Evolutionary Computation Assignment 1: Greedy Heuristics

BENGRICHE Tidiane (ER-2037)
Szczublińska Joanna Wiktoria (156070)

Contents

1 Problem

We are provided with a dataset consisting of three integer columns, where each row represents a node. The first two columns specify the x and y coordinates of each node in a two-dimensional plane, while the third column indicates the cost associated with each node.

The objective is to choose exactly half of the nodes (rounding up if the total number is odd) and construct a Hamiltonian cycle, a closed loop that visits each selected node exactly once. The aim is to minimize the combined total of the path length and the cumulative cost of the selected nodes.

For this assignment, we had to implement three algorithms:

- Random solution.
- Nearest neighbor considering adding the node only at the end of the current path.
- Nearest neighbor considering adding the node at all possible position, i.e. at the end, at the beginning, or at any place inside the current path.
- Greedy cycle

You can see the repository of our assignment on github:
`https://github.com/tbengric/EC_laboratory/tree/main/assignment_1`

2 Pseudocode

In this section, we will present the pseudocodes of the algorithms. For this assignment, we will code in C++.

2.1 Objective function

Our objective function passed correctly the solution checker tests for TSPA and TSPB.

Objective Function

```
computeObjective(path, distance matrix, nodes) :
    totalDist ← 0
    totalCost ← 0
    FOR i FROM 0 TO LENGTH(path) - 2 DO :
      totalCost ← totalCost + nodes[path[i]].cost
      totalDist ← totalDist + dist[path[i]][path[i+1]]

    RETURN totalDist + totalCost
```

2.2 Random

Random Solution

```
randomSolution(selectedNodes) :
    path ← COPY(selectedNodes)
    shuffle(path)
    APPEND path[0] TO path // return to start
    RETURN path
```

2.3 Nearest Neighbor

Nearest Neighbor

```
nearestNeighborEnd(selectedNodes, distance matrix, nodes, starting node):
    path ← [startNodeId]
    maxSize ← FLOOR(LENGTH(nodes) / 2)
    visited ← array of FALSE with size LENGTH(nodes)
    visited[startNodeId] ← TRUE
    WHILE LENGTH(path) < maxSize DO
      bestNode ← -1
      bestScore ← INF
      FOR each node IN nodes DO
        IF visited[node.id] THEN CONTINUE
        score ← dist[path[-1]][node.id] + node.cost
        IF score < bestScore THEN
          bestScore ← score
          bestNode ← node.id
        END IF
      END FOR
      APPEND bestNode TO path
      visited[bestNode] ← TRUE
    END WHILE
    APPEND startNodeId TO path
    RETURN path
```

2.4 Nearest Neighbor Flexible

Nearest Neighbor Flexible Heuristic

```
path ← [startNodeId]
maxSize ← FLOOR(LENGTH(nodes) / 2)
visited ← array of FALSE with size LENGTH(nodes)
visited[startNodeId] ← TRUE
WHILE LENGTH(path) < maxSize DO
  bestNode ← -1
  bestPos ← -1
  bestScore ← INF
  FOR each node IN nodes DO
    IF visited[node.id] THEN CONTINUE
    FOR i FROM 0 TO LENGTH(path) DO
      tempPath ← COPY(path)
      INSERT node.id AT position i IN tempPath
      score ← computeObjective(tempPath, dist, nodes)
      IF score < bestScore THEN
        bestScore ← score
        bestNode ← node.id
        bestPos ← i
      END IF
    END FOR
  END FOR
  INSERT bestNode AT bestPos IN path
  visited[bestNode] ← TRUE
END WHILE
APPEND path[0] TO path
RETURN path
```

