Evolutionary Computation

Assignment 1

Antoni Lasik 148287
Daniel Jankowski 148257
Source code: https://github.com/JankowskiDaniel/evolutionary-computation

Problem description

The task involves analyzing three columns of integers, each row corresponding to a single node. The initial two columns designate the x and y coordinates, pinpointing the nodes' locations on a plane, while the third column specifies the cost associated with each node. The objective is to meticulously choose an exact half of the total nodes (in cases where the total node count is an odd number, the count of nodes to be selected is adjusted upward to the nearest whole number) to construct a Hamiltonian cycle, which is essentially a continuous loop that passes through each member of the selected set of nodes. The criterion for this selection is that the aggregate of the complete path's length and the cumulative cost of the chosen nodes should be as low as possible.

To quantify the distances between nodes, we employ the Euclidean distance formula, and the resulting figures are rounded off to the nearest integer in a standard mathematical fashion. Moreover, as part of the distance between nodes, we take into account the cost of the destination node. This ensures that cost has a significant impact on the final results.

We implement three heuristic methods for solving the problem: random, nearest neighbor, and a greedy cycle.

Pseudocode of implemented algorithms

An additional method for calculating the distance matrix

```
Calculate_distance_matrix(coords, costs):
    dist_matrix = [][]
    FOR i in range(len(coords)):
        FOR j in range(len(coords)):
            dist_matrix[i][j] = round(sqrt((coords[i].x – coords[j].x)**2 + (coords[i].y – coords[j].y)**2) + costs[j]
    return dist_matrix
```

The random solution algorithm

```
Generate_random_solution(dist_matrix, costs):
    num_nodes = dist_matrix.shape[0]
    num_select = (num_nodes + 1) // 2
    selected_nodes = randomly select num_select nodes that will be in the solution and suffle them
    total_distance = 0
    FOR N in range(1, len(selected_nodes)):
        previous_node = selected_nodes[N-1]
        current_node = selected_nodes[N]
        total_distance += dist_matrix[previous_node][current_node]
    // add the distance between the last and the first node
    total_distance += dist_matrix[selected_nodes[-1], selected_nodes[0]]
    return selected_nodes, total_distance
```

The nearest neighbor algorithm

```
Generate_nearest_neighbor_solution(dist_matrix,  start_node):
    num_nodes = dist_matrix.shape[0]
    num_select = (num_nodes + 1) // 2
    selected_nodes = [start_node]
    unselected_nodes = {num_nodes} \ {start_node}
    total_distance = 0
    WHILE len(selected_nodes) < num_select:
        last_node = selected_nodes[-1]
        nearest_node = min(unselected_nodes,
                           key=lambda node: dist_matrix[last_node][node])
        ADD nearest_node to selected_nodes
        REMOVE nearest_node from unselected_nodes
        total_distance += dist_matrix[last_node][nearest_node]
    // add the distance between the last and the first node
    total_distance +=  dist_matrix[selected_nodes[-1], selected_nodes[0]]
    return selected_nodes, total_distance
```

The greedy cycle algorithm

```
Generate_greedy_cycle_solution(dist_matrix,  start_node):
    num_nodes = dist_matrix.shape[0]
    num_select = (num_nodes + 1) // 2
    selected_nodes = [start_node]
    unselected_nodes = {num_nodes} \ {start_node}
    total_distance = 0
    WHILE len(selected_nodes) < num_select:
        FOR node in unselected_nodes:
            FOR i in range(len(selected_nodes):
                next_index = (i +1)% len(selected_nodes)
                 increase = (dist_matrix[selected_nodes[i], node] +
                        dist_matrix[node, selected_nodes[next_i]] -
                         dist_matrix[selected_nodes[i], selected_nodes[next_i]])
                // calculate the increase of distance if the node is inserted after the
          node of index i
                IF  increase < min_increase:
                        min_increase = increase
                        best_node = node
                        best_position = next_i  # Insert before next_i
                // keeping track of the position with the minimal increase
        ADD best_node at the best_position to selected nodes
        REMOVE best_node from unselected_nodes
        total_distance += min_increase
    total_distance +=  dist_matrix[selected_nodes[-1], selected_nodes[0]]
    return selected_nodes, total_distance
```

Results

The random solution

	Min	Max	Mean
Instance A	237,941	288,302	264,028.49
Instance B	243,288	295,269	266,655.16
Instance C	191,705	241,451	214,929.07
Instance D	191,218	242,515	219,678.85

