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.

Methods to solve:

Random approach: select random order of vertices to visist

Pseudo code:

Class: RandomSolution

Attributes:

- distanceMatrix (2D list)

Methods:

Constructor(distanceMatrix)

- *Input:* A 2D list distanceMatrix
- *Action:* Initializes the distanceMatrix attribute with the input value.

Solve(iterations)

- *Input:* An integer iterations
- *Output:* A list of lists solutions
- *Action:*
    1. Initialize solutions as an empty list.
    2. For each i from 0 to iterations:
        - Call GenerateRandomPermutation(i) to generate a random permutation.
        - Add the generated permutation to solutions.
    3. Return solutions.

GenerateRandomPermutation(startingNode)

- *Input:* An integer startingNode
- *Output:* A list permutation
- *Action:*
    1. Initialize permutation as an empty list.
    2. For each index i from 0 to the size of distanceMatrix:

        - Add i to permutation.
    3. Shuffle permutation randomly.
    4. Return the first half of the shuffled permutation list.

Results:

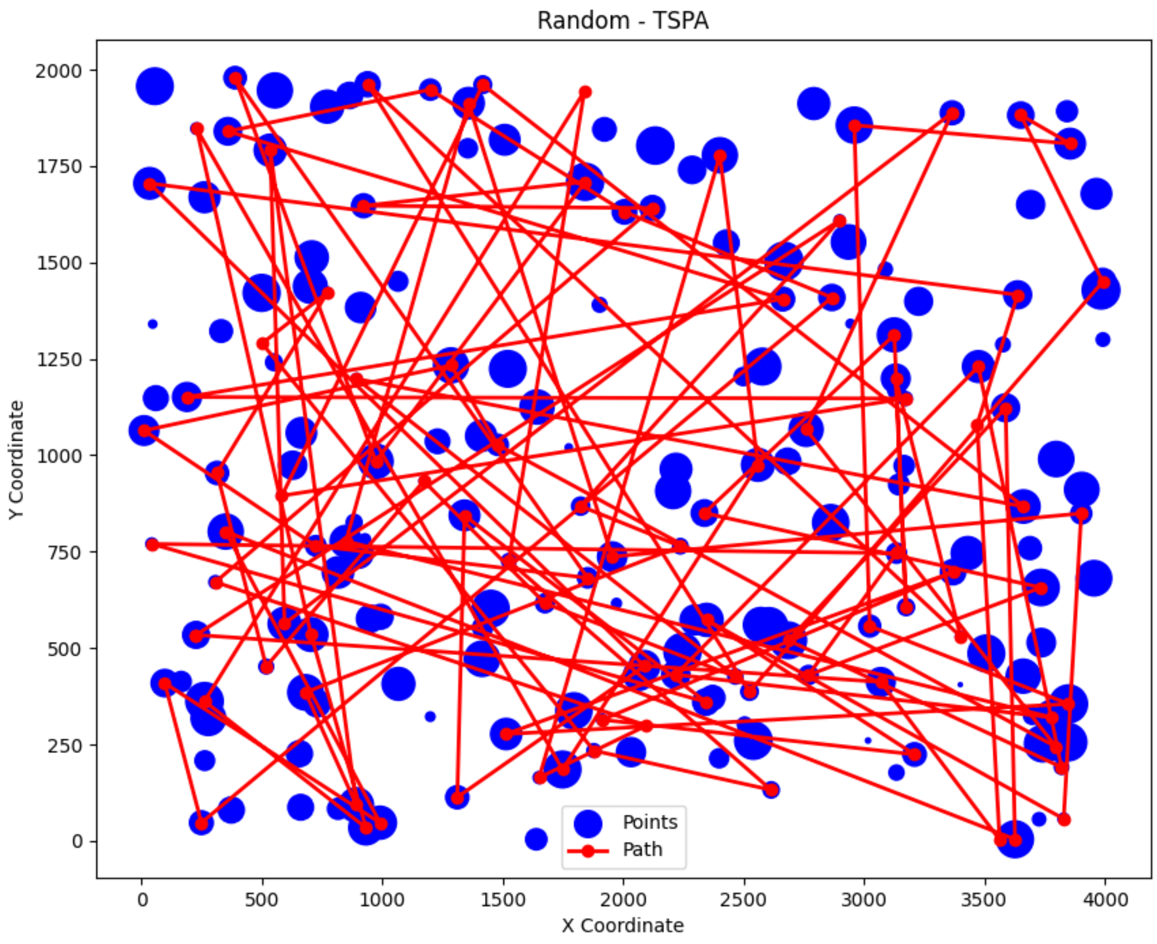
Problem Instance	Min Distance	Max Distance	Average Distance
TSPA	239 256	299 148	264 215.16
TSPB	187 901	239 651	213 411.075

Best Solutions:

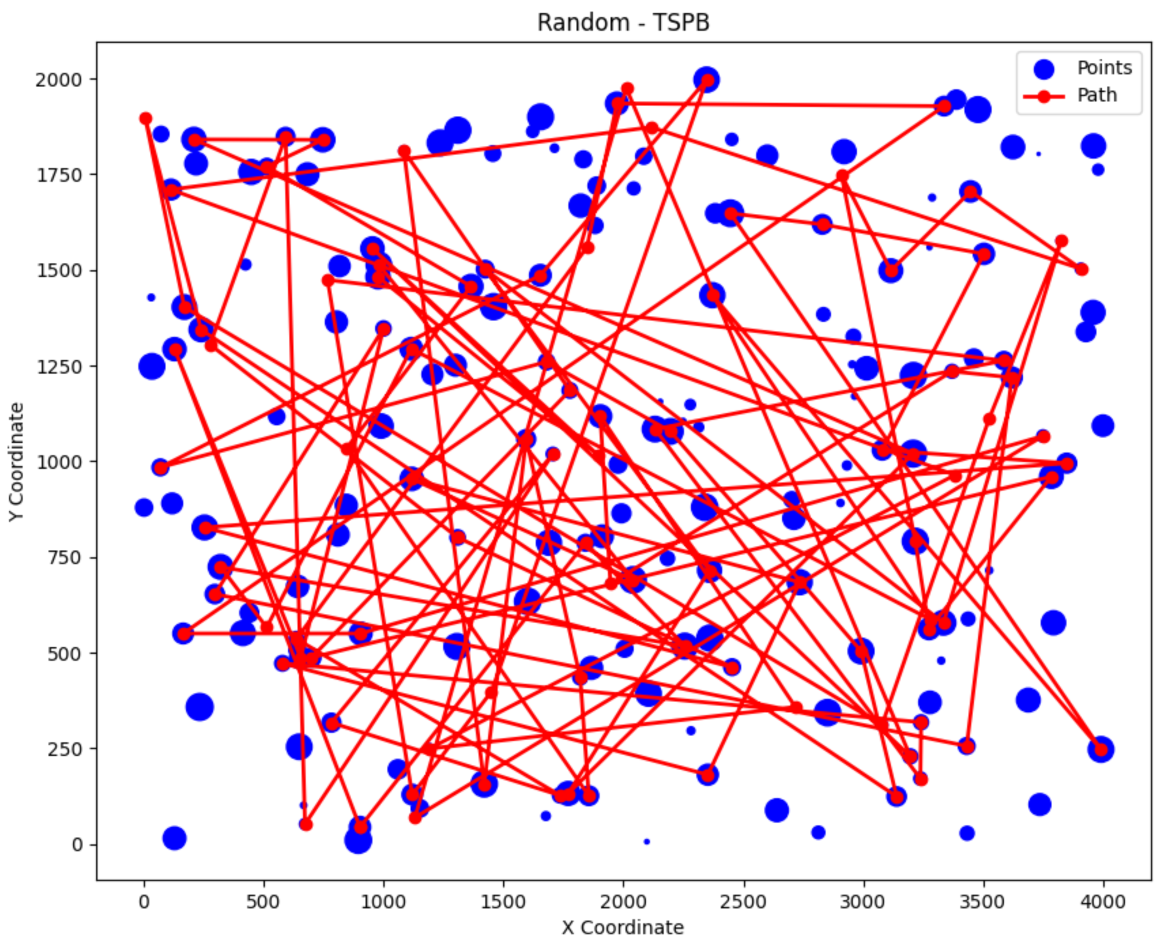
- TSPA: 48, 82, 16, 185, 39, 20, 182, 127, 72, 100, 34, 57, 2, 17, 124, 32, 92, 3, 145, 93, 1, 27, 7, 58, 132, 129, 188, 194, 179, 180, 25, 19, 175, 81, 122, 170, 105, 85, 121, 8, 113, 87, 119, 88, 192, 29, 30, 10, 102, 23, 117, 69, 9, 195, 106, 42, 143, 169, 115, 97, 133, 59, 184, 181, 139, 193, 154, 75, 151, 183, 147, 156, 166, 18, 6, 111, 64, 135, 53, 160, 186, 68, 76, 197, 108, 79, 43, 103, 60, 51, 56, 128, 21, 28, 163, 126, 11, 94, 80, 144

- TSPB: 162, 104, 50, 24, 117, 98, 176, 69, 93, 167, 170, 9, 128, 172, 197, 81, 25, 135, 10, 80, 138, 136, 61, 54, 168, 161, 4, 42, 173, 179, 175, 70, 89, 71, 33, 120, 67, 68, 158, 119, 97, 193, 103, 127, 39, 99, 90, 142, 177, 84, 43, 38, 139, 12, 129, 165, 32, 40, 96, 163, 185, 148, 88, 134, 41, 79, 110, 17, 115, 133, 85, 121, 91, 21, 182, 7, 147, 11, 73, 52, 180, 64, 152, 174, 199, 47, 3, 100, 86, 5, 153, 49, 8, 95, 22, 114, 156, 51, 164, 102

Instance A visualization:



Instance B visualization:



Nearest Neighbor: Selecting the node of lowest cost and least distance from either start or end of current path

Pseudo code:

Class: NearestNeighbor

Attributes:

- distanceMatrix (2D list)
- nodeCosts (list)

Methods:

Constructor(distanceMatrix, nodeCosts)

- *Input:* A 2D list distanceMatrix, a list nodeCosts
- *Action:* Initializes the distanceMatrix and nodeCosts attributes with the input values.

Solve(iterations)

- *Input:* An integer iterations
- *Output:* A list of lists solutions

- *Action:*
    1. Initialize solutions as an empty list.
    2. For each i from 0 to iterations:
        - Call GenerateSolution(i) to generate a solution starting from node i.
        - Add the generated solution to solutions.
    3. Return solutions.

GenerateSolution(startingNode)

- *Input:* An integer startingNode
- *Output:* A list solution
- *Action:*
    1. Initialize solution as an empty list.
    2. Add startingNode to solution.
    3. Set currentNode to startingNode.
    4. While the size of solution is less than half the size of distanceMatrix:
        - Call FindNearestNeighbor(currentNode, solution) to find the next node.
        - Add the next node to solution.
        - Update currentNode to the next node.
    5. Return solution.

