, Justin Bauer, Aude Pinson

April 21, 2024

Contents

Link to our GitHub

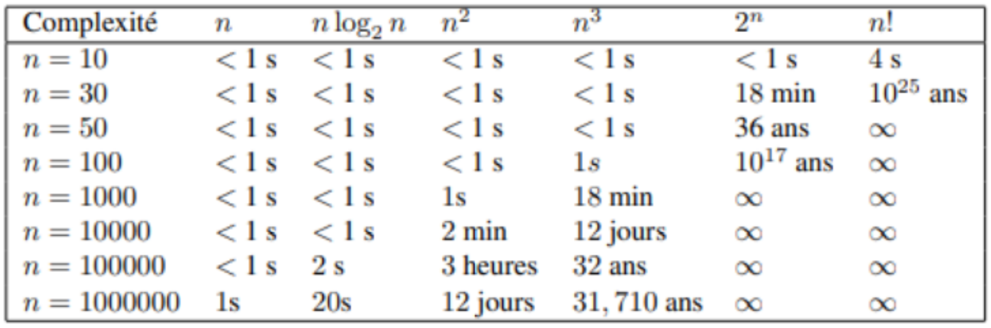
1 The problems of VRPTW

1.1 Description

The VRP problem is a complex optimization problem that was first proposed by Dantzig and Ramser in 1959, to model how a fleet of homogeneous trucks could serve the demand for oil from a number of gas stations from a central hub with a minimum of travel distance. It was then generalized to best fit with the complexity of the real world, and now it is generally known as the Vehicle Routing Problem with Time Windows (VRPTW). The VRPTW is a complex optimization problem that could be summed up as : how to look for the optimal set of routes to be performed by a fleet of vehicles to serve a set of customers within their assigned time windows. Beginning and ending at a given depot, a company has to satisfy the demand of several clients and optimize its costs (using the least trucks as possible and doing the smallest distance as possible) with the following constraints : each vehicle has maximum weight that he can take, the clients are only able to receive their supply at a given time-window. It is a generalization of the VRP, which consists of the same problem, but clients are ready to get their supply all day long. In the VRPTW, if a vehicle arrives too early and the window is not open, he has to wait for the window to open, and it is not allowed to arrive late. This problem is obviously important in the theoretical domain of optimization, but it also has a real impact on the real world. Indeed, the advances on this problem have enabled companies to improve their supply-chain services. This problem is interesting in a wide range of concrete domains : deliveries to supermarkets, security patrol service, school bus routing etc., so what is at stake in the research of this problem is the ability to optimize transport, delivery, and security... The VRPTW can be modeled as a directed graph, a fleet of homogeneous vehicles, and a set of N customers. The graph has N vertices between 1 and N that represent the clients, and two vertices, 0 and N+1 that represent the beginning and the ending of each cycle of the graph. The direct connections between the depot and the customers and among the customers are represented with a set of arcs. Each arc is related to a cost cij. Each client must be serviced once, every route originates at vertex 0 and ends at vertex N+1.

1.2 Complexiy

The main issue of the VRPTW is the complexity to find an optimal solution. The time complexity of a problem is defined by the number of operations that are required to solve it. Here is a table that explains why optimizing its complexity is so important :



Complexité	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} ans
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	36 ans	∞
$n = 100$	< 1 s	< 1 s	< 1 s	$1s$	10^{17} ans	∞
$n = 1000$	< 1 s	< 1 s	1s	18 min	∞	∞
$n = 10000$	< 1 s	< 1 s	2 min	12 jours	∞	∞
$n = 100000$	< 1 s	2 s	3 heures	32 ans	∞	∞
$n = 1000000$	1s	20s	12 jours	$31,710$ ans	∞	∞

Figure 1: complexity of algorithm

The VRPTW is a NP-hard problem, which means that it is a non-deterministic polynomial-hard problem. Coarsely, it is really hard to find a solution and really hard too to verify if the solution is optimal. The complexity of the problem grows exponentially with the size of the input data. That is why researchers don't use exact algorithms, but approximation algorithms to get as close as possible to the optimal solution. Indeed, as we can see with the previous table, it is absolutely unrealistic to use some algorithms with exponential complexity in order to solve problems with a great size of data. Even if the performance of the processing increases fast, for the next few years, the hardware and software won't be able to turn 10^{17} years into a realistic time for example. So several approximation algorithms are used to find acceptable solutions to the problem, such as simple algorithms, hybrid algorithms or machine-learning. This is, for the moment, the best compromise found to compute great solutions in an acceptable time. In this work, we will show the three that we used and tried to make them collaborate. In order to get a solution, we had to use a multi-agents approach which consists in finding solutions thanks to different algorithms, and then make them collaborate. The three meta-heuristics that we used are the tabou algorithm, the Recuit simulated and Genetics algorithm.

1.3 State of the art

Nowadays, the VRPTW problems are more realistic than ever, and they try to embrace real-world complexities. They are really aimed at solving concrete logistic issues. The research on the VRP problems grew up for the last decades and is still improving. Some exact methods are quite popular, even though they can only be used for small size problem. This is the case for the branch and bound approach, which involves partitioning the problem into subproblems and solving each subproblem to the optimal level, using bounds to eliminate the need to consider suboptimal solutions. The problem of this algorithm, as we said, is that it is unusable with a big size problem. However, it provides an optimal solution so it is a great way to solve smaller problems. Most of the time, some heuristics that give approximated optimal solutions are chosen to deal with a VRPTW problem.

