Assignment 1. Greedy Heuristics

Authors: Kiril Andrukh, 162069,
Uladzislau Lukashevich, 155671

October 13, 2024

Link to the source code: GitHub

1 Description of problem

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.

2 Random solution

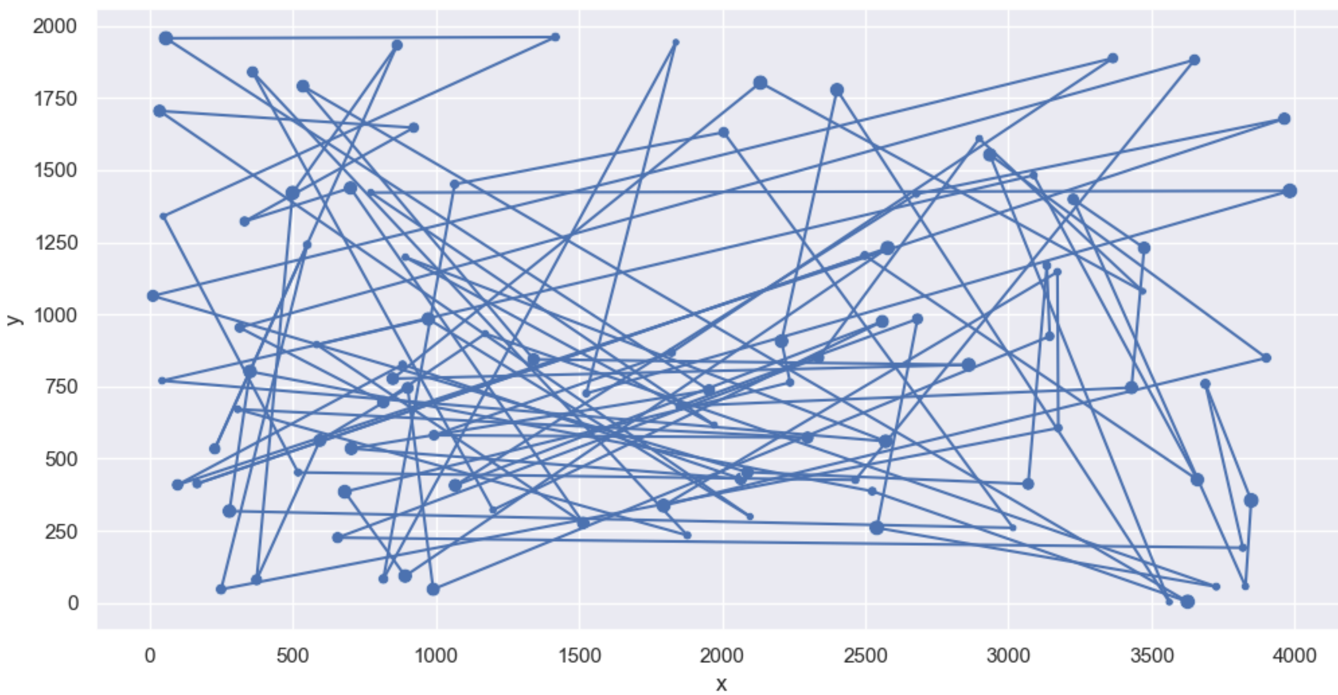
Pseudocode

Algorithm 1 Generate Random Solution

Procedure $\textsc{GenerateRandomSolution}(dataset)$
$size \leftarrow \textsc{RoundUp}(0.5 \times \textsc{Length}(dataset))$ {Calculate 50% of dataset size}
$selectedNodes \leftarrow \textsc{Sample}(dataset, size)$ {Sample without replacement}
return $selectedNodes$
End Procedure