The nearest neighbor algorithm

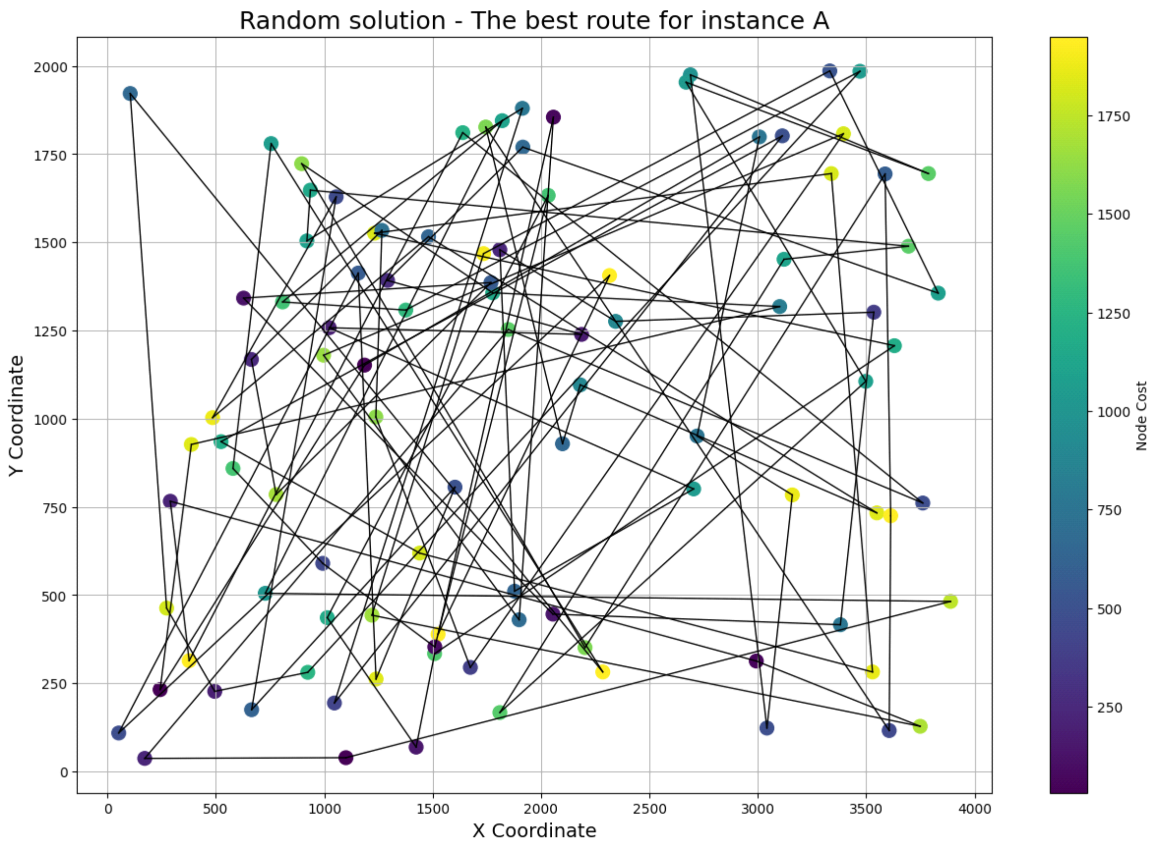
	Min	Max	Mean
Instance A	84,471	95,013	87,679.14
Instance B	77,448	82,631	79,282.58
Instance C	56,304	63,697	58,872.68
Instance D	50,335	59,846	54,290.68

