POZNAN UNIVERSITY OF TECHNOLOGY

Assignment 1: Greedy Heuristics

Mateusz Tabaszewski 151945

Bartłomiej Pukacki 151942

Source code:
https://github.com/MatTheTab/Evolutionary-Computation/blob/main/Assignment%201%20Greedy%20heuristics/Assignment_1_Greedy_Heuristics.ipynb

1. Description of the Problem

The report describes steps taken to solve the tasks required for assignment 1, which is based on testing different heuristic approaches for solving a slightly redefined Traveling Salesperson Problem (TSP). TSP is a combinatorial optimization problem in which, given a set of nodes with certain coordinates, we seek to find a cycle visiting all the nodes with the smallest possible cost and starting and ending in a chosen starting city, with the cost being associated with the total length of the cycle. It is an NP-hard problem. However, the problem described in this assignment is a slight redefinition of the original TSP, as in this version the cost is defined as the sum of the total length of the cycle and the sum of the weights of all the nodes included in the cycle. The cost should be minimized. Furthermore, in this version of the problem, only 50% of all nodes need to be visited in total. This redefined version of the problem in a formal notation can be seen here:

Decision Variables:

$x_{ij} \in \{0,\ 1\}$ - included edges

$y_i \in \{0,\ 1\}$ - visited nodes

Objective Function:

$$min(\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} + \sum_{i=1}^{n} w_i y_i)$$

sb. t.

$$\sum_{j \in V} x_{ij} = 2y_i \ , \forall i \in V \text{ ; } \textit{where V is a set of all vertices}$$

$$\sum_{i=1}^{n} y_i \geq \frac{n}{2}$$

$$x_{ij} \in \{0,\ 1\}, \forall i, j \in V$$

$$y_i \in \{0,\ 1\},\ \forall i \in V$$

Furthermore, two instances of the problem - TSPA.csv and TSPB.csv - were given to test the algorithms' performance and visualize the results. All algorithms described in the further sections were tested on both algorithms. Figures 1 and 2 show the nodes' x and y locations from the TSPA and TSPB instances.

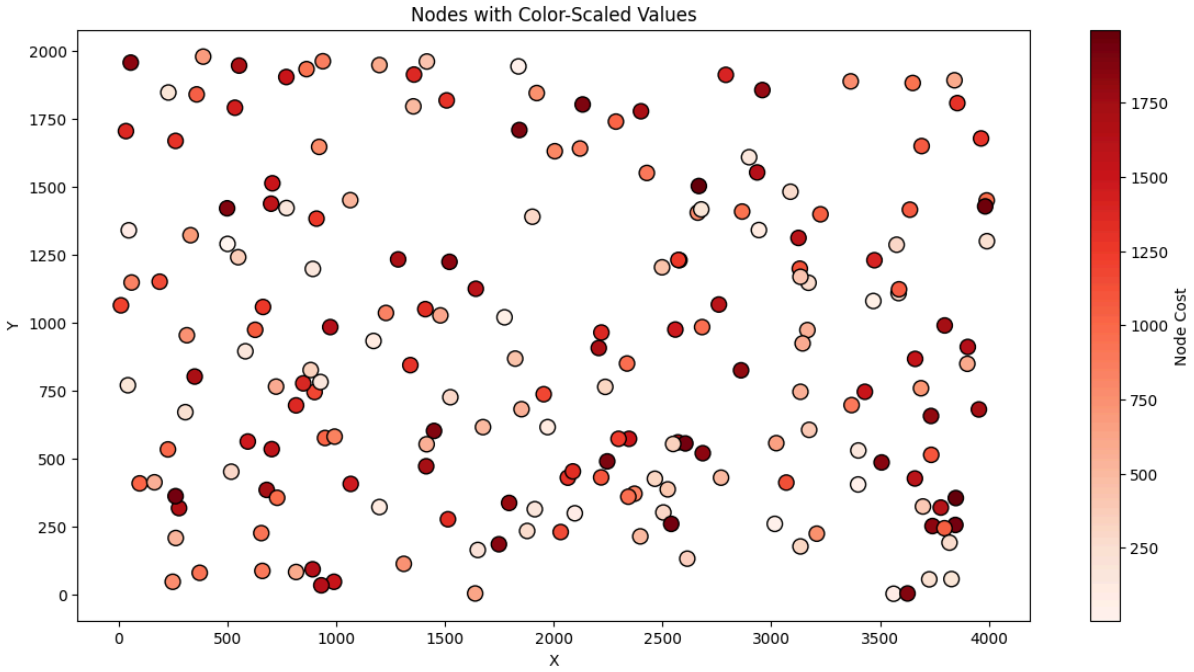

Fig 1. Visualization of the TSPA problem instance, each node's x and y locations on the plot correspond to their given x and y locations and the color intensity signifies the weight/cost of each node. The total length of the cycle and the sum of node weights should be minimized.

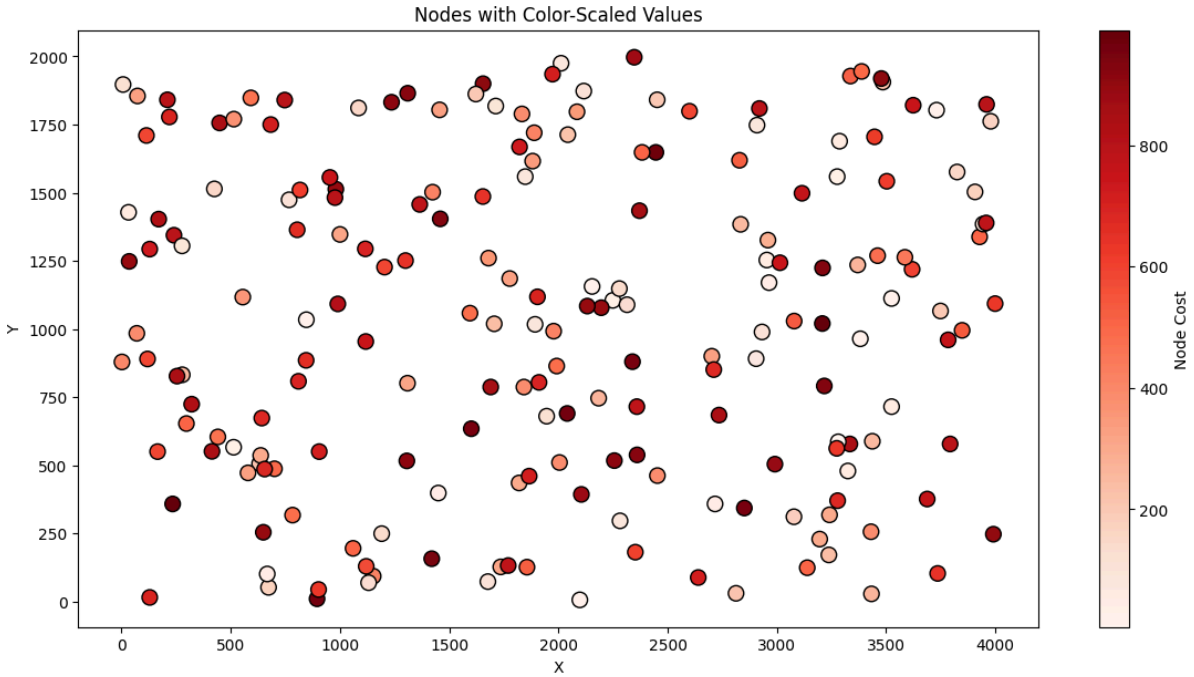

Fig 2. Visualization of the TSPB problem instance, each node's x and y locations on the plot correspond to their given x and y locations and the color intensity signifies the weight/cost of each node. The total length of the cycle and the sum of node weights should be minimized.