2 Optimization by Metaheuristics

2.1 Tabu and Classification

The Tabu Search is a general iterative method that was introduced in 1986 by Glover. It turns out to be really efficient, mostly for scheduling problems. It serves as a meta-heuristic, steering a local heuristic search process towards exploring solution spaces that extend beyond local optimality. A main aspect of Tabu Search lies in its adoption of adaptive memory, fostering a search behavior characterized by enhanced flexibility. As such, strategies anchored in memory become the cornerstone of Tabu search methodologies, arising from a pursuit of "integrating principles." This pursuit involves skillfully amalgamating diverse memory structures with potent strategies to effectively leverage them. The effective principle of the Tabu Search, in an optimization problem is the following : at the first iteration, a solution must be entered in the algorithm. Then, the neighborhood of this solution is computed and the next solution is chosen in this neighborhood in order to minimize the cost function. This process, which is the same as the descent gradient, is repeated until an local minimum is found. When it is the case, it is impossible to find a neighbor that has a smaller cost than the current solution. Once a local minimum is reached, the algorithm chooses the neighbor that has the "least bad" cost. In order to prevent the algorithm from cycling between these two solutions, a Tabu list was created, and the local minimum is added to it so that it is temporarily forbidden to come back to this solution. The solution may be removed from the Tabu list in the future if it satisfies an aspiration criterion, based on an aspiration function. This algorithm is repeated until the cost is satisfying or until the number of iterations without improving the cost function is too large. Some parameters of the Tabu Search algorithm can be modified in order to adapt to a precise problem, such as the aspiration function, whose choice is nearly infinite or even the constitution of neighborhood. Here is an illustration of an optimization problem where the Tabu Search method was used :

It illustrates well the importance of the Tabu list, which enables not to be stuck in a local minimum.

2.1.1 Specific Algorithms for the Vehicle Routing Problem with Time Windows (VRPTW)

As the Tabu Search algorithm requires us to identify movements that have already been done in order not to come back to a previously seen solution, we implemented a basic movement as follows : [type, index1, index2, assessment]. The type corresponds to one of the eight movement functions (Inter route swap, Inter

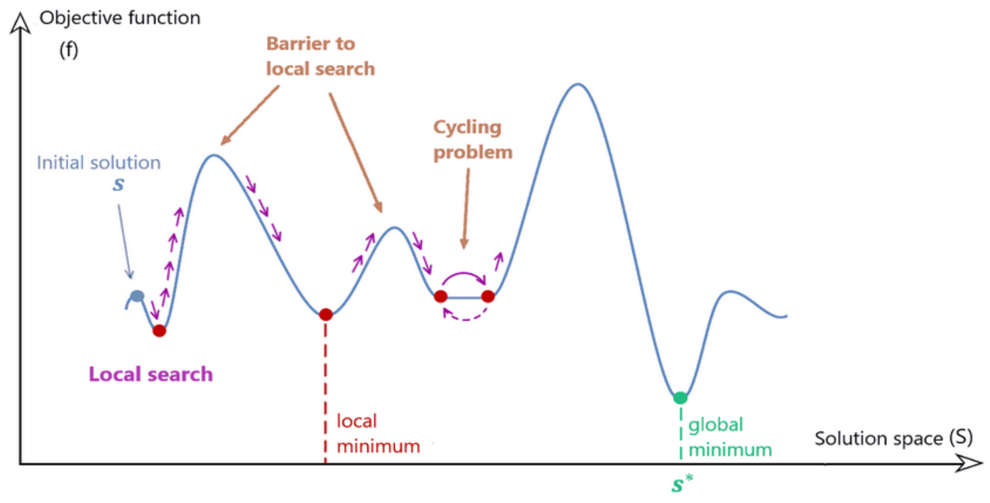


Figure 2: Tabu algorithm

route shift, Intra route swap, Intra route Shift, Two Intra Route Shift, Two Intra Route Swap, Pop small route, Pop random route), represented as integers. The indexes of the movement are in fact the positions of the two companies associated with the movement. The final assessment enables us to identify the movement because it is nearly unique for each solution.

The classification algorithm offers a significantly superior solution compared to the previous taboo version, and does so much faster. The complex movements performed by the classification algorithm initially led to the neglect of using the taboo list. However, it is interesting to reintroduce the latter at a later stage, as it can offer complementary benefits and further enhance the efficiency of the algorithm. Experimentally, it has been observed that executing the taboo after classification improves the proposed solution by 20%.

In the first step of the current algorithm, there are only two functions that are responsible for the movement : one enables to move clusters, and the other to move aberrant points. We can choose whether we should use one or no thanks to thresholdings that identify the parts of the solution to optimize. These thresholdings play an important role in the determination of the most appropriate strategy movement, helping to target specifically the areas that need an improvement.

An interesting way to choose the threshold is to use a sinusoidal function, as this allows for the movement of clusters of widely varying sizes.

Algorithm 1: Find Cluster

```
input  : $s$: sequence, $\epsilon$: threshold, $clients$: list of client data
output: $cluster$: list of clusters
$L\_distance \leftarrow$ inter_distance($s, clients$);
$n \leftarrow$ length of $L\_distance$;
$cluster \leftarrow []$;
$c \leftarrow 0$;
while $c < n - 1$ do
    $segment \leftarrow []$;
    $k \leftarrow c$;
    while $k < n$ and $L\_distance[k] < \epsilon$ do
        Append $k$ to $segment$;
        $k \leftarrow k + 1$;
    Append $k$ to $segment$;
    if length of segment $> 1$ then
        Append $segment$ to $cluster$;
    $c \leftarrow k + 1$;
return $cluster$;
```

Algorithm 2: Find Outlier Point

```
input  : $s$: sequence, $\epsilon$: threshold, $clients$: list of client data
output: $L\_outlier$: list of outlier points
$L\_distance \leftarrow$ inter_distance($s, clients$);
$n \leftarrow$ length of $L\_distance$;
$L\_outlier \leftarrow []$;
$c \leftarrow 0$;
while $c < n$ do
    $segment \leftarrow []$;
    $k \leftarrow c$;
    while $k < n - 1$ and $L\_distance[k] > \epsilon$ do
        Append $k$ to $segment$;
        $k \leftarrow k + 1$;
    if length of segment $> 1$ then
        Remove the first element from $segment$;
        Append $segment$ to $L\_outlier$;
    $c \leftarrow k + 1$;
return $L\_outlier$;
```

