# 42137 - Project Report 2020

May 18, 2020

| | |
|---|---|
| **Group ID** | 17 |
| **Group members** | s190050<br>s182550<br>s190031 |
| **Project topic** | VRPTW |
| **PURE mataheuristics implemented** | Simulated Annealing<br>TABU<br>GRASP |

# 1 Executive Summary

The VRPTW can be considered a somewhat easy problem for the human-mind to understand the dynamics, but not necessarily to generalize a solution for. Since this was the first time the author has implemented a metaheuristic, the easy interpretation of Simulated Annealing (SA) made this method the preferred one.

The construction of the initial solution was done by means of "exhausting" each vehicle as much as possible, iterating through every client and trying to insert it in any spot of the route. The order of the clients was simply the order that they were presented in the instance files, not following any other factors.

With the initial solution in hand and not breaking any constraints it was decided to attempt both intra and inter-relocation on the routes, randomly: suppose a solution has ten routes. Choosing two routes and two spots randomly (one to choose a customer and one to where it is going to be relocated), the chances that they are the same routes is 10% which. This is particularly of interest in this problem, since it represents higher chances of attempting inter-route relocation, therefore going hand-in-hand with the primary goal of reducing the number of vehicles used.

A reasonable run-time limit when considering commercial use for the algorithm can be defined by assuming that routing systems work overnight when all demands for the next day have been received. Therefore, assuming that this could be a data-set for a routing company operating in 40 different areas (cities, states, etc.), each represented by a file, a total run-time of around 7 hours should leave enough room for adjustments overnight. Therefore, the run-time for each instance was chosen to be 600 seconds.

A more precise and efficient method to define run-time could be by using a threshold of number of iterations without cost reduction (closely related to the value of the first derivative of the fitted function to the cost converge curve), potentially cutting loss time in instances that can't reach improvements with the current algorithm. That may be an improvement for future works.

With $T_0 = 100$ and $\alpha = 0.98$ obtained after more than 4 hours of tuning, the average gap improvement between the initially obtained solutions and the objective values (best known solutions) was of 81% (from 8,7 to 1,6), which can be considered expressive.

# 2 Algorithm - s190050

In order to make things easier to grasp (no pun intended), the core parts of the algorithm are separated in subsections below.

## 2.1 Solution Representation

Relating this problem with real life situations showed itself to be, obviously, the best option. For that matter, solution is represented as mutable structures corresponding to ***Car*** and ***Client***, each with their respective data:

- ***Car***
  - load: how much of product is left in the car;
  - route: list of the visited clients ids;
  - id: number of the car;
- ***Client***
  - demand: amount of product needed by the client;
  - stock: amount of product delivered;
  - locale: same as ID (warehouse is ID 1, for example, last client is 201);
  - t_init: beginning of the availability window;
  - t_end: end of the availability window;
  - serv_time: how long it takes to unload delivery;
  - delivery_car: ID of the car that made the visit;
  - delivery_time: moment in time when the vehicle can leave the client;

In this manner, it becomes very easy for human logic to operate with an array of cars (the fleet) and an array of clients. Final solution maybe obtained by accessing each route (which are arrays of clients IDs) in each car of the fleet. In other words, solutions are represented as arrays of integers corresponding to clients in their respective routes.

## 2.2 Construction Heuristic

As previously mentioned, this was done using two very basic iteration loops through the cars and the clients (in that order). The easiest way to explain the logic behind it can be seen on Algorithm 1.

```
Data: (fleet, clients)
for car in fleet do
    car.route[1] ← warehouse;
    for client in clients do
        if size(car.route) = 1 then
            makeDelivery(car,1,client);
            updateDeliveryTime(car.route,i);
            car.route[end] ← warehouse;
        else
            i ← 2;
            while i < size(car.route) and 0 < client.demand ≤ car.load do
                flag ← checkInsertion(car.route[i],client);
                if flag = True then
                    makeDelivery(car,i,client);
                    updateDeliveryTimes(car.route,i);
                end
                i ← i + 1 ;
            end
        end
    end
end
```

**Algorithm 1:** Construction Heuristic

And a brief explanation of what the functions inside the construction heuristic do:

- **makeDelivery** – inserts the client in the vehicle's route at location $i$, subtracts the client demand from the vehicle total load, sets the client stock to its demand and its demand to zero. Finally, assigns the car to the client's delivery log;
- **updateDeliveryTime** – sets/updates the delivery time for each client after position $i$ by summing the previous client's delivery time with the distance between them and its service time in case the vehicle arrives within the time window of availability, or its initial available time plus the service time in case it arrives before the availability window;
- **checkInsertion** – uses the same logic as updateDeliveryTime to check if the insertion is possible before performing it with makeDelivery.

## 2.3 Neighborhood Function

As previously mentioned, client relocation was chosen as the method to obtain neighbor solutions due to its simplicity. At first, only inter-route relocation was considered, since it enables the reduction of vehicles used and distance, but then small changes were made to the code in order to enable intra-relocation as well (further analysis on this change can be found in Section 5). The pseudo-code for the neighborhood function can be seen on Algorithm 2.

```
Data: (fleet, clients, time_limit, T, α)
t_s ← TIME;
while (TIME - t_s) ≤ time_limit do
    routes[ ];
    for car in fleet do
        if size(car.route) > 2 then
            append(routes, car)
        end
        old_car ← rand(routes);
        new_car ← rand(routes);
        old_spot ← rand(2,size(old_car.route)-1);
        new_spot ← rand(2,size(new_car.route)-1);
        p ← rand(0.0,1.0);
        if (old_car ≠ new_car AND new_car.load ≥ clients[old_spot].stock) OR
          (old_car = new_car AND old_spot ≠ new_spot) then
            SA(fleet, clients, old_spot, new_spot, new_car, old_car, T, p);
        end
        T ← T × α;
    end
end
```

