Authors: Weronika Zawadzka 151943 Eliza Czaplicka 151963

Date: October 13, 2024

Best solutions have been checked with the solution checker

Summary Report for w1_greedy_heuristics

Based on methods from [our github project]

Problem description

We are given three columns of integers with a row for each node. The first two columns contain x and y coordinates of the node positions in a plane. The third column contains node costs. The goal is to select exactly 50% of the nodes (if the number of nodes is odd we round the number of nodes to be selected up) and form a Hamiltonian cycle (closed path) through this set of nodes such that the sum of the total length of the path plus the total cost of the selected nodes is minimized.

The distances between nodes are calculated as Euclidean distances rounded mathematically to integer values. The distance matrix should be calculated just after reading an instance, and then only the distance matrix (no nodes coordinates) should be accessed by optimization methods to allow instances defined only by distance matrices.

Pseudocode of implemented algorithms

nodeList -> list of nodes with id and cost

distanceMatrix -> matrix of euclidean distances between all nodes in nodeList

- **Random solution**

```
ids = []
// add all node IDs to the list
for node in nodeList:
        ids.add(node)
// shuffle the ids randomly
ids.shuffle()
// leave 50% of nodes
return ids[:nodeList.size()//2]
```

- **Greedy cycle**

```
// select randomly the starting index
startNode = random node from nodeList
Add startNode to currCycle
Mark startNode as visited
// choose the nearest vertex and create two vertices cycle
Find the nearest neighbor to startNode and add to currCycle nearestNode
Add nearestNode to currCycle
Mark nearestNode as visited
// repeat until 50% of nodes in currCycle
for each unvisited node j do:
    for each possible position in currCycle do:
        prevNode = node at current position in currCycle
        nextNode = node at next position
        increase = distanceMatrix(prevNode, j) + distanceMatrix(j,
                nextNode) - distanceMatrix(prevNode,
            nextNode) + nodeList.getCost(j)
        if increase < bestIncrease then:
            bestIncrease = increase
            bestNode = j
            bestPosition = position + 1
Insert bestNode into currCycle at bestPosition
Mark bestNode as visited
// return currCycle
```

- **Nearest Neighbor adding nodes at the end**

```
// select randomly the starting index
startNode = random node from nodeList
Add startNode to currCycle
Mark startNode as visited
// repeat until 50% of nodes in currPath
    for all possible unvisited nodes:
        increase = distanceMatrix[lastNodeIndex][j] +
nodeList.getCost(j)
        if increase < bestIncrease:
            bestIncrease = increase
            bestNodeIndex = j
    Insert bestNodeIndex into currPath at the end
    Mark bestNodeIndex  as visited
// return currPath
```

- **Nearest Neighbor adding nodes at the beginning, end and every place inside**

```
// select randomly the starting index
startNode = random node from nodeList
Add startNode to currCycle
Mark startNode as visited
// repeat until 50% of nodes in currPath
        for i=0 to currPath.size():
                // repeat until 50% of nodes in currPath
                        currentNode = currPath.get(i)
                        for j=0 to nodeList.size() do:
                                // check all possible unvisited nodes
                                if not visited[j]:
                                        increase =
                distanceMatrix[currentNode][j] + nodeList.getCost(j)
                                        if increase < bestIncrease:
                                // track the best node and position
                                                bestIncrease = increase
                                                bestNodeIndex = j
                        // add best node to the best position in the path
                        currPath.add(i+1, bestNodeIndex)
                        visited[bestNodeIndex) = true
// return currPath
```

Summary performance of each method

Instance: A

Summary for Execution Time (ms) - bonus

method	min	max	mean
GreedyCycleMethod	2	91	9.21
NearNeighborEndMethod	0	16	0.145
NearNeighborMethod	0	15	0.26
RandomMethod	0	5	0.12

Summary for Objective Function Value

method	min	max	mean
GreedyCycleMethod	71719	75803	72987.8
NearNeighborEndMethod	83590	89433	85208.68
NearNeighborMethod	171521	207728	187048.41
RandomMethod	239099	289258	264976.085

We see that the winner for instance A was the method Greedy Cycle, which achieved the lowest objective function of 71,719. The worst one was the random solution, which later in 2D visualizations can be seen as why it is so - very entangled, paths don't make sense - but of course it was expected as it is random selection. Nearest neighbor method with adding nodes to the end achieved not far from the winner solution's score, while nearest neighbor with adding nodes anywhere - did not perform well.