Algorithm 3: Nearest from Cluster

```
input  : $s$: sequence, $one\_cluster$: cluster, $clients$: list of client data
output: $indice\_min$: index of nearest point in the cluster
$n \leftarrow$ length of $s$;
$s\_cluster \leftarrow$ copy of $one\_cluster$;
$G \leftarrow$ barycentre ($s$, $one\_cluster$, $clients$);
$d\_min \leftarrow$ distance_squared($clients[s[0]]$, $G$);
$indice\_min \leftarrow 0$;
for $k \leftarrow 0$ to $n - 1$ do
    if $s[k]$ not in $s\_cluster$ then
        $d \leftarrow$ distance_squared($clients[s[k]]$, $G$);
        if $d < d\_min$ then
            $d\_min \leftarrow d$;
            $indice\_min \leftarrow k$;
return $indice\_min$;
```

Algorithm 4: Shift Cluster

```
input  : $s$: sequence, $one\_cluster$: cluster, $clients$: list of client data
output: $n\_s$: shifted sequence
$restricted\_s \leftarrow$ elements of $s$ not in $one\_cluster$;
if length of $restricted\_s < 2$ then
    return $s$;
$G \leftarrow$ barycentre ($s$, $one\_cluster$, $clients$);
$indice\_insertion \leftarrow$ neer_from_cluster ($restricted\_s$, $one\_cluster$, $clients$);
$g \leftarrow restricted\_s[: indice\_insertion]$;
$d \leftarrow restricted\_s[indice\_insertion :]$;
$n\_s \leftarrow g + one\_cluster + d$;
```

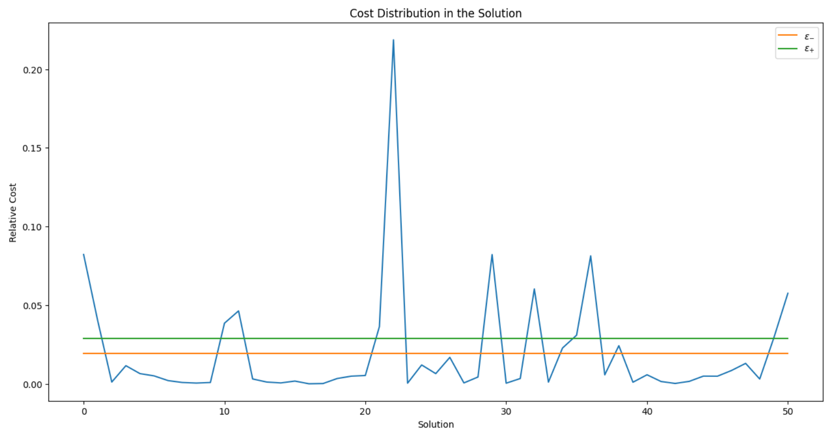


Figure 3: Use of thresholds to identify areas for improvement

2.1.2 Small-sized Database

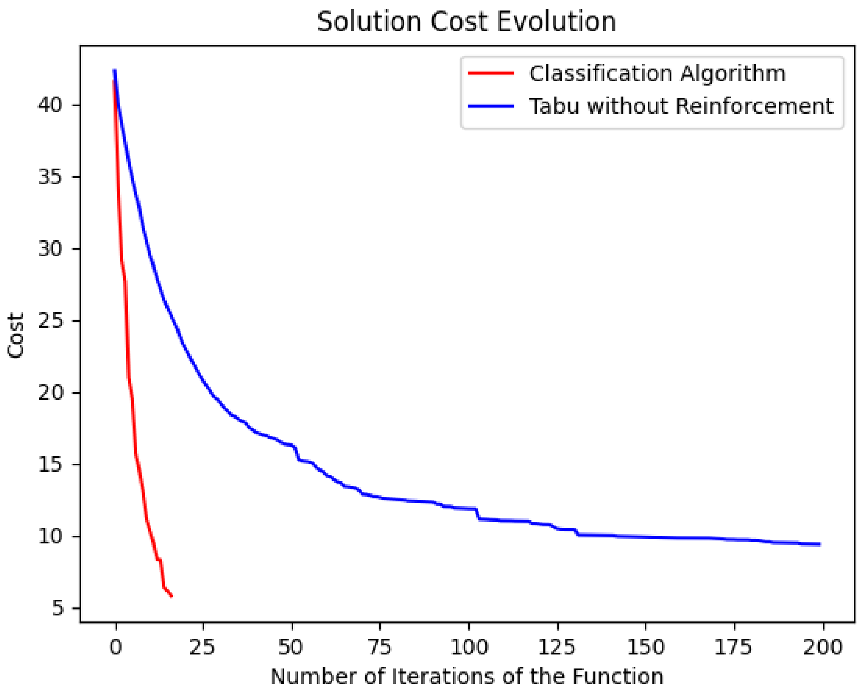


Figure 4: Comparison of algorithm versions' behavior (trial with 50 clients)
The classification algorithm is very fast but quickly gets stuck in the first local minimum (which sometimes corresponds to a very good solution when dealing with a small database). However, this premature convergence does not seem to affect the Tabu search algorithm.

2.1.3 Large-sized Database

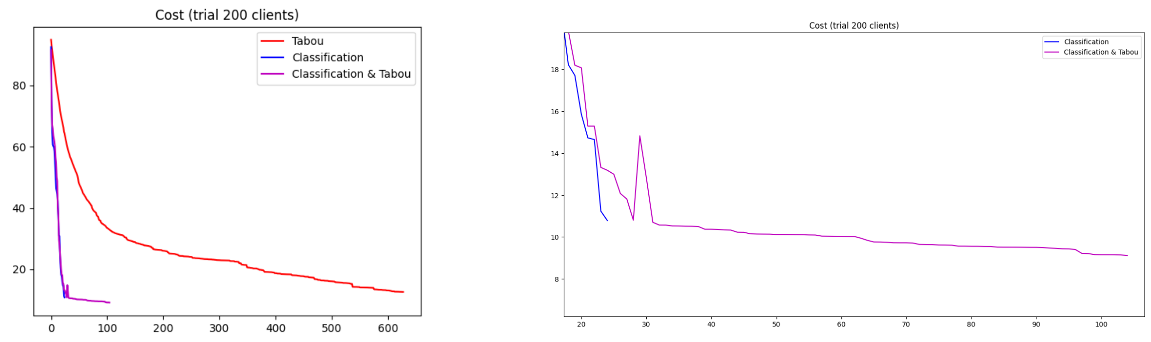


Figure 5: Cost for the heuristically in series.