2. Algorithms

The assignment included the use of the following algorithms: random, nearest neighbor considering adding the node only at the end of the current path, nearest neighbor considering adding the node at all possible positions, and the greedy cycle. All of the following algorithms will be described in text and using pseudocode. For the sake of clarity, a pseudocode of a function for calculating the score has also been added:

```
FUNCTION calculate_score(solution, matrix, weights):
    INPUT:
        solution - list or array representing the sequence of nodes in
the solution
        matrix - 2D matrix of distances between nodes
        weights - a 1D array of weights associated with each node

    score ← 0

    FOR i FROM 0 TO (length of solution) - 2:
        node_1 ← solution[i]
        node_2 ← solution[i + 1]
        score ← score + (matrix[node_1][node_2] + weights[node_1])

    node_1 ← last node in solution
    node_2 ← first node in solution
    score ← score + (matrix[node_1][node_2] + weights[node_1])

    RETURN score
```

The Random solution is an algorithm that, after being given the starting node from which the search should begin, picks the next one at random. After all the required nodes have been selected (half of all given nodes for our redefined TSP), the algorithm adds the first node as the last one to complete the cycle and terminates. The complexity of the greedy algorithm can be denoted as *O(n)*.

```
FUNCTION random_solution(distance_matrix, weights, start_node):
    INPUT:
        distance_matrix - matrix of distances between nodes
        weights - an array of weights associated with each node
        start_node - starting node for the cycle

    remaining_nodes ← list of all nodes except start_node
    SHUFFLE remaining_nodes randomly
    solution ← [start_node] + ((number of all nodes)/2 - 1) nodes from
remaining_nodes

    score ← calculate_score(solution, distance_matrix, weights)

    RETURN solution, score
```

The nearest neighbor considering adding the node only at the end of the current path, which we will from now on refer to as **The Nearest Neighbor Closest** is an algorithm in which we select each subsequent node after the first one by finding the nearest neighbor from the last added one. So let us assume we start with node_1 and then the closest node to it is node_2, so we will add it to the cycle, then we will try to find the node closest to node_2, considering all nodes other than those already in the cycle (node_1, node_2), this will continue until we reach the required number of nodes. In this case, we are not just adding the closest node in terms of the coordinates, but we are looking for the nodes with the smallest distance and smallest weight to the last added node, this is because our objective function takes both into account. The complexity of this algorithm is $O(n^2)$.

```
FUNCTION nearest_neighbor_end(distance_matrix, weights, start_node):
    INPUT:
        distance_matrix - matrix of distances between nodes
        weights - an array of weights associated with each node
        start_node - starting node for the cycle

    FILL (diagonal of distance_matrix) with INFINITY #prevents self-loops
    solution ← [zero FOR zero in RANGE((number of all nodes)//2)]
    solution[0] ← start_node
    next_node ← start_node
    i ← 1

    WHILE i < ((number of all nodes)//2):
        next_node ← FIND node with minimum value in column next_node of
distance matrix PLUS weight of this node
        FILL (next_node's row in distance_matrix) with INFINITY #prevents
reselection
        solutions[i] ← next_node
        i += 1

    score ← calculate_score(solution, distance_matrix, weights)

    RETURN solution, score
```

The nearest neighbor considering adding the node at all possible positions, from now on referred to as **The Nearest Neighbor All** is an algorithm in which we select each subsequent node after the first one by considering the closest not yet selected node from all selected nodes, not just the already selected ones. In other words, if the selected nodes include node_1 and node_2 then we will have to look for the closest node overall when comparing the distance to node_1 and node_2. The definition of distance here also includes the node's weight as it is something taken into account in the objective function. The complexity of this algorithm is $O(n^3)$.

```
FUNCTION nearest_neighbor_all(distance_matrix, weights, start_node):
    INPUT:
        distance_matrix - matrix of distances between nodes
        weights - an array of weights associated with each node
        start_node - starting node for the cycle

    FILL (diagonal of distance_matrix) with INFINITY #prevents self-loops
    FILL (start_node's row in distance_matrix) with INFINITY #prevents
reselection
    solution ← [start_node]

    i ← 1
    WHILE i < ((number of all nodes)//2):
        j ← 0
        best_found ← INFINITY #Assume we will find something better than
INFINITY
        insert_node_idx ← NONE
        insert_node_nearest  ← NONE
        WHILE j < i:
            tested_node = solution[j]
            nearest node ← FIND node with minimum value in column
tested_node of distance matrix PLUS the weight of this node
            min_cost ← distance from tested_node to nearest_node +
weight of nearest_node
            IF min_cost < best_found:
                best_found ← min_cost
                insert_node_idx ← j
                insert_node_nearest ←  nearest_node
            j += 1
        FILL (insert_node_nearest's row in distance_matrix) with INFINITY
#prevents reselection
        INSERT insert_node_nearest at position insert_node_idx in
solution
        i+= 1

    score ← calculate_score(solution, distance_matrix, weights)

    RETURN solution, score
```

The last of the tested algorithms was the **Greedy Cycle Algorithm**, this method constructs a closed cycle in every step choosing a node that increases the value of the objective function the least while still maintaining the closed cycle. To be more specific, the algorithm first finds a node that is the closest to the starting one, then a third node is selected which is the closest to both the first and the second node. The cycle is also closed, so the total cost calculated at this step is the length of the cycle (from the first node to the second, to the third,

and back to the first one) and the sum of the weights of all chosen nodes. Then the algorithm considers all possible places where the current cycle can be severed and a new node can be added, before closing the cycle. In other words, the algorithm seeks to find a new cycle by adding the node that increases the objective function the least. It is also important to mention that when considering the “closest” nodes in this case, much like in all previous algorithms, the algorithm seeks to find the node that is the closest in terms of the Euclidean distance while also having the smallest weight. The previous references to the “closest node” always referred to combining these two factors. The complexity of the Greedy Cycle Algorithm is $O(n^3)$.

```
FUNCTION greedy_cycle(matrix, weights, start_node):
    INPUT:
        matrix - distance matrix between nodes
        distance_matrix - matrix of distances between nodes
        weights - an array of weights associated with each node
        start_node - starting node for the cycle

    distance_matrix ← matrix
    FILL (diagonal of distance_matrix) with INFINITY #prevents self-loops
    FILL (start_node's row in distance_matrix) with INFINITY #prevents
reselection
    second_node ← FIND node with minimum value in column start_node of
distance matrix PLUS the weight of this node
    FILL (second_node's row in distance_matrix) with INFINITY #prevents
reselection
    third_node ← FIND node with minimum combined distance (starting_node
+ second_node) + weight
    FILL (third_node's row in distance_matrix) with INFINITY #prevents
reselection
    solution ← [start_node, second_node, third_node]
    score ← calculate_score(solution, distance_matrix, weights)

    num_iterations ← ((number of all the nodes//2) - 3)
    FOR i FROM 0 to num_iterations  - 1:
        best_score ← INFINITY
        FOR each node position in solution:
            new_score, new_solution, new_node ←
find_shortest_cycle(current_score, distance_matrix, matrix, weights,
solution, position)
            IF new_score < best_score:
                Update best_score, best_solution, and best_node
        Update solution and score with best solution and score
        FILL (best_node’s row in distance_matrix) with INFINITY
#prevents reselection

    RETURN solution, score
```

```
FUNCTION find_shortest_cycle(score, distance_matrix, matrix, weights,
solution, idx):
    INPUT:
        score - current total score
        distance_matrix - matrix of distances between nodes, modified
with infinities to prevent reselections and self-loops
        matrix - original distance matrix
        weights - an array of weights associated with each node
        solution - current node sequence
        idx - index of the new node to be inserted

    node_1 ← solution[idx -1]
    node_2 ← solution[idx]
    score ← SUBTRACT distance between node_1 and node_2

    new_node ← node with minimum summed distance to (node_1, node_2) +
weight
    INSERT new_node at position idx in solution
    score += distance between (node_1, new_node) + distance between
(new_node, node_2) + new_node's weight

    RETURN score, solution, new_node
```

3. Experiments

To quantify the performance of greedy methods, each algorithm was run 200 times on each of the 200 nodes as starting points on both instances available and the solutions and scores generated were collected. Additionally, the random algorithm was run 200 times to compare the results.