Instance: B

Summary for Execution Time (ms) - bonus

method	min	max	mean
GreedyCycleMethod	1	109	9.315

NearNeighborEndMethod	0	6	0.125
NearNeighborMethod	0	8	0.165
RandomMethod	0	1	0.08

Summary for Objective Function Value

method	**min**	**max**	**mean**
GreedyCycleMethod	49001	57271	51568.455
NearNeighborEndMethod	52319	59030	54359.255
NearNeighborMethod	117926	161912	141517.085
RandomMethod	189044	237132	213532.255

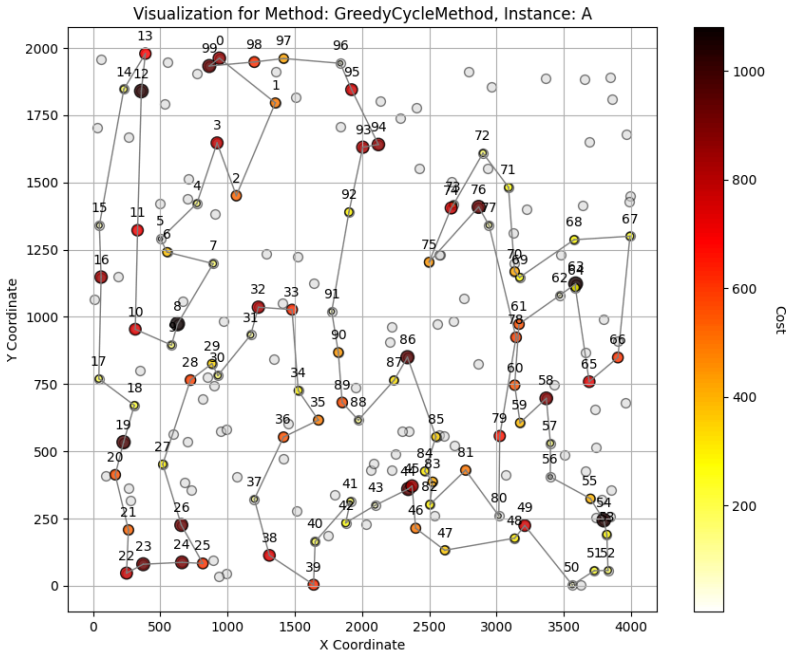
Again the winner is Greedy cycle method with a low best score of 49,001. The rest of the methods placed the same in this instance as well. This instance differs from the first one by having less extreme costs of nodes. This instance was also on average more time consuming for the methods (except random one).