**Algorithm 2:** Neighborhood Operator

## 2.4 Simulated Annealing

SA was performed as seen on [1] and classes. In as few words as possible: the cost (defined as the number of routes multiplied by the total travelled distance) of a neighbor solution is compared with the current solution (delta evaluation). If delta is negative (*i.e.*, neighbor solution is better than current solution), then this shall be set as the current solution. If delta is positive, then the neighbor solution is accepted as the current solution with probability $p_{T_i} = e^{(-\delta/T_i)}$, where $T_i$ is the temperature in iteration $i$, determined by the cooling schedule $T_i = T_{i-1} \times \alpha$. The pseudo-code for SA can be seen below, in Algorithm 3.

```
Data: (fleet, clients, old_spot, new_spot, new_car, old_car, T_i, p)
cfleet = copy(fleet);
cclients = copy(clients);
makeSwap(cfleet[old_car], cfleet[new_car], old_spot, new_spot, cclients);
if checkSolution(cfleet, cclients) = 1 then
    s_1 ← solutionCost(cfleet, cclients);
    s_0 ← solutionCost(fleet, clients );
    δ ← s_1 − s_0;
    p_T ← e^(−δ/T_i);
    if δ < 0 OR p ≤ p_T then
        makeSwap(fleet[old_car], fleet[new_car], old_spot, new_spot, clients);
    end
end
```

**Algorithm 3:** Simulated Annealing

And a brief explanation of what the functions inside SA do:

- **makeSwap** – inserts client into new spot in new car and deletes it from old spot in old car. Updates the delivery time for clients in both routes after the position where changes

occurred. If it happens to be the same car, makes the relocation and updates delivery times from the smallest position index. Also updates the delivery car to the client and the load of the car;

- **checkSolution** – returns true if all clients have been visited, no cars have negative load, and no route breaks time constraints (*i.e.*, the delivery time for a client plus the distance to the subsequent client cannot be greater than the subsequent client's availability window time ending);
- **solutionCost** – returns the total number of cars multipled by the total distance covered.

## 2.5 Parameter Tuning

As previously mentioned, in SA the probability with which a solution should be accepted in case its cost is bigger than the current solution is given by:

$$p_{t_i} = e^{-\frac{\delta}{T_i}} \tag{1}$$

Where $\delta$ is the cost difference between the current analyzed neighbor and the current solution, and $T_i$ is the temperature in iteration $i$, define by the cooling schedule from Section 2.4. In order to choose the initial temperature and $\alpha$, some educated guesses were performed:

1. Analysis of the average expected $\delta$ – when $\delta$ is positive, the number of cars will most likely be the same between the two solutions that are being compared (number of cars has order of magnitude $10^0$ – $10^1$, whereas distance has order of magnitude $10^3$ – $10^4$, so it is unlikely that a reduction in one car can yield such a negative impact on covered distance so that the cost is much worse). Therefore, it is expected that the value of delta will revolve around the average distance between clients. For instances with prefix *c1_2_* that average is of 74.93, for *c2_2_* is 67, and for both *r1_2_* and *r2_2_* is 72.67. Therefore, it is wished that the order of magnitude of T to be close to those distances, guaranteeing acceptance of worse solutions in the first iterations, but not towards the end of the run-time. Figure 1 shows the plot for $e^{-x/T}$ for $T = \{10^1, 10^2, 10^3, 10^4\}$, and where the expected values for $\delta$ would fall.

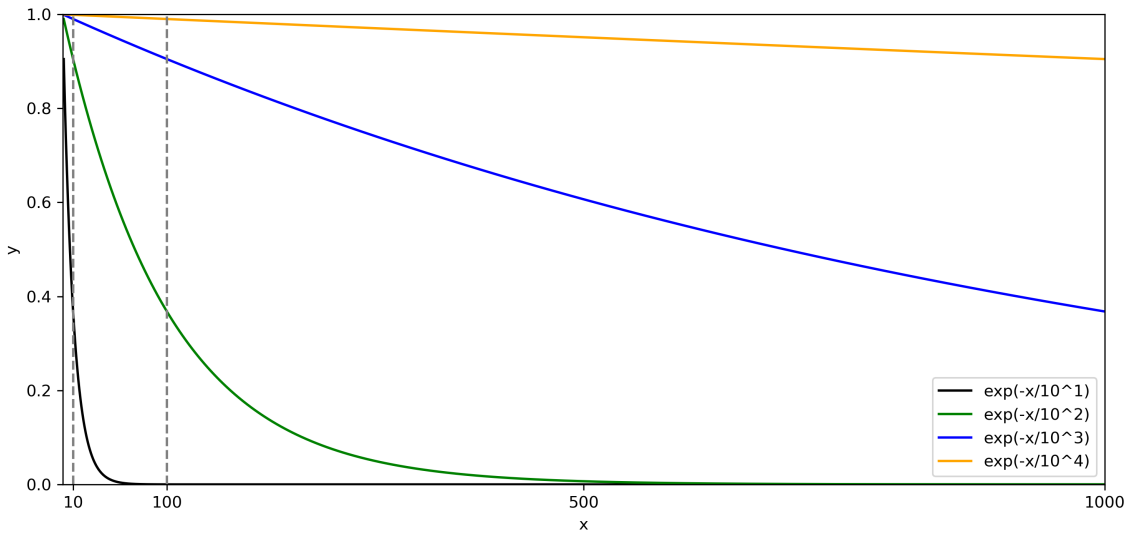


Figure 1: Plot for probability function with different initial temperatures.

The set of values chosen to tune the temperature was $T = \{100, 500, 1000\}$, due to the profile of the curves of 100 and 1000 in relation to $\delta$, and to attempt to add the halfway between those with 500.

2. Analysis of the behavior of the function in relation to $\alpha$ – if $\alpha$ is too close to one (*e.g.*, 0.9999), then the algorithm will accept "worse" solutions for more iterations than when compared to smaller values of $\alpha$, independently of the value of $T$. This can be seen on Figure 2, where it is evident that the area under the curve for all values but the extreme ones seem to sensibly correspond to a desirable percentage of the total area comprised by the straight lines for x between 10 and 100 (too little means less chances of escaping local minima, too much means higher chances of making the model too stochastic). Therefore, the set of possible values for $\alpha$ chosen for tuning was $\alpha = \{0.99, 0.98, 0.97\}$.

