Report from lab 1 – Greedy Heuristics

Aleksandra Krasicka 148254
Dominika Plewińska 151929

Description of the problem

The task involves selecting 50% of nodes from a set of nodes with x, y coordinates and costs, and forming a Hamiltonian cycle that minimizes the sum of the total path length and the total node costs. The distances between nodes are calculated using Euclidean distance, rounded to integers, and stored in a distance matrix. Optimization methods should only access the distance matrix, not the original node coordinates, for solving the problem.

Pseudocode

- **Random Solution**

```
Function generateRandomSolution(n):
  Create an empty list called nodes
  For i from 0 to n-1:
    Add i to nodes

  nodesToSelect = round_up(n / 2.0)

  Shuffle the nodes list randomly

  Return the first nodesToSelect elements of the shuffled nodes list

Function calculateCost(solution, distanceMatrix, nodeData):
  If solution is null or distanceMatrix is null or distanceMatrix is empty:
    Throw an exception (invalid input)

  If solution size is less than half the size of distanceMatrix:
    Throw an exception (solution size mismatch)
```

```
Initialize cost to 0

For i from 0 to solution.size() - 1:
    currentNode = solution[i]
    nextNode = solution[i + 1]

    Add distanceMatrix[currentNode][nextNode] to cost
    Add nodeData[currentNode][2] (the cost of the node) to cost

lastNode = solution[solution.size() - 1]
startNode = solution[0]

Add distanceMatrix[lastNode][startNode] to cost (to complete the cycle)
Add nodeData[lastNode][2] (the cost of the last node) to cost

Return total cost
```

- **Nearest Neighbor (at the end)**

```
function nearestNeighborEnd(distanceMatrix, startNode, nodes):

    n = length of distanceMatrix
    visited = array of false with size n
    nodesToSelect = ceiling(n / 2)
    path = empty list
    add startNode to path
    mark startNode as visited

    for i from 0 to nodesToSelect – 1:
        lastNode = last element in path
        nearestNode = -1
        minDistance = infinity
```

```
        for candidateNode from 0 to n-1:
            if candidateNode is not visited and (distanceMatrix[lastNode][candidateNode] +
nodes[candidateNode][2]) < minDistance:
                minDistance = distanceMatrix[lastNode][candidateNode] + nodes[candidateNode][2]
                nearestNode = candidateNode

        if nearestNode is not -1:
            add nearestNode to path
            mark nearestNode as visited

    return path
```

- **Nearest Neighbor (any position)**

```
function nearestNeighborAnyPosition(distanceMatrix, startNode, nodes):

    n = length of distanceMatrix
    visited = array of false with size n
    path = empty list
    add startNode to path
    mark startNode as visited
    nodesToSelect = ceiling(n / 2)

    for i from 0 to nodesToSelect - 1:
        bestNode = -1
        bestIncrementalCost = infinity
        bestPosition = -1

        for candidateNode from 0 to n-1:
            if candidateNode is not visited:
                for position from 0 to path size:
                    incrementalCost = calculateIncrementalCost(path, distanceMatrix, candidateNode,
position, nodes)
                    if incrementalCost < bestIncrementalCost:
                        bestIncrementalCost = incrementalCost
                        bestNode = candidateNode
                        bestPosition = position

        insert bestNode at bestPosition in path
        mark bestNode as visited

    return path
```

```
function calculateIncrementalCost(path, distanceMatrix, newNode, position, nodes):
  cost = 0
  previousNode = if position > 0 then path[position - 1] else last element in path
  nextNode = if position < path size then path[position] else first element in path

  cost += distanceMatrix[previousNode][newNode]
  cost += distanceMatrix[newNode][nextNode]
  cost += nodes[newNode][2]  // Adding node's cost

  if position > 0:
    cost -= distanceMatrix[previousNode][nextNode]

  return cost
```