2.5 Greedy Cycle

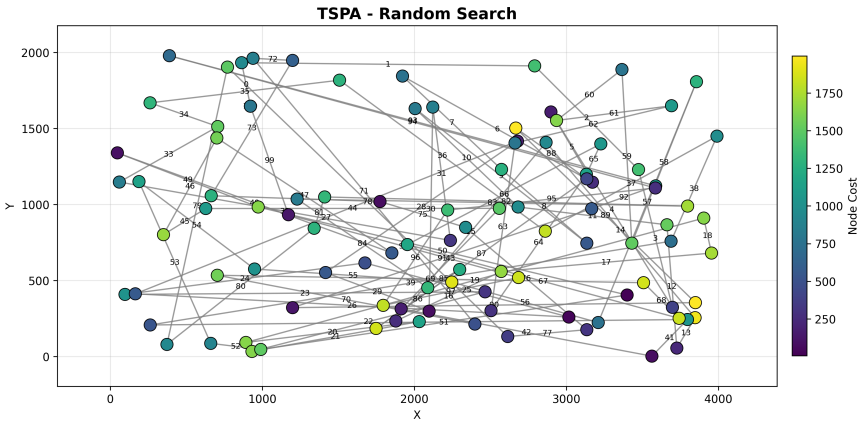
Greedy Cycle

```
path ← [startNodeId]
numToSelect ← FLOOR(LENGTH(nodes) / 2)
visited ← array of FALSE with size LENGTH(nodes)
visited[startNodeId] ← TRUE
// Select best second node to form initial cycle
bestSecondNode ← -1
bestInitialScore ← INF
FOR each node IN nodes DO
  IF visited[node.id] THEN CONTINUE
  score ← dist[startNodeId][node.id] + nodes[startNodeId].cost + node.cost
  IF score < bestInitialScore THEN
    bestInitialScore ← score
    bestSecondNode ← node.id
  END IF
END FOR
APPEND bestSecondNode TO path
visited[bestSecondNode] ← TRUE
APPEND startNodeId TO path // complete initial 2-node cycle
// Iteratively insert remaining nodes
WHILE LENGTH(path) < numToSelect + 1 DO
  bestNode ← -1
  bestPos ← -1
  bestScore ← INF
  FOR each node IN nodes DO
    IF visited[node.id] THEN CONTINUE
    FOR i FROM 0 TO LENGTH(path) - 2 DO
      tempPath ← COPY(path)
      INSERT node.id AT position i IN tempPath
      score ← computeObjective(tempPath, dist, nodes)
      IF score < bestScore THEN
        bestScore ← score
        bestNode ← node.id
        bestPos ← i
      END IF
    END FOR
  END FOR
  INSERT bestNode AT bestPos IN path
  visited[bestNode] ← TRUE
END WHILE
RETURN path
```

3 Results

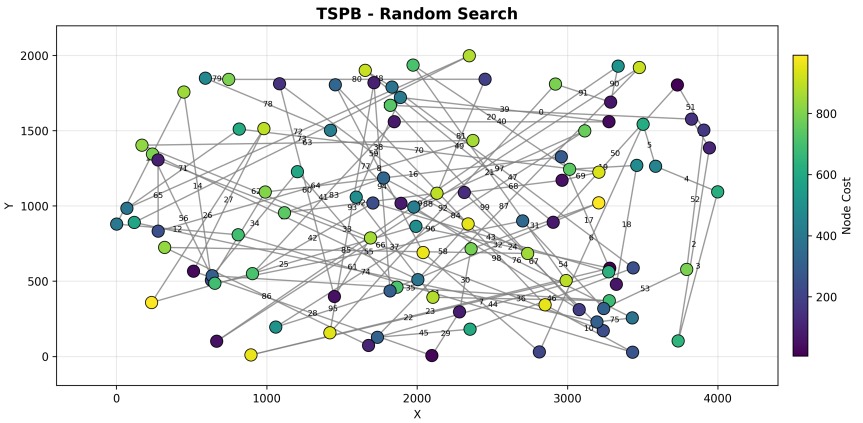
After trying these differents algorithms, we will check their effectiveness by showing the constructed cycle.

In order, we will start from the algorithm that seems least efficient to the one that shows the best results. In the plot, the cost of each node is represented using a color scale: yellow indicates higher cost, while purple indicates lower cost. Additionally, the edges show the sequence of nodes in the path starting from 0. Below the figure, we displayed the IDs of the nodes in the best path.



