UNIVERSITY OF ZAGREB

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

**Heuristic optimization methods**

Capacitated Vehicle Routing Problem with Time Windows (CVRPTW)

Toma Petrač

Zagreb, 2021.

Contents

[1. Problem description 3](#_Toc60788368)

[2. Simulated annealing 4](#_Toc60788369)

[2.1 Implementation 4](#_Toc60788370)

[2.2 Pseudocode 5](#_Toc60788371)

[3. Analysis of results 7](#_Toc60788372)

[3.1 Parameters and discussion 7](#_Toc60788373)

[4. Conclusion 9](#_Toc60788374)

# Problem description

Vehicle routing problem (VRP) is a combinatorial optimization problem which is a generalization of TSP (travelling salesman problem). Objective of this problem is to find the optimal set of routes for a fleet of vehicles to traverse in order to deliver the goods to a set of stationary customers, located on different positions. The most common objective is to minimize the total route cost.

Finding the optimal solution for this problem is NP-hard which means that any kind of brute force search is out of the question (due to the enormous size of the search space). Instead of that, problem is most often „attacked“ with some sort of heuristic algorithms.

Capacitated Vehicle Routing Problem (CVRP) adds an additional constraint to the problem: Every vehicle has a certain amount of capacity and can only deliver so many goods to the customers. Furthermore, this project will cover CVRPTW (capacitated vehicle routing problem with time windows) which adds even more constraints for delivering goods to customers. Every customer has a time interval in which he is allowed to be serviced by vehicles.

# Simulated annealing

Simulated annealing is a type of improvement metaheuristic, nature-inspired algorithm that applies principles of statistical mechanics. It simulates the energy changes in the system during the cooling process until an equilibrium state is reached.

The main idea of the algorithm is to stochasticly search by accepting even the non-improving solutions with a certain probabilty, so it can escape from local optima. The algorithm is memoryless, and only records incumbent solutions. It doesn't generate an entire neighborhood but creates one neighbor at random and evaluates it.

Most common method for selecting neighboring solutions is as follows:

* a random neighbor is generated
* if it improves the current solution we accept it as the new current solution
* otherwise, we accept it with a certain probability which depends on:
  + T (temperature) – control parameter
  + Δf = |f(current solution)-f(neighbor)| - relative fitness

## Implementation

Solution representation used for this problem is a list of lists, where the inner lists represent the routes (e.g. [[0, 6, 14, 25, 0], [0, 1, 3, 0], [0, 76, 23, 80, 97, 0], …]). Every list is actually a route for different vehicles, and the numbers in the routes represent the customers.

Primary objective function is to minimize the number of vehicles used for the problem (number of inner lists), and secondary objective is to minimize the total cost for all the routes.

Neighboring solutions are also lists of lists which are different from the current solution by 1 customer (1 customer is exchanged between 2 routes, or it's position in the current route is replaced).

Initial solution is obtained with a greedy algorithm with the following pseudocode:

while unassigned\_customer\_exists:  
 candidate = null  
 min\_cost = inf  
 curr\_vehicle = vehicles[veh\_index]  
 if curr\_vehicle.route.isEmpty():  
 curr\_vehicle.add\_customer(customers[0])  
   
 for customer in customers[1:]:  
 if not customer.is\_routed:  
 if curr\_vehicle.fits(customer):  
 candidate\_cost = calculate\_cost()  
 if min\_cost > candidate\_cost:  
 min\_cost = candidate\_cost  
 candidate = customer

if candidate != null:  
 if more\_vehicles\_left\_to\_assign:  
 if curr\_vehicle.position != depot\_position:  
 curr\_vehicle.add\_customer(customers[0])  
 veh\_index++  
 else:  
 print(„Problem can't be solved!“)  
 sys.exit(1)  
 else:  
 candidate.is\_routed = True  
 curr\_vehicle.add\_customer(candidate)

Neighbor selection used for this project was the one stated above the 2.1 heading. Random neighbor is accepted as the current solution with a probability as follows:

T is a control parameter (temperature) which is initially set to 100. The bigger it is, chances for accepting the non-improving solutions are also greater. I used a homogeneous cooling schedule with geometric decrement function, with . I also tried the very slow decrease with , but got better results with geometric decrement function.

Relative fitness , means that the worse the neighbor is, selection probability will also be lower. Here, fitness is calculated as the sum of all costs for all routes. Additionally, it's important to note that the difference in fitness between the 2 solutions which differ in only 1 vehicle used is set to 1000 (to indicate the primary objective).

Termination criterion would be when the final temperature gets to 0.01. Of course, since the task is to get the results in time, that's going to be our real termination criterion – time.