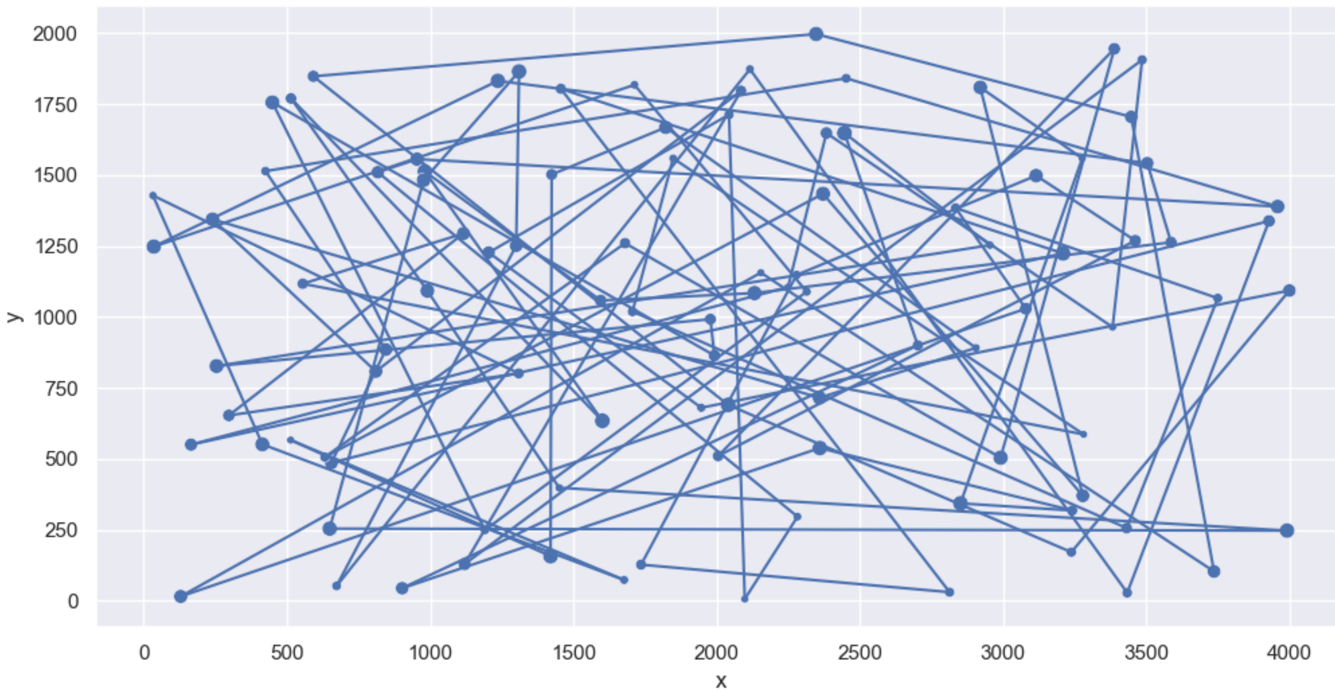
Results of a computational experiment

- For dataset A:
  - Best random solution: [166, 136, 116, 30, 71, 62, 24, 111, 123, 69, 154, 160, 189, 149, 80, 21, 103, 182, 82, 178, 55, 29, 47, 28, 190, 142, 140, 48, 41, 10, 92, 106, 173, 81, 155, 16, 64, 130, 94, 11, 126, 33, 54, 148, 38, 14, 42, 194, 191, 159, 68, 20, 53, 197, 34, 74, 115, 63, 139, 107, 151, 183, 112, 46, 23, 120, 104, 192, 1, 88, 163, 72, 172, 105, 83, 185, 144, 124, 84, 113, 196, 188, 175, 138, 8, 91, 79, 167, 150, 171, 59, 77, 181, 7, 97, 184, 22, 143, 134, 122]
  - Min objective function: 243393
  - Average objective function: 265009.32
  - Max objective function: 296391



- For dataset B:
  - Best random solution: [137, 66, 164, 112, 122, 155, 154, 71, 191, 77, 141, 145, 118, 8, 57, 127, 68, 10, 84, 199, 75, 29, 108, 128, 34, 179, 114, 138, 168, 176, 131, 98, 156, 25, 63, 30, 36, 117, 7, 134, 126, 106, 41, 32, 19, 15, 142, 28, 86, 189, 0, 42, 181, 6, 187, 61, 109, 174, 130, 82, 149, 129, 139, 31, 69, 26, 53, 183, 186, 103, 14, 162, 143, 167, 18, 197, 160, 144, 44, 5, 35, 169, 92, 192, 9, 99, 39, 182, 67, 171, 133, 116, 177, 93, 105, 120, 150, 74, 80, 3]
  - Min objective function: 190669

- Average objective function: 212375.76
- Max objective function: 241913



3 Distance matrix

Distance matrix was calculated once for the next algorithms, for them not to do redundant calculations and run faster with the same output.

Pseudocode

Algorithm 2 Calculate Distance Matrix

```
Procedure CalculateDistanceMatrix(dataset)
numNodes ← Length(dataset)
distanceMatrix ← ArrayOfZeros(numNodes, numNodes) {Initialize the distance matrix}
for i ← 1 to numNodes do
    for j ← 1 to numNodes do
        if i ≠ j then
            node1 ← dataset[i]
            node2 ← dataset[j]
            distance ← EuclideanDistance(node1, node2) {Compute Euclidean distance}
            cost ← NodesCost(node1, node2) {Get node-related cost}
            distanceMatrix[i, j] ← distance + cost {Sum of distance and node cost}
        end if
    end for
end for
return distanceMatrix
End Procedure
```