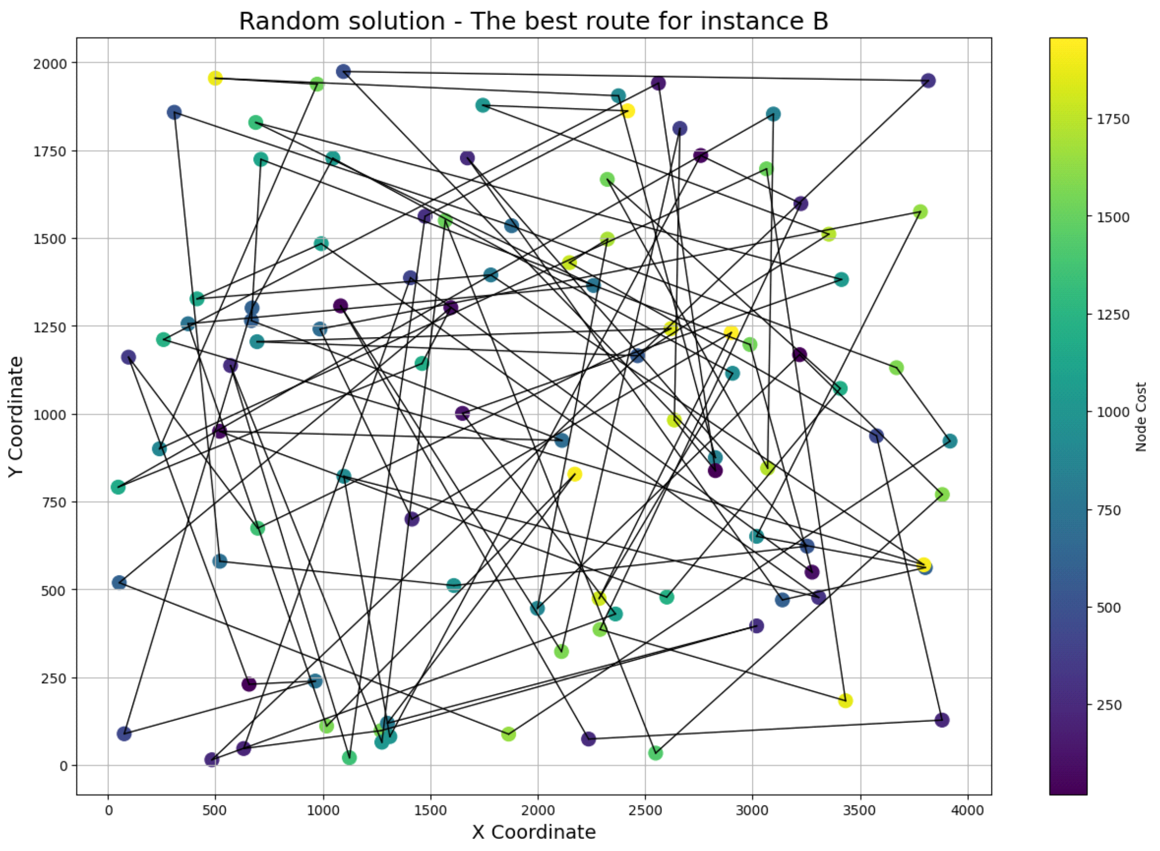
The greedy cycle algorithm

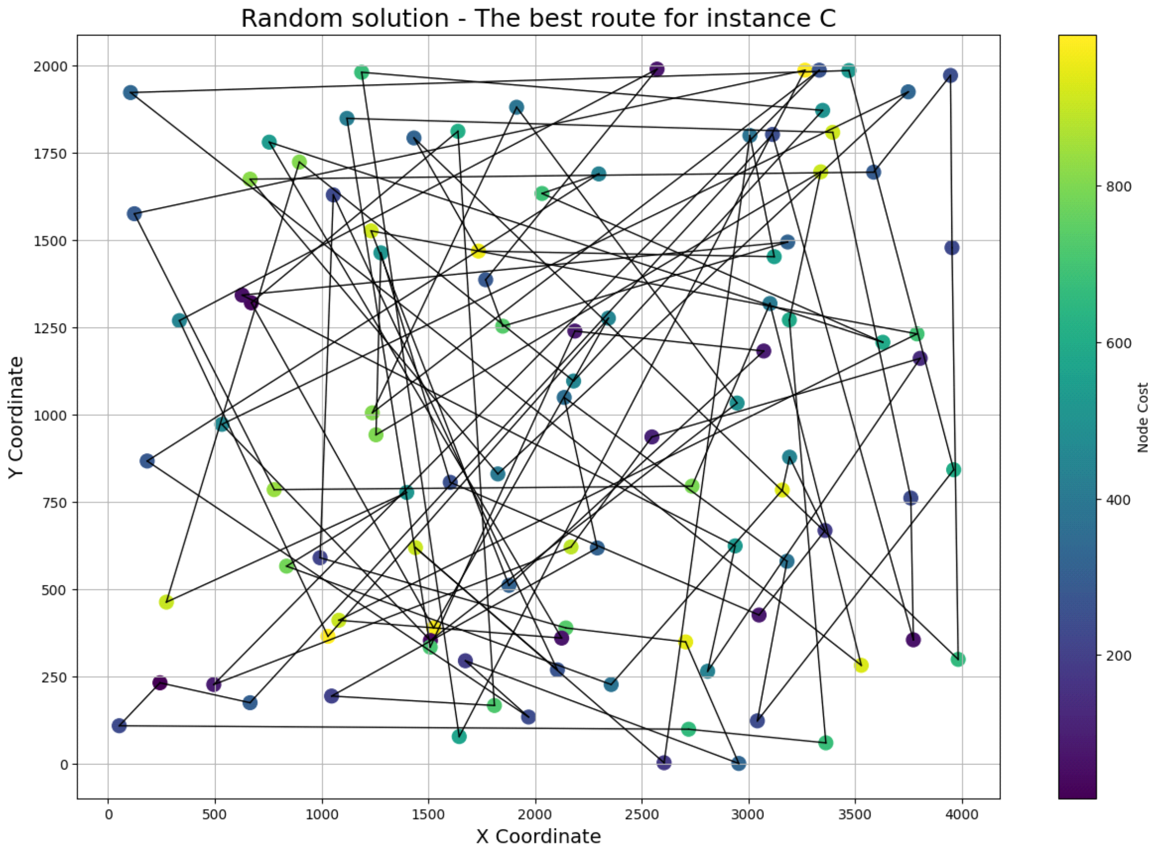
	Min	Max	Mean
Instance A	75,136	80,025	76,711.19
Instance B	67,896	76,096	70,464.27
Instance C	53,020	58,499	55,859.31
Instance D	50,288	60,208	54,931.05

