University of Southern Denmark

DM852

13 April 2020

Heuristics & Approximations Algorithms

Submitted To:
Marco Chiarandini
Lene Monrad Favrholdt
IMADA
Institute of Mathematics & Computer Science

Submitted By:
Alexander Lerche Falk
Spring - Master of Computer Science

Contents

1 Introduction

The goal of this project is to show the knowledge acquired by the student in regards to Heuristic- and Local Search Algorithms. The algorithms are developed to suit the Capacitated Vehicle Routing Problem (CVRP). CVRP is a combinatorial optimization problemm, where a set of n customers has to be visited by m vehicles. Each customer can only be visited one vehicle. The sum of demands by the customers assigned to a vehicle must not overseed the capacity of the vehicle. All vehicles start and end their route at the depot (the starting point). The objective is to minimise the travelling cost and try to use as few vehicles as possible. Heuristics and Local Search algorithms are applied to CVRP, where the heuristics are laying the foundation of possible solutions of routes, and the Local Search algorithms try to improve the solutions. Three heuristic algorithms have been developed:

- Nearest Neightbour - Pick nearest customer from the depot or current customer without breaking the capacity
- Furthest Neightbour (with nearest clustering) - Pick customer furthest away from the depot, find the closest customers until the capacity is reached, and generate a route by using Nearest Neightbour
- K-Nearest Neightbour - Pick k nearest customers and randomly pick one of the k options. Proceed until no more customers without breaking the capacity

Two local search algorithms have been developed:

- Two Opt - Pick a route generated by some heuristic. Pick two edges, swap the edges, and compare the distance from before and after the swap. If better, make the swap.
- Three Opt - Pick a route generated by some heuristic. Pick three edges, swap the edges, and compare the distance from before and after the swap. If better, make the swap.

At the end of the report, comparison of the heuristics and results are presented. All the code can be found at GitHub

2 Nearest Neighbour

The Nearest Neighbour algorithm is a simple, yet powerful, approach to tackle CVRP. The algorithm goes starts at the depot and picks the nearest customer by iterating through all of the customers. It marks the customer as visited, adds it to a vehicle, and continues until the capacity of the vehicle is reached (or nearly reached). It returns to the depot and start over with the remaining customers with a new vehicle.

The algorithm is described in pseudocode:

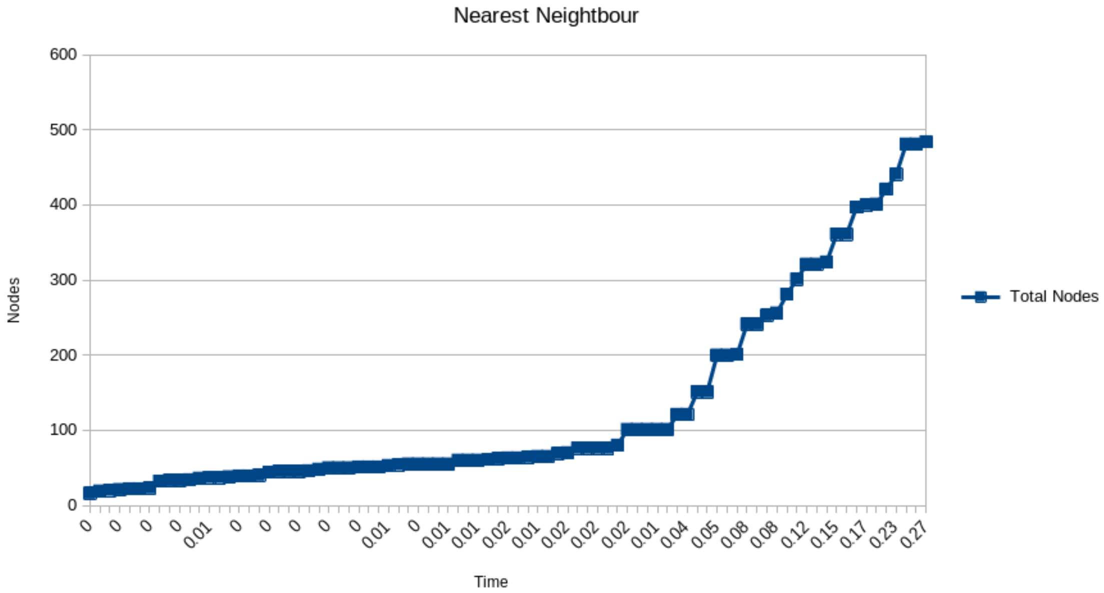
Algorithm 1 Nearest Neighbour

Require: Set of customers
Ensure: *Solution* (solution to the CVRP instance)