4 Nearest neighbor considering adding the node only at the end of the current path

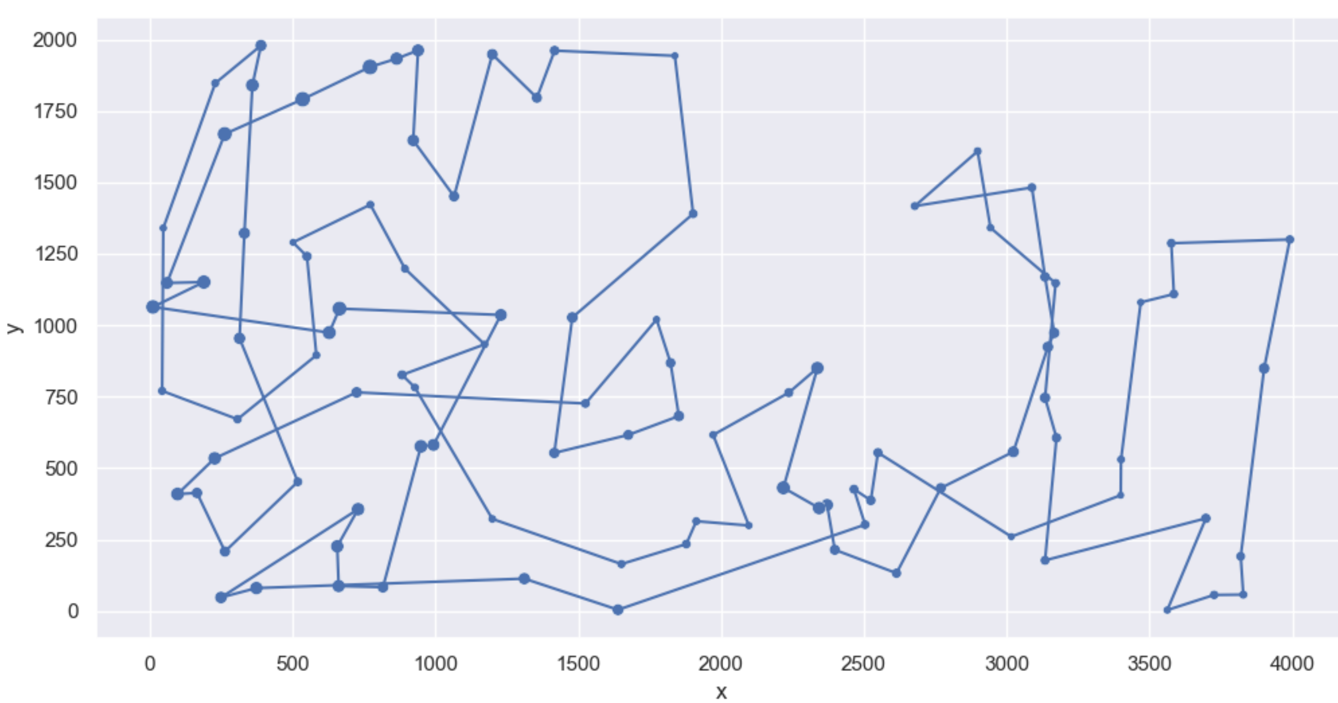
Pseudocode

Results of a computational experiment

- For dataset A:
  - Best random solution: [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]
  - Min objective function: 83182
  - Average objective function: 85108.51
  - Max objective function: 89433