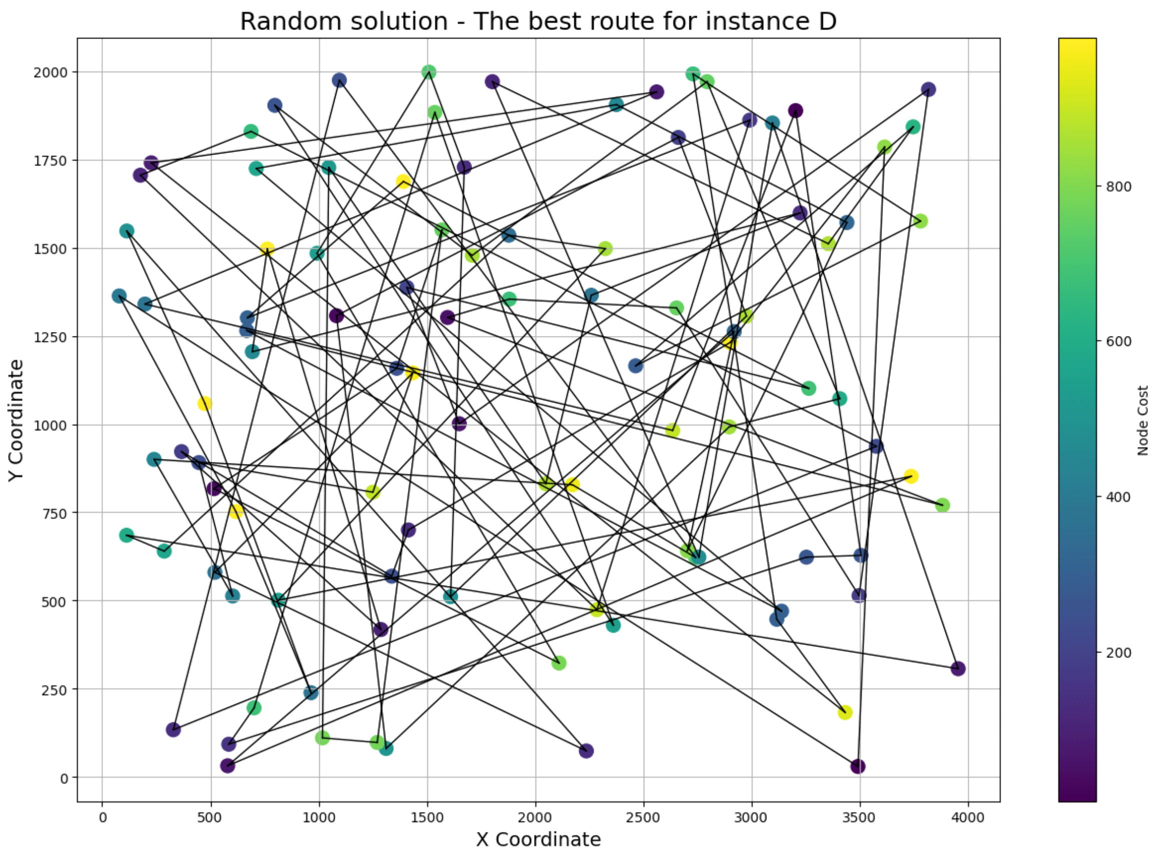
Visualizations

The random solution

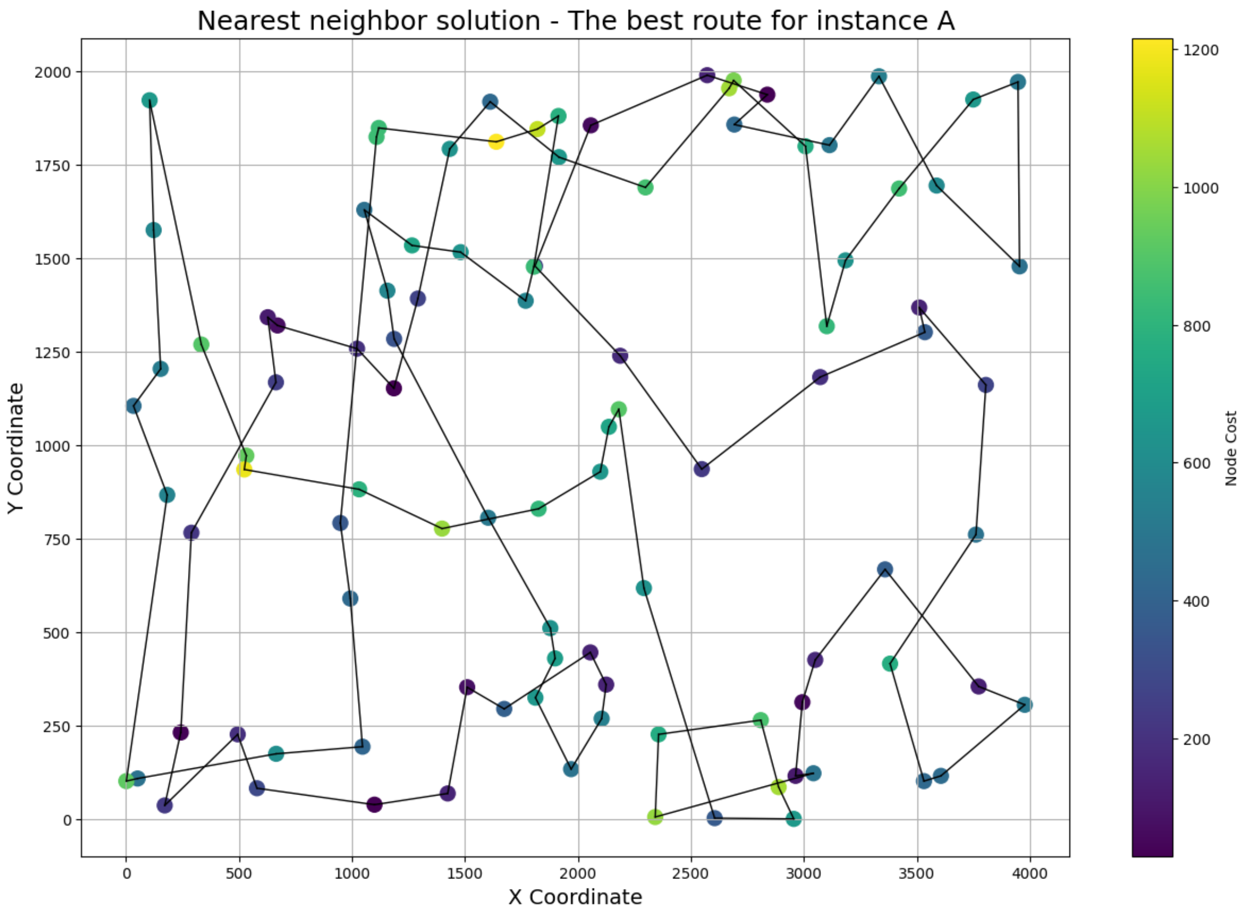


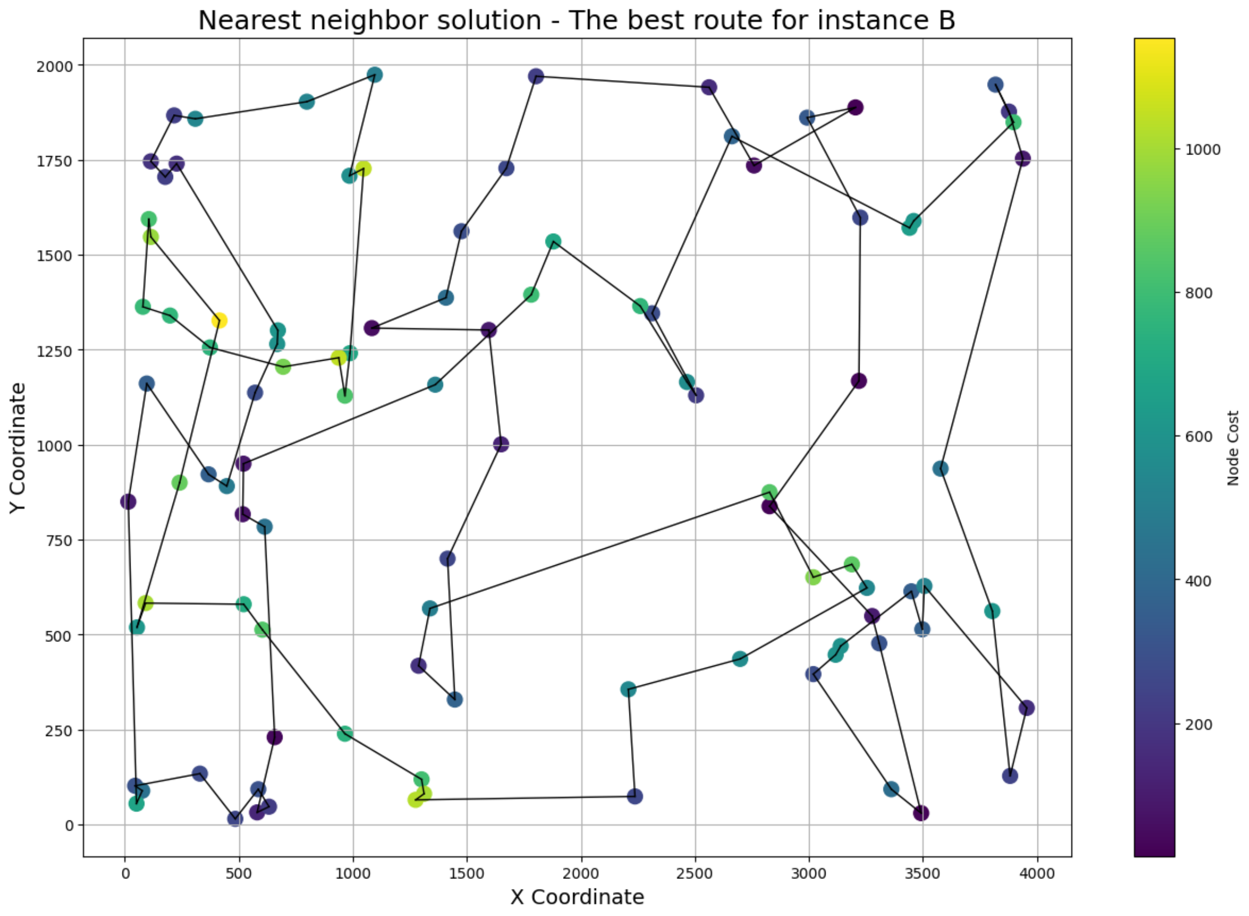

Random solution - The best route for instance B
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