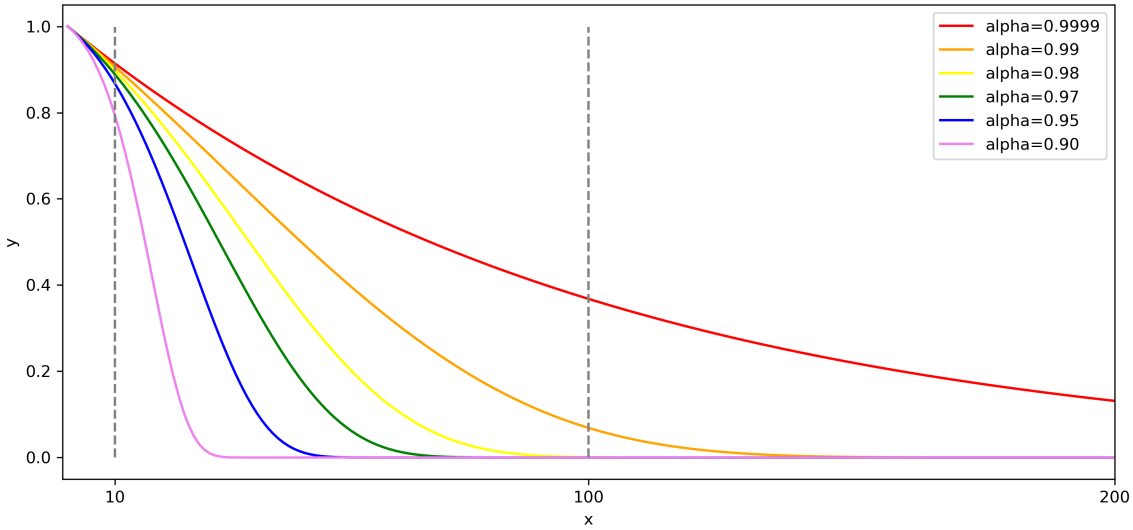


Figure 2: Plot for probability function with $T = 100$ for different values of $\alpha$.

Using those sets for both parameters, tuning was performed with three different instances as a training set and one as test set from each group of instances, totalling 12 for the training set and 4 for the test set, a split of 75%. 5 samples for each instance were used, with 30 seconds run-time for each. Then, the best average gap result for each combination was chosen, and the average test set gap registered for future benchmarking.

# 3 Algorithm - s182550

The other algorithm to be applied in the VRPTW problem was the TABU search. The construction heuristics used for the TABU search was the same as the one used in the Simulated Annealing. Therefore the same mutable structures can be found: Client and Car. As mentioned before, the creation of this type of variable facilitates the interpretation and consequently the computation of the algorithms.
Based on the mutable structures all the functions are built. Therefore the solution can be given by Clients and Fleet from which is possible to extract total distance driven, number of cars used and most importantly, the cost of the solution. Since the construction method was already explained previously the report continues to the application of the TABU search algorithm.

### 3.1 Neighborhood Function

The neighborhood structure is created by applying a "move" in the current solution. Initially, the chosen neighborhood method was an inter-route swap of two random clients. Even though the method seems promising and rather complex it contained a huge drawback. No optimization of the number of vehicles was taking place. Considering that the cost function was the result of $cars * distance$ it was quite obvious that the number of cars should also be optimized since it would highly impact the cost of the solution.

Hence, the change to a relocation method inter-route and intra-route. Using this methodology one client is moved into a car that already contains clients in its route, which allows to reduce both distance and number of cars. Since the meta-heuristic implemented is a TABU search the algorithm that finds the relocation move had at the same time check if this improvement had already taken place. Therefore, an algorithm was created in order to find the relocation having a TABU list has constrain. There are 3 factors of randomness: the client picked, the new car and the new position in the car.

The pseudo-code for to find the non TABU relocation move can be found bellow. The function represented is **getNonTABUmove**.

```
Data: (fleet, clients, TABUlist)
0 ← k;
0 ← nonTabu;
while k = 0 do
    for car in fleet do
        if size(car.route) > 2 then
            append(non_empty_routes, car.id)
        end
    end

    // Get new car and location for random client

    randomClient ← rand(client.id);
    new_car ← rand(non_empty_routes);
    new_spot ← rand(2,size(new_car.route)-1);
    new_Move ← [randomClient, new_car, new_spot];
    if (new_Move in TABUlist) == False then
        nonTabu ← new_Move ;
        1 ← k;
    end

    // Record old information

    oldcar ← randomClient.car;
    oldspot ← findall(x → x == randomClient, oldcar.route)
    oldInfo ← [randomClient, oldcar, oldspot];

    return nonTabu, oldInfo
end
```

**Algorithm 4:** Algorithm for non TABU move

The following algorithm describes how a single Neighbor solution is obtained. It uses the algorithm described previously in order to execute a relocation move that is not TABU.

```
Data: (fleet, clients, TABUlist)
0 ← k;
while k = 0 do
    nonTabu, oldInfo ← getNonTABUmove(fleet, clients, TABUlist);
    clients_cop = copy(client);
    fleet_cop = copy(client);
    if (new_car.load) >= client.stock) then
        clients_cop, fleet_cop = relocateCliente(clients_cop, fleet_cop, nonTabu,
          oldInfo);

        flagSol = checkSolution(clients_cop,fleet_cop);

        if flagSol == 0 then
            clients, fleet = relocateCliente(clients, fleet, nonTabu, oldInfo);
        end
    end
    return clients, fleet, nonTabu
end
```

**Algorithm 5:** Algorithm to get Neighbour

And a brief explanation of what the functions inside **getNeighbor**:

- **getNonTABUmove** – This function was described in the previous algorithm. Given a TABUlist and the set of car and clients, it returns two arrays, one with the information of the future location in the route for a specific client and another with the old location. Both arrays are used in the *relocateClient* function.
- **relocateClient** – this function combines a set of other functions. Firstly, deletes the client from its old position in the route and inserts back the demand (*deleteCliente*). Then, updates the delivery times for the subsequent clients in the old route (*updateDelivery*). Only the new car executes the delivery to this client (*makeDelivery*) and the delivery times are updated.
- **checkSolution** – returns 1 if a car arrives after the delivery time of a client, 2 if the car has negative load, 3 if the car does not finish in the depot and 4 if any of the clients have not been visited. In case all the constrains mentioned before are met, it returns 0.