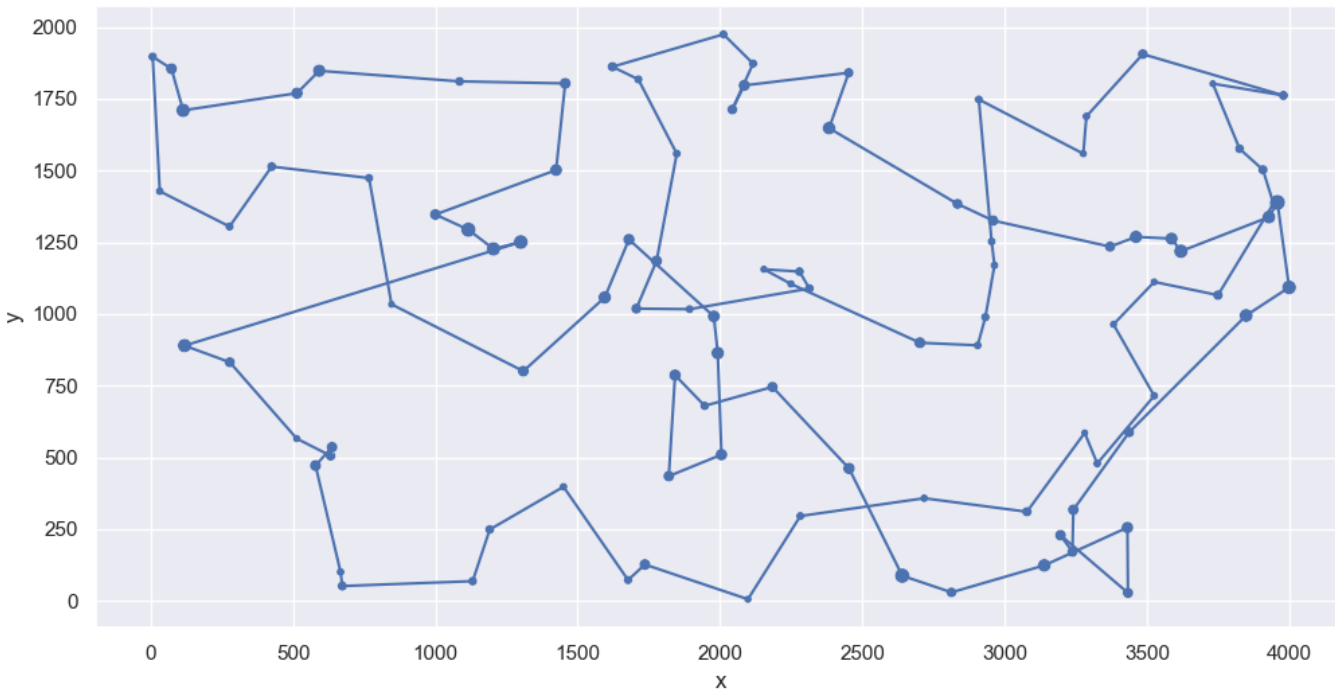
Algorithm 3 Generate Nearest Neighbor End

Procedure GENERATENEARESTNEIGHBOREND(*dataset*)
size ← ROUNDUP(0.5 × LENGTH(*dataset*)) {Calculate 50% of the dataset size}
numNodes ← LENGTH(*dataset*)
remainingMask ← ARRAYOFONES(*numNodes*) {Boolean mask array initialized to True}
solution ← [*startingNode*] {Initialize solution with the starting node}
remainingMask[*startingNode*] ← **False** {Mark the starting node as used}
while LENGTH(*solution*) < *size* **do**
    lastNode ← LASTELEMENT(*solution*) {Get the last node in the current solution}
    distancesToLastNode ← *distanceMatrix*[*lastNode*] {Extract distances from the last node}
    distancesToLastNode[¬*remainingMask*] ← ∞ {Set distances to used nodes as infinity}
    nearestNode ← ARGMIN(*distancesToLastNode*) {Find the nearest unused node}
    APPEND(*solution*, *nearestNode*) {Add the nearest node to the solution}
    remainingMask[*nearestNode*] ← **False** {Mark the nearest node as used}
end while
return GETSOLUTIONSUBSET(*dataset*, *solution*)
End Procedure



- For dataset B:

- Best random 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]
- Min objective function: 52319
- Average objective function: 54390.43
- Max objective function: 59030