FindNearestNeighbor(currentNode, visitedNodes)

- *Input:* An integer currentNode, a list visitedNodes
- *Output:* An integer nearestNode
- *Action:*
    1. Initialize nearestNode to -1 and minDistance to the maximum integer value.
    2. For each i from 0 to the size of distanceMatrix:
        - If i is not in visitedNodes and the sum of distanceMatrix[currentNode][i] and nodeCosts[i] is less than minDistance:
            - Set nearestNode to i.
            - Update minDistance to the sum of distanceMatrix[currentNode][i] and nodeCosts[i].
    3. Return nearestNode.

Results:

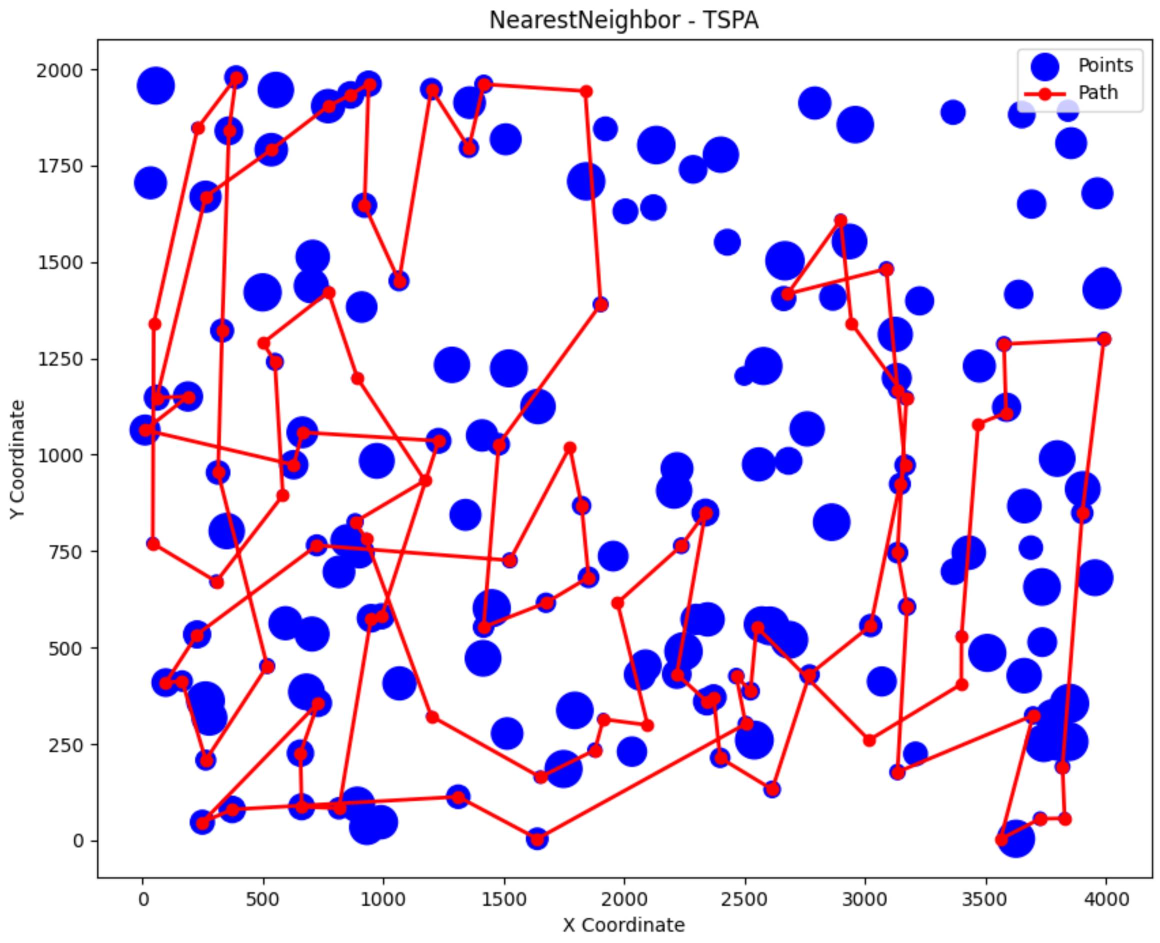
Type	Min Distance	Max Distance	Average Distance
TSPA	83,182	89,433	85,108.51
TSPB	52,319	59,030	54,390.43

Best Solutions:

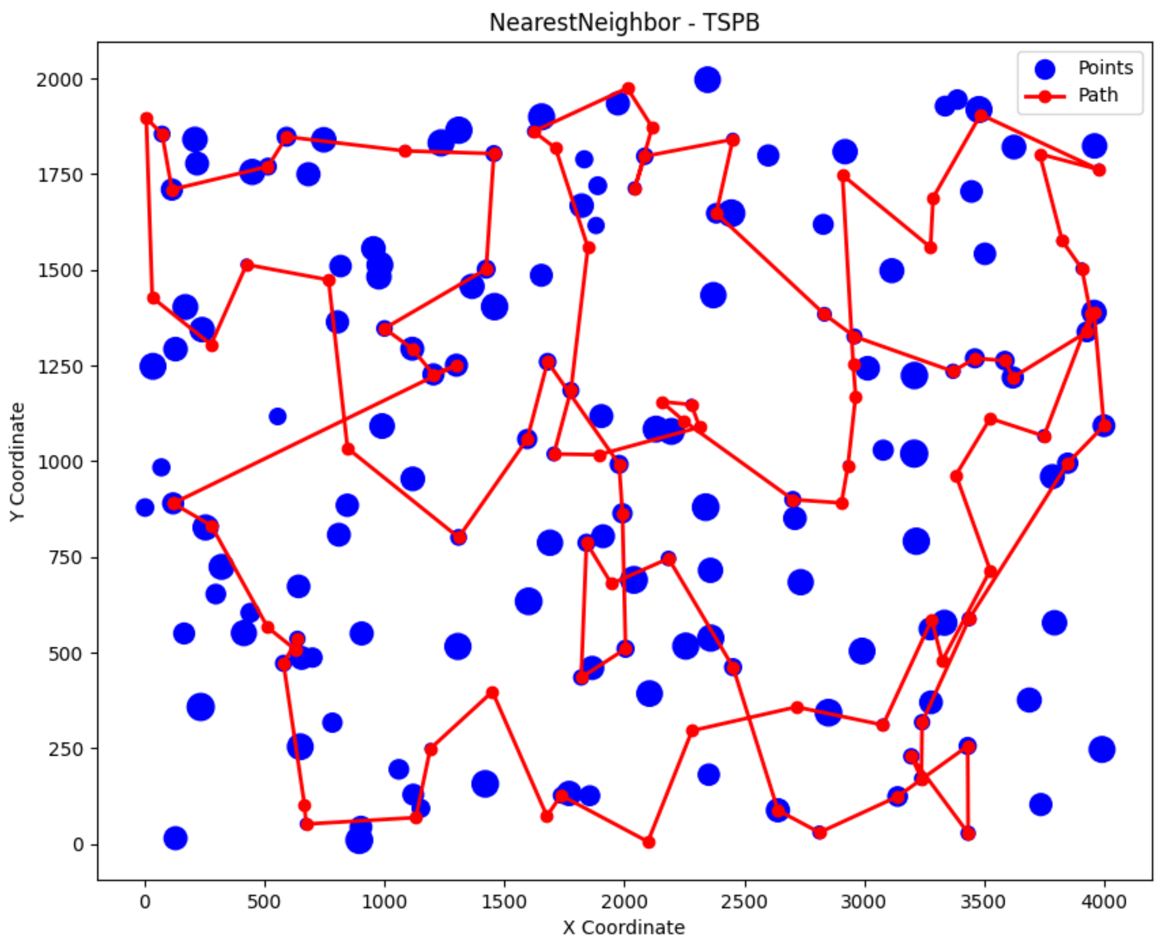
- TSPA: 124, 94, 63, 53, 180, 154, 135, 123, 65, 116, 59, 115, 139, 193, 41, 42, 160, 34, 22, 18, 108, 69, 159, 181, 184, 177, 54, 30, 48, 43, 151, 176, 80, 79, 133, 162, 51, 137, 183, 143, 0, 117, 46, 68, 93, 140, 36, 163, 199, 146, 195, 103, 5, 96, 118, 149, 131, 112, 4, 84, 35, 10, 190, 127, 70, 101, 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

- TSPB: 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

Instance A visualization:



Instance B visualization:



Any Nearest Neighbor: Selecting node which put anywhere inside current path leads to least ammount of increase in objective function

Pseudo code:

Class: NearestNeighborAny

Attributes:

- distanceMatrix (2D list)
- nodeCosts (list)

Methods:

Constructor(distanceMatrix, nodeCosts)

- *Input:* A 2D list distanceMatrix, a list nodeCosts
- *Action:* Initializes the distanceMatrix and nodeCosts attributes with the input values.

Solve(iterations)

- *Input:* An integer iterations
- *Output:* A list of lists solutions

- *Action:*
    1. Initialize solutions as an empty list.
    2. For each i from 0 to iterations:
        - Call GenerateSolution(i) to generate a solution starting from node i.
        - Add the generated solution to solutions.
    3. Return solutions.

GenerateSolution(startingNode)

- *Input:* An integer startingNode
- *Output:* A list solution
- *Action:*
    1. Initialize solution as an empty list and objective as an array with one element set to 0.
    2. Add startingNode to solution.
    3. Add the cost of startingNode to objective.
    4. While the size of solution is less than half the size of distanceMatrix:
        - Call FindAnyNearestNeighbor(solution, objective) to update the solution.
    5. Return solution.

FindAnyNearestNeighbor(visitedNodes, objective)

- *Input:* A list visitedNodes, an array objective
- *Output:* A list visitedNodes
- *Action:*
    1. Initialize tempObjective to objective[0], addingNode to -1, beforeNode to -1, and minObjective to the maximum integer value.
    2. For each node i from 0 to the size of distanceMatrix:
        - If i is not in visitedNodes:
            1. Set prevNode to -1.
            2. For each node in visitedNodes:
                - Calculate distanceToNext from node to i.
                - If prevNode is -1 or node is the last node in visitedNodes:
                    - Calculate the new objective value using tempObjective, distanceToNext, and nodeCosts[i].
                    - If the new objective is less than minObjective, update addingNode and minObjective.
                - Otherwise:
                    - Calculate distanceToPrev from prevNode to i, and distaneInCycle from prevNode to node.
                    - Calculate the new objective value using tempObjective, distanceToPrev, distanceToNext, and nodeCosts[i] while subtracting distaneInCycle.
                    - If the new objective is less than minObjective, update addingNode, beforeNode, and minObjective.
                - Set prevNode to the current node.
    3. Update objective[0] to minObjective.
    4. Return the result of AddNode(visitedNodes, addingNode, beforeNode).

AddNode(visitedNodes, addingNode, beforeNode)

- *Input:* A list visitedNodes, integers addingNode, beforeNode
- *Output:* A list newVisitedNodes
- *Action:*
    1. Initialize newVisitedNodes as an empty list.
    2. If beforeNode is -1, add addingNode to newVisitedNodes.
    3. For each node in visitedNodes:
        - Add node to newVisitedNodes.
        - If node is beforeNode, add addingNode after it.
    4. Return newVisitedNodes.