`j` $\leftarrow$ random node j
`i` $=$ `j` $\leftarrow$ starting point
`W` $= \{1, 2, ..., n\} \backslash \{j\} \leftarrow$ set of customers minus the randomly picked
function NearestNeighbour($instance, W$)
    while $W \neq \varnothing$ **do**
        $let\ j \in W \mid c_{ij} = min\{c_{ij} \mid j \in W\}$
        Shortest node by edge (i, j) is added to vehicle route
        Set $W = W \backslash \{j\} \leftarrow$ Customer i is marked visited
        $i = j \leftarrow$ setting i to next in line

The Nearest Neighbour has a running time of $\mathcal{O}(n^2)$, which is a quadric function. This means, that for every execution step done by the algorithm, the input size n has to be checked: $n * n$. In the developed code, a parameter has been added to perform performance testing on the algorithms. Every instance is executed by each of the algorithms, where time, cost, and numbers of nodes are saved. In Figure 1 (see appendix A for data), it is shown when the input size grows, the execution time grows quadric:

Figure 1: Algorithm analysis of Nearest Neighbour. $X = Time$, $Y = Nodes$



3 K Nearest Neightbours with a randomized twist

The K-Nearest Neightbour algorithm is the same as Nearest Neightbour but with a twist. The Nearest Neightbour algorithm has to downside of ending in a local optimum. This means, the solution is optimal within the set of candidate solutions. It is different from the global optimum, which is the optimal solution for all possible solutions. KNN tries to escape the local optimum by taking an unforseen step instead of always following the closest node. It starts by picking the k nearest neightbours and then by voting, it picks one of the k nearest neightbours. The vote in this case is done by randomly choosing a customer and then continue with the same approach.

The algorithm is described in pseudocode:

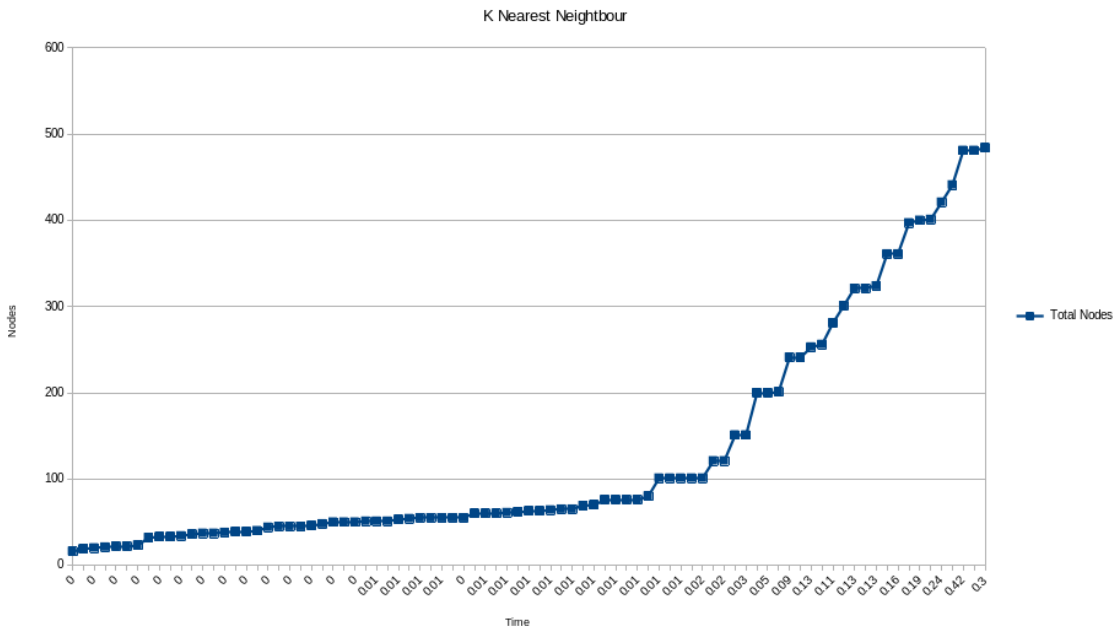
Algorithm 2 K-Nearest Neightbour

Require: Set of customers
Ensure: *Solution* (solution to the CVRP instance)