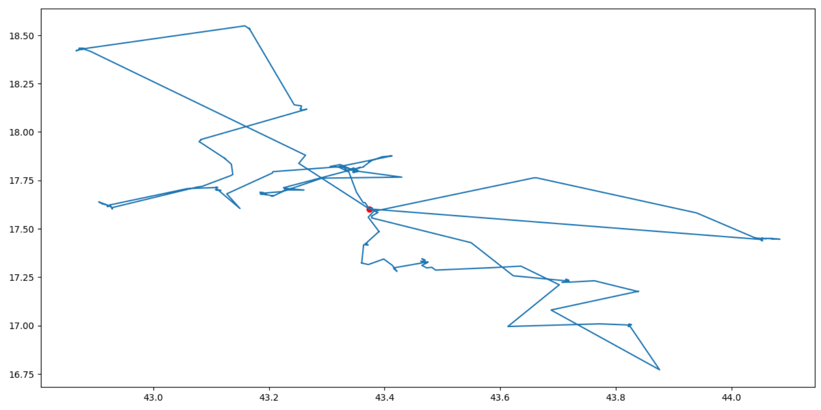


Figure 6: The solution path for 300 clients (Classification & Tabu)
Adjustment of the initial solution increased the value of the tabu search hyperparameters, allowing to propose a solution within an acceptable time frame.

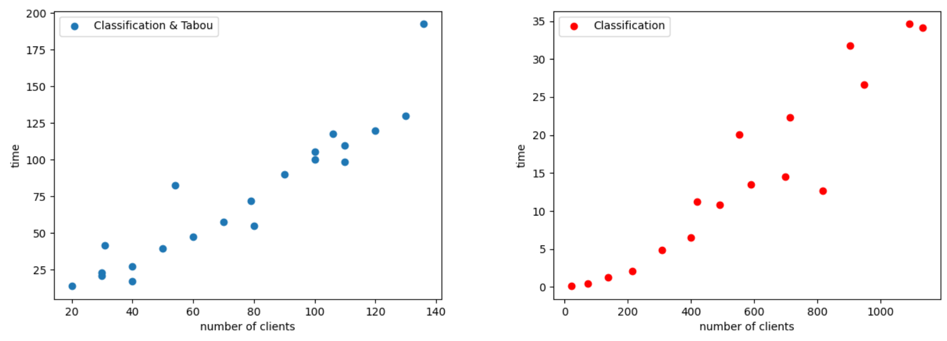


Figure 7: Time of execution (Classification & Tabu, Classification)

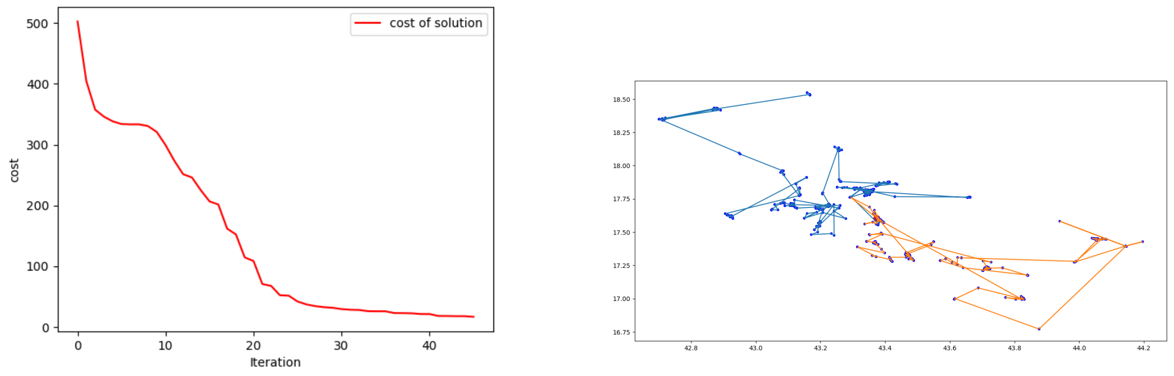


Figure 8: Evolution of the cost for and path Classification solo (1158 clients)

We notice that the optimization curve of the classification algorithm presents two plateaus. During the first descent, the algorithm takes on the task of ordering the initially randomly ordered data. Once the data is sufficiently ordered, it manages to identify clusters and outliers, which is represented by the second descent on the graph.

2.2 Simulated Annealing

The annealing simulated algorithm was first studied with the Metropolis experiments in 1953, whose work ended up as a simple way to describe how an unstable physical system comes to a state of balance, at a given temperature. Here, the algorithm mimics the evolution of the system when a small perturbation influences it. If the energy of the system goes lower, the system evolves in this way. In contrast, if the energy level increases, the new state may be accepted, but only with a certain probability :

$$prob(dE, T) = e^{-\frac{dE}{k.T}}$$

Where dE is the variation of energy due to the perturbation, T the temperature of the system, and k the Boltzmann constant. The annealing technique consists in heating up the system to get it liquid, and then cooling it slowly, to get successive states of balance. This method was first used on optimization problems in the 1980's, with the solution replacing the system, the energy level replaced by the cost function, and the set of possibles next states are represented by the neighborhood of the current solution. For the simulated annealing, if the cost function is reduced by a perturbation of the previous solution, the new solution obtained thanks to this perturbation is conserved, and if the cost function increases, the same probability as in the real annealing is used to choose whether the solution is conserved or not. This heuristics based on this physical model will progressively reduce the cost function to get closer to an optimal solution.