Results:

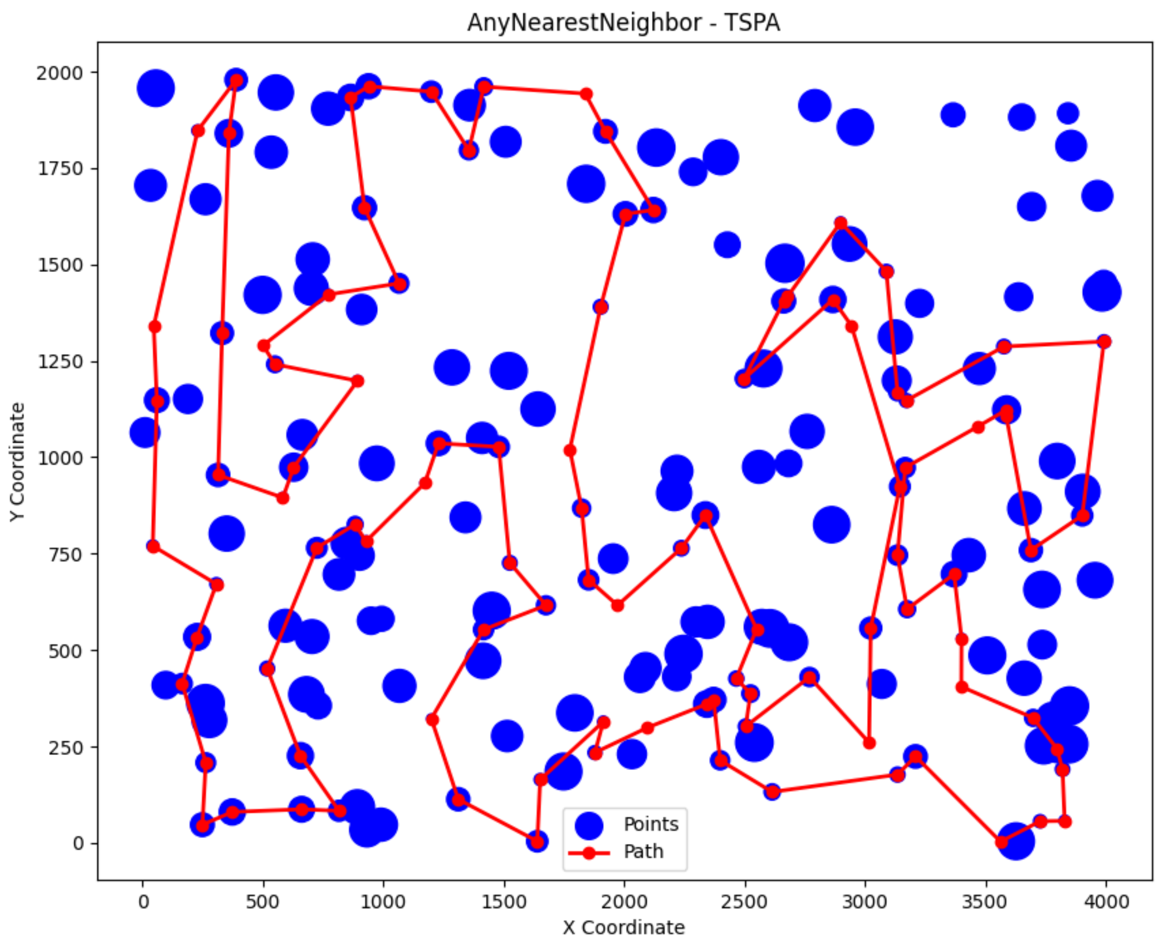
Type	Min Distance	Max Distance	Average Distance
TSPA	71,237	83,347	75,140.72
TSPB	49,125	58,800	52,401.99

Best Solutions:

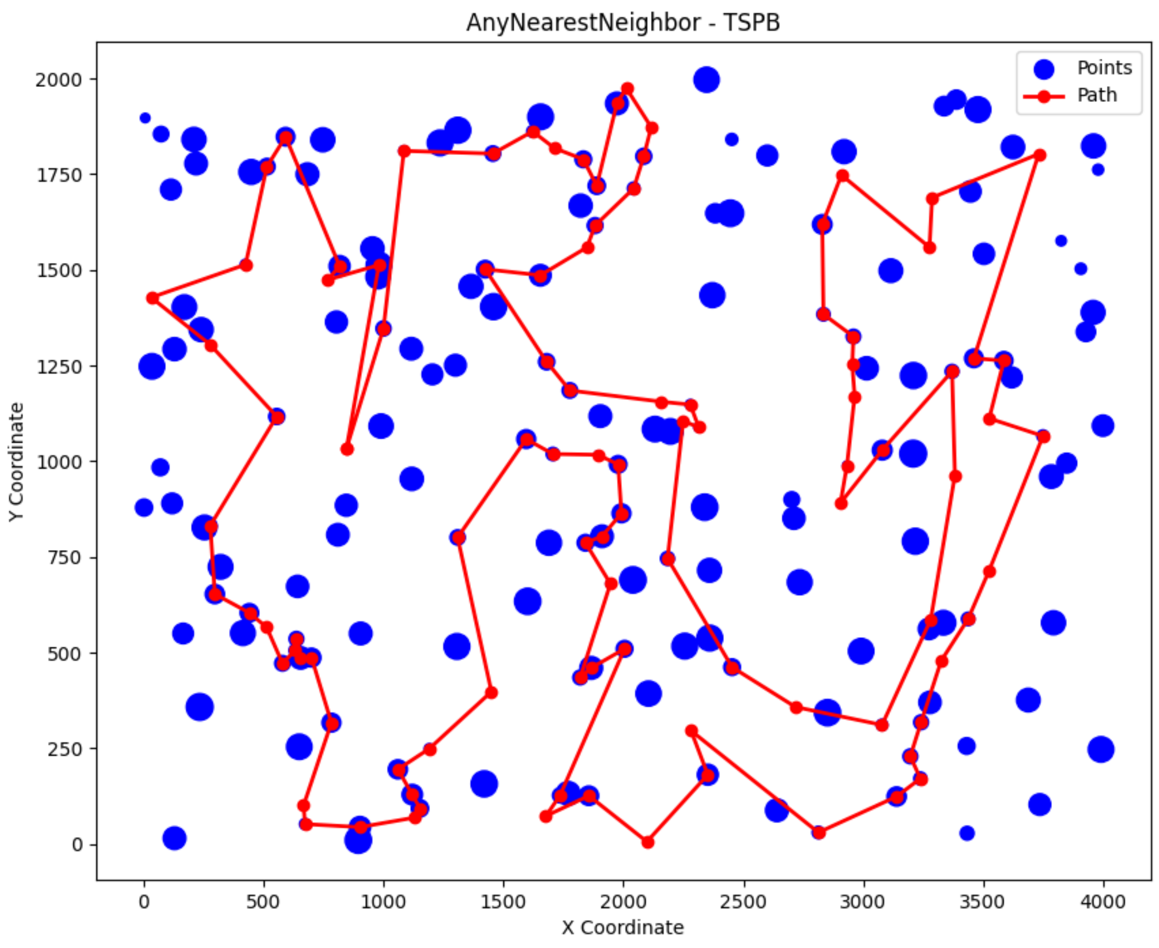
- TSPA: 0, 117, 93, 140, 68, 46, 139, 193, 41, 115, 5, 42, 181, 159, 69, 108, 18, 22, 146, 34, 160, 48, 54, 177, 10, 190, 4, 112, 84, 184, 43, 116, 65, 59, 118, 51, 151, 133, 162, 123, 127, 70, 135, 180, 154, 53, 100, 26, 86, 75, 44, 25, 16, 171, 175, 113, 56, 31, 78, 145, 179, 92, 57, 52, 185, 119, 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

- TSPB: 121, 51, 147, 6, 188, 169, 132, 13, 161, 70, 3, 15, 145, 195, 168, 43, 134, 139, 11, 29, 109, 35, 0, 111, 81, 153, 163, 176, 86, 95, 128, 106, 124, 62, 18, 55, 34, 170, 152, 183, 140, 20, 130, 99, 185, 179, 166, 194, 113, 103, 89, 127, 165, 187, 77, 97, 141, 91, 36, 61, 82, 87, 21, 8, 104, 56, 144, 160, 33, 138, 182, 25, 177, 5, 45, 142, 78, 175, 162, 80, 190, 136, 73, 164, 54, 31, 193, 117, 198, 156, 1, 131, 135, 63, 122, 133, 10, 191, 90, 67

Instance A visualization:



Instance B visualization:



Greedy Cycle: simmilar to any nearest neighbor, however we also compute distance between starting and ending node so we create cycle instead of path

Pseudo Code:

Class: GreedyCycle

Attributes:

- distanceMatrix (2D list)
- nodeCosts (list)

Methods:

Constructor(distanceMatrix, nodeCosts)

- *Input:* A 2D list distanceMatrix, a list nodeCosts
- *Action:* Initializes the distanceMatrix and nodeCosts attributes with the input values.

Solve(iterations)

- *Input:* An integer iterations

- *Output:* A list of lists solutions
- *Action:*
    1. Initialize solutions as an empty list.
    2. For each i from 0 to iterations:
        - Call GenerateSolution(i) to generate a solution starting from node i.
        - Add the generated solution to solutions.
    3. Return solutions.

GenerateSolution(startingNode)

- *Input:* An integer startingNode
- *Output:* A list solution
- *Action:*
    1. Initialize solution as an empty list and objective as an array with one element set to 0.
    2. Add startingNode to solution.
    3. Add the cost of startingNode to objective.
    4. While the size of solution is less than half the size of distanceMatrix:
        - Call FindAnyNearestNeighbor(solution, objective) to update the solution.
    5. Return solution.

FindAnyNearestNeighbor(visitedNodes, objective)

- *Input:* A list visitedNodes, an array objective
- *Output:* A list visitedNodes
- *Action:*
    1. Initialize tempObjective to objective[0], addingNode to -1, beforeNode to -1, and minObjective to the maximum integer value.
    2. For each node i from 0 to the size of distanceMatrix:
        - If i is not in visitedNodes:
            1. Set prevNode to -1.
            2. For each node in visitedNodes:
                - Calculate distanceToNext from node to i.
                - If prevNode is -1 or node is the last node in visitedNodes:
                    - Calculate closingCycleDistance:
                        - If prevNode is -1, set closingCycleDistance to the distance from i to the last node in visitedNodes.
                        - Otherwise, set closingCycleDistance to the distance from i to the first node in visitedNodes.
                    - Calculate the new objective value using tempObjective, distanceToNext, nodeCosts[i], and closingCycleDistance.
                    - If the new objective is less than minObjective, update addingNode and minObjective.
                - Otherwise:
                    - Calculate distanceToPrev from prevNode to i, and distanceInCycle from prevNode to node.
                    - Calculate the new objective value using tempObjective, distanceToPrev, distanceToNext, and nodeCosts[i], subtracting distanceInCycle.

```
            ▪ If the new objective is less than minObjective, update addingNode, beforeNode, and minObjective.
        ▪ Set prevNode to the current node.
    3. Update objective[0] to minObjective.
    4. Return the result of AddNode(visitedNodes, addingNode, beforeNode).
```