## Pseudocode

while !termination\_criterion\_reached:

vehicle\_to\_remove\_from = random.choice(used\_vehicles)  
 vehicle\_to\_insert\_into = random.choice(used\_vehicles)  
  
 index\_rmv = random.randint(1, len(vehicle\_to\_remove\_from.route) – 2)  
 index\_ins = random.randint(1, len(vehicle\_to\_insert\_into.route) – 2)  
  
 if transfer\_customer(vehicle\_to\_remove\_from, index\_rmv, vehicle\_to\_insert\_into, index\_ins):  
 if no\_used\_vehicles < incumbent\_number or no\_used\_vehicles == incumbent\_number and total\_cost < incumbent\_cost:

update\_incumbent\_solutions()  
 else:  
 diff = 1000 \* (current\_number – neighboring\_number)  
 diff += abs(current\_cost – neighboring\_cost)  
 prob = exp(-diff / temperature)  
 if prob < random.random():  
 reverse\_transfer()

temperature \*= alpha

# Analysis of results

Number of used vehicles:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 minute | 5 minutes | unlimited (> 15 min) |
| i1 | 21/25 | 20/25 | 19/25 |
| i2 | 21/50 | 21/50 | 21/50 |
| i3 | 14/100 | 14/100 | 13/100 |
| i4 | 13/150 | 13/150 | 12/150 |
| i5 | 30/200 | 29/200 | 29/200 |
| i6 | 98/250 | 95/250 | 94/250 |

Total distance:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 minute | 5 minute | unlimited (> 15 min) |
| i1 | 2463.68 | 2526.06 | 2345.64 |
| i2 | 5993.69 | 5924.62 | 5642.67 |
| i3 | 5998.57 | 5683.55 | 5582.60 |
| i4 | 25829.33 | 25042.71 | 23730.82 |
| i5 | 25284.81 | 25807.71 | 24927.45 |
| i6 | 87740.80 | 88042.85 | 85340.39 |

Number of iterations (successful transfers – with feasible solutions):

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 minute | 5 minutes | unlimited (> 15 min) |
| i1 | 7411 | 48787 | 197469 |
| i2 | 5754 | 23138 | 122900 |
| i3 | 5806 | 68775 | 252058 |
| i4 | 5621 | 79018 | 191236 |
| i5 | 8140 | 109892 | 289255 |
| i6 | 34288 | 115523 | 279724 |

## Parameters and discussion

**Initial temperature** - A couple of different values for initial temperature have been tried (50, 75, 80, 100, 120, 200), but overall, the best results were achieved when it was set to 100.

**Choosing neighboring solutions** - I tried using the alternative method for choosing neighboring solutions (the one where even improving solutions were chosen with a certain probability in order to escape the local minima), but it performed slightly worse.

**Temperature decrement function** – Overall, geometric decrement function with α = 0.95 worked the best. If α was too small, in a large number of iterations, the temperature would fall to almost 0 which would be a problem (division by zero). I also tried using very slow decrease function (T <- 1 + (β \* T), where β should be close to 0 – I used 0.001), but the performance was a bit worse, especially in the 1 minute runs. In the longer runs it seemed it was about the same as using the geometric decrement function.

It's important to note that the temperature gets updated only after a successful transfer between the vehicles. Successful transfer means that the new routes of the vehicles taking part in the transfer are feasible, and meet all the constraints. Now, perhaps I could have ignored some of the constraints in order to escape the local optima more easily, but, as I've found, once you let go off some constraints in a difficult problem such as this, it seems very difficult to get back to the solution which satisfies all of those constraints.

# Conclusion

Due to the stochasticity of the algorithm, every simulation can yield different results. Also, because of the problem difficulty, with many constraints, it is often very hard for the algorithm to escape from local optima.

Since I used Python programming language, it was expected that the performance would be quite slow, considering Python's dynamic nature, so it's rather obvious that the results for the same algorithm would be better if it was written in Java or C++.

Probably, the better solution could be obtained by combining the algorithm with another one, e.g. Ant colony optimization. Regarding the simulated annealing, perhaps some advanced techniques could be used, such as re-heating or order of searching (maybe, after the initial solution is found, we could try to search first for the shortest routes and transfer the customers from there to the ones that are a bit longer in order to try and discard any vehicles which don't take that long routes).

Also, we could use the 2-opt swap between the routes, but it seemed to me that technique would be more appropriate if the primary objective was to minimize the total distance of all routes.

All in all, simulated annealing is a fast algorithm that is often used for the vehicle routing problem because it produces a solid solution, especially when combined with some other algorithms.