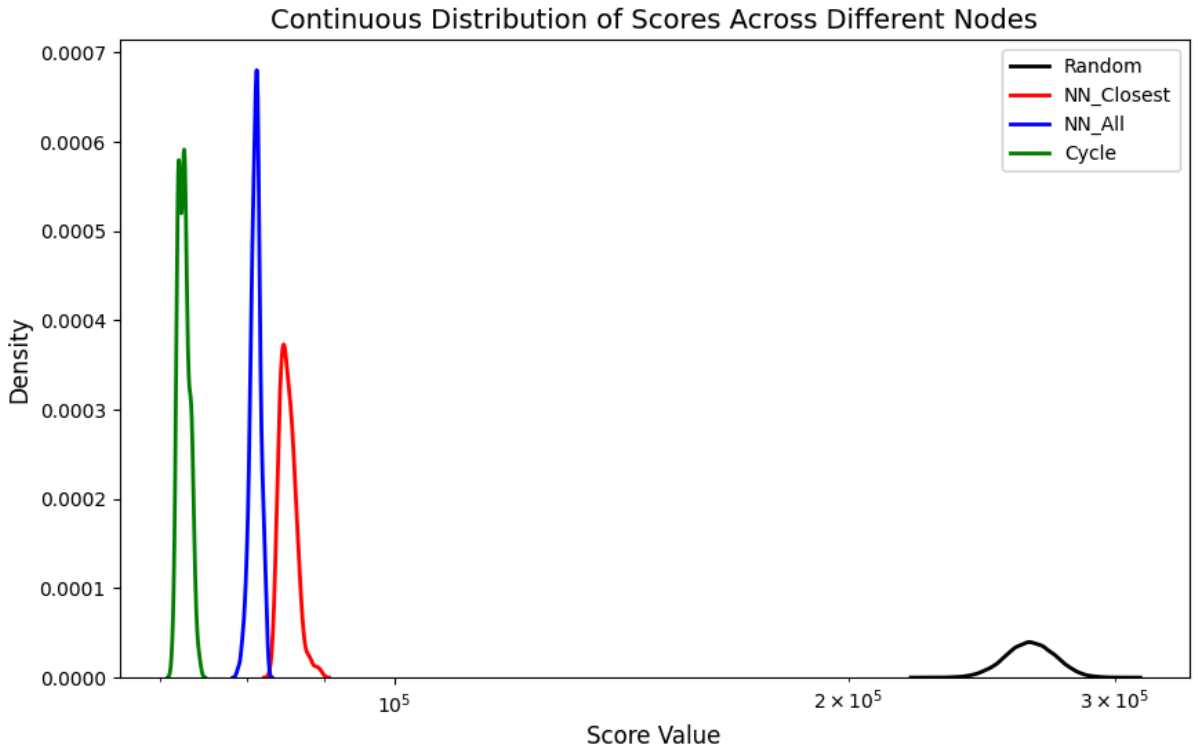

Fig 3. Visualization of the distribution of scores achieved by each algorithm run 200 times on each node as a starting point, on the TSPA problem instance.

The best scores achieved are visualized below.

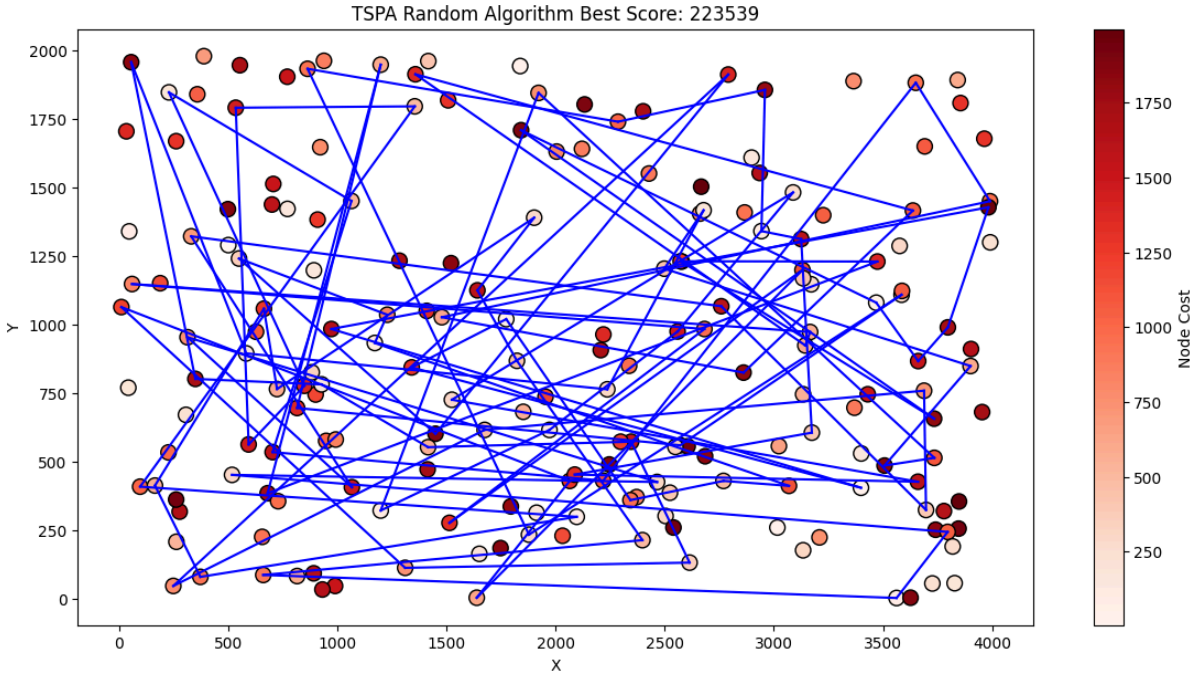

Fig 4. Visualization of the best solution found by the Random algorithm on the TSPA problem instance.

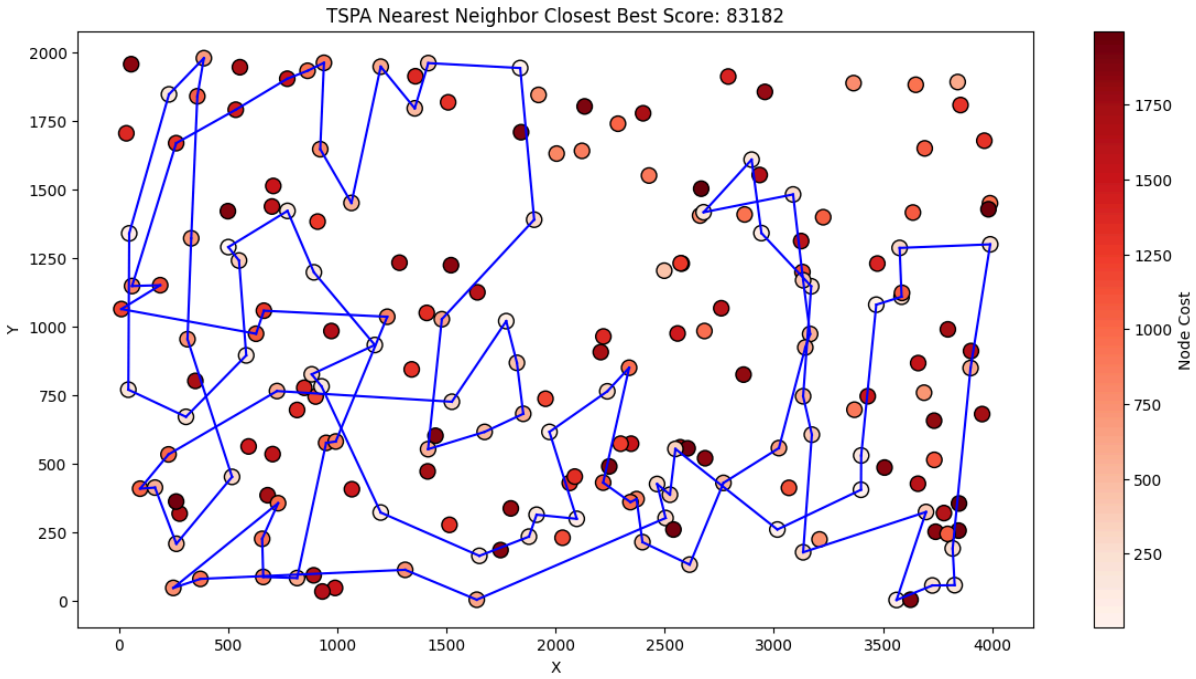

Fig 5. Visualization of the best solution found by the Nearest Neighbor Closest algorithm on the TSPA problem instance. Starting node: 124.

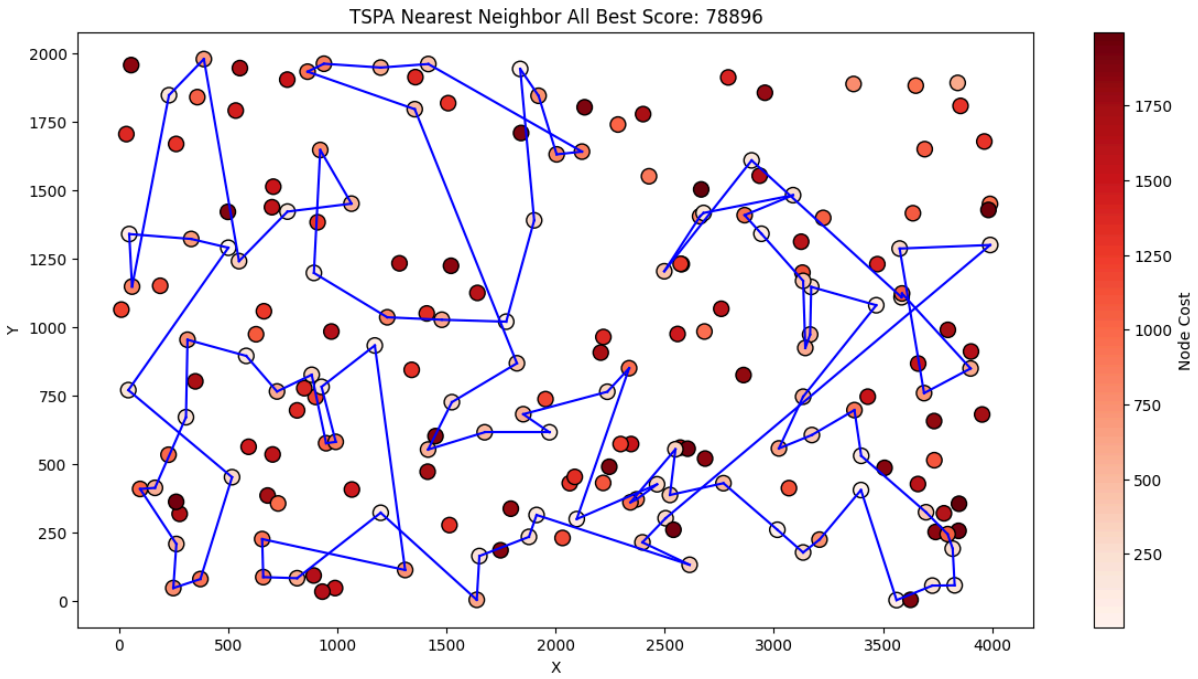

Fig 6. Visualization of the best solution found by the Nearest Neighbor All algorithm on the TSPA problem instance. Starting node: 118.