- **Greedy Cycle**

```
function greedyCycle(distanceMatrix, startNode, nodes):

  n = length of distanceMatrix
  visited = array of false with size n
  path = empty list
  add startNode to path
  mark startNode as visited
  nodesToSelect = ceiling(n / 2)

  while path size is less than nodesToSelect:
    bestNode = -1
    bestIncrementalCost = infinity
    bestPosition = -1

    for candidateNode from 0 to n-1:
      if candidateNode is not visited:
        for position from 0 to path size:
          incrementalCost = calculateIncrementalCost(path, distanceMatrix, candidateNode,
position, nodes)
          if incrementalCost < bestIncrementalCost:
            bestIncrementalCost = incrementalCost
            bestNode = candidateNode
            bestPosition = position

    if bestNode is at least 0 and bestPosition is at least 0:
      insert bestNode at bestPosition in path
      mark bestNode as visited
  return path
```

```
function calculateIncrementalCost(path, distanceMatrix, newNode, position, nodes):
 cost = 0
 previousNode = if position > 0 then path[position - 1] else path[-1] (last node)
 nextNode = if position < path size then path[position] else path[0] (first node)
 cost += distanceMatrix[previousNode][newNode]
 cost += distanceMatrix[newNode][nextNode]
 cost += nodes[newNode][2]
 if position > 0 and path size is at least 2:
     cost -= distanceMatrix[previousNode][nextNode]
 return cost
```

Results

- **Dataset A - TSPA.csv**

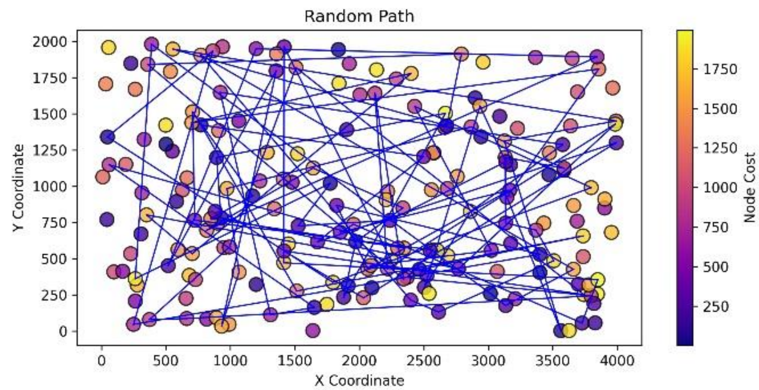
	Min	**Max**	**Average**
Random Solution	235 361.0	291 848.0	264 276.205
NN End	83 182.0	89 433.0	85 108.51
NN Any	71 488.0	74 410.0	72 635.12
Greedy	71 488.0	74 410.0	72 609.075

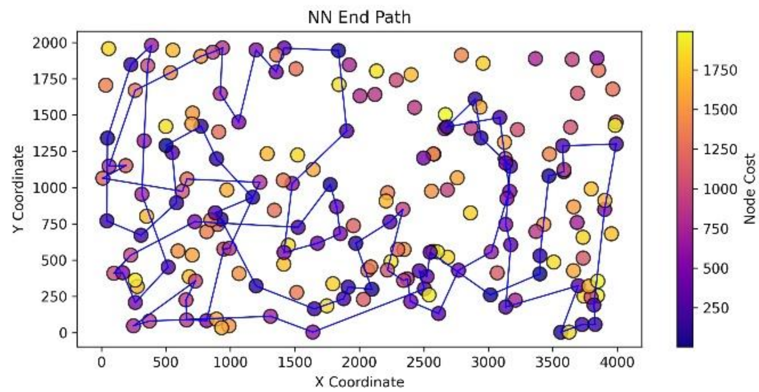
- **Dataset B - TSPB.csv**

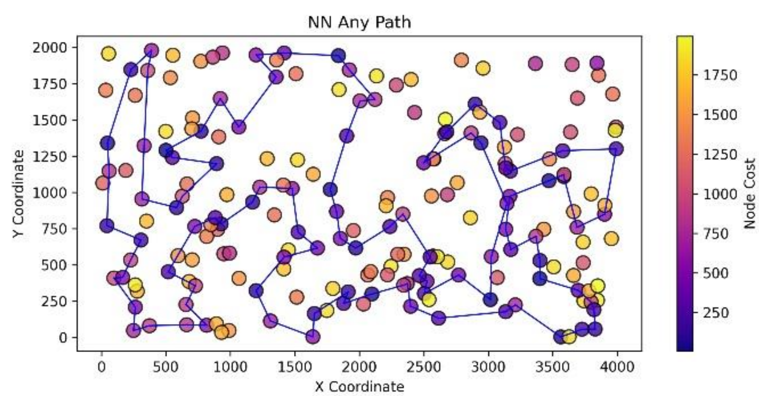
	Min	**Max**	**Average**
Random Solution	191 290.0	232 832.0	212 368.47
NN End	52 319.0	59 030.0	54 390.43
NN Any	49 001.0	57 262.0	51 397.595
Greedy	48 765.0	57 324.0	51 301.73