2.2.1 Specific Algorithms for the Vehicle Routing Problem with Time Windows (VRPTW)

In the case of VRPTW, the annealing simulated algorithm must be adapted to our problem. Which means creating adequate cost function and a neighbouring solution generator. The cost function is defined by the distance traveled by the vehicle. For the generation of a neighbouring solution we created a function that swap 2 random points between or inside a road. The stop parameter of this function is a previously fixed number of iterations without improvement. To make the processing of the large amount of data easier, we decided to create our own dictionary with the customer number and its latitude and longitude. The solution it returns is a list as follows : [Cost of the solution, [[route1],[route2],...]] In order to make this algorithm work we needed to generate an initial solution. For that we created another function that randomly fills a number of vehicles whilst taking into account the capacity of the vehicle.

2.2.2 Small-sized Database

We can try and run our algorithm on 20 points, which gives us the following result. For this graph, the cost is 120. If we run our code on a slightly bigger amount of points (here 70), where we have more than one vehicle, we observe a satisfying result. There are no aberrant links between two points or loops in this solution. In this case the cost is 303, which is consistent regarding the one we had for 20 points.

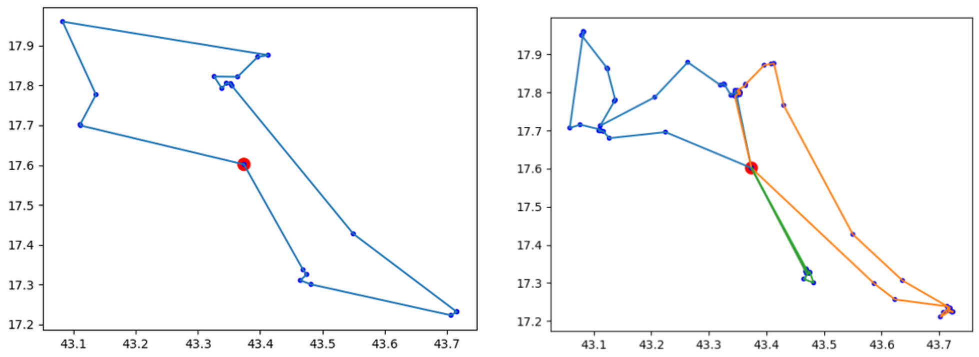


Figure 9: Solution for 20 and 70 clients.

Analysis of the results With these applications we can conclude that this algorithm works quite well on small databases. However, it doesn't always give us a satisfying solution which means it gets stuck in local minimums and doesn't manage to escape this local minimum. It always gives a satisfying (and not aberrant) solution but it might not be the most optimized solution. In spite of the database being quite small, the algorithm takes quite a long time to run and give a solution. It is important to note that the way the first solution is generated and the swap function works result in the creation of some biases. Given that the swap function only exchanges two points intra or inter route but never creates or suppresses a route constrains the final solution.

Interest of the study The use of an annealing simulated algorithm to solve a vehicle routing problem is of interest because it will always return a correct solution even though it might not be the best, most optimized one. This method gives correct results and is quite easy to implement however the adjustment of the parameter can be challenging.

2.2.3 Large-sized Database

We can run our code on 300 points, which gives us the following graph. For this solution, the cost is 1037.

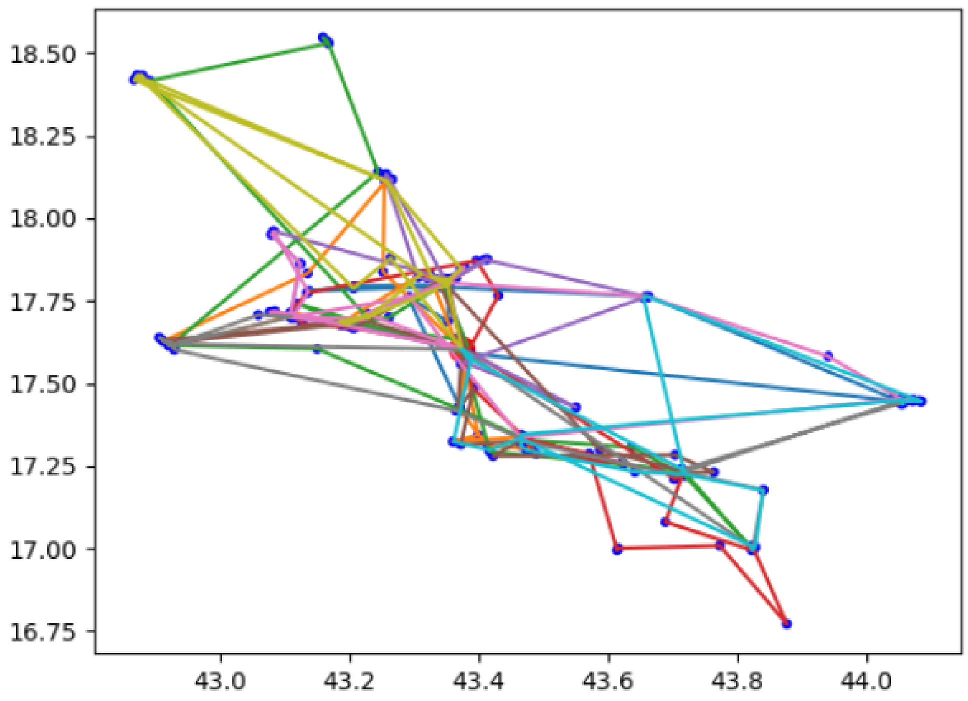


Figure 10: Solution for 300 clients.

Analysis of the results On a bigger database we can observe that the solution given by the annealing simulated algorithm is not optimal. That can be explained by the biases induced at the initialization and in research of a neighboring solution as explained for the small databases. Since running this algorithm was taking a very long time, we decided to make an extrapolation in order to determine the temporal complexity of our algorithm. It is represented in the following graph and led us to determine a polynomial complexity of O(n^2) which is a satisfactory complexity.

Interest of the study Working on a bigger database helps us better see the coherence and the accuracy of our algorithm. Despite our algorithm being slow and the solution not always being optimized, we can conclude that annealing simulated algorithm is a functional way to solve a vehicle routing problem.