5 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

Pseudocode

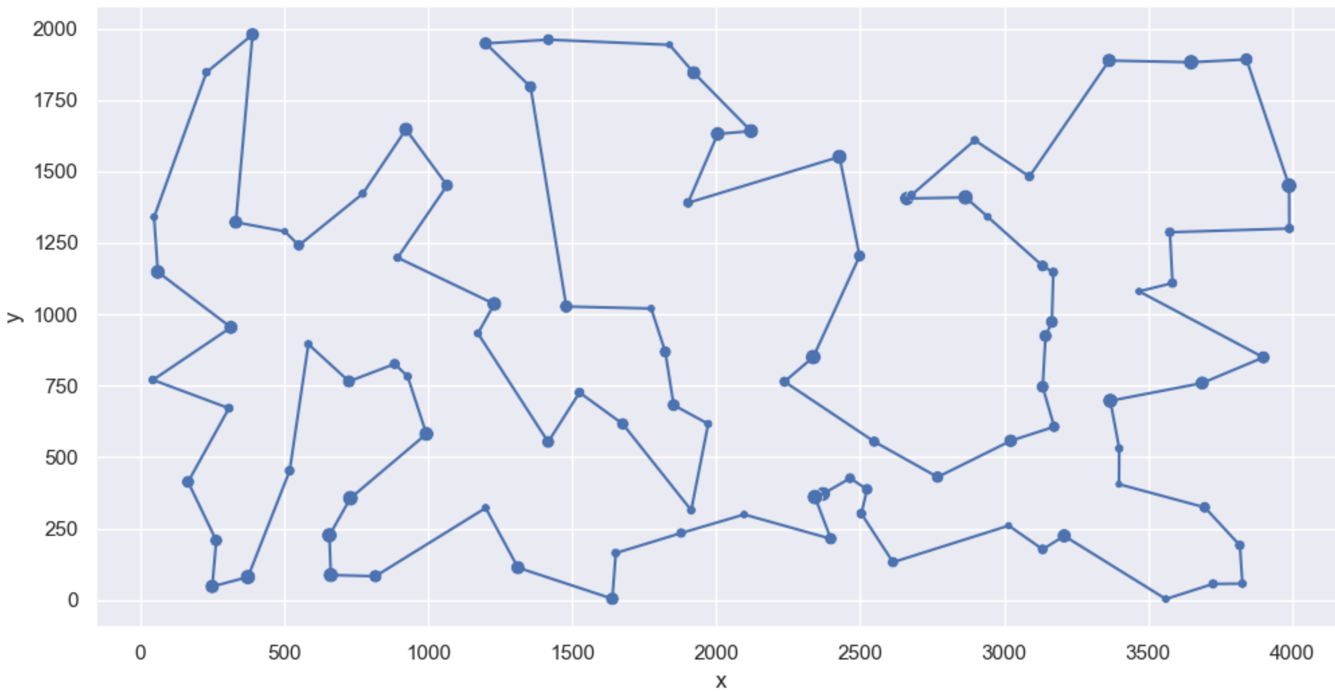
Results of a computational experiment

- For dataset A:

Algorithm 4 Generate Nearest Neighbor At The Best Position

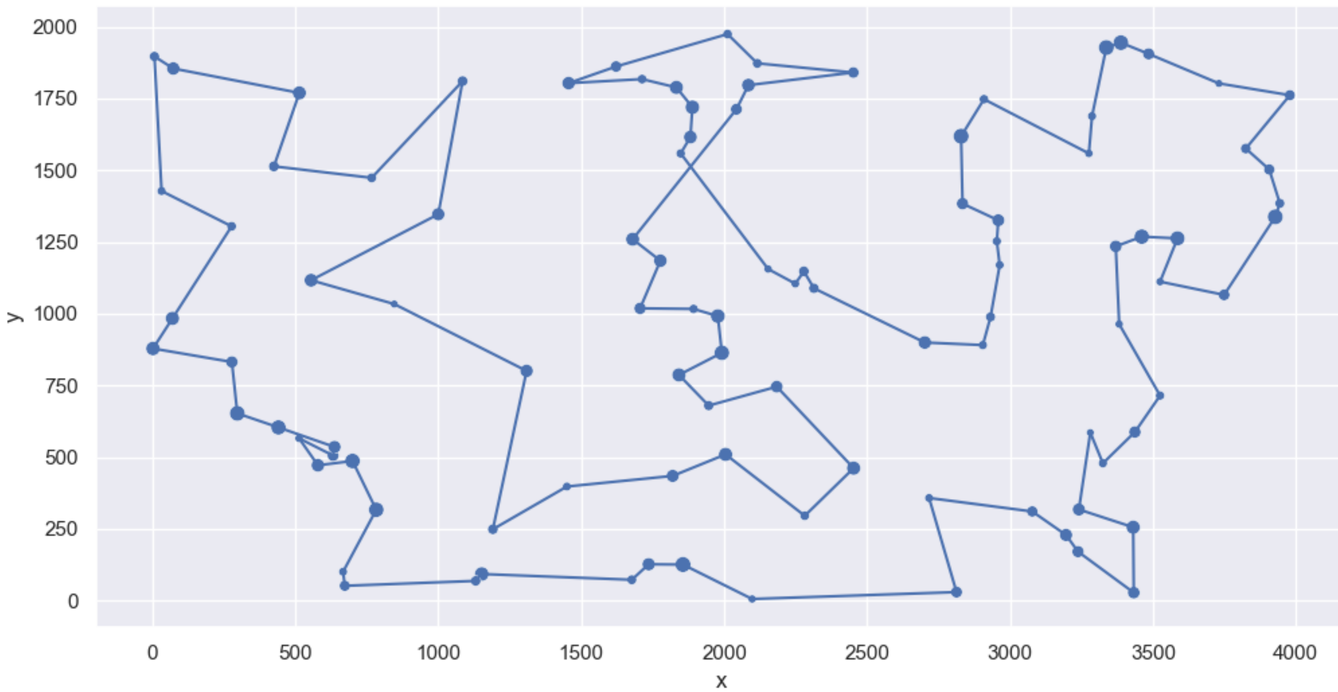
Procedure GenerateNearestNeighborAtTheBestPosition($dataset$)
$size \leftarrow$ RoundUp($0.5 \times$ Length($dataset$)) {Calculate 50% of the dataset size}
$numNodes \leftarrow$ Length($dataset$)
$solution \leftarrow [startingNode]$ {Initialize solution with the starting node}
$remainingNodes \leftarrow$ Set($[0, 1, \ldots, numNodes - 1]$) {Initialize set of all node indices}
Remove($startingNode$, $remainingNodes$) {Remove the starting node from remaining nodes}
while Length($solution$) $< size$ **do**
  $bestInsertionCost \leftarrow \infty$
  $bestInsertion \leftarrow$ **None**
  for each $nodeIdx \in remainingNodes$ **do**
    $nodeCost \leftarrow$ GetCost($dataset, nodeIdx$)
    for $i \leftarrow 0$ **to** Length(solution) **do**
      if $i = 0$ **then**
        $prevNode \leftarrow$ LastElement($solution$) {Wrap around for circular Hamiltonian cycle}
      else
        $prevNode \leftarrow solution[i - 1]$
      end if
      if $i =$ Length($solution$) **then**
        $nextNode \leftarrow solution[0]$ {The next node wraps to the first node in a cycle}
      else
        $nextNode \leftarrow solution[i]$
      end if
      $insertCost \leftarrow distanceMatrix[prevNode, nodeIdx] + distanceMatrix[nodeIdx, nextNode] - distanceMatrix[prevNode, nextNode]$
      $totalCost \leftarrow insertCost + nodeCost$
      if $totalCost < bestInsertionCost$ **then**
        $bestInsertionCost \leftarrow totalCost$
        $bestInsertion \leftarrow (nodeIdx, i)$
      end if
    end for
  end for
  Insert($bestInsertion[0]$, $solution$, $bestInsertion[1]$) {Insert the best node at the best position}
  Remove($bestInsertion[0]$, $remainingNodes$)
end while
return GetSolutionSubset($dataset$, $solution$)
End Procedure