`j` $\leftarrow$ random node j
`i` $=$ `j` $\leftarrow$ starting point
`W` $= \{1, 2, ..., n\} \backslash \{j\} \leftarrow$ set of customers minus the randomly picked
`C` $= \varnothing \leftarrow$ empty set to store closest points to i
function KNearestNeightbour($instance, W$)
    while $W \neq \varnothing$ **do**
        $let\ j \in W \mid c_{ij} = min\{c_{ij} \mid j \in W\}$
        Every j is added to C
        Sort C ascending and take the first k customers
        Set j to a random pick in C
        Set $W = W \backslash \{j\} \leftarrow$ Customer j is marked visited
        $i = j \leftarrow$ setting i to next in line

The K-Nearest Neightbour has a running time of $\mathcal{O}(n^2 + k)$, which is a quadric function. The execution steps for the input size is the same as the nearest neightbour, but it adds the extra execution by randomly picking one of the k nearest neigtbours. In Figure 2 (see appendix B for data), it is shown when the input size grows, the execution time grows quadric:

Figure 2: Algorithm analysis of K-Nearest Neightbour. $X = Time$, $Y = Nodes$



4 Furthest Neightbour Cluster - Custom

NN and KNN takes the nearest neightbours into consideration and keeps growing by using the nearest neightbour. This can lead to issues with the later vehicles only having the customers left, which are far away from each other. Depending on the instance, KNN - and especially - NN can grow in one direction, and not taking the rest of the instance into account. The project has tried to accommodate the issue by picking the customer furthest away from the depot. The algorithm has been named Furthest Neightbour Cluster - lacking of better naming. The algorithm picks the customer furtest away from the depot, finds the closest nodes to the picked customer (like a cluster), and uses NN to find a suitable route within this cluster. Capacity is taking into considartion when picking the closest customers.

The algorithm is described in pseudocode:

Algorithm 3 Furthest Neightbour Cluster

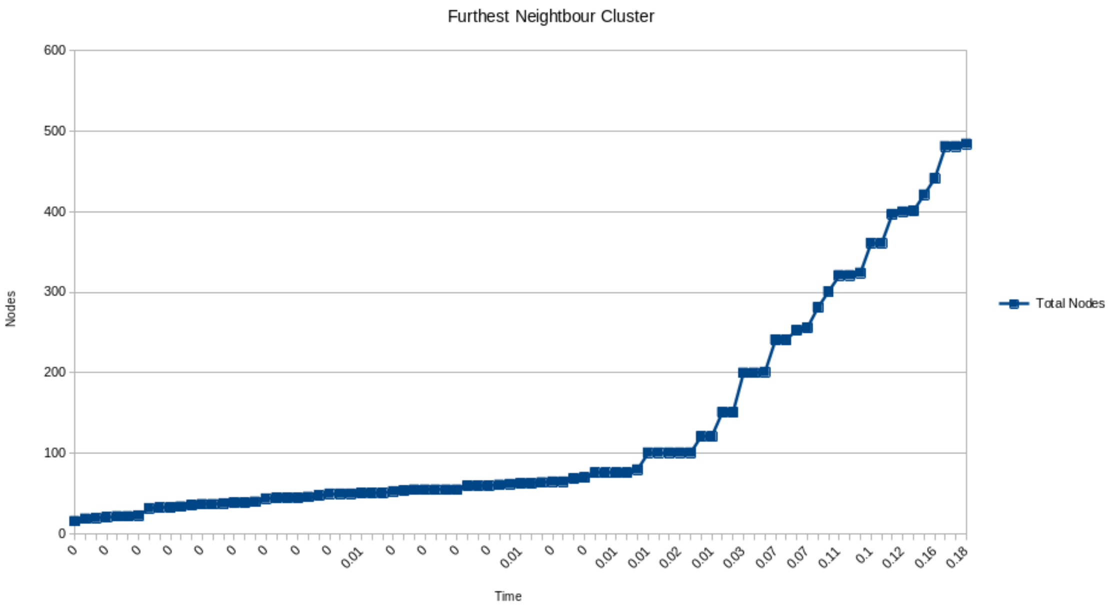
Require: Set of customers
Ensure: *Solution* (solution to the CVRP instance)