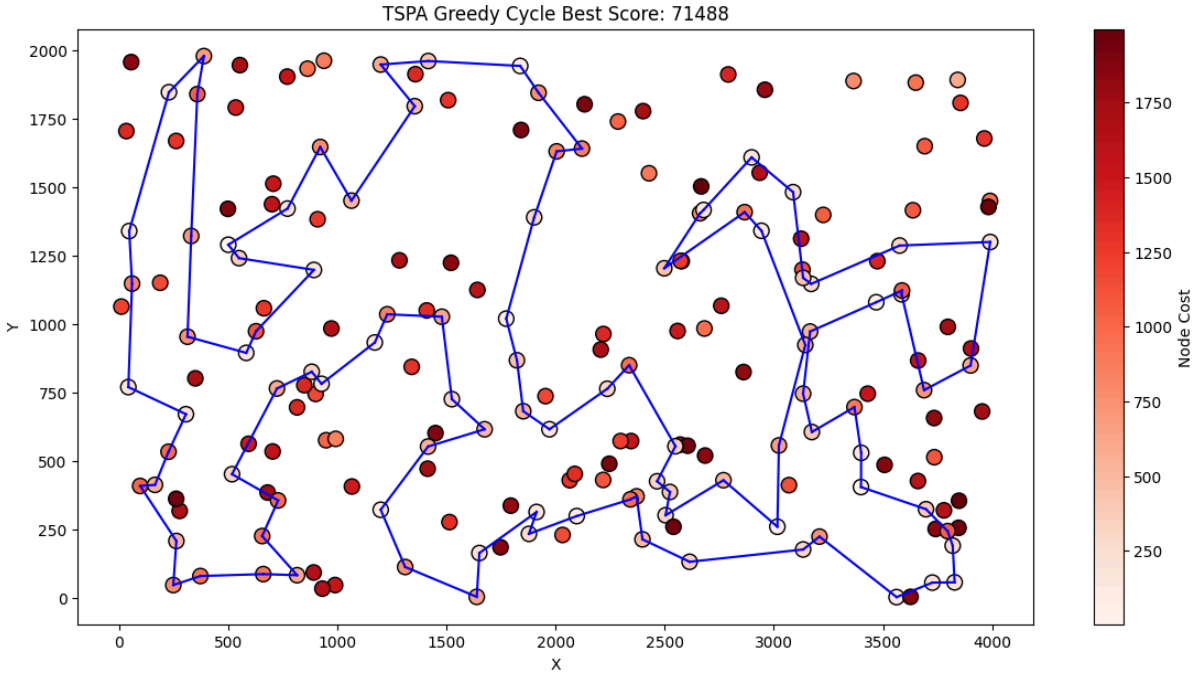

Fig 7. Visualization of the best solution found by the Greedy Cycle algorithm on the TSPA problem instance. Starting node: 0.

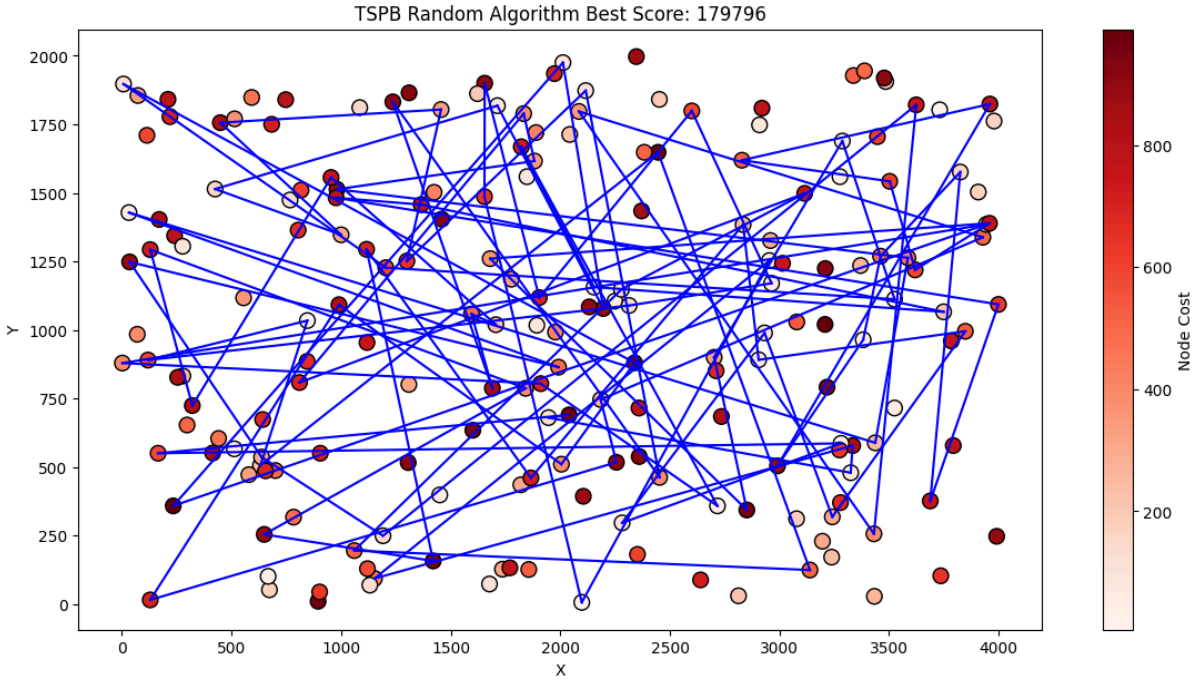

Fig 8. Visualization of the best solution found by the Random algorithm on the TSPB problem instance.

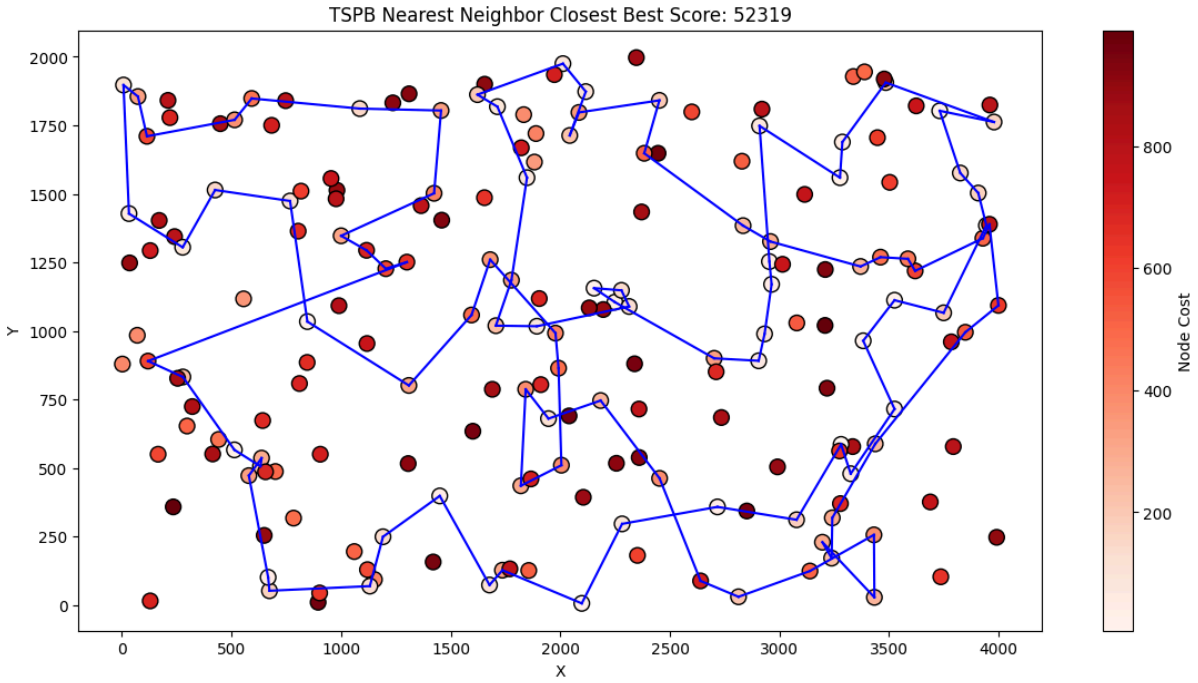

Fig 9. Visualization of the best solution found by the Nearest Neighbor Closest algorithm on the TSPB problem instance. Starting node: 16.