2D visualizations of best solutions

Instance: A

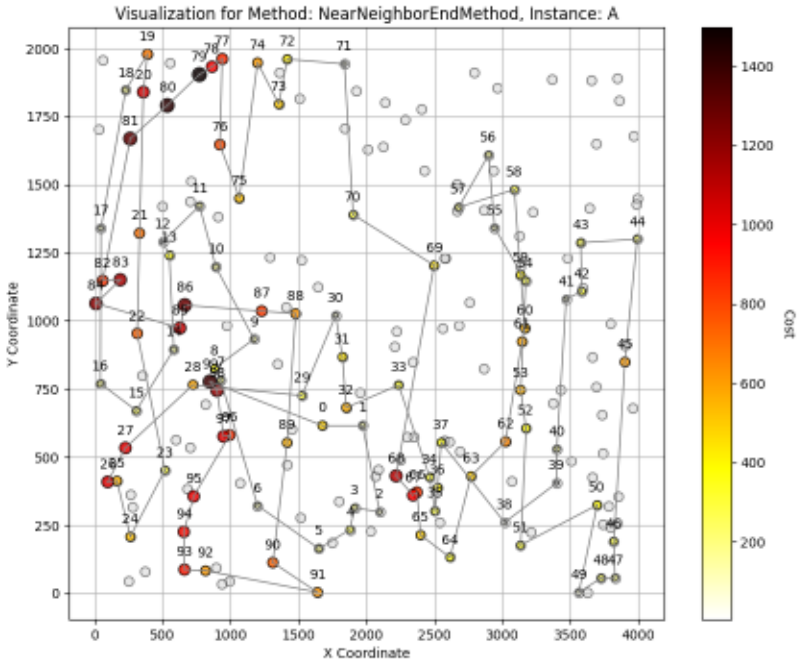
Method: GreedyCycleMethod

Obj.f. value: 71719



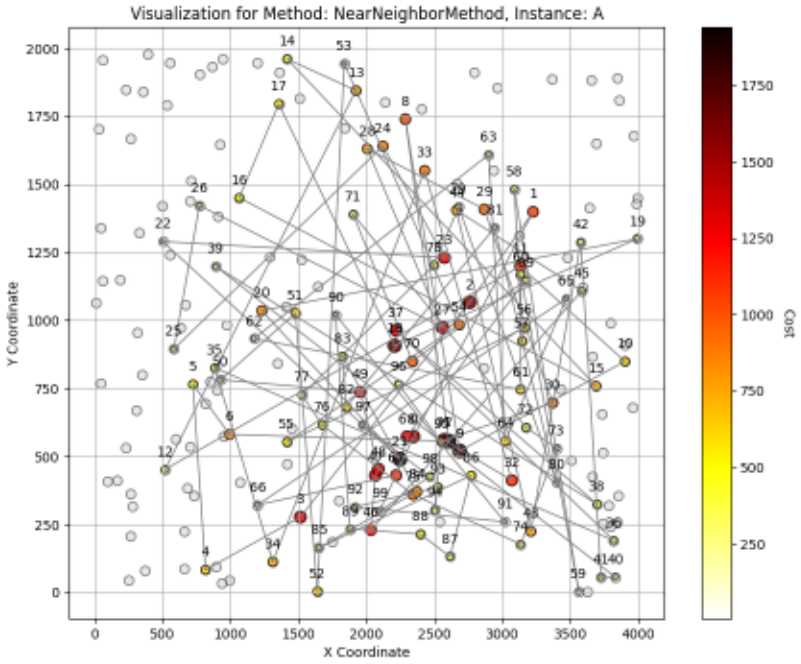
Method: NearNeighborEndMethod

Obj.f. value: 83590



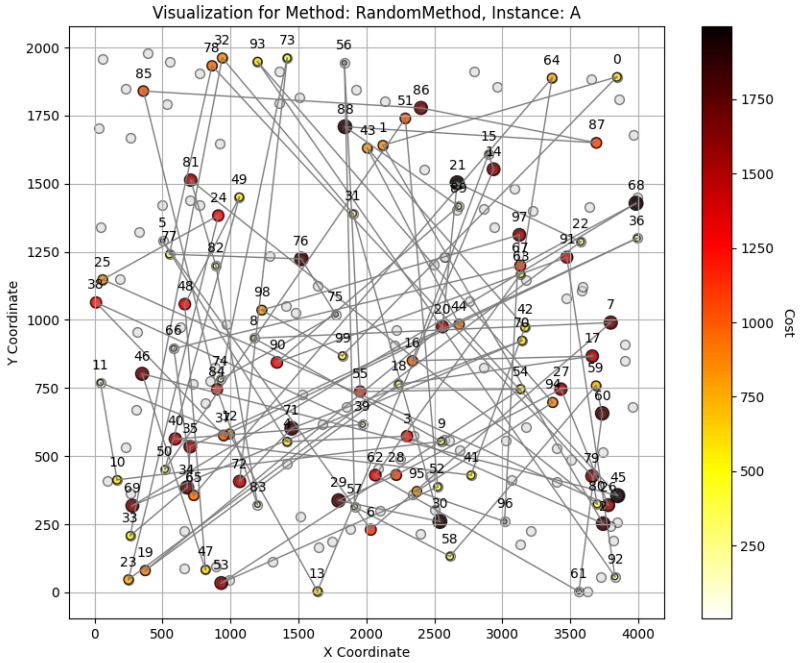
Method: NearNeighborMethod