Best Path: 68 - 140 - 73 - 8 - 31 - 144 - 106 - 89 - 37 - 57 - 23 - 167 - 169 - 171 - 50 - 102 - 154 - 74 - 98 - 187 - 126 - 158 - 156 - 97 - 30 - 133 - 120 - 54 - 62 - 162 - 53 - 186 - 94 - 146 - 110 - 199 - 153 - 172 - 27 - 196 - 177 - 56 - 16 - 180 - 111 - 5 - 192 - 191 - 79 - 22 - 197 - 78 - 29 - 4 - 195 - 131 - 99 - 25 - 58 - 91 - 21 - 155 - 95 - 9 - 101 - 3 - 138 - 122 - 13 - 168 - 166 - 86 - 93 - 117 - 96 - 176 - 124 - 44 - 75 - 36 - 190 - 72 - 109 - 174 - 118 - 182 - 189 - 6 - 178 - 61 - 188 - 123 - 52 - 119 - 108 - 40 - 12 - 173 - 87 - 59 - 68

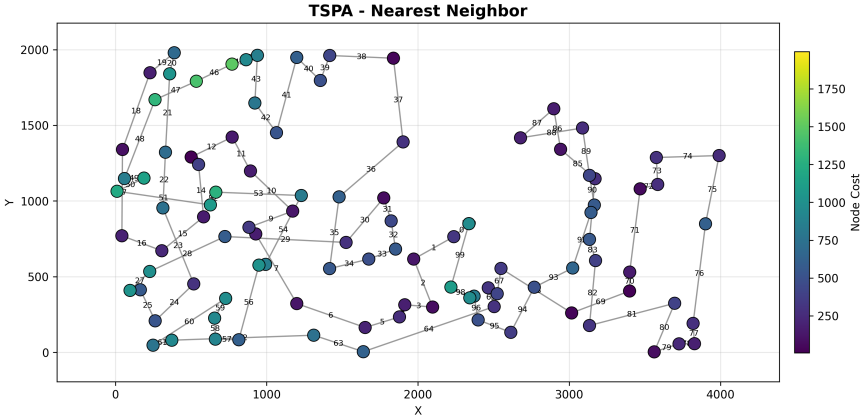
Figure 1: Random Search TSPA



Best Path: 126 - 148 - 94 - 75 - 57 - 99 - 9 - 187 - 68 - 6 - 186 - 137 - 19 - 27 - 44 - 31 - 116 - 55 - 113 - 130 - 83 - 161 - 129 - 46 - 176 - 144 - 196 - 67 - 117 - 141 - 77 - 41 - 110 - 87 - 118 - 54 - 26 - 58 - 182 - 13 - 183 - 168 - 173 - 138 - 180 - 61 - 97 - 194 - 132 - 65 - 106 - 20 - 47 - 48 - 103 - 143 - 164 - 32 - 82 - 37 - 147 - 177 - 157 - 96 - 69 - 1 - 135 - 127 - 35 - 62 - 181 - 191 - 38 - 84 - 16 - 89 - 114 - 33 - 134 - 10 - 115 - 155 - 39 - 158 - 11 - 119 - 24 - 36 - 101 - 190 - 4 - 140 - 53 - 7 - 169 - 21 - 45 - 174 - 160 - 163 - 126

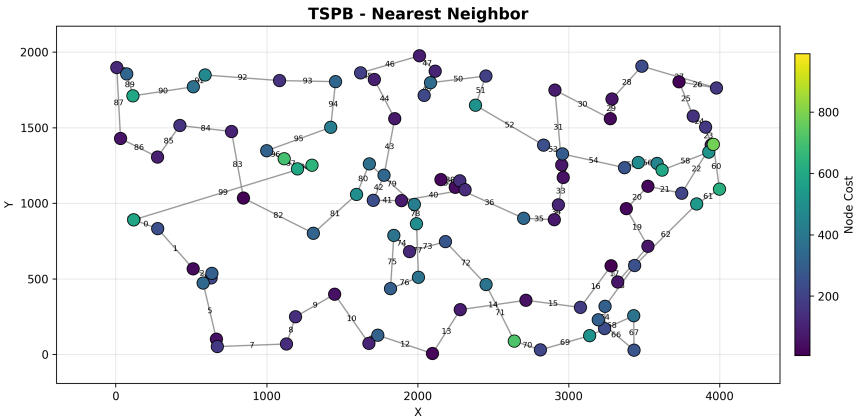
Figure 2: Random Search TSPB

For the Random Search algorithm, we quickly realize its inefficiency because the cycle is cut everywhere and there is no cost optimization as we can see many yellow nodes.