250
500
750
1000
1250
1500
1750

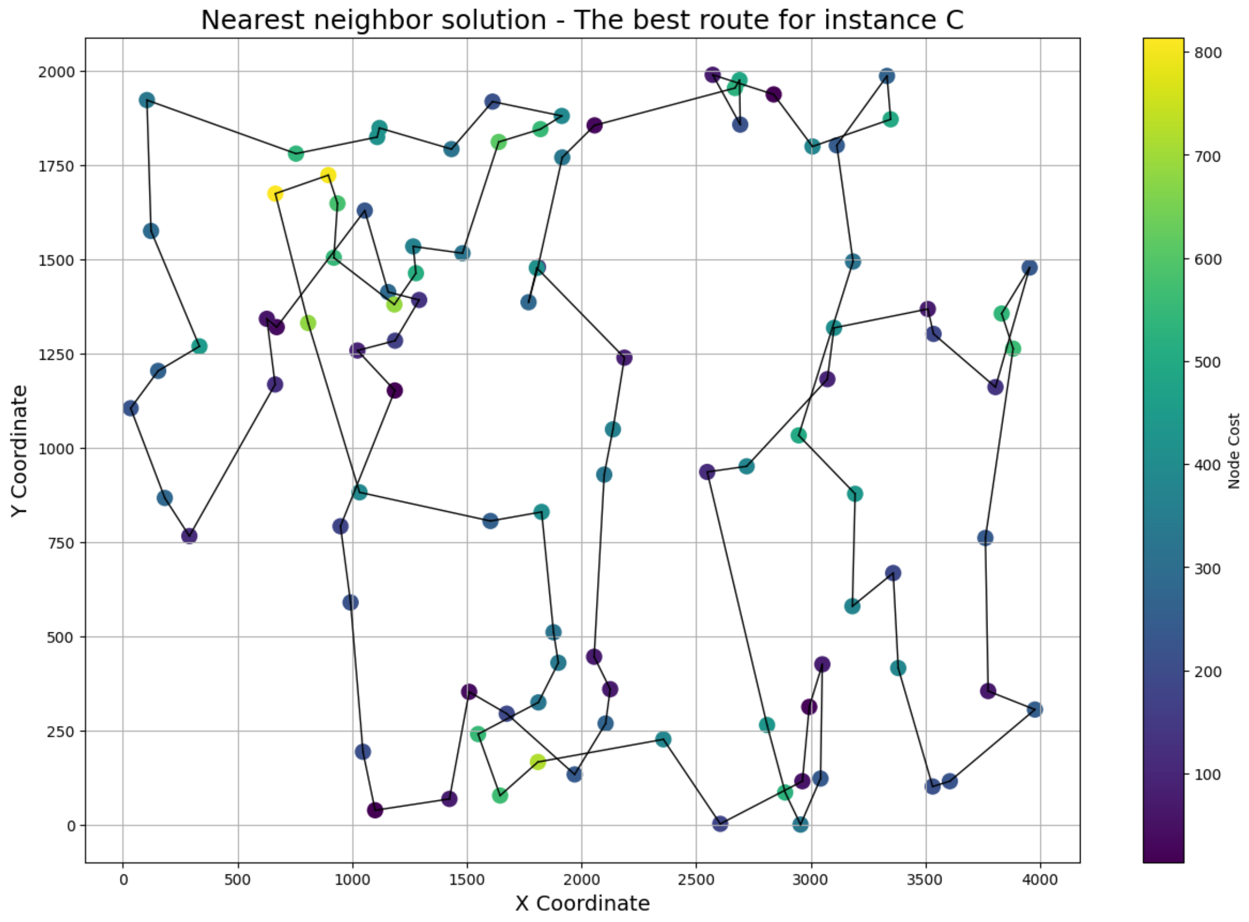

Random solution - The best route for instance C
Y Coordinate
X Coordinate
Node Cost
0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500
3000
3500
4000
200
400
600
800