`j` $\leftarrow$ depot
`i` $=$ `j` $\leftarrow$ starting point
`W` $= \{1, 2, ..., n\} \backslash \{j\} \leftarrow$ set of customers minus the randomly picked
`F` $= \varnothing \leftarrow$ empty set to store closest customer to furhtest customer
function FurthestNeightbourCluster($instance, W$)
  while $W \neq \varnothing$ **do**
    let $j \in W \mid c_{ij} = max\{c_{ij} \mid j \in W\}$
    Furthest node by edge (i, j) is stored in j
    Set $W = W \backslash \{j\} \leftarrow$ Customer j is marked visited
    while $W \neq \varnothing$ & Capacity not reached **do**
      let $k \in W \mid c_{jk} = min\{c_{jk} \mid k \in W\}$
      Shortest node by edge (j, k) is stored in k
      Set $W = W \backslash \{k\} \leftarrow$ Customer k is marked visited
      $i = depot$ as starting point

The Furthest Neightbour Cluster has a running time of $\mathcal{O}(3n^2)$, which is a quadric function. We can calculate it by looking into the code and analyze. The first While-loop takes up $\mathcal{O}(n)$ and the For-loop inside it takes up $\mathcal{O}(n)$. Then the While-loop inside the outer While-loop takes up the same as before: $\mathcal{O}(n)$ and the same with the For-loop inside: $\mathcal{O}(n)$. The last While- and For-loop takes up the same: $\mathcal{O}(n)$. This gives us the following: $(\mathcal{O}(n) * \mathcal{O}(n)) + (\mathcal{O}(n)) * \mathcal{O}(n)) + (\mathcal{O}(n) * \mathcal{O}(n)) = \mathcal{O}(3n^2)$.
In Appendix D, the code for the algorithm is shown.

In Figure 3 (see appendix E for data), it is shown when the input size grows, the execution time grows quadric:

Figure 3: Algorithm analysis of K-Nearest Neightbour. $X = Time$, $Y = Nodes$



5 Two Opt & Three Opt

Heuristics generates solutions which can be global optimal, but tends to end up in a local optimum. One way of getting out of this is by using Local Search Optimization algorithms. These algorithms takes an existing solution to an optimization problem and try to optimize the solution. In this project, the goal is to minimize the total cost of the vehicle routes, and we can use Two Opt and Three Opt to try improving our existing solutions.
Two opt takes two routes and compare their distances. A swap is possible if the distance is being reduced in at least one route, while still preserving the maximum capacity limit of both routes.
Three Opt does the same as Two Opt, but instead of looking and two routes, it takes three into consideration.

Using the Nearest Neighbour heuristics to analysise the cost reduction using local search showed an improvement of $171424 - 167016 = 4408$ (sum of total costs from

.csv files / see appendix F for results) over all of the instances. And this is by only running Two Opt once on every solution of each instance.

6 Closing Notes

Heuristics give fast and good results. They are not global optimum but if the objective is to find a solution, and not the best, they are good for this. To get closer to global optimum, local search algorithms tries to get us closer. The issue with local search algorithms are: when to stop improving. Shown in the above results, Two Opt was only ran once to improve existing solutions, but it could have been run continiously and set a kill switch, when it is no longer improving. One issue with this is, when it gets stuck in a local optimum. In that case, other approaches should be taken to try and get away from the local optimum. An example of this could be by making bad moves.