Best Path: 124 - 94 - 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 - 133 - 162 - 51 - 137 - 183 - 143 - 0 - 117 - 46 - 68 - 93 - 140 - 36 - 163 - 199 - 146 - 195 - 103 - 5 - 96 - 118 - 149 - 131 - 112 - 4 - 84 - 35 - 10 - 190 - 127 - 70 - 101 - 97 - 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 - 124

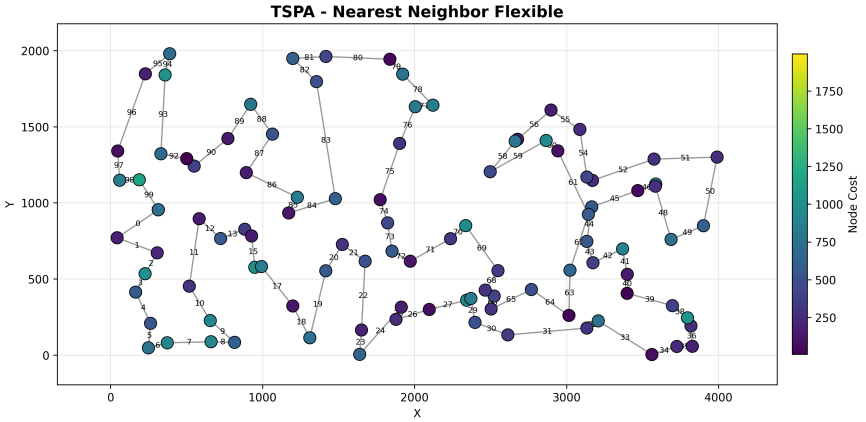
Figure 3: Nearest Neighbor TSPA (End)



Best Path: 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 - 16

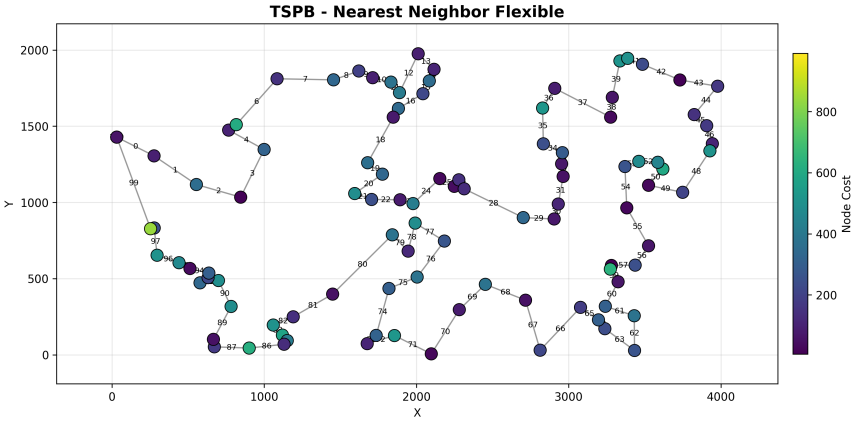
Figure 4: Nearest Neighbor TSPB (End)

For the Nearest Neighbor algorithm, which considers all end possible positions when constructing the path to the next node, we can also observe its inefficiency. Although it performs much better than Random Search what we can be seen with cost optimization by more purple/blue nodes, the paths are often crossed and appear quite messy.