- Best random solution: [164, 27, 90, 165, 40, 185, 81, 196, 179, 145, 78, 31, 113, 175, 171, 16, 25, 44, 120, 75, 101, 1, 97, 26, 100, 86, 53, 154, 135, 70, 127, 123, 112, 4, 84, 35, 149, 65, 116, 43, 42, 184, 190, 10, 177, 54, 160, 34, 181, 146, 22, 18, 108, 159, 193, 41, 139, 68, 46, 115, 118, 59, 162, 151, 133, 180, 63, 79, 80, 176, 51, 0, 117, 143, 183, 89, 186, 23, 137, 15, 148, 124, 94, 152, 2, 129, 92, 57, 55, 52, 106, 178, 49, 102, 9, 62, 144, 14, 21, 7]
- Min objective function: 71329
- Average objective function: 72183.035
- Max objective function: 73282



- For dataset B:

  - Best random solution: [149, 28, 20, 60, 148, 47, 94, 66, 179, 185, 99, 130, 95, 86, 166, 194, 113, 176, 103, 114, 137, 127, 89, 163, 153, 187, 141, 91, 61, 36, 78, 175, 80, 190, 136, 73, 193, 117, 31, 54, 198, 156, 1, 27, 38, 135, 63, 40, 107, 133, 122, 90, 147, 51, 131, 121, 25, 5, 177, 21, 82, 77, 81, 111, 8, 104, 144, 160, 33, 138, 11, 139, 145, 15, 155, 3, 70, 188, 6, 169, 132, 13, 195, 168, 29, 0, 109, 35, 143, 106, 124, 62, 18, 55, 34, 170, 152, 183, 140, 4]
  - Min objective function: 46193
  - Average objective function: 47038.35
  - Max objective function: 48330