Appendices

A Nearest Neighbour Results

Instance	Cost	Time	Total Nodes
Golden_04.xml	15746.0	0.26	481
Golden_13.xml	980.0	0.1	253
Golden_17.xml	924.0	0.08	241
Golden_19.xml	1756.0	0.15	361
Golden_05.xml	9443.0	0.07	201
Golden_09.xml	575.0	0.08	256
Golden_01.xml	5824.0	0.07	241
Golden_18.xml	1413.0	0.13	301
Golden_15.xml	1545.0	0.17	397
Golden_20.xml	2288.0	0.21	421
Golden_07.xml	11380.0	0.14	361
Golden_14.xml	1281.0	0.12	321
Golden_16.xml	1897.0	0.26	481
Golden_10.xml	764.0	0.13	324
Golden_11.xml	974.0	0.17	400
Golden_03.xml	12054.0	0.19	401
Golden_08.xml	12876.0	0.23	441
Golden_06.xml	9689.0	0.13	281
Golden_12.xml	1165.0	0.27	484
Golden_02.xml	9245.0	0.22	321
A-n63-k10.xml	1505.0	0.02	63
A-n61-k09.xml	1294.0	0.01	61
A-n37-k05.xml	965.0	0.01	37
A-n54-k07.xml	1690.0	0.01	54
A-n69-k09.xml	1555.0	0.02	69
A-n62-k08.xml	1486.0	0.01	62
A-n53-k07.xml	1270.0	0.01	53
A-n38-k05.xml	966.0	0.0	38
A-n65-k09.xml	1422.0	0.01	65
A-n60-k09.xml	1583.0	0.01	60
A-n64-k09.xml	1715.0	0.01	64
A-n45-k07.xml	1614.0	0.0	45
A-n63-k09.xml	2056.0	0.01	63
A-n39-k06.xml	1054.0	0.0	39
A-n33-k05.xml	765.0	0.0	33
A-n34-k05.xml	969.0	0.0	34
A-n33-k06.xml	903.0	0.0	33
A-n39-k05.xml	1076.0	0.0	39
A-n46-k07.xml	1159.0	0.0	46
A-n45-k06.xml	1158.0	0.0	45
A-n37-k06.xml	1164.0	0.0	37
A-n55-k09.xml	1392.0	0.0	55
A-n44-k06.xml	1251.0	0.0	44
A-n36-k05.xml	1059.0	0.0	36
A-n48-k07.xml	1316.0	0.01	48
A-n80-k10.xml	2054.0	0.01	80
A-n32-k05.xml	926.0	0.0	32
P-n051-k10.xml	962.0	0.0	51

B K-Nearest Neighbour Results

Instance	Cost	Time	Total Nodes
Golden_04.xml	27923.0	0.42	481
Golden_13.xml	1411.0	0.13	253
Golden_17.xml	1207.0	0.09	241
Golden_19.xml	2299.0	0.18	361
Golden_05.xml	13912.0	0.08	201
Golden_09.xml	965.0	0.09	256
Golden_01.xml	10802.0	0.07	241
Golden_18.xml	1686.0	0.14	301
Golden_15.xml	2165.0	0.19	397
Golden_20.xml	3025.0	0.24	421
Golden_07.xml	19552.0	0.16	361
Golden_14.xml	1773.0	0.13	321
Golden_16.xml	2633.0	0.29	481
Golden_10.xml	1231.0	0.13	324
Golden_11.xml	1538.0	0.19	400
Golden_03.xml	20616.0	0.22	401
Golden_08.xml	23087.0	0.24	441
Golden_06.xml	17759.0	0.11	281
Golden_12.xml	1920.0	0.3	484
Golden_02.xml	15024.0	0.14	321
A-n63-k10.xml	2060.0	0.01	63
A-n61-k09.xml	1742.0	0.01	61
A-n37-k05.xml	1170.0	0.0	37
A-n54-k07.xml	1860.0	0.0	54
A-n69-k09.xml	1755.0	0.01	69
A-n62-k08.xml	2189.0	0.01	62
A-n53-k07.xml	1616.0	0.01	53
A-n38-k05.xml	1202.0	0.0	38
A-n65-k09.xml	1882.0	0.01	65
A-n60-k09.xml	2285.0	0.01	60
A-n64-k09.xml	2292.0	0.01	64
A-n45-k07.xml	1585.0	0.0	45
A-n63-k09.xml	2487.0	0.01	63
A-n39-k06.xml	1441.0	0.0	39
A-n33-k05.xml	1205.0	0.0	33
A-n34-k05.xml	1397.0	0.0	34
A-n33-k06.xml	1107.0	0.0	33
A-n39-k05.xml	1289.0	0.0	39
A-n46-k07.xml	1453.0	0.0	46
A-n45-k06.xml	1613.0	0.0	45
A-n37-k06.xml	1509.0	0.0	37
A-n55-k09.xml	1812.0	0.01	55
A-n44-k06.xml	1503.0	0.0	44
A-n36-k05.xml	1307.0	0.0	36
A-n48-k07.xml	1717.0	0.0	48
A-n80-k10.xml	2732.0	0.01	80
A-n32-k05.xml	1282.0	0.0	32
P-n051-k10.xml	1146.0	0.0	51

