Large Scale Optimization

M.Sc. Business analytics – Fall Quarter 2016 | Athens University of Economics and Business

Project Report

Stamatis Pitsios – BAPT1502

2016

Table of Contents

[Introduction 1](#_Toc468820997)

[Component 1 1](#_Toc468820998)

[Component 2 1](#_Toc468820999)

[Component 3 1](#_Toc468821000)

[Component 4 3](#_Toc468821001)

[Component 5 5](#_Toc468821002)

[Component 6 7](#_Toc468821003)

# Introduction

The vehicle routing problem (VRP) is a combinatorial optimization and integer programming problem which asks "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?". It generalizes the well-known travelling salesman problem (TSP), in the way that now we have a set of vehicles with a given capacity and we should serve a set of customers each one of them with a specific demand in product units. Determining the optimal solution is an NP-hard problem in combinatorial optimization, so the size of problems that can be solved optimally is limited. For the purposes of this assignment, we will solve an instance of the VRP problem using various optimization methods such as greedy search, local search and tabu search.

Before moving on, it would be a good a good idea to give a brief description about the structure of the implementation. The whole project is written in Java and consists of six components, each one designed by extending the previous one to offer more capabilities. Each component is actually a Java package which contains one or more classes that tackle the problem requested at the given time. Every next component that requires input from previous ones, simply imports the necessary classes of other components and makes calls to the appropriate methods. This design was chosen in order to minimize code duplication. Let us now proceed with the details of each component.

# Component 1

In the first component were simply called to design and implement the necessary classes that will be used in the representation of an instance of the VRP. The representation should include the depot (the starting and ending point of every vehicle), the customers the vehicles and the solution. To deal with these requirements, 3 classes were developed. A class named **Node** that will be used to represents the customers and the depot. A class **Route** used for the representation of vehicles and finally a **Solution** class that contains a set of routes and a cost.

# Component 2

At this point, we were simply requested to generate an instance of the problem that contains a depot, 30 customers and ten vehicles with a capacity of 50. The coordinates of the depot were given. For the initialization of the coordinates of the customers, as well as their demand, a random generator was used. Finally, after obtaining the coordinates of all customers, we initialized the distance matrix that will be useful for the cost calculation of the solution.

# Component 3

This component consists of the implementation of a given greedy algorithm that gives a solution to the problem (obviously not the optimal one). The algorithm is based on the idea of the “Nearest Neighbor”, while also dealing with the demand of each customer and the capacity of every vehicle. The solution we obtained after implementing the algorithm, can be seen in the next page.

Solution{totalCost=931.0, routes=[

Route{capacity=50, load=50, cost=186.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=52, y=47, id=8, demand=7}

Node{x=46, y=54, id=19, demand=5}

Node{x=47, y=57, id=15, demand=4}

Node{x=41, y=63, id=21, demand=10}

Node{x=39, y=71, id=16, demand=4}

Node{x=44, y=70, id=23, demand=5}

Node{x=34, y=53, id=26, demand=5}

Node{x=12, y=62, id=10, demand=5}

Node{x=72, y=85, id=13, demand=5}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=48, cost=155.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=54, y=38, id=28, demand=9}

Node{x=56, y=36, id=18, demand=9}

Node{x=65, y=30, id=12, demand=8}

Node{x=79, y=29, id=2, demand=9}

Node{x=98, y=31, id=24, demand=9}

Node{x=96, y=2, id=27, demand=4}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=50, cost=175.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=65, y=58, id=4, demand=7}

Node{x=66, y=86, id=22, demand=6}

Node{x=90, y=82, id=6, demand=5}

Node{x=95, y=71, id=9, demand=9}

Node{x=98, y=70, id=5, demand=8}

Node{x=93, y=95, id=20, demand=8}

Node{x=87, y=97, id=29, demand=7}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=44, cost=187.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=45, y=25, id=17, demand=10}

Node{x=40, y=19, id=11, demand=9}

Node{x=39, y=2, id=30, demand=5}

Node{x=48, y=3, id=7, demand=4}

Node{x=73, y=7, id=25, demand=10}

Node{x=15, y=36, id=3, demand=6}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=17, cost=228.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=4, y=94, id=1, demand=7}

Node{x=99, y=98, id=14, demand=10}

Node{x=50, y=50, id=0, demand=0}]}]}

The solution can be read as follows. We have a solution with a total cost of 931 units which consists of a set of 5 routes-vehicles. Every route has a cost, the total capacity, the load it has when serving the customers, as well as a set of nodes that represent the customers. Customers within a route, are shown in the order they are served. For example, let’s have a look at the first route. The sequence of nodes that the first vehicles visits, is 0 🡪 8 🡪 19 🡪 15 🡪 21 🡪 16 🡪 23 🡪 26 🡪 10 🡪 13 🡪 0.