## 3.2 TABU Search

The most fundamental part of the the TABU search algorithm is the TABU list, which distinguishes it from the Local Search algorithm. Opposite to Local Search, the success of TABU search is in part due to the fact that it can avoid getting stuck in a local minimum. This happens since the algorithm allows to move to a neighboring solution that may result in deterioration of the objective function but at the same time avoids going back to previous moves.

The same points that make it a powerful algorithm can also become its drawbacks. The TABU list might also prevent beneficial moves for the algorithm and even saturate the neighborhood.

However, in order to tackle this last problem some further improvements on the algorithm can be applied. The improvement method that was used in this example case is called Intensification. Like mentioned in the class, if no progress is achieved for a number of iterations the algorithm returns to the previous best solution and erases the TABU list.

Another note on the TABU Search algorithm concerns the neighborhood size. In order to avoid making the problem very extensive only a subset of the neighborhood was considered. By default the size of the subset is 300 neighboor solutions, but with more time and computational power one could increase the neighborhood size in order to include more solutions.

The following pseudo-code illustrates the structure of the TABU search.

```
Data: (fleet, clients, time_limit, L)
bestFleet, bestClients ← ConstructionsHeuristics(fleet, client)
bestcost ← solutionCost(bestFleet, bestClients)
t_s ← TIME;
[ ] ← TABUlist;
while (TIME - t_s) ≤ time_limit do
    [ ] ← Neighborhood;
    [ ] ← Neighborhood_cost;
    0 ←intensification;
    // Create Tabu Neighborhood
     while neighborhood_size < 300 do
        neighFleet, neighClients, nonTABU
          ← getNeighbor(bestFleet, bestClients, TABUlist)

        neighcost ← solutionCost(neighFleet, neighClients)
        neighbor_trio = [neighFleet, neighClients, nonTABU];
        append(Neighborhood_cost, neighcost)
        append(Neighborhood, neighbor_trio)
    end
    // Get best solution in Neighborhood and update TABUlist
     clients_ngb_best, fleet_ngb_best = selectBestNeighbour(Neighborhood,
      Neighborhood_cost)
    TABUlist ← updateTabuList(TABUlist, L, nonTABU)
     // Compare with best solution and check intensification
    if intensification < Int_lim then
        if neighcost < bestcost then
            previous_bestclients = copy(bestclients)
            previous_bestfleet = copy(bestfleet)
            bestFleet = copy(neighFleet)
            bestClients = copy(neighClients)
            bestcost = copy(neighcost)
            intensification = 0
        else
            intensification += 1
        end
    else
        // Go back to preivous best solution and reset TABU to zero
        bestclients = copy(previous_bestclients);
        bestfleet = copy(previous_bestfleet);
        [ ] ← TABUlist;
        intensification = 0
    end
    return bestclients, bestfleet
end
```

**Algorithm 6:** TABU Search

- **solutionCost** – A simple function that given the set of clients and cars, takes the total distance and number of used cars and returns the cost as $total\_distance * nr\_cars$.
- **updateTabuList** – this function receives the TABUlist and the new TABU. It updates the TABUlist according to the tabu ternure, $L$.
- **selectBestNeighbour** – in this case is not a function but instead a loop that extracts the best neighboor from the neighborhood. The decision to put as a function in the pseudo-code had the intention of simplify the ideas.

### 3.3 Parameter Tuning

As mentioned above, the most important feature of the TABU Search is the TABU list. The length of the list is directly proportional with the memory capacity of the algorithm, also called TABU tenure. A short list means the algorithm may "recycle" moves and go back to previous visited solutions. A long list means many spaces not explored. Therefore this is an important parameter to be tested. For this project the TABU tenure changed betweeen [5, 10, 20, 30].

The parameter involved in the tunning was the *Intensification*. This improvement is a mechanism that allows the algorithm to go back to the previous best solution erasing simultaneously the TABU List if no improvement was seen for a few iteration. In this problem the values change between [50, 100, 200].

Unfortunately the parameter tuning is very computationally expensive and time consuming. Hence, the decision of running the code approximately 10 seconds for each instance with each set of parameters. Is expected that the increase in the running time for one instance would lead to better results like explained before.

The results of the parameter tuning can be found in the following table.

| Tabu Lenght | Intensification | GAP | Cars | Distance |
|---|---|---|---|---|
| 5 | 50 | 7,15 | 22,72 | 11364,06 |
| 5 | 100 | 7,09 | 22,75 | 11292,81 |
| 5 | 200 | 7,14 | 22,89 | 11323,56 |
| 10 | 50 | 7,06 | 22,72 | 11294,24 |
| 10 | 100 | 7,06 | 22,86 | 11238,19 |
| 10 | 200 | 7,11 | 22,83 | 11282,49 |
| 20 | 50 | 7,11 | 22,64 | 11324,02 |
| 20 | 100 | 7,06 | 22,81 | 11251,38 |
| 20 | 200 | 7,15 | 22,86 | 11346,46 |
| 30 | 50 | 7,19 | 22,94 | 11379,19 |
| 30 | 100 | 7,11 | 22,83 | 11278,90 |
| 30 | 200 | 7,11 | 22,89 | 11279,21 |

Looking at the table above it can observed that the lowest values of GAP are a bit scattered. It doesn't seem to exist a clear best set of parameters which might be explained but the short running time for each instance. However, we can see that the set [10, 50] and [10,100] are similar and have low GAP, so one of them was chosen to get the final results. The choice is considered bit arbitrary since there is no difference significant between them.

## 4 Algorithm - s190031

Third algorithm built in the project was Greedy Randomized Adaptive Search Procedure (GRASP). The construction heuristic is same as the one explained in the section 2.2 of the report where the

solution is constructed by creating mutable structures Car and Client. Although the metaheuristic built in this section has few errors, this section of the report explains the approach in building the metaheuristic.