C Furthest Neightbour Cluster - Results

Instance	Cost	Time	Total Nodes
Golden_04.xml	74042.0	0.19	481
Golden_13.xml	3137.0	0.05	253
Golden_17.xml	3050.0	0.07	241
Golden_19.xml	6218.0	0.1	361
Golden_05.xml	36673.0	0.03	201
Golden_09.xml	2521.0	0.07	256
Golden_01.xml	21912.0	0.05	241
Golden_18.xml	4350.0	0.07	301
Golden_15.xml	5432.0	0.15	397
Golden_20.xml	8151.0	0.14	421
Golden_07.xml	49258.0	0.1	361
Golden_14.xml	4211.0	0.11	321
Golden_16.xml	6905.0	0.17	481
Golden_10.xml	3519.0	0.11	324
Golden_11.xml	4729.0	0.12	400
Golden_03.xml	51120.0	0.12	401
Golden_08.xml	54002.0	0.16	441
Golden_06.xml	43663.0	0.07	281
Golden_12.xml	6295.0	0.18	484
Golden_02.xml	34681.0	0.1	321
A-n63-k10.xml	3343.0	0.01	63
A-n61-k09.xml	3200.0	0.01	61
A-n37-k05.xml	1762.0	0.0	37
A-n54-k07.xml	3374.0	0.0	54
A-n69-k09.xml	3756.0	0.0	69
A-n62-k08.xml	4130.0	0.0	62
A-n53-k07.xml	2817.0	0.0	53
A-n38-k05.xml	1982.0	0.0	38
A-n65-k09.xml	3420.0	0.0	65
A-n60-k09.xml	3699.0	0.01	60
A-n64-k09.xml	3872.0	0.0	64
A-n45-k07.xml	2592.0	0.0	45
A-n63-k09.xml	4233.0	0.0	63
A-n39-k06.xml	2262.0	0.0	39
A-n33-k05.xml	1970.0	0.0	33
A-n34-k05.xml	2076.0	0.0	34
A-n33-k06.xml	1608.0	0.0	33
A-n39-k05.xml	1942.0	0.0	39
A-n46-k07.xml	2592.0	0.0	46
A-n45-k06.xml	2617.0	0.0	45
A-n37-k06.xml	2217.0	0.0	37
A-n55-k09.xml	3250.0	0.0	55
A-n44-k06.xml	2345.0	0.0	44
A-n36-k05.xml	1990.0	0.0	36
A-n48-k07.xml	2639.0	0.0	48
A-n80-k10.xml	4686.0	0.01	80
A-n32-k05.xml	2018.0	0.0	32
P-n051-k10.xml	1814.0	0.01	51