# Component 4

In this component, we use local search method in order to find a better solution, based on the one we got from component 3. The local search operator we use at this point, is Intra-Relocations. That is, we take every route and we check all relocations of every customer within this specific route. We check all relocations within every route and at each step we choose the one that gives the best results. We repeat the process, until no better solution was found. In our Java implementation, we provide the initial cost, as well as we print the new cost for every iteration the local search method performed. The output of the program can be seen below.

Initial Cost: 931.0

Iteration 1 - New Total Cost: 917.0

Iteration 2 - New Total Cost: 905.0

Iteration 3 - New Total Cost: 896.0

Iteration 4 - New Total Cost: 891.0

Iteration 5 - New Total Cost: 876.0

Iteration 6 - New Total Cost: 875.0

Iteration 7 - New Total Cost: 874.0

Iteration 8 - New Total Cost: 873.0

Iteration 9 - New Total Cost: 872.0

We can see that we started from an initial cost of 931 units and after 9 iterations we came up with a solution that has a cost of 872 units. After iteration number 9, no better instance of the problem found using Intra-Relocation moves, so the algorithm terminated. The whole solution we got at iteration 9, can be found on the next page.

Solution{totalCost=872.0, routes=[

Route{capacity=50, load=50, cost=165.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=46, y=54, id=19, demand=5}

Node{x=47, y=57, id=15, demand=4}

Node{x=41, y=63, id=21, demand=10}

Node{x=34, y=53, id=26, demand=5}

Node{x=12, y=62, id=10, demand=5}

Node{x=39, y=71, id=16, demand=4}

Node{x=44, y=70, id=23, demand=5}

Node{x=72, y=85, id=13, demand=5}

Node{x=52, y=47, id=8, demand=7}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=48, cost=153.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=54, y=38, id=28, demand=9}

Node{x=56, y=36, id=18, demand=9}

Node{x=65, y=30, id=12, demand=8}

Node{x=96, y=2, id=27, demand=4}

Node{x=98, y=31, id=24, demand=9}

Node{x=79, y=29, id=2, demand=9}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=50, cost=149.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=65, y=58, id=4, demand=7}

Node{x=95, y=71, id=9, demand=9}

Node{x=98, y=70, id=5, demand=8}

Node{x=90, y=82, id=6, demand=5}

Node{x=93, y=95, id=20, demand=8}

Node{x=87, y=97, id=29, demand=7}

Node{x=66, y=86, id=22, demand=6}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=44, cost=177.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=15, y=36, id=3, demand=6}

Node{x=40, y=19, id=11, demand=9}

Node{x=39, y=2, id=30, demand=5}

Node{x=48, y=3, id=7, demand=4}

Node{x=73, y=7, id=25, demand=10}

Node{x=45, y=25, id=17, demand=10}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=17, cost=228.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=4, y=94, id=1, demand=7}

Node{x=99, y=98, id=14, demand=10}

Node{x=50, y=50, id=0, demand=0}]}]}

# Component 5

At this part, we also used local search technique to improve the solution from component 3, but this time instead of looking only Intra-Relocation moves we also check Inter-Relocation moves. That is, we also create solutions that relocate a customer from a route j, and place them after another customer in a route k. Obviously, this gives much more candidate solutions that applying only Intra-Relocations. However, we should be careful that when relocating a customer into a different route, the load of the vehicle plus the demand of the relocated customer, does not exceed the capacity of the vehicle. The cost at every iteration, as this was outputted by the program is the following:

Initial Cost: 931.0

Iteration 1 - New Total Cost: 868.0

Iteration 2 - New Total Cost: 829.0

Iteration 3 - New Total Cost: 799.0

Iteration 4 - New Total Cost: 785.0

Iteration 5 - New Total Cost: 773.0

Iteration 6 - New Total Cost: 761.0

Iteration 7 - New Total Cost: 754.0

Iteration 8 - New Total Cost: 750.0

Iteration 9 - New Total Cost: 749.0

Iteration 10 - New Total Cost: 748.0

Iteration 11 - New Total Cost: 747.0

Iteration 12 - New Total Cost: 746.0

Iteration 13 - New Total Cost: 730.0

This time the algorithm performed 13 iterations and came up with a new solution with cost 730, much lower than the initial cost of 931 units and also lower than the solution of component 4 which had a cost of 872 units. The whole solution can be seen in the next page.