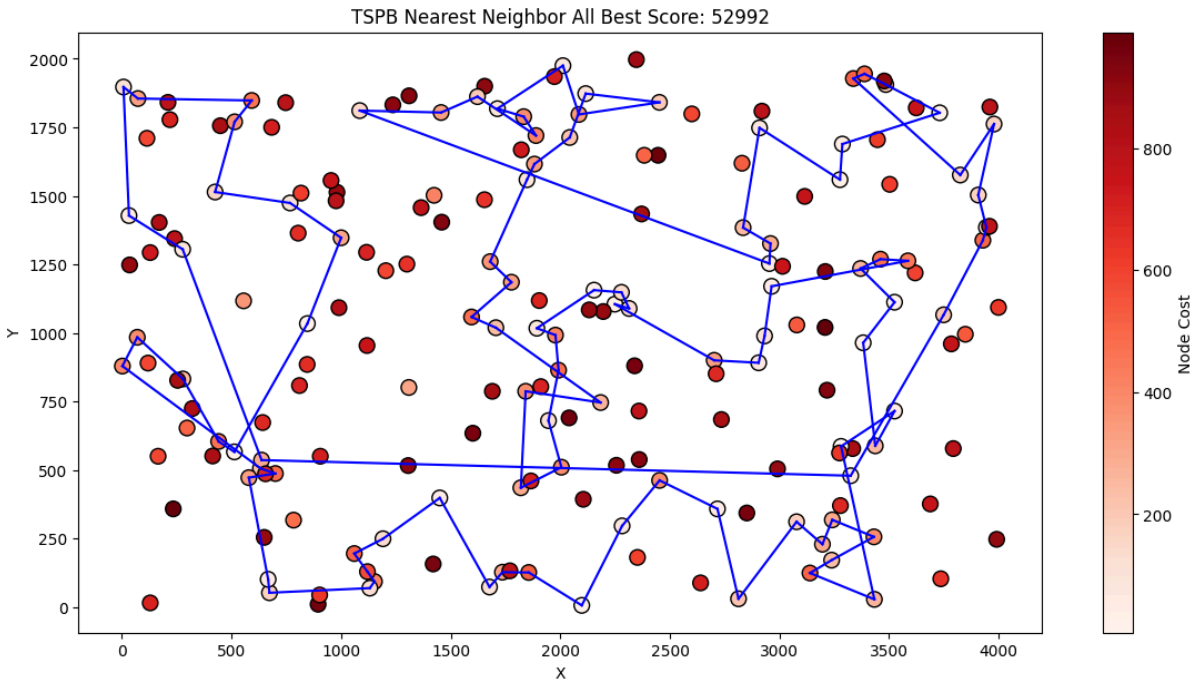

Fig 10. Visualization of the best solution found by the Nearest Neighbor All algorithm on the TSPB problem instance. Starting node: 10.

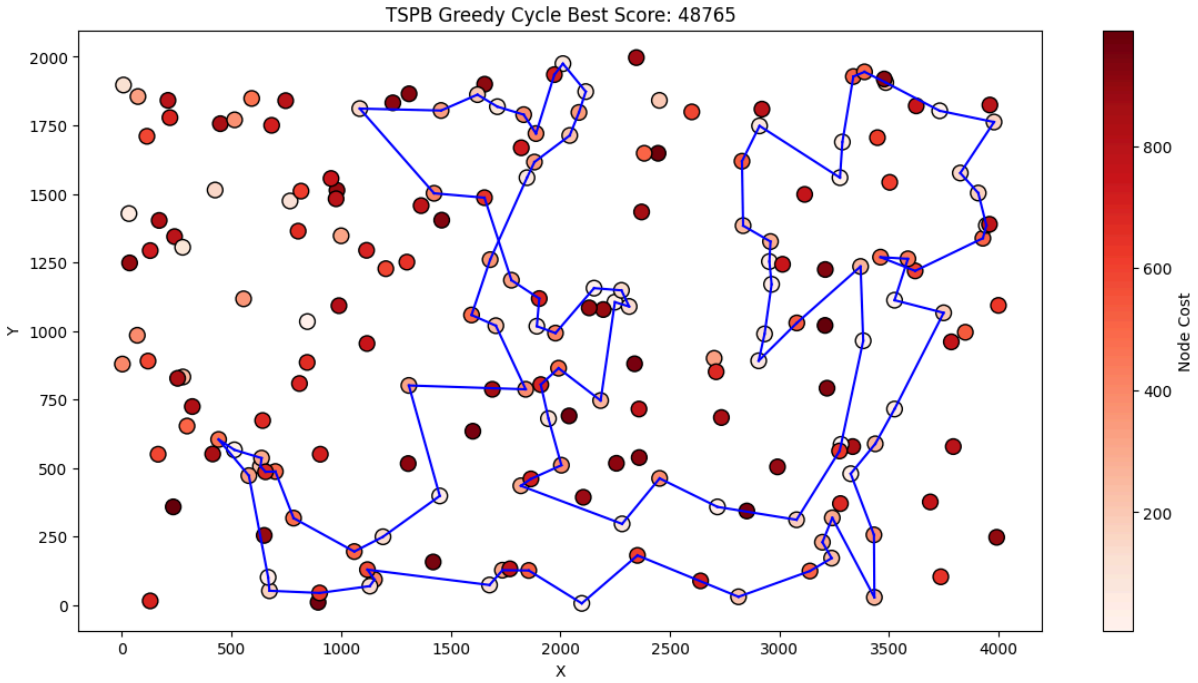

Fig 11. Visualization of the best solution found by the Greedy Cycle algorithm on the TSPB problem instance. Starting node: 136.

All best solutions were checked using the solution checker spreadsheet available on eKursy. The lists of node indices in the best solutions and their scores are presented in the table below.

Problem instance	Algorithm	Score	Solution
TSPA	Random	223539	14, 111, 63, 123, 89, 157, 168, 81, 148, 62, 94, 42, 134, 192, 65, 162, 19, 75, 127, 103, 136, 70, 3, 194, 167, 146, 52, 55, 170, 39, 172, 51, 27, 7, 121, 166, 46, 18, 105, 28, 163, 0, 30, 53, 190, 54, 96, 43, 137, 66, 80, 86, 4, 16, 56, 184, 97, 181, 24, 159, 128, 31, 196, 133, 10, 73, 45, 41, 118, 59, 82, 2, 100, 176, 72, 78, 197, 107, 174, 169, 185, 76, 17, 37, 8, 11, 117, 77, 74, 40, 154, 140, 114, 132, 49, 32, 92, 182, 38, 151
	Nearest Neighbor Closest	83182	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
	Nearest Neighbor All	78896	118, 51, 176, 137, 183, 89, 23, 186, 143, 117, 93, 140, 0, 80, 151, 162, 133, 63, 79, 94, 124, 53, 97, 26, 100, 152, 1, 2, 120, 44, 25, 78, 16, 171, 175, 113, 56, 31, 145, 179, 92, 129, 57, 185, 106, 52, 55, 178, 49, 102, 14, 62, 9, 148, 144, 40, 119, 81, 196, 165, 90, 101, 86, 75, 180, 154, 135, 70, 123, 112, 4, 84, 127, 59, 65, 149, 131, 116, 43, 42, 181, 160, 54, 30, 177, 10, 190, 184, 34, 193, 159, 22, 146, 18, 108, 41, 139, 46, 68, 115
	Greedy Cycle	71488	117, 0, 46, 68, 139, 193, 41, 115, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 30, 177, 10, 190, 4, 112, 84, 35, 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
TSPB	Random	179796	78, 18, 141, 43, 65, 49, 184, 62, 35, 16, 121, 31, 167, 165, 45, 109, 174, 19, 132, 195, 67, 99, 194, 63, 144, 92, 54, 5, 59, 114, 15, 66, 111, 50, 108, 116, 82, 37, 40, 118, 185, 140, 143, 186, 139, 154, 22, 9, 170, 23, 129, 86, 130, 148, 76, 57, 120, 85, 179, 29, 126, 153, 56, 27, 94, 196, 70, 12, 169, 122, 51, 44, 6, 74, 3, 81, 192, 157, 182, 138, 71, 24, 102, 104, 105, 7, 98, 87, 34, 106, 172, 103, 124, 77, 176, 42, 68, 8, 113, 88
	Nearest Neighbor Closest	52319	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
	Nearest Neighbor All	52992	10, 133, 122, 90, 51, 121, 117, 198, 1, 38, 27, 31, 73, 193, 190, 80, 175, 78, 142, 45, 5, 177, 36, 61, 91, 141, 77, 81, 153, 187, 163, 89, 103, 114, 127, 165, 137, 176, 166, 194, 86, 185, 95,