6 Greedy Cycle

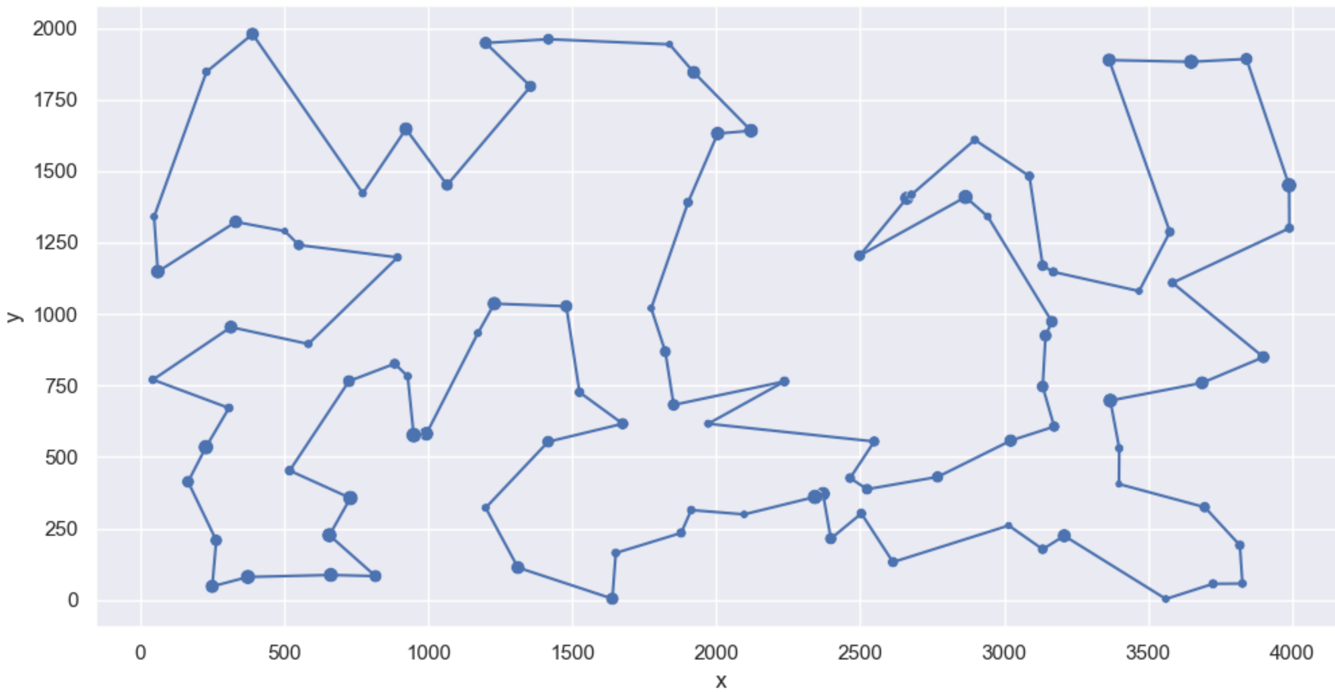
Pseudocode

Results of a computational experiment

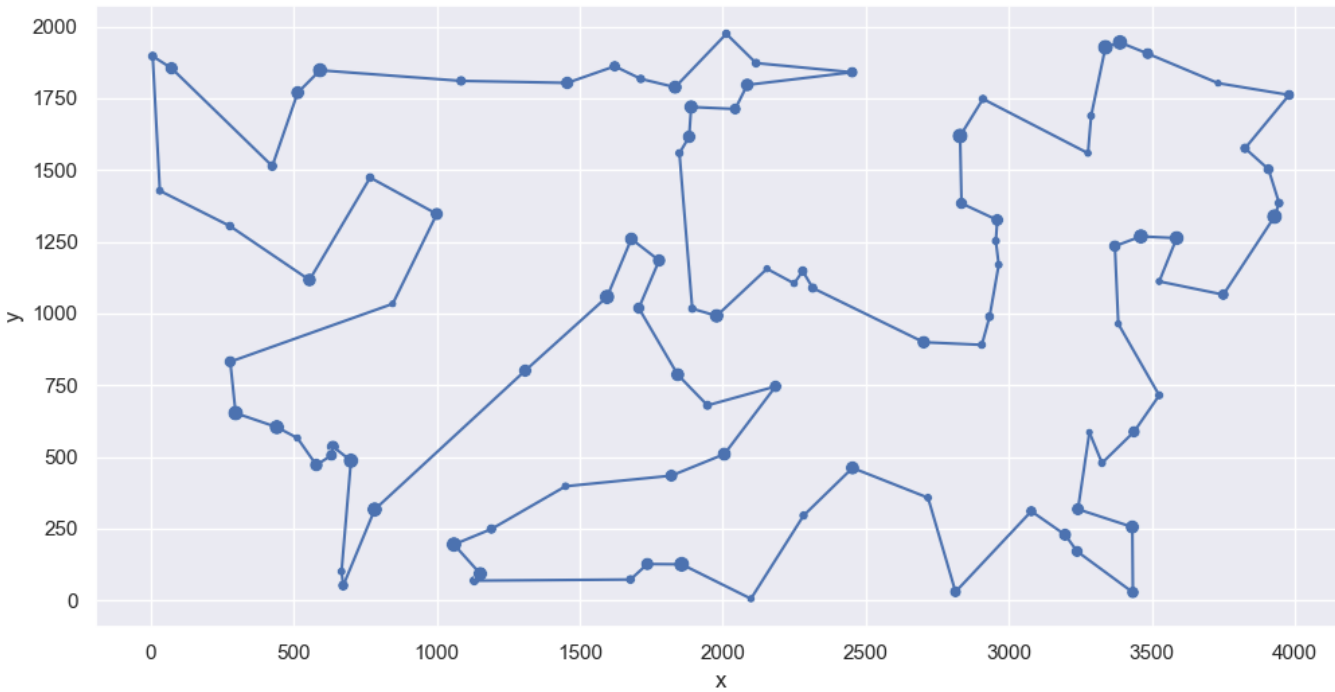
- For dataset A:
  - Best random solution: [0, 117, 143, 183, 89, 186, 23, 137, 176, 80, 79, 94, 63, 152, 97, 1, 2, 129, 92, 57, 55, 52, 49, 102, 148, 9, 62, 144, 14, 178, 106, 185, 165, 21, 7, 164, 27, 90, 40, 81, 196, 179, 145, 78, 31, 113, 175, 171, 16, 25, 44, 120, 75, 101, 86, 26, 100, 53, 180, 154, 135, 70, 127, 123, 162, 133, 151, 51, 118, 59, 149, 131, 65, 116, 43, 184, 35, 84, 112, 4, 190, 10, 177, 54, 48, 160, 34, 181, 42, 115, 41, 193, 159, 146, 22, 18, 108, 139, 68, 46]
  - Min objective function: 71263
  - Average objective function: 72071.915
  - Max objective function: 73154