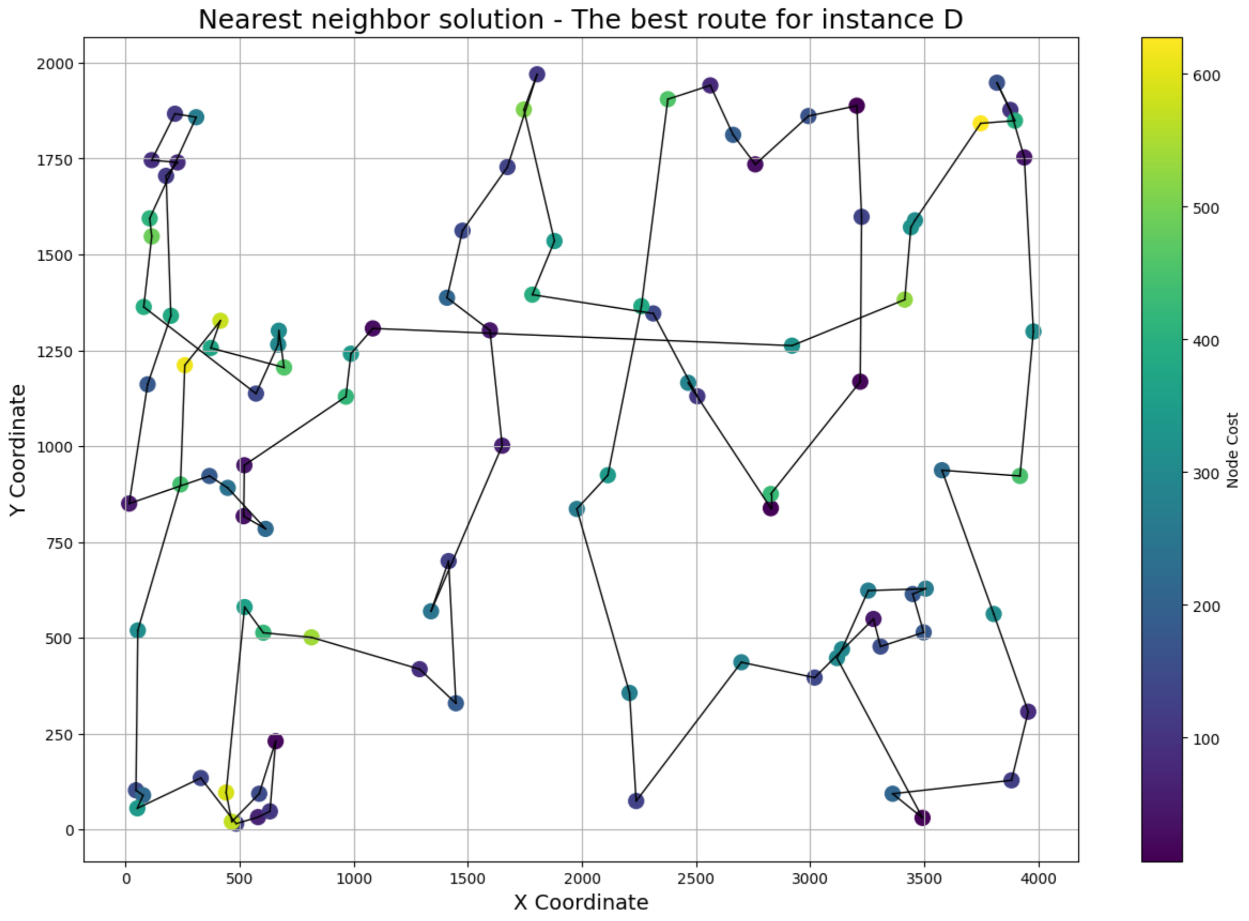

Random solution - The best route for instance D
Y Coordinate
X Coordinate
Node Cost
0
500
1000
1500
2000
2500
3000
3500
4000
250
750
1250
1750
200
400
600
800