Solution{totalCost=730.0, routes=[

Route{capacity=50, load=28, cost=50.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=46, y=54, id=19, demand=5}

Node{x=41, y=63, id=21, demand=10}

Node{x=39, y=71, id=16, demand=4}

Node{x=44, y=70, id=23, demand=5}

Node{x=47, y=57, id=15, demand=4}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=48, cost=153.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=54, y=38, id=28, demand=9}

Node{x=56, y=36, id=18, demand=9}

Node{x=65, y=30, id=12, demand=8}

Node{x=96, y=2, id=27, demand=4}

Node{x=98, y=31, id=24, demand=9}

Node{x=79, y=29, id=2, demand=9}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=40, cost=130.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=65, y=58, id=4, demand=7}

Node{x=95, y=71, id=9, demand=9}

Node{x=98, y=70, id=5, demand=8}

Node{x=90, y=82, id=6, demand=5}

Node{x=72, y=85, id=13, demand=5}

Node{x=66, y=86, id=22, demand=6}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=45, cost=133.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=45, y=25, id=17, demand=10}

Node{x=40, y=19, id=11, demand=9}

Node{x=39, y=2, id=30, demand=5}

Node{x=48, y=3, id=7, demand=4}

Node{x=73, y=7, id=25, demand=10}

Node{x=52, y=47, id=8, demand=7}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=48, cost=264.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=34, y=53, id=26, demand=5}

Node{x=15, y=36, id=3, demand=6}

Node{x=12, y=62, id=10, demand=5}

Node{x=4, y=94, id=1, demand=7}

Node{x=87, y=97, id=29, demand=7}

Node{x=99, y=98, id=14, demand=10}

Node{x=93, y=95, id=20, demand=8}

Node{x=50, y=50, id=0, demand=0}]}]}

# Component 6

At this final component, we apply a different approach known as Tabu search. What this technique does, is to start from an original solution that comes from a greedy method. Then it proceeds like we did in component 5, by searching the best intra-relocation or inter-relocation move and applies it. The difference however, is that applies the best move, regardless if that move increases the total cost of solution! The idea is to try to escape from local minima by temporarily accepting moves that increase the cost, with the hope that we will result in a better solution. During the search, of course it is possible that we will not end up into a better solution, so we store in a variable the best solution ever found during the search. This process is called Tabu search.

Nevertheless, a problem it is possible to arise. This has to do with the fact that we temporarily proceed with a solution that is a little bit worse that the current solution. But if in the new state, the next best move is to go back, this will result into doing moves like the “pendulum swing” by constantly moving forward or backwards. To tackle this issue, we follow the next steps. We keep a 2-D array that represents all possible arc combinations between all nodes. At each of these cells we store a number indicating for how many iterations this arc has been forbidden. When we apply a relocation move (either intra or inter), we conceptually delete 3 arcs in the node sequences, so in the corresponding cells of the table described above, we forbid these 3 arcs for a predefined number of iterations. Then, when looking for a better move to apply, except the cost we also check if the arcs that are going to be created are forbidden in the current iterations. **Careful: we check if both 3 arcs have been forbidden to mark a move as Tabu**.

By following the approach described above, we can be sure that we will avoid states that we have visited earlier. It is also very possible to forbid moves that we haven’t visited before, but this is a tradeoff of the specific search method. For the need of this assignment, we use tabu search for a total number of 200 iterations, as this was requested in the description. **The tabu policy that was chosen under these circumstances, is to forbid an arc for a fixed number of 20 iterations**, which is the 1/10 of the total number of iterations.

When we run the algorithm, we got a solution with a total cost of 710. The best solution was found in the iteration number 45 out of 200 iterations. The whole solution can be seen in the next page.

Solution{totalCost=710.0, routes=[

Route{capacity=50, load=28, cost=50.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=47, y=57, id=15, demand=4}

Node{x=44, y=70, id=23, demand=5}

Node{x=39, y=71, id=16, demand=4}

Node{x=41, y=63, id=21, demand=10}

Node{x=46, y=54, id=19, demand=5}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=49, cost=158.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=56, y=36, id=18, demand=9}

Node{x=65, y=30, id=12, demand=8}

Node{x=73, y=7, id=25, demand=10}

Node{x=96, y=2, id=27, demand=4}

Node{x=98, y=31, id=24, demand=9}

Node{x=79, y=29, id=2, demand=9}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=40, cost=130.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=65, y=58, id=4, demand=7}

Node{x=95, y=71, id=9, demand=9}

Node{x=98, y=70, id=5, demand=8}

Node{x=90, y=82, id=6, demand=5}

Node{x=72, y=85, id=13, demand=5}

Node{x=66, y=86, id=22, demand=6}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=44, cost=108.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=45, y=25, id=17, demand=10}

Node{x=40, y=19, id=11, demand=9}

Node{x=39, y=2, id=30, demand=5}

Node{x=48, y=3, id=7, demand=4}

Node{x=54, y=38, id=28, demand=9}

Node{x=52, y=47, id=8, demand=7}

Node{x=50, y=50, id=0, demand=0}]}

Route{capacity=50, load=48, cost=264.0, route=[

Node{x=50, y=50, id=0, demand=0}

Node{x=34, y=53, id=26, demand=5}

Node{x=15, y=36, id=3, demand=6}

Node{x=12, y=62, id=10, demand=5}

Node{x=4, y=94, id=1, demand=7}

Node{x=87, y=97, id=29, demand=7}

Node{x=99, y=98, id=14, demand=10}

Node{x=93, y=95, id=20, demand=8}

Node{x=50, y=50, id=0, demand=0}]}]}