Algorithm 5 Generate Greedy Cycle Solution

```
Procedure GenerateGreedyCycleSolution(dataset)
size ← RoundUp(0.5 × Length(dataset)) {Calculate 50% of the dataset size}
numNodes ← Length(dataset)
remainingNodes ← Set(Range(numNodes)) {Create a set of remaining nodes}
remainingNodes.Remove(startingNode)
solution ← [startingNode] {Initialize the solution with the starting node}
nearestNode ← ArgMin(distanceMatrix[startingNode, List(remainingNodes)]) {Find nearest node to starting node}
nearestNodeIdx ← List(remainingNodes)[nearestNode]
Append(solution, nearestNodeIdx) {Add nearest node to the solution}
remainingNodes.Remove(nearestNodeIdx)
while Length(solution) < size do
  bestInsertionCost ← ∞ {Initialize best insertion cost}
  bestInsertion ← None {Initialize best insertion tuple}
  for nodeIdx ← List(remainingNodes) do
    for i ← 0 to Length(solution) − 1 do
      nextI ← (i + 1) mod Length(solution)
      currentCost ← distanceMatrix[solution[i], nodeIdx] + distanceMatrix[nodeIdx, solution[nextI]] − distanceMatrix[solution[i], solution[nextI]] {Calculate insertion cost}
      if currentCost < bestInsertionCost then
        bestInsertionCost ← currentCost
        bestInsertion ← (nodeIdx, i) {Update best insertion}
      end if
    end for
  end for
  Insert(solution, bestInsertion[1] + 1, bestInsertion[0]) {Insert best node into the solution}
  remainingNodes.Remove(bestInsertion[0]) {Remove inserted node from remaining nodes}
end while
return GetSolutionSubset(dataset, solution)
End Procedure
```



- For dataset B:
  - Best random solution: [4, 149, 28, 20, 60, 148, 47, 94, 66, 179, 185, 99, 130, 95, 86, 166, 194, 113, 176, 103, 114, 137, 127, 89, 163, 187, 153, 81, 77, 141, 91, 61, 36, 175, 78, 45, 5, 177, 21, 82, 111, 8, 104, 138, 11, 139, 182, 25, 136, 80, 190, 73, 54, 31, 193, 117, 198, 156, 1, 121, 51, 90, 131, 135, 63, 40, 107, 122, 133, 10, 147, 6, 188, 169, 132, 70, 3, 155, 15, 145, 13, 195, 168, 33, 160, 29, 0, 109, 35, 143, 106, 124, 62, 18, 55, 34, 170, 152, 183, 140]
  - Min objective function: 45312
  - Average objective function: 46903.73
  - Max objective function: 48623



7 Conclusion

Comparing all the methods together, the best performance was reached by Greedy cycle, with following best objective function values:

Dataset A: 71263; Dataset B: 45312.

The best solutions were checked by solution checker and were calculated align with the value of objective function.

Nearest Neighbor with best possible position choice on the current path was the algorithm longest to execute. The reason to this is due to Nearest Neighbor with best possible position having $O(n^2 * k)$ complexity, where ‘n‘ is the number of remaining nodes and ‘k‘ is the number of visited nodes. Greedy cycle on the other hand has $O(n^2)$ complexity, while having better performance.

Obviously, the random solution worked the worst, and the less sophisticated Nearest Neighbor with end node on the current path was the second least optimized method.