## 4.1 Neighborhood Function

In this section, both intra and inter route swaps were applied. Since, only cars were interchanged in the neighbourhood there was no reduction in trucks. But the distance travelled decreased.

```
Data: (clients, fleet, distnace, dimension, rClient1, rClient2)
K ← 0;
while k = 0 do
    client1 = clients[rclient1];
    client2 = clients[rclient2];
    car1 = fleet[client1.delivery_car];
    car2 = fleet[client2.delivery_car];
    car1_cop = copy(client1);
    car2_cop = copy(client2);
    client1_cop = copy(car1);
    client2_cop = copy(car2);
    clients_cop = copy(clients);
    // Save location in the route before deleting them

    if (checkRoute(distance,clients_cop,car1_cop) =0 AND
     (checkRoute(distance,clients_cop,car2_cop)=0 then
        // Delete old information
        // Insert client2 in car1 and car2 in client1
        flag1 = checkInsertion(clients_cop[old], clients_cop[new],car1_cop, rClient2,
         distance)
        flag1 = checkInsertion(clients_cop[old], clients_cop[new],car2_cop, rClient1,
         distance)
        // If both the flags are bigger then 0 and the dmeand of the
           client is not bigger then the capacity of the car, make
           delivery

        client2, car1 = makeDelivery(client2,car1,i);
        client1, car2 = makeDelivery(client1,car2,j);
    end
    (clients[rcleint1].delivery_car]= clients[rcleint2].delivery_car
end
 // Change only one car and make delivery
return clients, fleet
```

**Algorithm 7:** Algorithm to get Neighbour

Brief explanation of the neighbourhood function.

Initially, a function LocalSearch was created were a list consisting of pairs of clients was created using Julia's inbuilt "Combinatorics" library. Later, the neighbourhood funcction was called inside the LocalSearch.

- **checkRoute** – Checks if any constraints are being broken for routes before inserting the clients to the route.
- **makeDelivery** – Inserts a client in a route and delivers it. Updates the capacity of the car by clients demand and updates the capacity in the car.

### 4.2 GRASP

A Greedy Randomized Search Procedure is a metaheuristic for combinatorial optimization. It has two phases, a construction phase(Pseudo code below) and a local search. A local search function was built by calling neighbourhood function inside and by making a list of best pairs of client.

```
Data: (fleet, clients, Distance, Dimension)
nextClient ← clients[1];
for car in fleet do
 | car.route[1] ← warehouse
end
;
while x < 10 do
 | possibleClient, distance ← nearestUnvisitedClient(clients, car, prevClient,
 |  dimensions, distance );
 | RestrictedListClients[] ← possibleClient;
 | nextClient ← rand(RestrictedListClients);
 | if size(car.route) = 1 then
 |  | makeDelivery(car,1,client);
 |  | car.route[end] ← warehouse;
 | else
 |  | i ← 2;
 |  | while i < size(car.route) and 0 < nextClient.demand ≤ car.load do
 |  |  | flag ← checkInsertion(car.route[i],client);
 |  |  | if flag = True then
 |  |  |  | makeDelivery(car,i,client);
 |  |  |  | updateDeliveryTimes(car.route,i);
 |  |  | end
 |  |  | i ← i + 1 ;
 |  | end
 | end
 | return clients, fleet
end
```

**Algorithm 8:** GRASP Heuristic

And a brief explanation of what the functions inside GRASP do:

- **nearestUnvisitedClient** – This function sees the top 10 clients and inserts it in the route.
- **RestrictedCandidateList** – So this list consists of the top 10 clients picked by nearestUnvisitedClient. A random client is picked from this list and a delivery is made.

## 5 Results and Conclusions

The results obtained for the Section 2 algorithm can be seen on Figure 3. A few things may be discussed:

- Although parameter tuning was carried with a smaller time-limit per instance when compared to the best run (30 and 600 seconds, respectively), reduction of the gap between the results and the best known objective values could be seen on both cases. Here, gap was defined as:

$$\frac{f(s') - f(s_b)}{f(s_b)} \tag{2}$$

Where $f(s')$ is the cost of the neighbor solution, and $f(s_b)$ is the cost of the best solution (remembering that cost is defined as the number of cars multiplied by the total covered distance). This is corroborated by Figure 4, where it can be seen that the cost does seem to converge (for that particular example, but the same may be inferred for the other instances).

It is interesting to see in Figure 4 that it is clear where there are drops in cost due to reduction in number of cars, such as around 2000, between 3000 and 4000, and right after 10000 iterations, amongst other (11 car reductions in this case);

- As previously mentioned, the neighborhood function initially only generated inter-route ($Ie$) relocation possibilities, but a small change was made in order to allow for both inter and intra-route ($Ie + Ia$) relocation. The results can be seen on Figure 3 where, even with a very small time-limit per instance (10 seconds), allowing both types of relocation yielded, on average, better results;

- The average gap for the test set was of 4.2, while the average gap for the instances when running for the defined time-limit (600 seconds) was of 1.6, as it can be seen on Figure 3, a 81% reduction from the initial gap.

It is hard to assess the quality of the results obtained, since here comparisons are made with the best known objective values reached by probably much more complex algorithms, written by much more skilled programmers and ran in more capable machines, especially when taking into account that improvement is clearly not linear, as can be seen from Figure 4, rendered by the algorithm from Section 2, but rather reduces exponentially with time. Nevertheless, when comparing the results with the initial construction, these have shown an average gap reduction of more than 81%, earning the right to be defined by the author as a success from the optic of optimization using meta-heuristics.

Figure 5. summarizes the results of the **TABU Search** for the VRPTW problem. Looking at the table with closer detail it can be observed that TABU Search has no impact in the reduction of number of cars for the instances "r2_2_" and "c2_2_". It only achieves a slight reduction of the distances. Even though better results are achieved in the two groups of instances, "r1_1_" and "c1_1_", they are still very far from the objective distances and cars. The results shown on Figure 5. can have the following explanations:

- Starting with the parameter tuning. As mentioned before, the running time for parameter tuning was only of 10 seconds per instance which may have created an incorrect evaluation of the best set o parameters. It have been necessary to have a higher running time to have more accurate results.

  Still related to the parameter tuning is the values of parameters that was tested. Only 3 values the $intensification$ and 4 for the *TABU lenght* which is also indicative that a deeper analysis could have been performed.