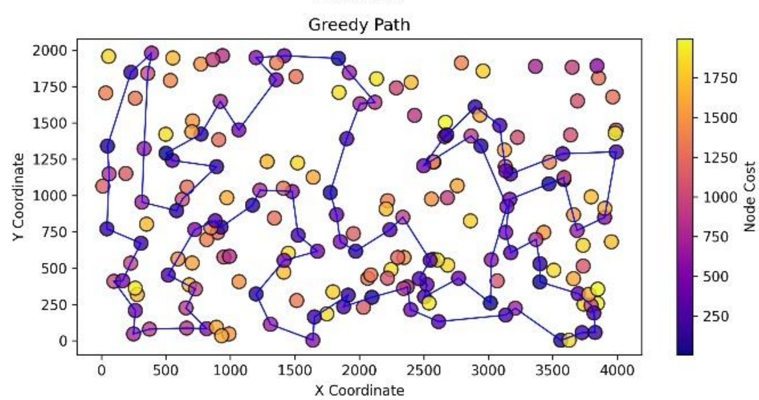
Visualizations

- **Dataset A - TSPA.csv**

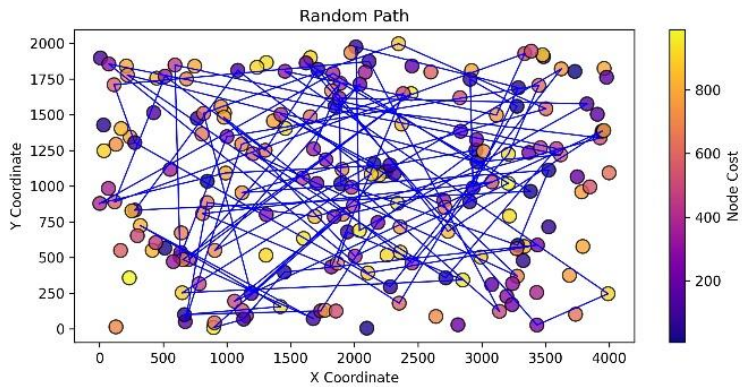


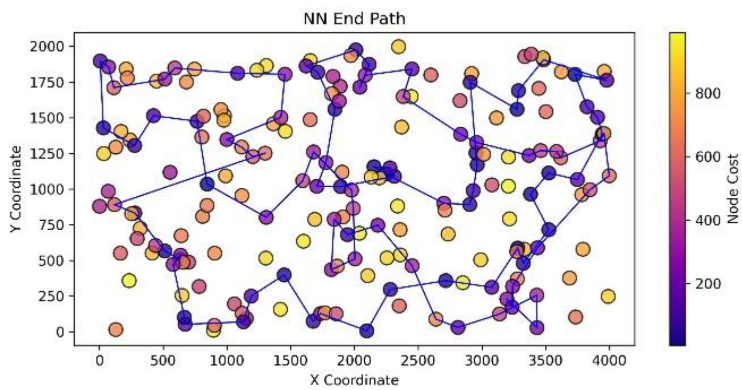


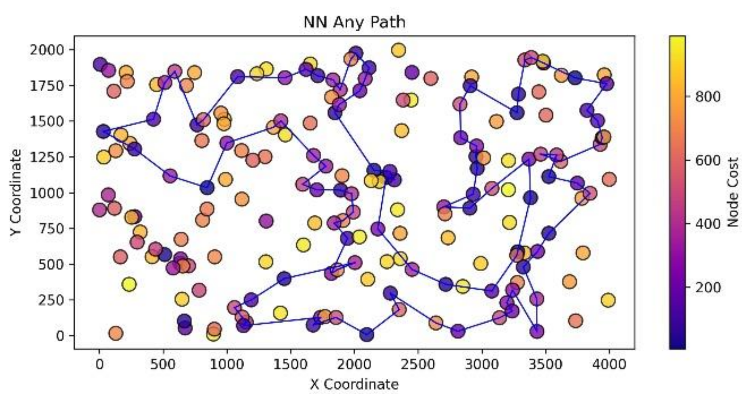


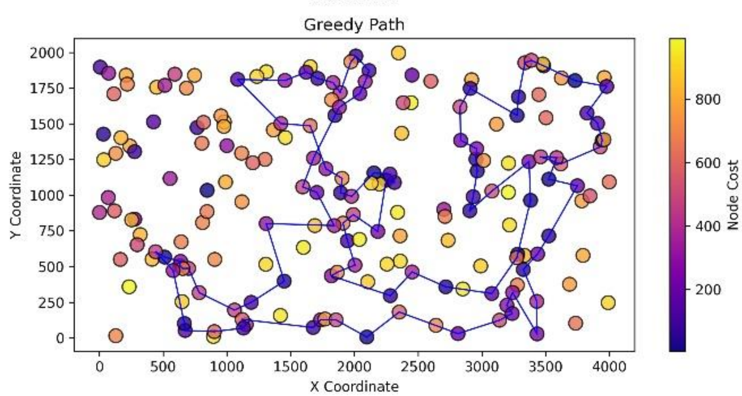


- **Dataset B - TSPB.csv**









Best solutions

- **Dataset A - TSPA.csv**

Random Solution

73,69,100,186,121,107,2,65,115,61,154,63,124,66,77,164,117,184,0,156,24,197,102,119,3,57,179,168,75,86,36,38,44,120,108,147,176,151,94,178,155,101,172,165,68,127,135,146,19,105,59,110,43,74,26,56,4,190,149,153,13,162,158,90,133,148,139,194,17,97,182,129,15,174,173,22,140,10,145,80,136,52,79,122,171,188,161,143,99,192,166,64,191,58,55,16,32,27,67,62

Nearest Neighbor (at the end)

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 (any position)

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

Greedy

183,89,186,23,137,176,80,79,63,94,124,152,97,1,101,2,120,129,55,49,102,148,9,62,144,14,178,106,165,90,81,196,40,119,185,52,57,92,179,145,78,31,56,113,175,171,16,25,44,75,86,26,100,53,154,180,135,70,127,123,162,133,151,51,118,59,65,116,43,184,35,84,112,4,190,10,177,30,54,48,160,34,146,22,18,108,69,159,181,42,5,115,41,193,139,68,46,0,117,143

- **Dataset B - TSPB.csv**

Random