Best Path: 181 - 34 - 160 - 48 - 54 - 177 - 10 - 190 - 4 - 112 - 84 - 184 - 42 - 43 - 116 - 65 - 131 - 149 - 123 - 127 - 162 - 151 - 133 - 135 - 70 - 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 - 0 - 51 - 59 - 118 - 115 - 46 - 68 - 139 - 41 - 193 - 159 - 69 - 108 - 18 - 22 - 146 - 195 - 181

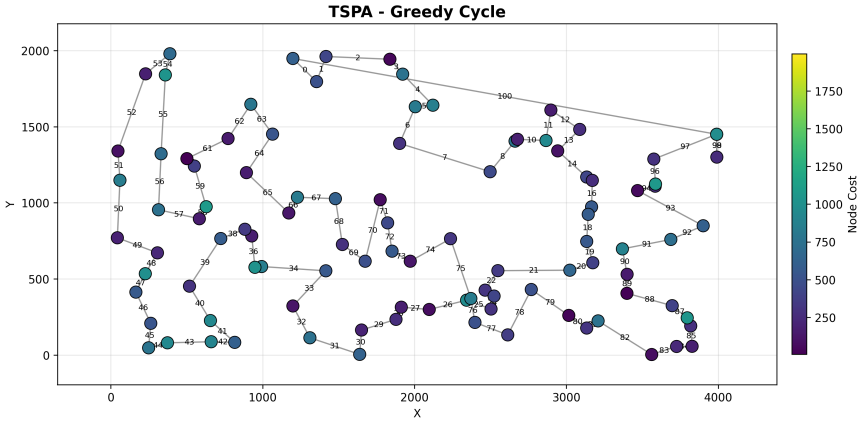
Figure 5: Nearest Neighbor Flexible TSPA



Best Path: 63 - 135 - 131 - 121 - 51 - 90 - 191 - 147 - 6 - 188 - 169 - 132 - 13 - 70 - 3 - 15 - 145 - 195 - 168 - 139 - 11 - 182 - 138 - 33 - 160 - 29 - 0 - 109 - 35 - 143 - 106 - 124 - 62 - 18 - 55 - 34 - 170 - 152 - 183 - 140 - 4 - 149 - 28 - 20 - 60 - 148 - 47 - 94 - 66 - 179 - 185 - 22 - 99 - 130 - 95 - 86 - 166 - 194 - 176 - 180 - 113 - 103 - 114 - 137 - 127 - 89 - 163 - 187 - 153 - 81 - 77 - 141 - 91 - 36 - 61 - 21 - 82 - 111 - 144 - 8 - 104 - 177 - 5 - 45 - 142 - 78 - 175 - 162 - 80 - 190 - 136 - 73 - 193 - 31 - 54 - 117 - 198 - 156 - 1 - 197 - 63

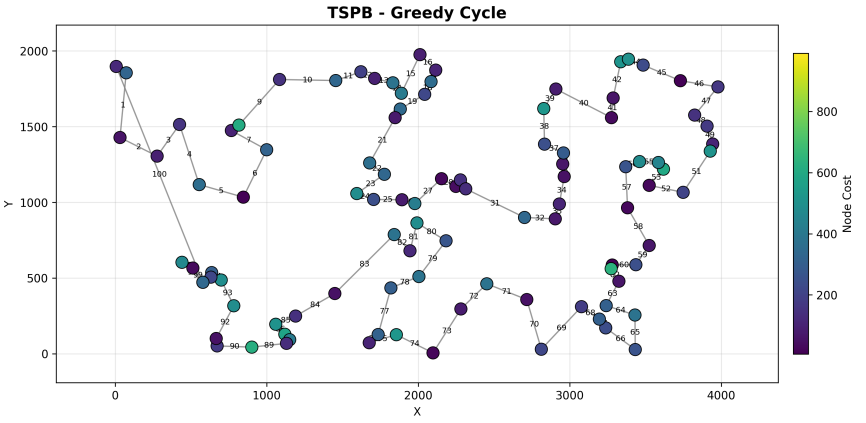
Figure 6: Nearest Neighbor Flexible TSPB

For the Nearest Neighbor Flexible algorithm, where the next node can be added at any position in the existing path, the results are quite good. Cost optimization is achieved through the inclusion of low cost nodes, and the paths are less crossed, making them appear more natural.