The nearest neighbor algorithm

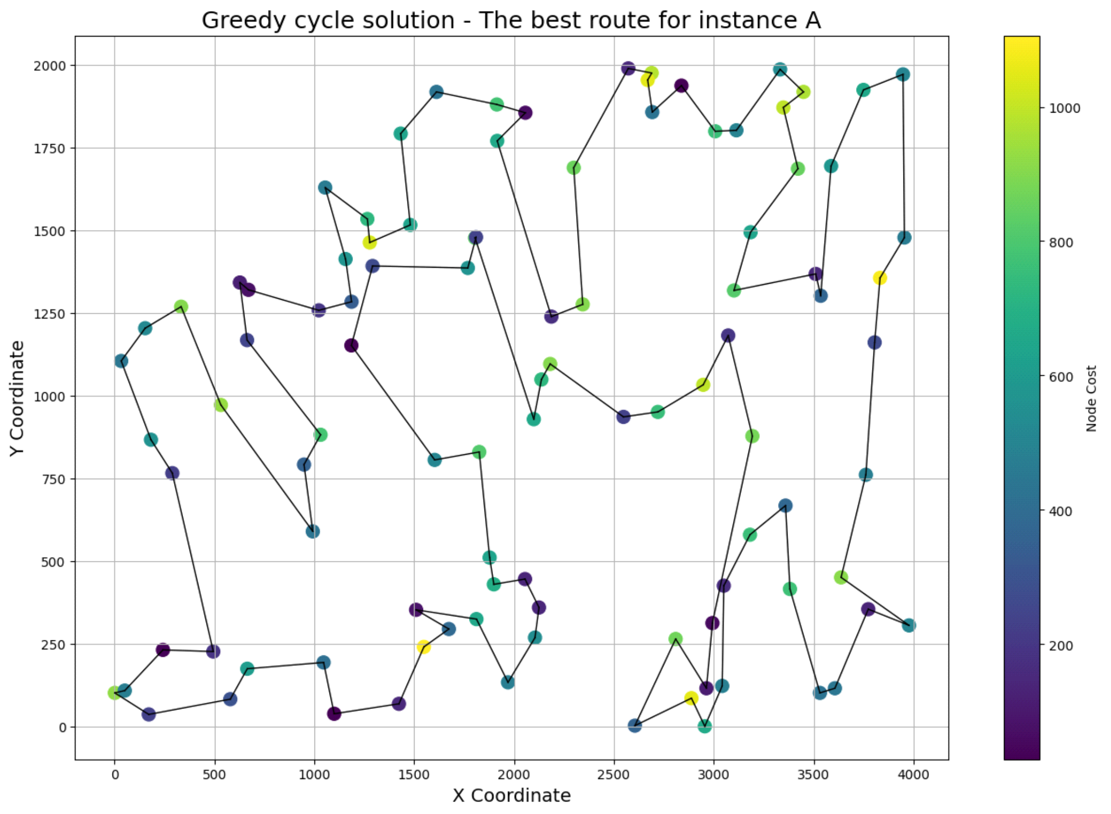


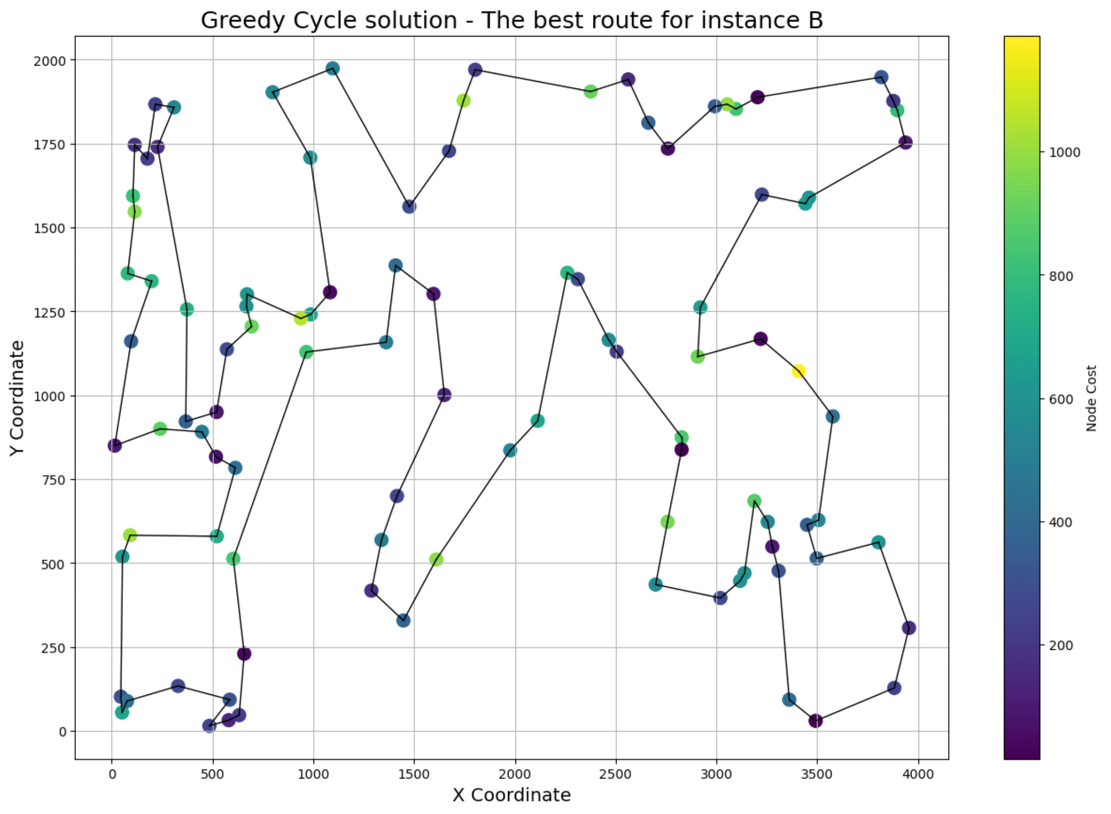



Nearest neighbor solution - The best route for instance C
Y Coordinate
X Coordinate
Node Cost