AddNode(visitedNodes, addingNode, beforeNode)

- *Input:* A list visitedNodes, integers addingNode, beforeNode
- *Output:* A list newVisitedNodes
- *Action:*

```
    1. Initialize newVisitedNodes as an empty list.
    2. If beforeNode is -1, add addingNode to newVisitedNodes.
    3. For each node in visitedNodes:
        ▪ Add node to newVisitedNodes.
        ▪ If node is beforeNode, add addingNode after it.
    4. Return newVisitedNodes.
```

Results:

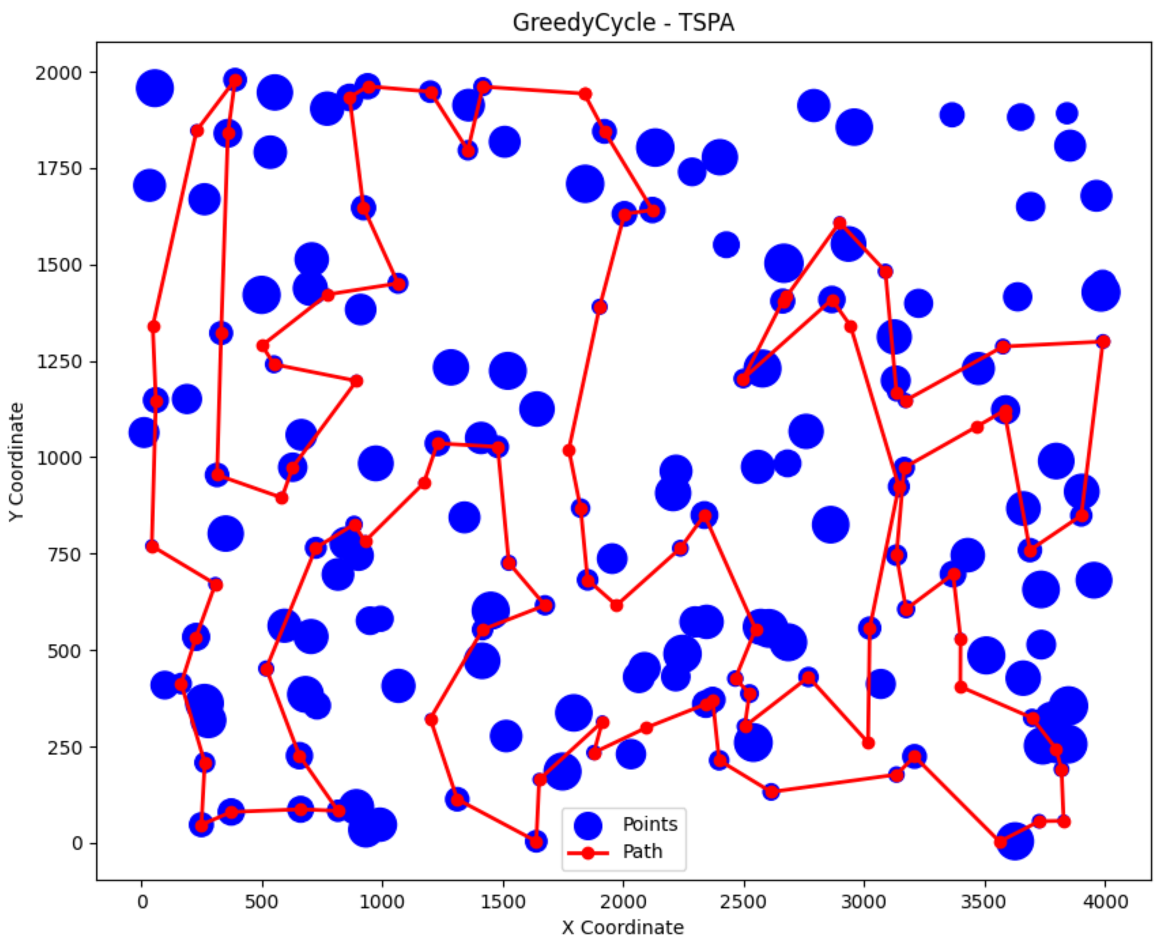
Type	Min Distance	Max Distance	Average Distance
TSPA	71,237	74,967	72,958.235
TSPB	49,306	58,453	51,780.94

Best Solutions:

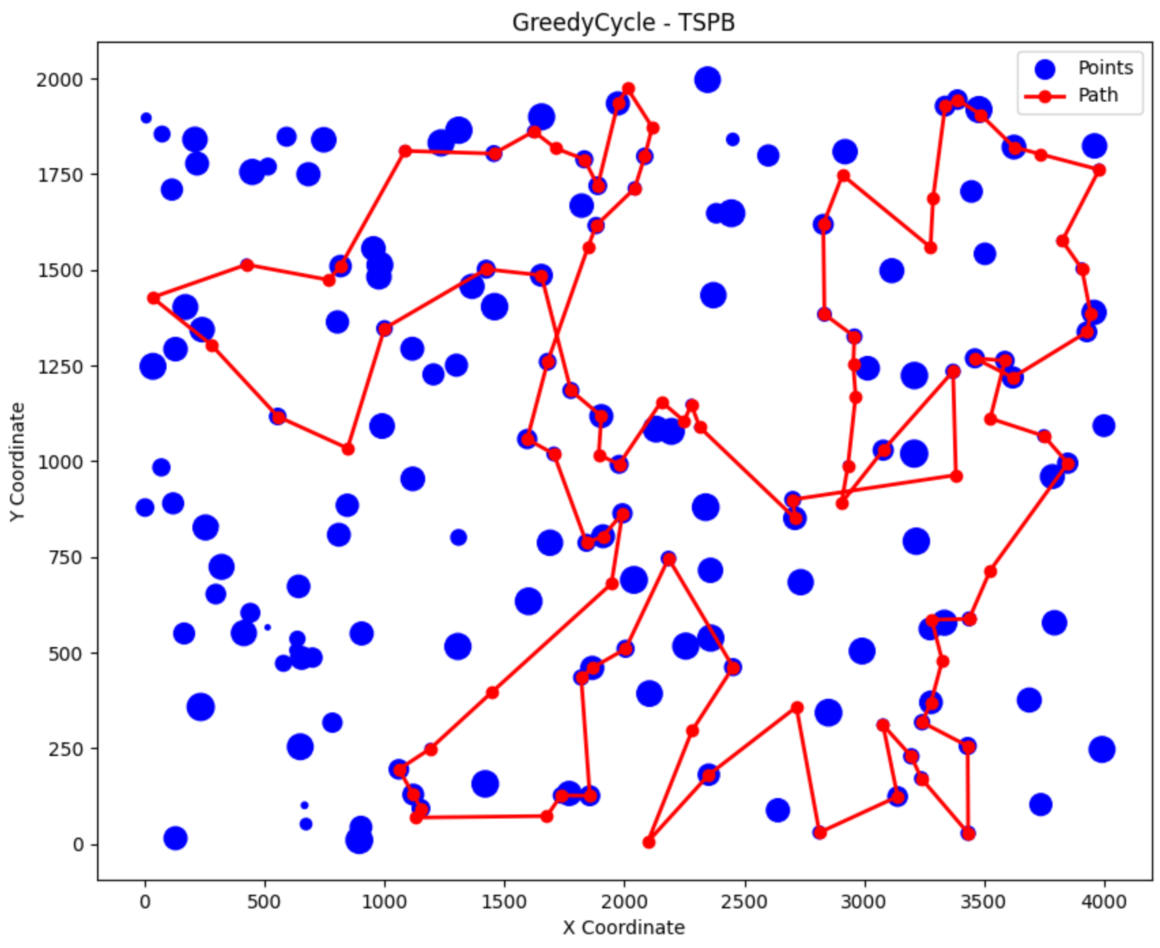
- TSPA: 183, 89, 186, 23, 137, 176, 80, 79, 63, 94, 124, 152, 97, 1, 101, 2, 120, 129, 55, 49, 102, 148, 9, 62, 144, 14, 178, 106, 165, 90, 81, 196, 40, 119, 185, 52, 57, 92, 179, 145, 78, 31, 56, 113, 175, 171, 16, 25, 44, 75, 86, 26, 100, 53, 154, 180, 135, 70, 127, 123, 162, 133, 151, 51, 118, 59,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

- TSPB: 194, 166, 172, 179, 185, 99, 130, 22, 66, 94, 47, 148, 60, 20, 59, 28, 149, 4, 140, 183, 152, 170, 34, 55, 18, 62, 124, 106, 128, 95, 86, 143, 159, 35, 109, 0, 29, 160, 33, 49, 11, 43, 134, 51, 121, 131, 135, 63, 122, 90, 191, 147, 6, 188, 169, 132, 13, 161, 70, 3, 15, 145, 195, 168, 139, 182, 138, 104, 56, 144, 8, 177, 5, 45, 142, 78, 175, 36, 61, 91, 21, 87, 82, 111, 81, 77, 141, 97, 153, 187, 165, 163, 89, 127, 137, 114, 103, 26, 113, 176

Instance A visualization:



Instance B visualization:



(All solutions were checked with solution checker)

Source code link: Github

Summary

Comparison of all methods

Method	Instance	Min Distance	Max Distance	Average Distance
RANDOM	TSPA	233,799	293,158	264,015.79
	TSPB	184,182	240,770	212,528.985
NEARESTNEIGHBOR	TSPA	83,182	89,433	85,108.51
	TSPB	52,319	59,030	54,390.43
ANYNEARESTNEIGHBOR	TSPA	71,237	83,347	75,140.72
	TSPB	49,125	58,800	52,401.99
GREEDYCYCLE	TSPA	71,237	74,967	72,958.235
	TSPB	49,306	58,453	51,780.94

Conclusion

The GREEDYCYCLE algorithm performed the best in minimizing distances across both instances, achieving the lowest average distances. This effectiveness can be attributed to its systematic approach to cycle optimization, which reduces whole cycle length instead of path length.

The ANYNEARESTNEIGHBOR algorithm followed closely, demonstrating better performance than both the NEARESTNEIGHBOR and RANDOM algorithms. This improvement is likely due to its ability to consider multiple nearest neighbors, allowing for more flexibility and reduced distances.

NEARESTNEIGHBOR algorithm didn't perform stunningly however even such a simple approach can yield much better results then random approach.

In contrast, the RANDOM algorithm yielded the highest distances, reflecting its lack of optimization and reliance on chance rather than a structured approach to pathfinding. This inefficiency makes it unsuitable for tasks requiring effective distance minimization.

Overall, GREEDYCYCLE's structured method makes it the top choice for optimization tasks in this comparison.