D Furthest Neightbour Cluster - Code

```
        while unvisited_nodes:
            # Find the node furthest away from the depot
            for index, p in enumerate(unvisited_nodes):
                dist = sol.instance.distance(current_node['pt'], p['pt'])

                if dist > furthest_distance or furthest_distance == 0:
                    furthest_distance = dist
                    furthest_node = p

            unvisited_nodes.remove(furthest_node)
            furthest_node_closest_nodes += [furthest_node]
            capacity += furthest_node['rq']
            # As long as the capacity is not reached
            capacity_not_reached = True
            while capacity_not_reached and unvisited_nodes:

                # Loop through the nodes closest to the furthest node
                for index, p in enumerate(unvisited_nodes):
                    dist = sol.instance.distance(furthest_node['rq'], p['rq'])
                    if dist < closest_distance_from_furthest_node or closest_distance_from_furthest_node == 0:
                        closest_distance_from_furthest_node = dist
                        closest_node_from_furthest_node = p

                    # If the closest node can fit the vehicle, then add it to its own list
                    if capacity + closest_node_from_furthest_node['rq'] <= self.instance.capacity:
                        capacity += closest_node_from_furthest_node['rq']
                        furthest_node_closest_nodes += [closest_node_from_furthest_node]
                        unvisited_nodes.remove(closest_node_from_furthest_node)
                        closest_distance_from_furthest_node = 0
                    else:
                        capacity_not_reached = False

            # Find the shortest route in the furthest node list of closest nodes
            sub_current_node = furthest_node
            closest_distance = 0
            closest_node = None
            while furthest_node_closest_nodes:

                for node in furthest_node_closest_nodes:
                    dist = sol.instance.distance(sub_current_node['rq'], node['rq'])

                    if dist < closest_distance or closest_distance == 0:
                        closest_distance = dist
                        closest_node = node

                route += [self.instance.nodes.index(closest_node)]
                sub_current_node = closest_node
                furthest_node_closest_nodes.remove(closest_node)
                closest_distance = 0
```

E Two Opt - Local Search Algorithm

Algorithm 4 Two_Opt

Require: $dist(A, B)$ returns distance from point A to B

function TWOOPT($route\_i, route\_j, i\_to\_j, j\_to\_i$)
    for all i, j such that i, j are routes index combination **do**
        evaluate d1 = total length of the two edges before being swapped
        evaluate d2 = total length of the two edges after being swapped
        if `d1` $> d2$ **then**
            Make the swap between two the routes
        else
            return

F Nearest Neighbour with Two Opt - Results

Instance	Cost	Time	Total Nodes
P-n016-k08.xml	497	0	16
P-n019-k02.xml	247	0	19
P-n020-k02.xml	285	0	20
P-n021-k02.xml	219	0	21
P-n022-k02.xml	238	0	22
P-n022-k08.xml	690	0	22
P-n023-k08.xml	669	0	23
A-n32-k05.xml	911	0	32
A-n33-k05.xml	749	0	33
A-n33-k06.xml	890	0	33
A-n34-k05.xml	943	0	34
A-n36-k05.xml	919	0	36
A-n37-k05.xml	931	0	37
A-n37-k06.xml	1111	0	37
A-n38-k05.xml	954	0	38
A-n39-k06.xml	1018	0	39
A-n39-k05.xml	996	0	39
P-n040-k05.xml	542	0	40
A-n44-k06.xml	1201	0	44
A-n45-k07.xml	1485	0	45
A-n45-k06.xml	1104	0	45
P-n045-k05.xml	633	0	45
A-n46-k07.xml	1107	0	46
A-n48-k07.xml	1277	0	48
P-n050-k10.xml	791	0	50
P-n050-k07.xml	650	0	50
P-n050-k08.xml	705	0.01	50
P-n051-k10.xml	942	0	51
CMT06.xml	722	0	51
CMT01.xml	722	0.01	51
A-n53-k07.xml	1227	0	53
A-n54-k07.xml	1554	0	54
A-n55-k09.xml	1324	0	55
P-n055-k08.xml	649	0.01	55
P-n055-k07.xml	700	0.01	55
P-n055-k15.xml	1079	0.01	55
P-n055-k10.xml	761	0	55
A-n60-k09.xml	1528	0.01	60
P-n060-k10.xml	846	0	60
P-n060-k15.xml	1119	0.01	60
A-n61-k09.xml	1202	0.01	61
A-n62-k08.xml	1456	0.01	62
A-n63-k10.xml	1482	0	63
A-n63-k09.xml	1985	0.01	63
A-n64-k09.xml	1630	0.01	64
A-n65-k09.xml	1331	0.01	65
P-n065-k10.xml	992	0.01	65
A-n69-k09.xml	1501	0.01	69