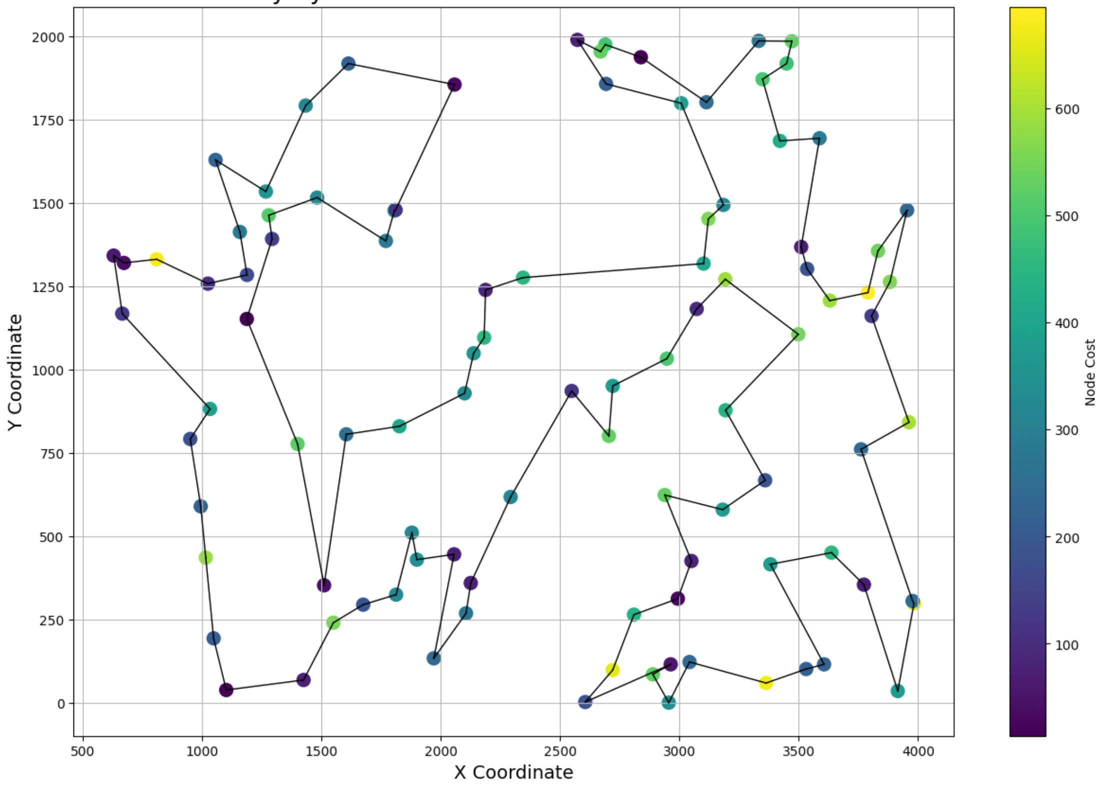

Nearest neighbor solution - The best route for instance D
Y Coordinate
X Coordinate
Node Cost

The greedy cycle algorithm

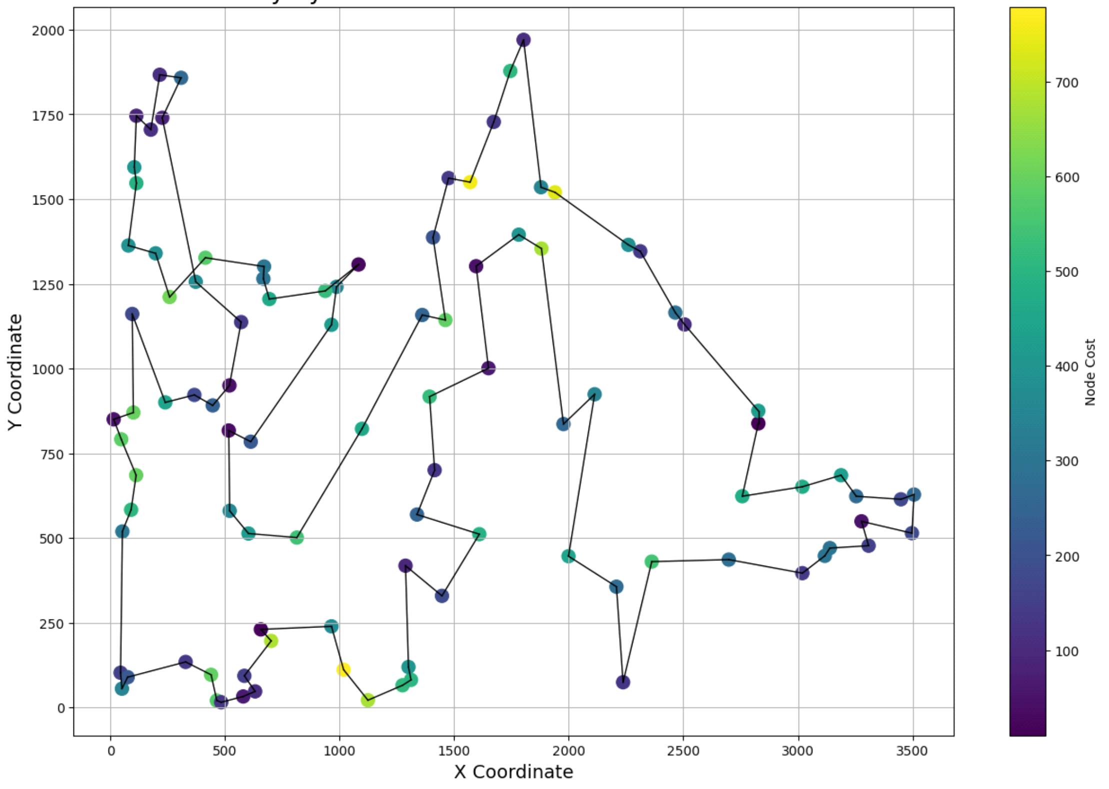




Greedy Cycle solution - The best route for instance C



Greedy Cycle solution - The best route for instance D



Conclusions

The three approaches we have experimented with are not so sofisticated, pretty simple in concept. The first one - random, is far from the optimum in the average case because the probability that it outputs the optimal solution is n!(where n is the number of nodes taken under consideration), but it's very fast and can be used to visualize in what order of magnitude the worse part of optimization landscape is expressed.

The second algorithm the Nearest Neighbor is deterministic in opposite to the first one, produces far better solutions, and can be treated as a baseline for future improvements. It selects the nearest neighbor(based on the distance in our case Euclidean) of the last selected node and adds it into the path. In plain sight we can see the weaknesses of this approach in the visualisations, the last step in which we connect the starting point with the distant last selected node offen crosses other edges and is usually far from optimum.

Here comes the third algorithm, Greedy Cycle, which has a solution to this problem because it optimizes the whole cycle length rather than the last connection as the nearest neighbor does. When the decision comes on which node to add to the Hamiltonian cycle it iterates over all of the possible positions in which we can insert the new node and calculates the increase of cycle length for every candidate from the unselected nodes. This algorithm has the highest complexity but solves the problem of the last step which was the issue of the NN. It is worth noticing, that this algorithm is the only one, that in most cases produces solutions without crossing paths in the graph. Only for instance D, paths for this algorithm crossed once. Going further, based on the provided visualizations, it’s easy to see, that paths produced by this method are the shortest among all solutions provided by implemented methods.