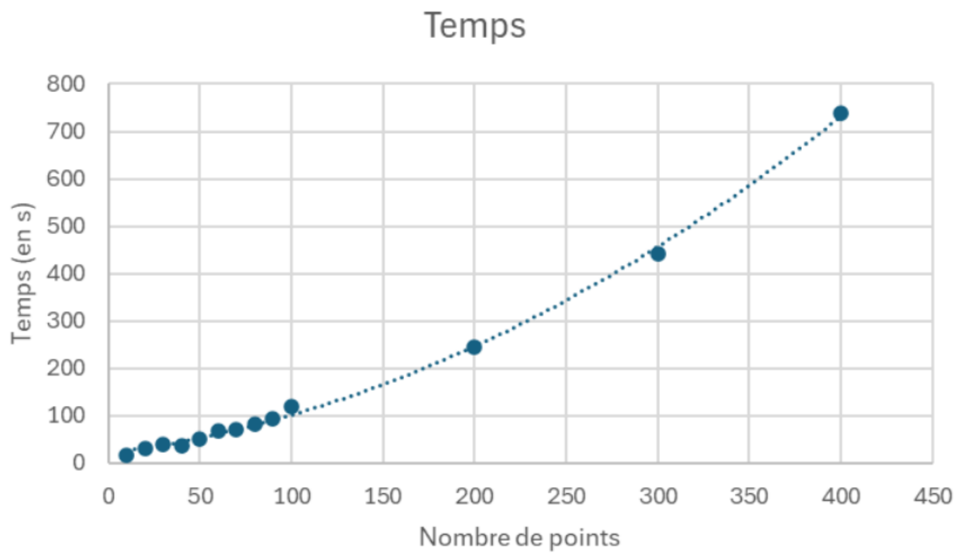


Figure 11: Time of execution
Not optimal result on big database (biases) and very slow but polynomial complexity : $O(n^2) \rightarrow$ good thing.

2.3 Genetic Algorithm

The genetic algorithms are based on the mechanism of natural selection : the survival of the fittest. A genetic algorithm repeatedly modifies a population of individual solutions. At every stage, the genetic algorithm picks some individuals from the current group to become parents and then makes new babies for the next group. As time goes on, this group changes bit by bit, moving closer to the best possible answer. You can use the genetic algorithm to solve different kinds of problems that regular methods struggle with, like when the goal changes suddenly, or there's no clear direction, or it's really complicated. It's also handy for problems where some parts have to be integers. Here is an image illustrating this :

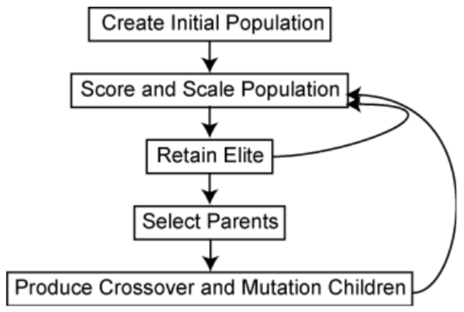

Create Initial Population
Score and Scale Population
Retain Elite
Select Parents
Produce Crossover and Mutation Children

Figure 12: the genetic algorithms

To make the population evolving, the genetic algorithms rely on three main rules:

1. The selection rule, which will enable to select the best parents to create the next generation.
2. The crossover rule that will define how should the parents be combined to form the next generation.
3. The mutation rule, which simulates a random modification to simulate children.

As we can see, these algorithms really rely on real-life principles.

2.3.1 Specific Algorithms for the Vehicle Routing Problem with Time Windows (VRPTW)

The global idea of our genetic algorithm is to create a population where each individual represents a potential optimal solution to the VRPTW problem. The population is located in an initial environment, and at each iteration, the best solutions are selected thanks to a stochastic method. Then they reproduce together to form the next generation. The selection enables the algorithm to get closer to an optimal solution and make the algorithm converge. As in a natural environment, having a large population enables us to have a wide range of solutions, which prevent us from getting stuck to local minimas.

Algorithm 5: Generate Next Generation

```
input  : pop: list of cycles
output: next_gen: list of cycles
// Select random individuals in pop
// Class these individuals by fitness (priority queue insertion)
// Reproduce best individuals while next_pop size ≤ pop_size
next_gen ← [];
while len(next_gen) < self.pop_size do
    sub_select ← sorted(sample(pop, 10), key=lambda x: self.fitness(x));
    sub_select ← sub_select[:5];
    p1 ← choice(sub_select);
    p2 ← choice(sub_select);
    next_gen.append(self.crossover(p1, p2));
self.scores.append(self.fitness(next_gen[0]));
return next_gen;
```

Algorithm 6: Evolve

```
input  : None
output: None
pop ← self.init_pop;
for _ in range(self.iteration) do
    pop ← self.next_gen(pop);
self.final_pop ← pop;
```

Algorithm 7: Crossover

```
input  : p1: cycles, p2: cycles
output: crossover: cycles
crossover ← [p1[i] if uniform(0, 1) ≥ self.cross_proba else p2[i] for i in range(self.n)];
return crossover;
```