- Secondly, the robustness of the TABU Search. One of the main advantages of this algorithm is the possibility of developing the algorithm further into a more complex model since many different features and mechanisms can be add, for example, aspiration level, reactive tenure, diversification, etc. In this specific project only the TABU tenure and intensification where implemented. Therefore one could say that a further development of the model would most likely turn into better results.

- Last but not least, is the structure of the code in itself. The formulation of the meta-heuristic in theory not always meet the practice while coding. Inefficiencies in the code lead to worst results for the same amount of time. Which gives even more importance to the factor of running time in this project.

# References

[1] F. Taner, A. Galić, T. Carić. *Solving practical vehicle routing problem with time windows using metaheuristic algorithms.* University of Zagreb, Faculty of Transport and Traffic Sciences. Croatia, 2011.

| Var. | Vehicles | | | | | Distance | | | | | Gap | | | | Iter. (x100) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Avg. | 23 | 20 | 20 | 18 | 12 | 12033 | 7598 | 7569 | 4311 | 2773 | 8,7 | 4,5 | 4,4 | 1,6 | 15 | 16 | 444 |
| Inst. | Init. | Ie | Ie+Ia | **Best** | Obj. | Init. | Ie | Ie+Ia | **Best** | Obj. | Init. | Ie | Ie+Ia | **Best** | Ie | Ie+Ia | **Best** |
| c1_2_1 | 40 | 35 | 36 | **30** | 20 | 14224 | 12648 | 11705 | **5802** | 2705 | 9,5 | 7,2 | 6,8 | **2,2** | 1 | 2 | **69** |
| c1_2_2 | 39 | 38 | 37 | **28** | 18 | 14501 | 10277 | 9865 | **4649** | 2918 | 9,8 | 6,4 | 5,9 | **1,5** | 5 | 6 | **133** |
| c1_2_3 | 38 | 35 | 36 | **28** | 18 | 14535 | 9808 | 10107 | **4153** | 2707 | 10,3 | 6,0 | 6,5 | **1,4** | 6 | 7 | **325** |
| c1_2_4 | 34 | 27 | 28 | **24** | 18 | 15032 | 5808 | 6880 | **3744** | 2643 | 9,7 | 2,3 | 3,0 | **0,9** | 27 | 22 | **656** |
| c1_2_5 | 38 | 35 | 35 | **27** | 20 | 13672 | 10983 | 11186 | **5778** | 2702 | 8,6 | 6,1 | 6,2 | **1,9** | 2 | 2 | **41** |
| c1_2_6 | 35 | 34 | 34 | **28** | 20 | 12566 | 10597 | 9979 | **6183** | 2701 | 7,1 | 5,7 | 5,3 | **2,2** | 2 | 3 | **75** |
| c1_2_7 | 35 | 34 | 33 | **26** | 20 | 14495 | 9858 | 11159 | **5088** | 2701 | 8,4 | 5,2 | 5,8 | **1,4** | 4 | 3 | **74** |
| c1_2_8 | 35 | 34 | 33 | **27** | 19 | 14079 | 9867 | 10773 | **5335** | 2775 | 8,3 | 5,4 | 5,7 | **1,7** | 5 | 4 | **106** |
| c1_2_9 | 34 | 32 | 33 | **26** | 18 | 14175 | 8071 | 7595 | **4308** | 2688 | 9,0 | 4,3 | 4,2 | **1,3** | 11 | 12 | **213** |
| c1_2_10 | 34 | 31 | 30 | **24** | 18 | 14775 | 6823 | 7859 | **4294** | 2644 | 9,6 | 3,4 | 4,0 | **1,2** | 21 | 15 | **229** |
| c2_2_1 | 11 | 11 | 11 | **11** | 6 | 10261 | 8222 | 8042 | **3411** | 1931 | 8,7 | 6,8 | 6,6 | **2,2** | 3 | 3 | **116** |
| c2_2_2 | 14 | 13 | 14 | **11** | 6 | 12049 | 6892 | 7407 | **3262** | 1863 | 14,1 | 7,0 | 8,3 | **2,2** | 16 | 16 | **416** |
| c2_2_3 | 13 | 13 | 12 | **13** | 6 | 12360 | 6228 | 6087 | **3173** | 1775 | 14,1 | 6,6 | 5,9 | **2,9** | 33 | 26 | **874** |
| c2_2_4 | 13 | 13 | 10 | **10** | 6 | 12102 | 12102 | 5944 | **2812** | 1703 | 14,4 | 14,4 | 4,8 | **1,8** | 0 | 26 | **1328** |