			130, 99, 62, 124, 106, 143, 0, 35, 109, 29, 33, 160, 144, 8, 82, 21, 104, 111, 138, 182, 11, 139, 168, 195, 145, 3, 155, 15, 70, 169, 132, 13, 188, 6, 147, 18, 55, 34, 152, 183, 140, 20, 28, 149, 4, 148, 60, 47, 94, 66, 179, 113, 54, 135, 63, 40, 107
	Greedy Cycle	48765	162, 175, 78, 142, 36, 61, 91, 141, 97, 187, 165, 127, 89, 103, 137, 114, 113, 194, 166, 179, 185, 99, 130, 22, 66, 94, 47, 148, 60, 20, 28, 149, 4, 140, 183, 152, 170, 34, 55, 18, 62, 124, 106, 128, 95, 86, 176, 180, 163, 153, 81, 77, 21, 87, 82, 8, 56, 144, 111, 0, 35, 109, 29, 160, 33, 49, 11, 43, 134, 147, 6, 188, 169, 132, 13, 161, 70, 3, 15, 145, 195, 168, 139, 182, 138, 104, 25, 177, 5, 45, 136, 73, 164, 31, 54, 117, 198, 193, 190, 80, 162

Table 1. Best solutions and their scores found by each algorithm in both instances.

For further comparison, we have included the plots of all algorithm's solutions for both test instances when starting from node 0, this way we can compare the results for all algorithms when placed in the same starting location. The results can be seen in Figure 12. - 19.

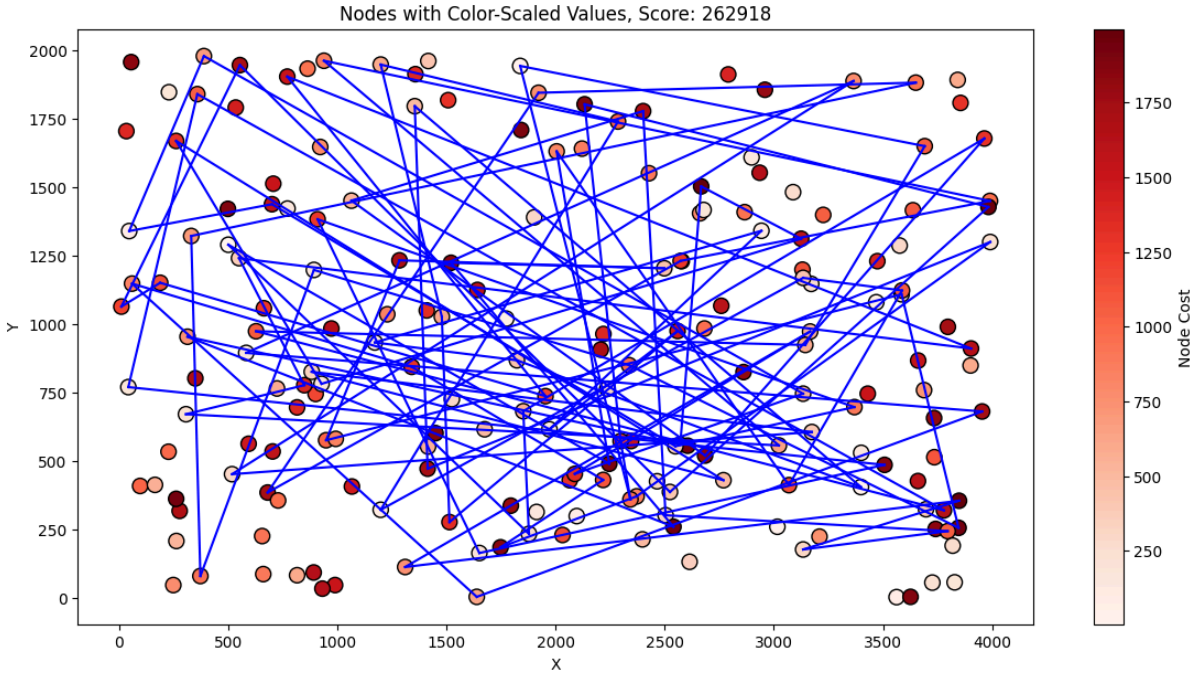

Fig 12. Visualization of the solution found by the Random algorithm on the TSPA problem instance starting from node 0.

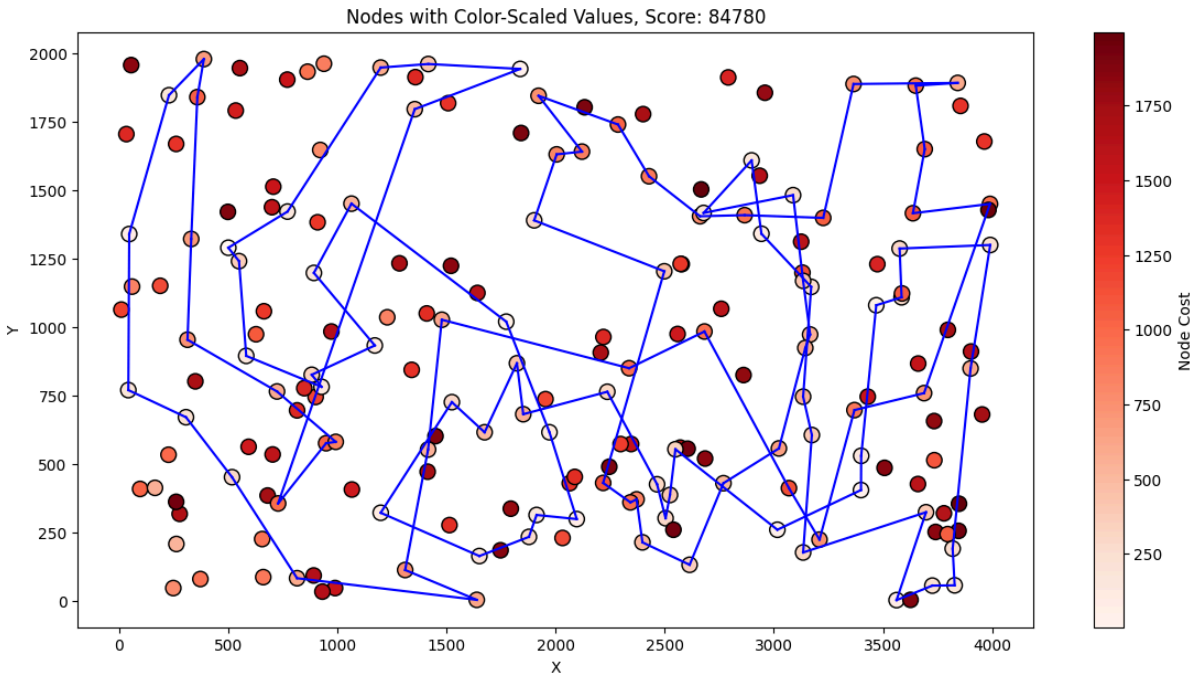

Fig 13. Visualization of the solution found by the Nearest Neighbor Closest algorithm on the TSPA problem instance starting from node 0.