2.3.2 Small-sized Database

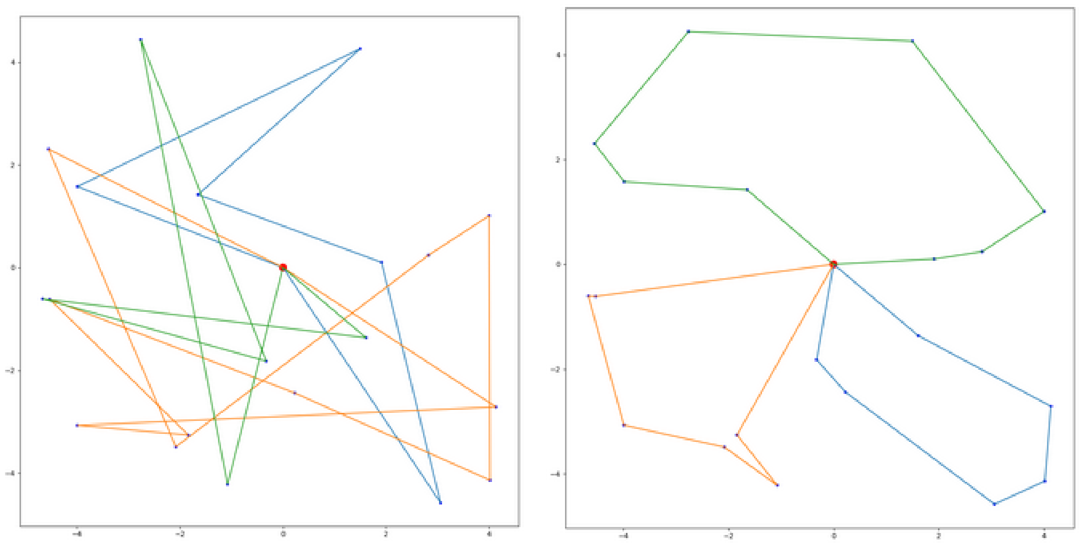


Figure 13: Solution optimization using genetic algorithm.

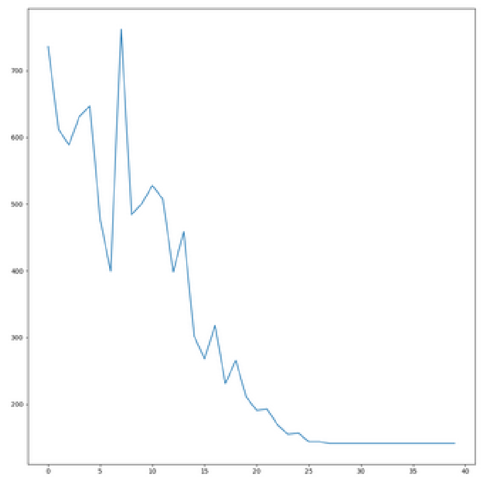


Figure 14: Cost of solution with genetic algorithm.

Figure 13 illustrates the execution of our algorithm on random data. On the left we observe a random individual in the initial population while on the right we observe any individual in the final population. Noted that there may be a better individual in the final population. Figure 14 demonstrates an interesting behavior: peaks in solution costs come directly from bad mutations introduced into a population, but this guarantees not getting stuck in a local minimum. The larger the population size, the lower the chances of getting stuck.

3 Collaborative Optimization: MAS + Metaheuristics

A multi-agent system (MAS) is composed of multiple interacting intelligent agents - autonomous entities that can sense, learn models of their environment, make decisions and act upon them. Agents in an MAS can be software programs, robots, drones, sensors, humans or a combination. Each agent has specialized capabilities and goals, they can also work together, share information or divide tasks in a customizable way. In a VRPTW, the agents can be algorithms that are individually capable of giving an acceptable solution to the problem. In order to implement our metaheuristics into a multi-agent system, we need to create 3 agents for the three different algorithms.each of those agents have a contact function in order to allow them to exchange information when they encounter. This will be used for the reinforcement and is known as friend-or-foe reinforcement. The agents work together either in a cooperative or a competitive way. In a cooperative functioning, the agents take both the cost and the solution of the best solution between the three and in a competitive functioning they only take the best cost without the solution. We also created a model for the agents to interact with a SimultaneousActivation to orchestrate the multi-agent system as well as a data collector in order to retrieve the best solution at each step. During that stage we faced a compatibility problem. Our algorithm, since coded with different formalisms, had trouble communicating together. Even if they all wrote the solution the same : [cost, [[route1], [route2],...], they were not operating the same which made the execution of the multi-agent system way harder. Moreover, since all the algorithms were individually fairly long to run, running them altogether several times wasn't possible.

Algorithm 8: ModelVRPTW

```
input  : population, coord_dict, t0 = 10, nbiter_cycle = $10^2$, a = 0.99, N_RS = 1, N_TB = 1, N_AG
         = 1
output: None
// Initialize
super.__init__();
self.dict ← coord_dict;
self.t_init ← t0;
self.nbiter_cycle ← nbiter_cycle;
self.a ← a;
self.population ← population;
self.NRS ← N_RS;
self.NTB ← N_TB;
self.NAG ← N_AG;
self.best_cost ← random.choice(self.population)[0];
// Schedule
self.schedule ← SimultaneousActivation(self);
// Create Recuit agents
for i ← 0 to int(self.NRS) do
    solution_test ← random.choice(self.population);
    agent ← AgentRecuit(i, self, solution_test, self.dict, self.t_init, self.nbiter_cycle, self.a);
    self.schedule.add(agent);
// Create Tabou agents
for i ← 0 to int(self.NTB) do
    solution_test ← random.choice(self.population);
    agent ← OptTabouAgent(solution_test, nb_iter_max, max_tabou, taille_vois, Capacite);
    self.schedule.add(agent);
// Create Genetic agents
// Datacollector
self.datacollector ← DataCollector(model_reporter = {"Meilleur coût" : lambda m : m.best_cost});
```

Algorithm 9: step

```
input  : None
output: None
// Advance the schedule by one step
self.schedule.step();
// Update the best cost
self.best_cost ← random.choice(self.schedule.agents).cost;
// Collect data
self.datacollector.collect(self);
```

4 Collaborative Optimization: MAS + Metaheuristics + RL

An illustrative example of generating a solution neighborhood involves using reinforcement learning, as in the case of the taboo algorithm. In this context, the algorithm explores different actions to generate neighboring solutions based on learned information over time. For instance, the algorithm may choose to explore new actions randomly to break away from current solution configurations, or to exploit learned knowledge to select actions based on their past performance. By using this approach, the algorithm can generate a diverse set of neighboring solutions while effectively leveraging learned information to guide the search towards higher-quality solutions.