Obj.f. value: 171521



Method: RandomMethod

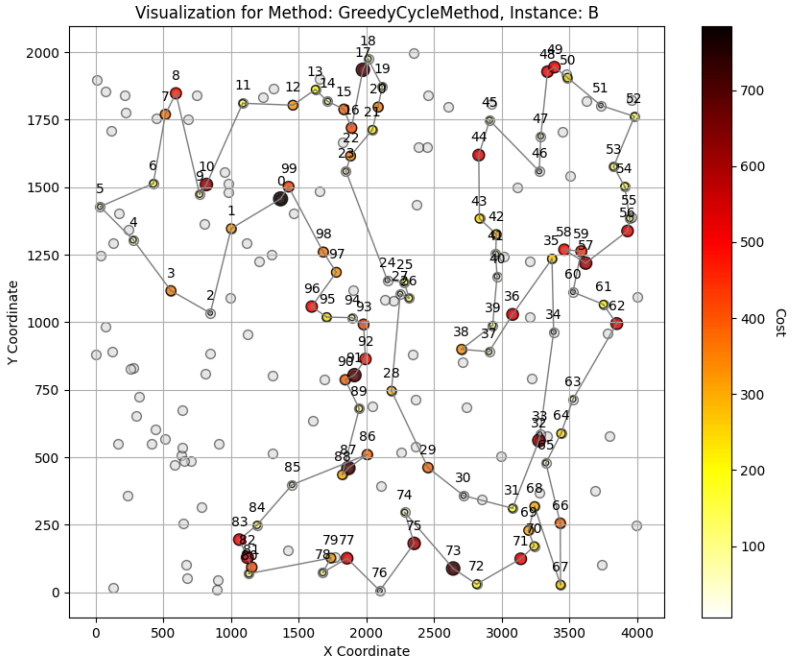
Obj.f. value: 239099



Instance: B

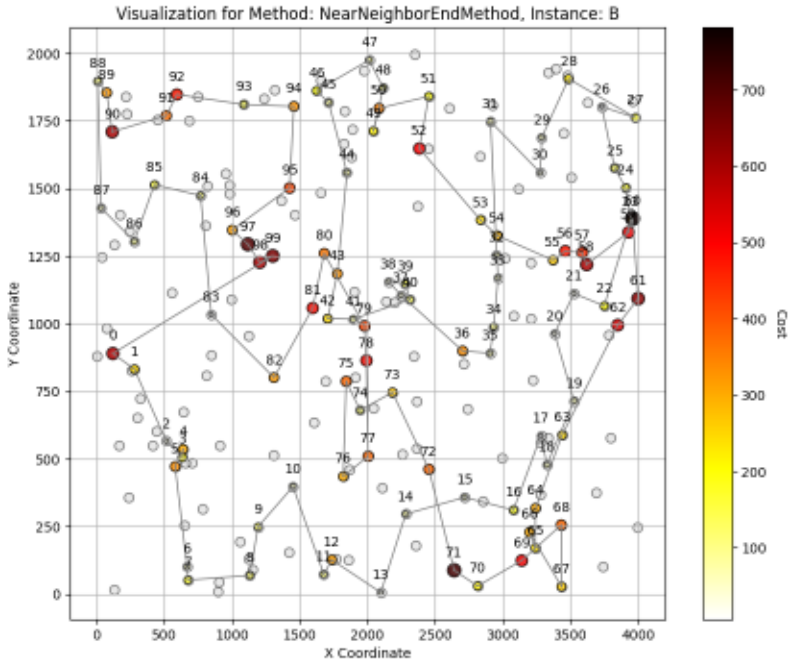
Method: GreedyCycleMethod

Obj.f. value: 49001



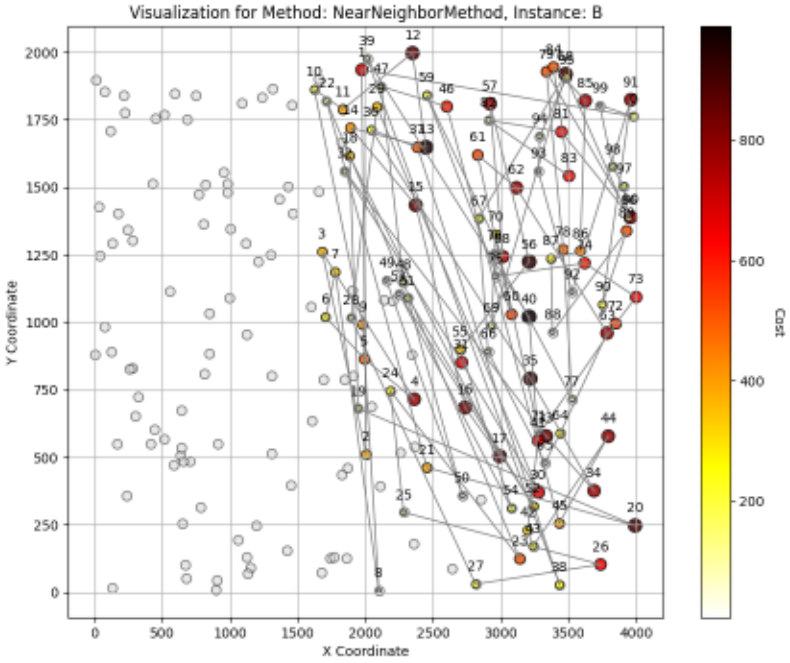
Method: NearNeighborEndMethod