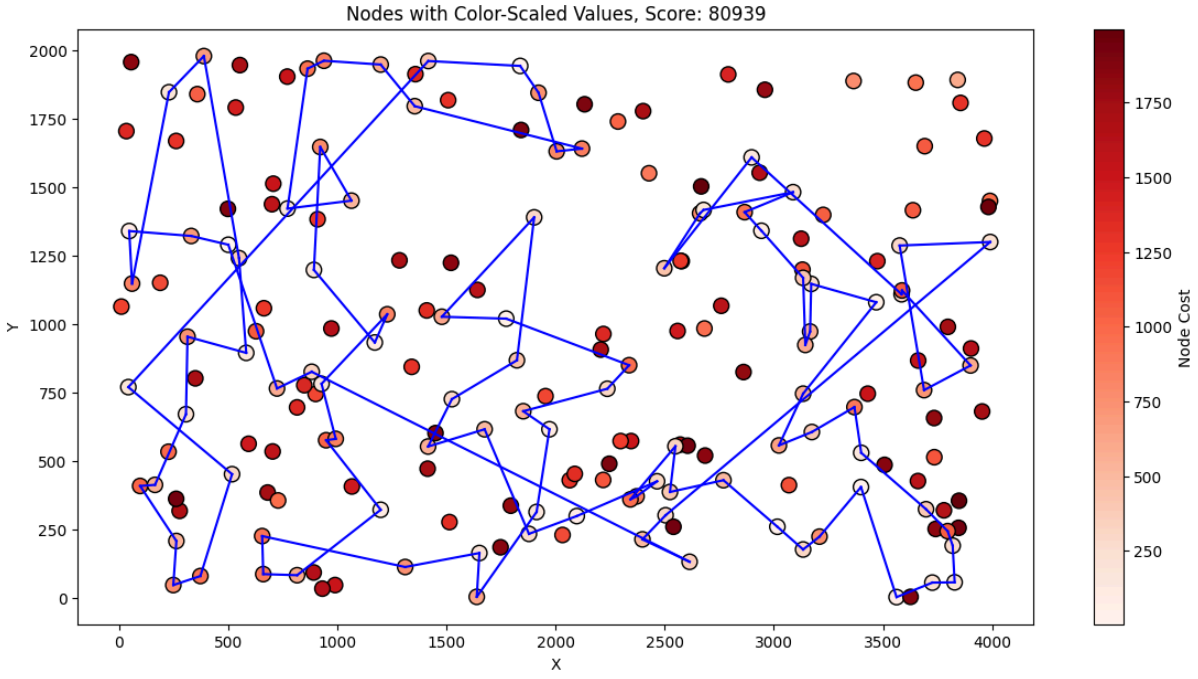

Fig 14. Visualization of the solution found by the Nearest Neighbor All algorithm on the TSPA problem instance starting from node 0.

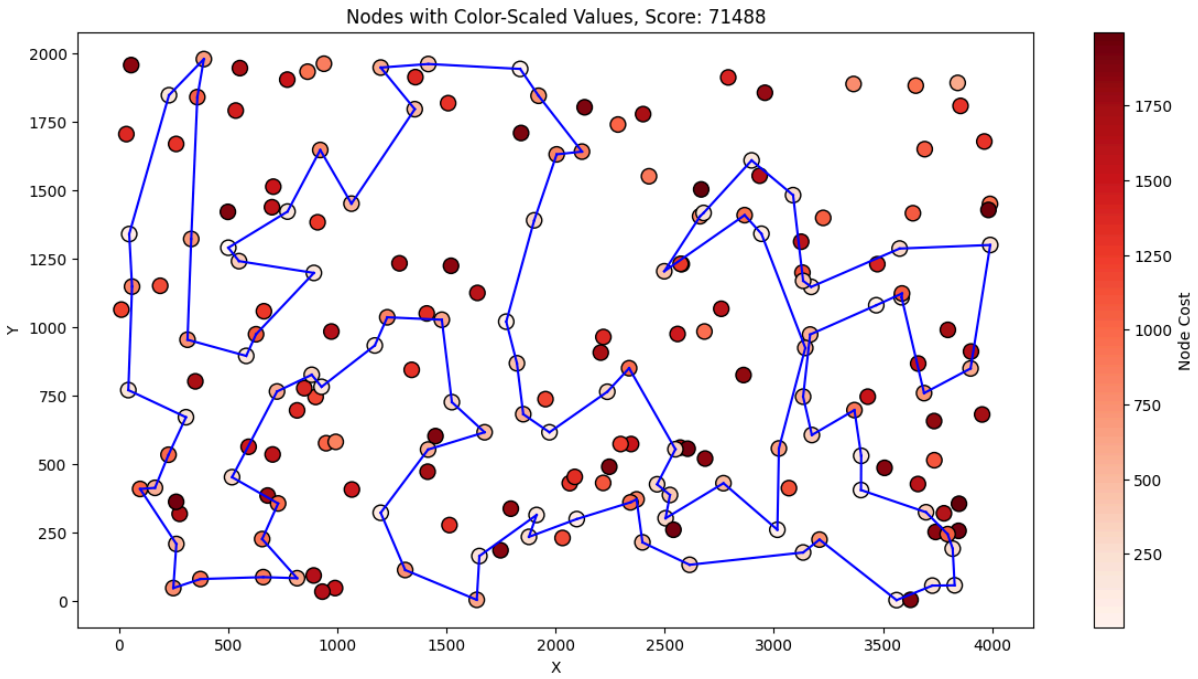

Fig 15. Visualization of the solution found by the Greedy Cycle algorithm on the TSPA problem instance starting from node 0.

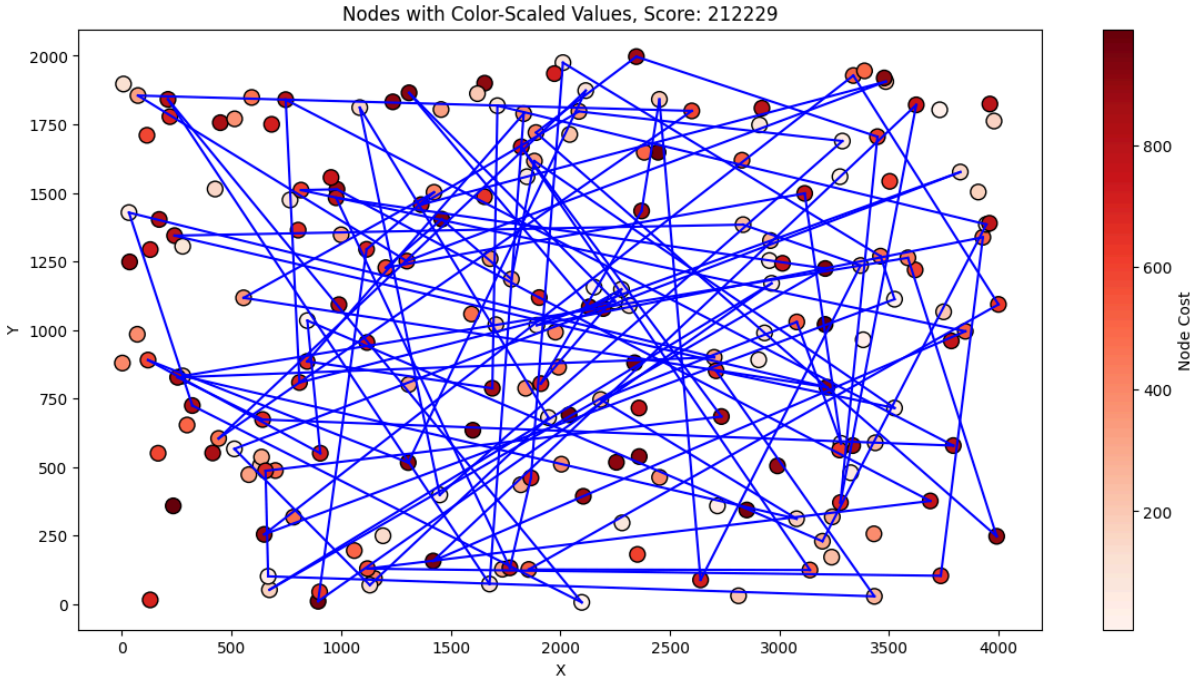

Fig 16. Visualization of the solution found by the Random algorithm on the TSPB problem instance starting from node 0.