To improve the reinforcement algorithm, we tried using a q-learning system to improve the use of the neighboring function; Basically, the q-learning algorithm consist of calculating a 'reward' for each state the function comes to after choosing an action, using a factor named q, and saving it in a q-table. This would allow us to know, after a few iterations, what kind of action would be the most rewarding for each kind of state; for our example, it would allow us to choose the 'best' and most rewarding kind of neighboring function. However, we did not succeed in using the q-learning algorithm for the function. Firstly, using git hub as a platform sometimes got in the way of sharing code, with some functions being lost over the course of merging and using different branches. But our main issues was that as the different pairs worked on the three heuristics, we did not use the same way of coding solutions and the functions itself. As a result, we had to use differents neighboring function for each heuristic, which complicated a lot the q-learning algorithm, to the point where we had to drop the algorithm and use reinforcement without it.

5 Conclusion and Perspectives

The classification algorithm is a promising avenue to explore to save time and obtain a reasonably accurate initial solution. One way to improve it would be to introduce recursion when a sufficiently large cluster is identified. This would help address the issue of points within the cluster being too close in relative distance. By adding recursion, the algorithm could subdivide clusters into smaller sub-clusters, which could assist in better distributing the points and achieving more precise results.

Another way to enhance the algorithm would be to increase the problem resolution by considering fictitious points located at the barycenter of each pair of clients in the current solution. This approach would allow clusters to be shifted to much more precise locations, taking into account not only the positions of the clients but also the centers of gravity of different regions. This could lead to a more balanced distribution of clients and better optimization of traveled distances.

Lastly, reinforcement learning has not been implemented in the classification algorithm. It might be interesting to use a reinforcement learning algorithm to choose and adjust the value of the classification threshold ε_+ and ε_-.

These avenues had begun to be explored and could represent promising directions for improving the classification algorithm but did not have time to materialize within the allotted time frame. By combining recursion to better handle large clusters and the use of fictitious points to increase the precision of cluster

Algorithm 10: Generate Neighborhood

```
input  : $s$: current solution, $taille\_vois$: neighborhood size,
$Tabou$: tabu list, $max\_tab$: maximum tabu size
output: $V$: neighborhood solutions
$mvt \leftarrow$ mouvement ();
$eval \leftarrow$ cout ($s$, $clients$);
$c \leftarrow 0$;
$V \leftarrow []$;
$state \leftarrow q\_agent.previous\_state$;
while $c < taille\_vois$ do
    $a \leftarrow$ e_greedy ($state$);
    $(new\_sol, mv) \leftarrow$ perform_action ($s$, $a$);
    while $mv == []$ do
        $a \leftarrow$ e_greedy ($state$);
        $(new\_sol, mv) \leftarrow$ perform_action ($s$, $a$);
        $cout\_solution \leftarrow$ cout ($new\_sol$, $clients$);
        $reward \leftarrow$ reward ($cout\_solution$, $eval$);
        update_q_table ($q\_agent.previous\_state - 1$, $a$, $reward$, $a$);
        $q\_agent.previous\_state \leftarrow a - 1$;
    $mv.append(eval)$;
    $mv\_opposer \leftarrow [mv[0], mv[2], mv[1], eval]$;
    if $new\_sol$ not in $V$ and $mv\_opposer$ not in $Tabou$ then
        $V.append((new\_sol, mv))$;
        $c \leftarrow c + 1$;
return $V$;
```

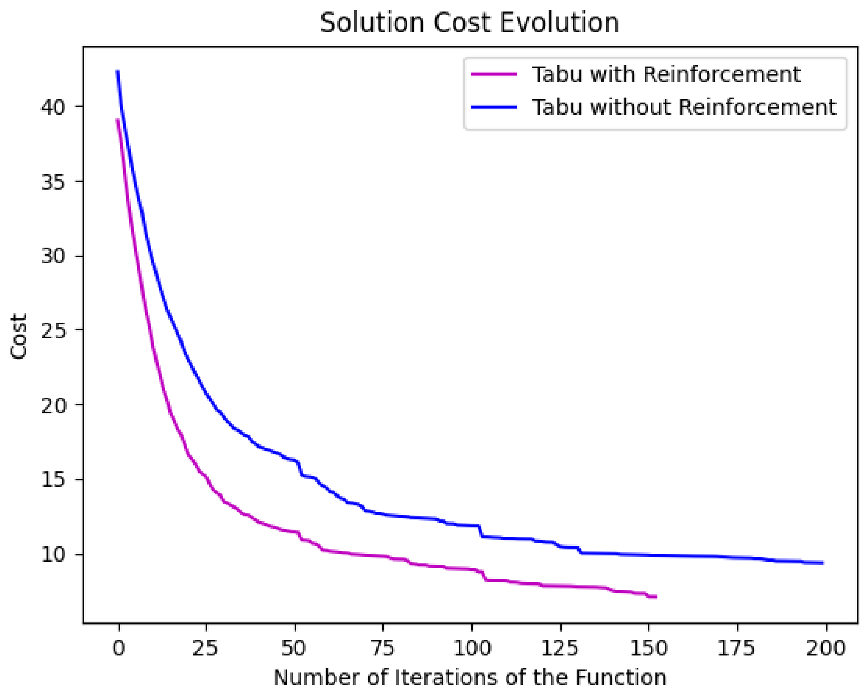


Figure 15: Comparison with reinforcement learning implementation (50 clients).

localization, we could potentially obtain initial solutions of better quality and more quickly.

6 Annexes

[1] Vehicle Routing Problem with Time Windows - ScienceDirect

[2] Vehicle Routing Problem with Time Windows - AlvarezTech

[3] Vehicle Routing Problem with Time Windows - MDPI

[4] Vehicle Routing Problem with Time Windows - UV.es

[5] Genetic Algorithm - MathWorks

[6] Genetic Algorithm - NCBI

[7] What is Annealing? - TWI Global

[8] A Simulated Annealing Algorithm for the Vehicle Routing Problem with Time Windows and Synchronization Constraints. 7th International Conference, Learning and Intelligent Optimization Sohaib Afifi, Duc-Cuong Dang, Aziz Moukrim.

[9] Genetic Algorithms and Random Keys for Sequencing and Optimizations - James C. Bean - 1992

Link to our GitHub