| c2_2_5 | 15 | 14 | 15 | **14** | 6 | 12077 | 6080 | 6875 | **3403** | 1879 | 15,1 | 6,6 | 8,1 | **3,2** | 9 | 9 | **352** |
| c2_2_6 | 12 | 12 | 12 | **11** | 6 | 12505 | 7049 | 7053 | **2966** | 1857 | 12,5 | 6,6 | 6,6 | **1,9** | 8 | 9 | **368** |
| c2_2_7 | 11 | 11 | 11 | **11** | 6 | 11368 | 6785 | 6842 | **2999** | 1849 | 10,3 | 5,7 | 5,8 | **2** | 7 | 8 | **363** |
| c2_2_8 | 11 | 10 | 11 | **10** | 6 | 12025 | 6623 | 6794 | **2999** | 1821 | 11,1 | 5,1 | 5,8 | **1,7** | 10 | 10 | **456** |
| c2_2_9 | 12 | 11 | 12 | **11** | 6 | 12751 | 5343 | 6110 | **3248** | 1830 | 12,9 | 4,4 | 5,7 | **2,3** | 20 | 20 | **559** |
| c2_2_10 | 11 | 11 | 11 | **10** | 6 | 12384 | 5354 | 5831 | **2923** | 1807 | 11,6 | 4,4 | 4,9 | **1,7** | 21 | 17 | **614** |
| r1_2_1 | 35 | 31 | 30 | **28** | 20 | 10757 | 9430 | 8788 | **7268** | 4784 | 2,9 | 2,1 | 1,8 | **1,1** | 2 | 3 | **60** |
| r1_2_2 | 33 | 29 | 29 | **25** | 18 | 9824 | 7913 | 7567 | **5809** | 4040 | 3,5 | 2,2 | 2,0 | **1** | 8 | 8 | **147** |
| r1_2_3 | 33 | 30 | 27 | **24** | 18 | 9795 | 7279 | 7228 | **4785** | 3382 | 4,3 | 2,6 | 2,2 | **0,9** | 10 | 14 | **275** |
| r1_2_4 | 34 | 26 | 28 | **21** | 18 | 11644 | 5923 | 5882 | **4085** | 3058 | 6,2 | 1,8 | 2,0 | **0,6** | 39 | 37 | **883** |
| r1_2_5 | 32 | 28 | 27 | **22** | 18 | 9932 | 8006 | 7804 | **6109** | 4108 | 3,3 | 2,0 | 1,8 | **0,8** | 3 | 4 | **52** |
| r1_2_6 | 33 | 29 | 29 | **24** | 18 | 10562 | 7438 | 7996 | **5410** | 3583 | 4,4 | 2,3 | 2,6 | **1** | 9 | 8 | **151** |
| r1_2_7 | 34 | 28 | 30 | **24** | 18 | 11060 | 7179 | 7366 | **4935** | 3150 | 5,6 | 2,5 | 2,9 | **1,1** | 13 | 13 | **243** |
| r1_2_8 | 33 | 27 | 26 | **22** | 18 | 12271 | 5801 | 5832 | **3953** | 2952 | 6,6 | 1,9 | 1,9 | **0,6** | 40 | 40 | **788** |
| r1_2_9 | 38 | 29 | 29 | **27** | 18 | 11180 | 7879 | 7497 | **5434** | 3761 | 5,3 | 2,4 | 2,2 | **1,2** | 7 | 8 | **150** |
| r1_2_10 | 25 | 23 | 24 | **21** | 18 | 9757 | 6684 | 6523 | **4509** | 3301 | 3,1 | 1,6 | 1,6 | **0,6** | 9 | 10 | **202** |
| r2_2_1 | 9 | 9 | 9 | **9** | 4 | 10362 | 7939 | 8100 | **5149** | 4483 | 4,2 | 3,0 | 3,1 | **1,6** | 4 | 6 | **94** |
| r2_2_2 | 8 | 8 | 8 | **8** | 4 | 10418 | 7573 | 7477 | **4515** | 3621 | 4,8 | 3,2 | 3,1 | **1,5** | 12 | 11 | **269** |
| r2_2_3 | 9 | 9 | 9 | **8** | 4 | 11335 | 7048 | 7106 | **3960** | 2881 | 7,9 | 4,5 | 4,6 | **1,7** | 26 | 31 | **598** |
| r2_2_4 | 10 | 9 | 9 | **9** | 4 | 12605 | 5472 | 5212 | **2924** | 1981 | 14,9 | 5,2 | 4,9 | **2,3** | 57 | 57 | **1072** |
| r2_2_5 | 8 | 8 | 8 | **8** | 4 | 11090 | 7158 | 7554 | **4537** | 3367 | 5,6 | 3,3 | 3,5 | **1,7** | 7 | 8 | **185** |
| r2_2_6 | 9 | 8 | 8 | **8** | 4 | 11325 | 6211 | 6755 | **4086** | 2913 | 7,7 | 3,3 | 3,6 | **1,8** | 20 | 17 | **746** |
| r2_2_7 | 10 | 9 | 9 | **7** | 4 | 11334 | 6423 | 6204 | **3457** | 2451 | 10,6 | 4,9 | 4,7 | **1,5** | 37 | 44 | **1138** |
| r2_2_8 | 11 | 8 | 9 | **9** | 4 | 11821 | 4894 | 4920 | **2965** | 1850 | 16,6 | 4,3 | 5,0 | **2,6** | 65 | 68 | **2287** |
| r2_2_9 | 8 | 8 | 8 | **8** | 4 | 10459 | 6273 | 6754 | **4139** | 3092 | 5,8 | 3,1 | 3,4 | **1,7** | 12 | 12 | **423** |
| r2_2_10 | 10 | 9 | 9 | **9** | 4 | 11851 | 5973 | 6083 | **3893** | 2655 | 10,2 | 4,1 | 4,2 | **2,3** | 19 | 21 | **637** |