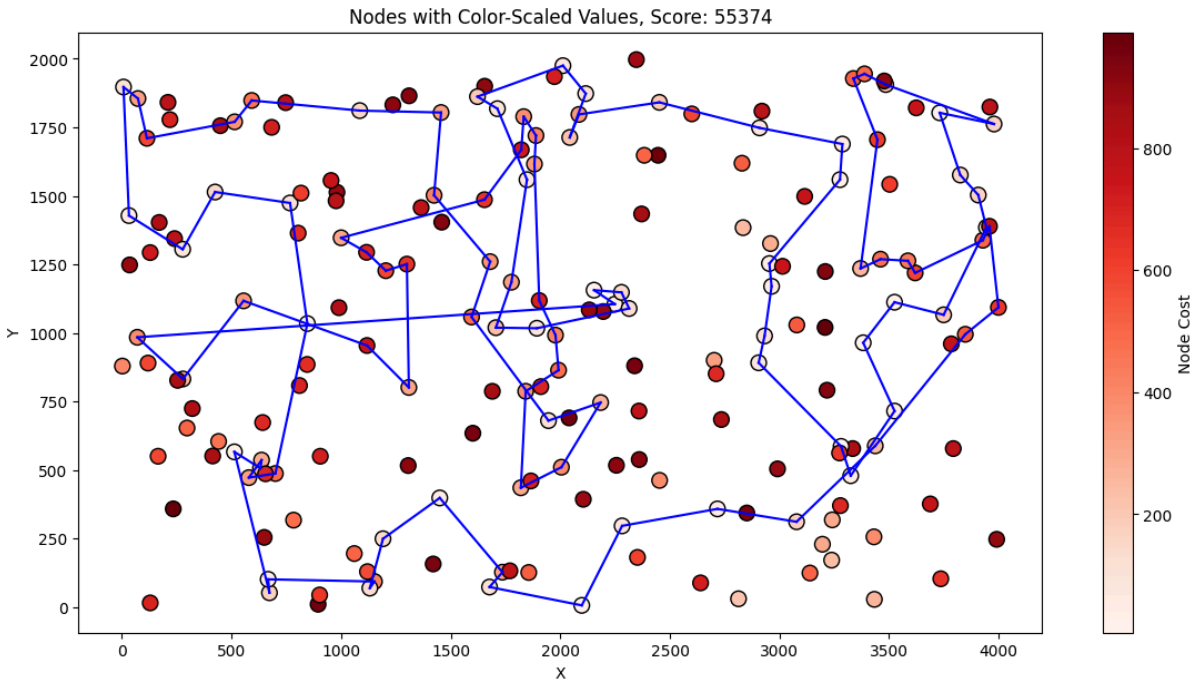

Fig 17. Visualization of the solution found by the Nearest Neighbor Closest algorithm on the TSPB problem instance starting from node 0.

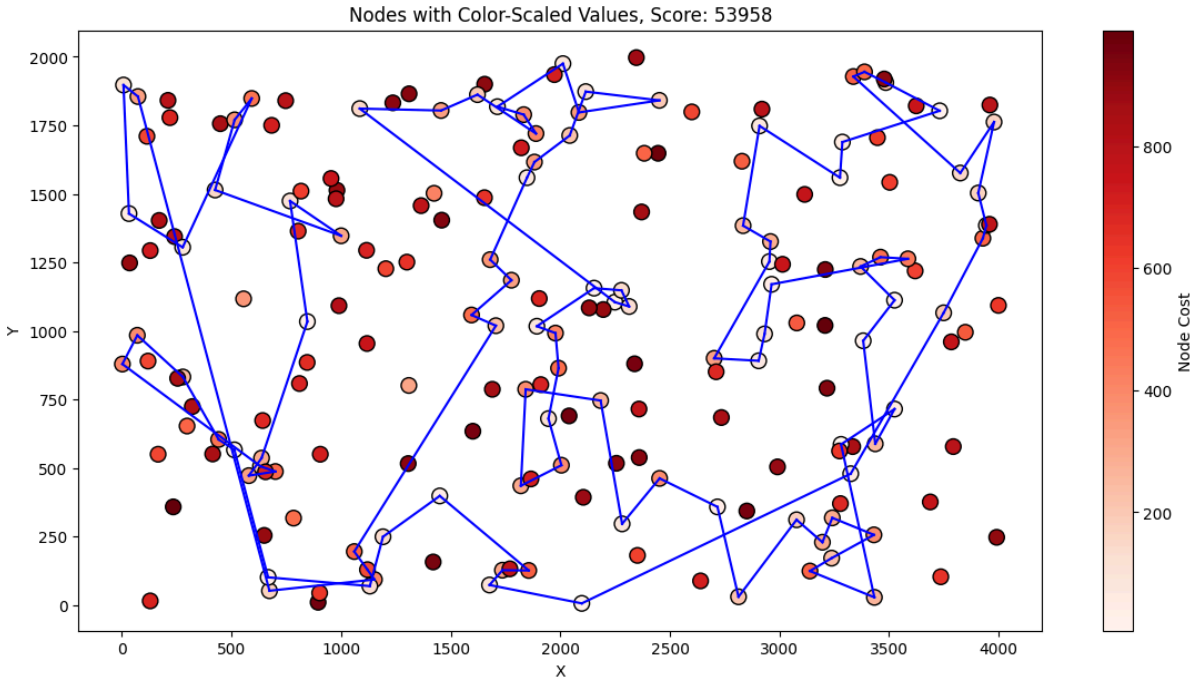

Fig 18. Visualization of the solution found by the Nearest Neighbor All algorithm on the TSPB problem instance starting from node 0.

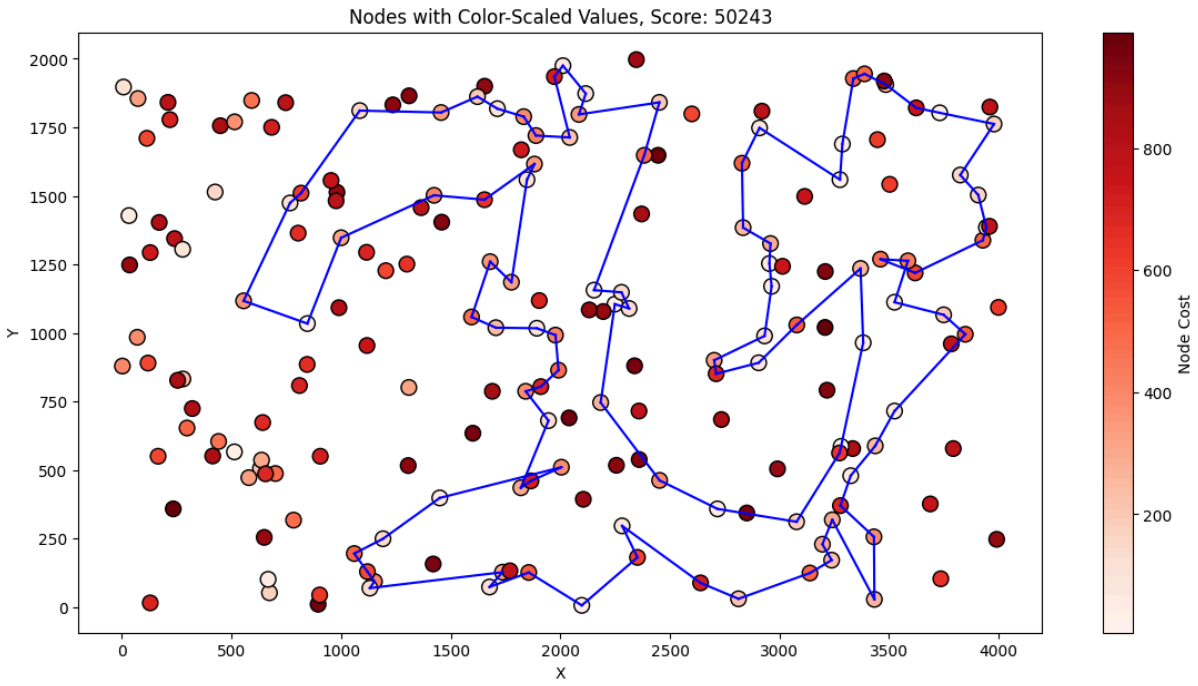

Fig 19. Visualization of the solution found by the Greedy Cycle algorithm on the TSPB problem instance starting from node 0.

4. Conclusions

Experiments conducted on the available instances of the problem proved that the greedy heuristic algorithms perform significantly better than random solutions with the greedy cycle method achieving the best results consistently, at the cost of complexity. The nearest neighbor algorithm appending nodes only to the end of the current path creates a visibly suboptimal solution, even in the best case, with many nonoptimal backtracks and intersections. Since the algorithm may finish on the opposite side of the instance space than the beginning, often a major distance cost is present in the closing of the cycle. In the TSPA problem instance, the nearest neighbor algorithm which is able to find neighbors closest to any node in the solution performed notably better, selecting paths that slightly closer resembled the greedy cycle approach. Although the algorithm was able to incorporate nodes in a smarter way, the inability to calculate all changes in the distances created by adding a new node meant that many backtracks and intersections in the path were still present when good choices were exhausted in a particular area. When considering the best results overall across all nodes, surprisingly, the approach considering all nodes in the solution performed on average slightly worse than the algorithm considering only the nearest node at the end in the TSPB instance of the problem which might be due to its inability to consider the changes in path distances to other nodes which are modified after new node insertion and the fact that the algorithm is more likely to exhaust good choices in a single area instead of moving forward further and reaching groups of more appealing nodes. In other words, for specific instances of the problem, the exploitation aspect may lead to worse results due to a lack of exploration. The greedy cycle method outperformed other algorithms in both instances. Its ability to consider the changes in path distance from both nodes of the broken connection

allows it to make more optimal decisions when adding a new node to the solution. The results often lack intersections (lines produced by the connections between nodes do not cross).