Obj.f. value: 52319



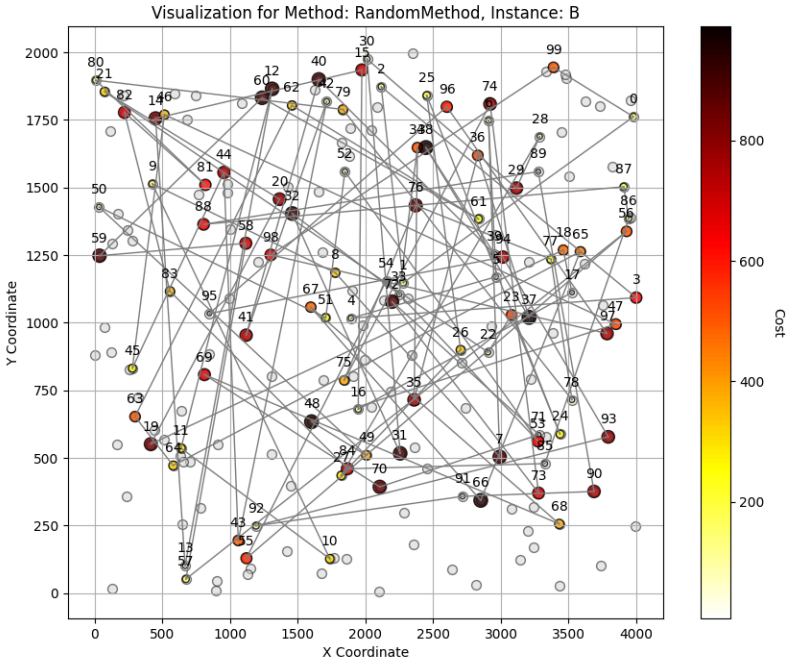
Method: NearNeighborMethod

Obj.f. value: 117926



Method: RandomMethod

Obj.f. value: 189044



Best solutions, indices

Instance: A

Method: GreedyCycleMethod

Lowest Objective Function Value (f_val): 71719

Solution:

93, 0, 46, 68, 139, 193, 41, 115, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 177, 10, 190, 4, 112, 84, 184, 43, 116, 65, 59, 118, 51, 151, 133, 162, 123, 127, 70, 135, 180, 154, 53, 100, 26, 86, 75, 44, 25, 16, 171, 175, 113, 56, 31, 78, 145, 179, 92, 57, 52, 185, 119, 40, 196, 81, 90, 165, 106, 178, 14, 144, 62, 9, 148, 102, 49, 55, 129, 120, 2, 101, 1, 97, 152, 124, 94, 63, 79, 80, 176, 137, 23, 186, 89, 183, 143, 117, 140

Method: NearNeighborEndMethod

Lowest Objective Function Value (f_val): 83590

Solution:

133, 63, 53, 180, 154, 135, 123, 65, 116, 59, 115, 139, 193, 41, 42, 160, 34, 22, 18, 108, 69, 159, 181, 184, 177, 54, 30, 48, 43, 151, 176, 80, 79, 94, 97, 101, 1, 152, 120, 78, 145, 185, 40, 165, 90, 81, 113, 175, 171, 16, 31, 44, 92, 57, 106, 49, 144, 62, 14, 178, 52, 55, 129, 2, 75, 86, 26, 100, 121, 148, 137, 183, 143, 0, 117, 46, 68, 93, 140, 36, 163, 199, 146, 195, 103, 5, 96, 118, 51, 162, 127, 70, 112, 4, 84, 35, 149, 131, 47, 105

Method: NearNeighborMethod

Lowest Objective Function Value (f_val): 171521

Solution:

19, 138, 128, 194, 112, 43, 149, 125, 114, 87, 81, 3, 184, 89, 143, 196, 46, 0, 130, 90, 118, 99, 193, 37, 186, 42, 139, 111, 23, 102, 179, 74, 82, 15, 127, 116, 113, 12, 31, 115, 175, 171, 165, 25, 9, 40, 158, 136, 182, 122, 65, 51, 70, 183, 167, 162, 52, 55, 14, 16, 178, 57, 59, 144, 129, 185, 123, 121, 189, 106, 124, 137, 92, 145, 44, 100, 133, 151, 148, 62, 78, 49, 79, 80, 26, 135, 2, 75, 86, 154, 176, 120, 180, 1, 101, 152, 94, 63, 97, 53

Method: RandomMethod

Lowest Objective Function Value (f_val): 239099

Solution:

164, 186, 13, 189, 162, 193, 158, 174, 59, 152, 54, 34, 149, 70, 155, 144, 124, 169, 94, 190, 111, 61, 165, 10, 198, 146, 85, 91, 121, 173, 150, 137, 93, 177, 11, 166, 90, 131, 103, 63, 28, 2, 52, 23, 167, 188, 192, 112, 96, 46, 184, 114, 1, 156, 57, 122, 183, 180, 75, 196, 17, 16, 136, 178, 21, 35, 42, 3, 107, 104, 55, 45, 24, 143, 65, 176, 141, 41, 140, 38, 31, 110, 115, 123, 47, 69, 64, 95, 76, 62, 72, 8, 175, 117, 179, 26, 120, 32, 118, 80

Instance: B

Method: GreedyCycleMethod

Lowest Objective Function Value (f_val): 49001

Solution:

85, 51, 121, 131, 135, 63, 122, 133, 10, 90, 191, 147, 6, 188, 169, 132, 13, 161, 70, 3, 15, 145, 195, 168, 29, 109, 35, 0, 111, 81, 153, 163, 180, 176, 86, 95, 128, 106, 143, 124, 62, 18, 55, 34, 170, 152, 183, 140, 4, 149, 28, 20, 60, 148, 47, 94, 66, 22, 130, 99, 185, 179, 172, 166, 194, 113, 114, 137, 103, 89, 127, 165, 187, 146, 77, 97, 141, 91, 36, 61, 175, 78, 142, 45, 5, 177, 82, 87, 21, 8, 104, 56, 144, 160, 33, 138, 182, 11, 139, 134

Method: NearNeighborEndMethod

Lowest Objective Function Value (f_val): 52319

Solution:

16, 1, 117, 31, 54, 193, 190, 80, 175, 5, 177, 36, 61, 141, 77, 153, 163, 176, 113, 166, 86, 185, 179, 94, 47, 148, 20, 60, 28, 140, 183, 152, 18, 62, 124, 106, 143, 0, 29, 109, 35, 33, 138, 11, 168, 169, 188, 70, 3, 145, 15, 155, 189, 34, 55, 95, 130, 99, 22, 66, 154, 57, 172, 194, 103, 127, 89, 137, 114, 165, 187, 146, 81, 111, 8, 104, 21, 82, 144, 160, 139, 182, 25, 121, 90, 122, 135, 63, 40, 107, 100, 133, 10, 147, 6, 134, 51, 98, 118, 74

Method: NearNeighborMethod

Lowest Objective Function Value (f_val): 117926

Solution:

60, 161, 82, 139, 41, 144, 138, 11, 141, 160, 188, 132, 84, 167, 13, 69, 119, 129, 195, 8, 93, 81, 169, 165, 111, 77, 75, 187, 33, 15, 26, 159, 168, 88, 76, 64, 145, 189, 137, 70, 110, 180, 89, 127, 48, 114, 184, 3, 109, 29, 153, 35, 103, 0, 163, 143, 181, 53, 83, 155, 128, 170, 174, 52, 194, 113, 106, 34, 101, 124, 55, 176, 172, 57, 22, 62, 18, 166, 130, 4, 154, 199, 152, 9, 149, 59, 99, 95, 86, 66, 179, 23, 185, 183, 140, 28, 94, 47, 148, 20

Method: RandomMethod

Lowest Objective Function Value (f_val): 189044

Solution:

60, 109, 3, 57, 33, 62, 152, 129, 11, 122, 61, 54, 150, 190, 44, 161, 8, 185, 130, 30, 85, 107, 106, 128, 194, 155, 143, 21, 140, 174, 70, 50, 2, 0, 189, 41, 170, 110, 167, 18, 65, 158, 169, 45, 71, 1, 133, 172, 171, 82, 63, 138, 168, 180, 29, 142, 66, 80, 98, 92, 192, 34, 6, 156, 193, 99, 186, 182, 114, 19, 58, 176, 12, 26, 53, 104, 69, 95, 166, 132, 40, 191, 72, 131, 87, 113, 94, 47, 125, 183, 76, 153, 5, 48, 83, 121, 184, 52, 74, 149

Conclusions

The best method was the greedy cycle construction heuristics, for both of the instances. Interestingly, the ranking of methods in our case stayed the same in instance B. The worst one was obviously the random one, but after random, the second worst performing was the NN method considering adding the node at all possible positions, i.e. at the end, at the beginning, or at any place inside the current path. Other observations include the fact the greedy cycle algorithm produces a very characteristic and structured shape. This is due to the fact that at each step, it tries to construct the shortest possible cycle by selecting the node that causes the smallest increase in the overall cycle length. Random is highly entangled, with no clear pattern or form. The NN algorithm, when nodes are inserted at all possible positions in the path, tends to explore primarily in the area of the first randomly selected node esp. in instance B - the exploration remains limited to the initial area. Plain NN, since it always looked at the nearest neighbor at the end of the path, it did explore a further out from the initial area.