188,137,93,6,174,4,16,133,74,169,119,94,45,56,46,175,177,194,58,107,22,62,72,104,128,19,21,189,98, 80,112,157,66,168,145,173,29,134,159,26,118,153,99,10,31,51,199,181,81,120,82,110,132,27,116,109,18,111,61,190,7,24,36,164,167,73,13,139,165,3,100,135,77,59,0,5,191,84,9,158,185,176,33,195,148,68,70,140,124,152,183,97,87,8,186,105,39,147,1,86

Nearest Neighbor (at the end)

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 (any position)

134,139,11,182,138,33,160,144,56,104,8,21,87,82,177,5,45,142,78,175,61,36,91,141,97,77,146,187,165,127,89,103,137,114,113,194,166,172,179,185,99,130,22,66,94,47,148,60,20,28,149,4,140,183,152,170,34,55,18,62,124,143,106,128,95,86,176,180,163,153,81,111,0,35,109,29,168,195,145,15,3,70,161,13,132,169,188,6,147,191,90,10,133,122,63,135,131,121,51,85

Greedy

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

Source code

https://github.com/senketsutsu/Evolutionary-computation

Conclusions

• The random solutions algorithm produced the worst results in all cases.

• The Nearest Neighbor algorithm performed slightly better when nodes were added at any position than when adding at the end. This improvement is due to its ability to minimize the incremental cost at each step, leading to a more efficient path formation.

• The Greedy algorithm performs similarly to the Nearest Neighbor (any position). Still, it achieves the best outcomes out of all the methods due to its systematic approach of optimizing the cycle by inserting nodes at the most advantageous positions.