Figure 3: Final results for best run with 600s of time-limit per instance and comparison between inter-route relocation only (Ie) and inter/intra-route (Ie+Ia) with a time-limit of 20s per instance.

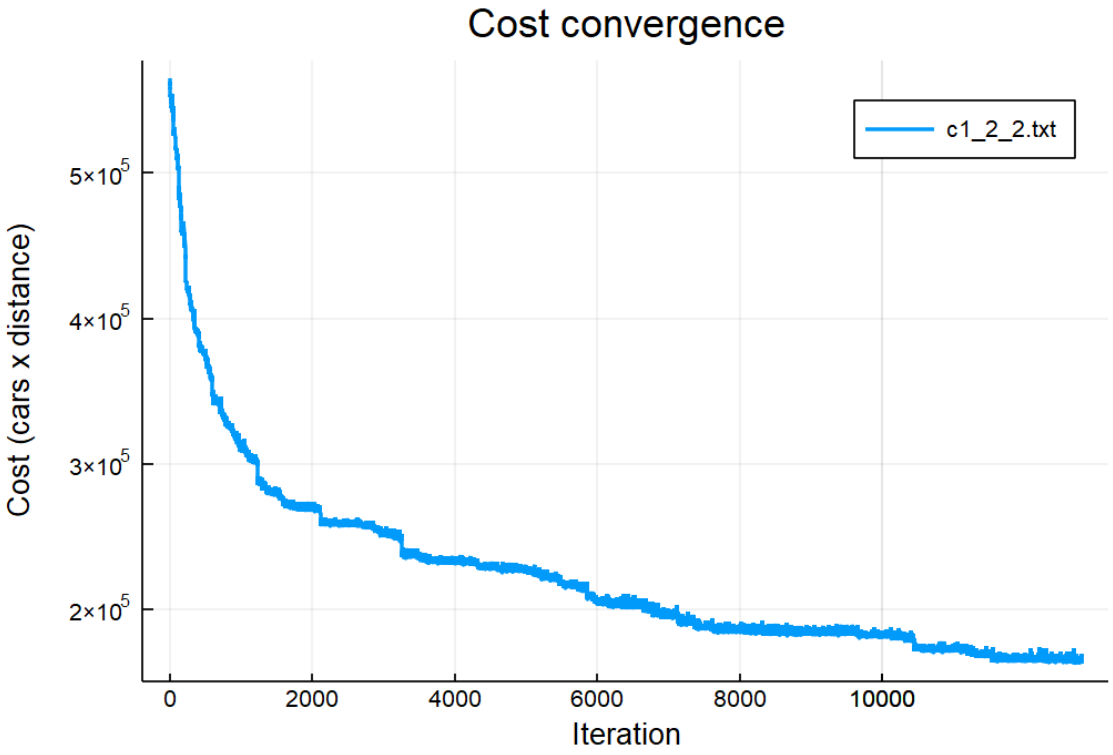


Figure 4: Cost convergence from Algorithm from section 2.

| Instance | Initial Cars | Optimal Cars | Objective Car | Initial Dist | Optimal Dist | Objective Dist | Initial GAP | Optimal GAP |
|---|---|---|---|---|---|---|---|---|
| c1_2_1 | 40 | 35 | 20 | 14224 | 10956 | 2705 | 9,5 | 6,1 |
| c1_2_2 | 39 | 36 | 18 | 14501 | 11895 | 2918 | 9,8 | 7,2 |
| c1_2_3 | 38 | 34 | 18 | 14535 | 12501 | 2707 | 10,3 | 7,7 |
| c1_2_4 | 34 | 31 | 18 | 15032 | 12834 | 2643 | 9,7 | 7,4 |
| c1_2_5 | 38 | 34 | 20 | 13672 | 11181 | 2702 | 8,6 | 6,0 |
| c1_2_6 | 35 | 33 | 20 | 12566 | 11000 | 2701 | 7,1 | 5,7 |
| c1_2_7 | 35 | 32 | 20 | 14495 | 10275 | 2701 | 8,4 | 5,1 |
| c1_2_8 | 35 | 34 | 19 | 14079 | 11525 | 2775 | 8,3 | 6,4 |
| c1_2_9 | 34 | 33 | 18 | 14175 | 11722 | 2688 | 9 | 7,0 |
| c1_2_10 | 34 | 33 | 18 | 14775 | 12272 | 2644 | 9,6 | 7,5 |
| c2_2_1 | 11 | 11 | 6 | 10261 | 10124 | 1931 | 8,7 | 8,6 |
| c2_2_2 | 14 | 14 | 6 | 12049 | 11054 | 1863 | 14,1 | 12,8 |
| c2_2_3 | 13 | 13 | 6 | 12360 | 11318 | 1775 | 14,1 | 12,8 |
| c2_2_4 | 13 | 13 | 6 | 12102 | 11532 | 1703 | 14,4 | 13,7 |
| c2_2_5 | 15 | 14 | 6 | 12077 | 11049 | 1879 | 15,1 | 12,7 |
| c2_2_6 | 12 | 12 | 6 | 12505 | 10870 | 1857 | 12,5 | 10,7 |
| c2_2_7 | 11 | 11 | 6 | 11368 | 10517 | 1849 | 10,3 | 9,4 |
| c2_2_8 | 11 | 11 | 6 | 12025 | 10813 | 1820 | 11,1 | 9,9 |
| c2_2_9 | 12 | 12 | 6 | 12751 | 11129 | 1830 | 12,9 | 11,2 |
| c2_2_10 | 11 | 11 | 6 | 12384 | 10817 | 1807 | 11,6 | 10,0 |
| r1_2_1 | 35 | 27 | 20 | 10757 | 8960 | 4784 | 2,9 | 1,5 |
| r1_2_2 | 33 | 31 | 18 | 9824 | 9851 | 4040 | 3,5 | 3,2 |
| r1_2_3 | 33 | 31 | 18 | 9795 | 10302 | 3382 | 4,3 | 4,2 |
| r1_2_4 | 34 | 29 | 18 | 11644 | 11158 | 3058 | 6,2 | 4,9 |
| r1_2_5 | 32 | 25 | 18 | 9932 | 8471 | 4108 | 3,3 | 1,9 |
| r1_2_6 | 33 | 30 | 18 | 10562 | 9932 | 3583 | 4,4 | 3,6 |
| r1_2_7 | 34 | 29 | 18 | 11060 | 10288 | 3150 | 5,6 | 4,3 |
| r1_2_8 | 33 | 31 | 18 | 12271 | 11724 | 2952 | 6,6 | 5,8 |
| r1_2_9 | 38 | 27 | 18 | 11180 | 8659 | 3761 | 5,3 | 2,5 |
| r1_2_10 | 25 | 25 | 18 | 9757 | 8808 | 3301 | 3,1 | 2,7 |
| r2_2_1 | 9 | 9 | 4 | 10362 | 10193 | 4483 | 4,2 | 4,1 |
| r2_2_2 | 8 | 8 | 4 | 10418 | 10077 | 3621 | 4,8 | 4,6 |
| r2_2_3 | 9 | 9 | 4 | 11335 | 11044 | 2881 | 7,9 | 7,6 |
| r2_2_4 | 10 | 10 | 4 | 12605 | 12114 | 1981 | 14,9 | 14,3 |
| r2_2_5 | 8 | 8 | 4 | 11090 | 10329 | 3367 | 5,6 | 5,1 |
| r2_2_6 | 9 | 9 | 4 | 11325 | 11145 | 2913 | 7,7 | 7,6 |
| r2_2_7 | 10 | 10 | 4 | 11334 | 11289 | 2451 | 10,6 | 10,5 |
| r2_2_8 | 11 | 11 | 4 | 11821 | 11821 | 1850 | 16,6 | 16,6 |
| r2_2_9 | 8 | 8 | 4 | 10459 | 10459 | 3092 | 5,8 | 5,8 |
| r2_2_10 | 10 | 10 | 4 | 11851 | 11520 | 2655 | 10,2 | 9,8 |

Figure 5: Results table for TABU Search