Best Path: 117 - 0 - 143 - 183 - 89 - 186 - 23 - 137 - 148 - 9 - 62 - 102 - 144 - 14 - 49 - 178 - 106 - 52 - 55 - 57 - 92 - 129 - 152 - 97 - 1 - 101 - 100 - 53 - 180 - 154 - 135 - 70 - 127 - 123 - 162 - 149 - 131 - 65 - 116 - 43 - 184 - 84 - 112 - 4 - 190 - 10 - 177 - 54 - 48 - 160 - 34 - 146 - 22 - 18 - 108 - 69 - 159 - 181 - 42 - 5 - 41 - 193 - 139 - 68 - 46 - 115 - 59 - 118 - 51 - 151 - 133 - 176 - 80 - 79 - 63 - 94 - 26 - 86 - 75 - 2 - 120 - 44 - 25 - 16 - 171 - 175 - 113 - 56 - 31 - 78 - 145 - 179 - 196 - 81 - 185 - 40 - 119 - 165 - 27 - 90 - 27

Figure 7: Greedy Cycle TSPA



Best Path: 40 - 107 - 63 - 135 - 122 - 131 - 121 - 51 - 90 - 191 - 147 - 6 - 188 - 169 - 132 - 13 - 70 - 3 - 15 - 145 - 195 - 168 - 139 - 11 - 182 - 138 - 33 - 160 - 29 - 0 - 109 - 35 - 143 - 106 - 124 - 62 - 18 - 55 - 34 - 170 - 152 - 183 - 140 - 4 - 149 - 28 - 20 - 60 - 148 - 47 - 94 - 66 - 179 - 185 - 22 - 99 - 130 - 95 - 86 - 166 - 194 - 176 - 180 - 113 - 103 - 114 - 137 - 127 - 89 - 163 - 187 - 153 - 81 - 77 - 141 - 91 - 36 - 61 - 21 - 82 - 111 - 144 - 8 - 104 - 177 - 5 - 45 - 142 - 78 - 175 - 162 - 80 - 190 - 136 - 73 - 54 - 31 - 193 - 198 - 117 - 193

Figure 8: Greedy Cycle TSPB

For the Greedy Cycle algorithm, the results are very similar to those of the Nearest Neighbor Flexible algorithm. Additionally, we can see that almost all nodes have low costs and the paths are not crossed, so they are forming a more natural cycle.

Method	Avg (Min, Max)
Random Search	264431.57 (233614, 296772)
Nearest Neighbor End	85108.51 (83182, 89433)
Nearest Neighbor Flexible	73594.10 (71868, 75953)
Greedy Cycle	72055.95 (70212, 74540)

Table 1: Average, minimum, and maximum objective values for TSPA

Method	Avg (Min, Max)
Random Search	212716.98 (182577, 237263)
Nearest Neighbor End	54390.43 (52319, 59030)
Nearest Neighbor Flexible	48694.25 (44609, 57283)
Greedy Cycle	47365.54 (43599, 57925)

Table 2: Average, minimum, and maximum objective values for TSPB

Thanks to these tables, we can see that the Greedy Cycle is the most effective method in both scenarios for the TSP type. The Nearest Neighbor Flexible algorithm ranks second, and its performance is not significantly different from that of the Greedy Cycle. Based on the minimum and maximum values, we can also conclude that the Greedy Cycle algorithm has the smallest range between these values.

4 Conclusion

In this study, we compared four algorithms for constructing TSP cycles: Random Search, Nearest Neighbor, Nearest Neighbor Flexible, and Greedy Cycle. Random Search proved the least effective, producing highly crossed paths with little cost optimization. The standard Nearest Neighbor algorithm improved upon Random Search but still generated messy paths.

The Nearest Neighbor Flexible algorithm resulted in more natural, less crossed paths and better cost optimization. The Greedy Cycle algorithm produced results very similar to Nearest Neighbor Flexible, including almost all low-cost nodes and forming well-structured cycles. Among these algorithms, the one that is stronger than the others is the **Greedy Cycle**.

Based on both cost metrics and path structure, the Greedy Cycle algorithm consistently demonstrated the best performance, with the smallest range between minimum and maximum values. Nearest Neighbor Flexible ranked as a second, showing only minor differences in efficiency.

This assignment allowed us to observe the differences between the search algorithms and provided a foundation for introducing more advanced concepts in class. By analyzing their performance, strengths, and limitations across different scenarios, we gained a deeper understanding of how algorithmic efficiency impacts problem-solving in computer science.