EC Laboratory 1
Assignment 1: Greedy heuristics

BENGRICHE Tidiane (ER-2037), Szczublińska Joanna Wiktoria (156070)

Contents

1 Problem

We are provided with a dataset consisting of three integer columns, where each row represents a node. The first two columns specify the x and y coordinates of each node in a two-dimensional plane, while the third column indicates the cost associated with each node.

The objective is to choose exactly half of the nodes (rounding up if the total number is odd) and construct a Hamiltonian cycle, a closed loop that visits each selected node exactly once. The aim is to minimize the combined total of the path length and the cumulative cost of the selected nodes.

For this assignment, we had to implement three algorithms:

- Random solution.
- Nearest neighbor considering adding the node only at the end of the current path.
- 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.
- Greedy cycle

You can see the repository of our assignment on github : https://github.com/tbengric/EC_lab_1

2 Pseudocode

In this section, we will present the pseudocodes of the algorithms. For this assignment, we will code in C++.

2.1 Objective function

Objective function

computeObjective(path, distance matrix, nodes) :
  for each node in path:
    add the cost by looking in nodes to the total cost
    add the distance with between the node and the next one by looking in distance matrix to the total distance
  return total cost + total distance

2.2 Random

Objective function

randomSolution(selectedNodes) :
  we shuffle the selectedNodes to make a random path
  return path

2.3 Nearest Neighbor

Objective function

nearestNeighborEnd(selectedNodes, distance matrix, nodes, starting node):
  the path is initialize with the starting node
  we create a list called visited to know if each node was already visited and we initialize it with the starting node
  While the cycle is not build entirely:
    for each node in selectedNodes:
      if the node is already visited, we go to the next node
      Otherwise we calculate the score of the node
      if he is a better node, we keep him.
    we push the bestNode in the path and he is considered as visited
  return path

2.4 Nearest Neighbor Flexible

Objective function

nearestNeighborFlexible(selectedNodes, distance matrix, nodes, starting node):
the path is initialize with the starting node
we create a list called visited to know if each node was already visited and we initialize it with the starting node
While the cycle is not build entirely:
for each node in selectedNodes:
if the node is already visited, we go to the next node
for each position in path
if the position permits to optimize the path with the new node, we keep the position
we push the bestNode in the path with the best position and he is considered as visited
return path

2.5 Greedy cycle

Objective function

greedyCycle(selectedNodes, distance matrix, nodes, starting node):
the path is initialize with the starting node
we create a list called visited to know if each node was already visited and we initialize it with the starting node
While the cycle is not build entirely:
for each node in selectedNodes:
if the node is already visited, we go to the next node
for each node in path
we look the the distance between the node in path with the new node in path and the distance between the next node in path with the new node
we add them, then we substract the distance between the node in path with the next node in path
then we add the cost of the new node to finally make the score
if the score is better, we keep him
we push the bestNode in the path with the best position and he is considered as visited
return path

3 Results

After trying these differents algorithms, we will check their effectiveness by showing the constructed cycle.

In order, we will start from the algorithm that seems least efficient to the one that shows the best results.

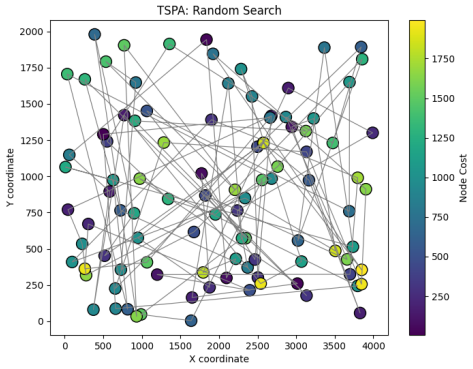

Figure 1: Random Search TSPA

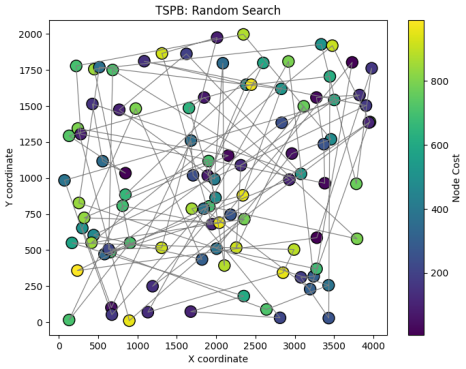

Figure 2: Random Search TSPB

For the random algorithm, we quickly realize its inefficiency because the cycle is cut everywhere and there is no cost optimization.

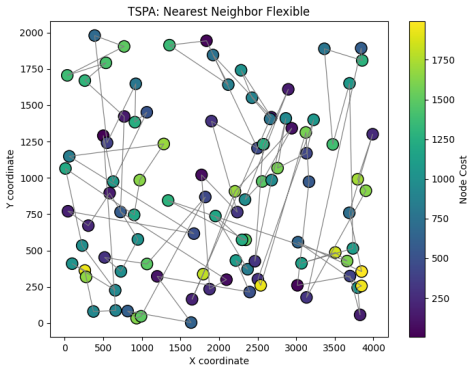

Figure 3: NN Flexible TSPA

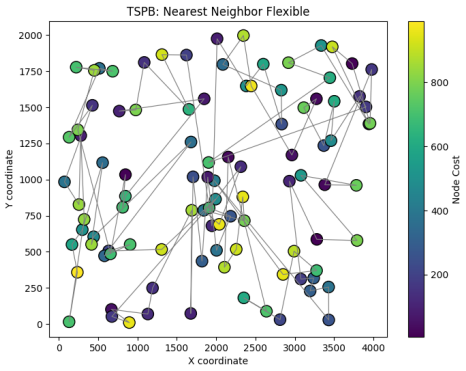

Figure 4: NN Flexible TSPB

For the nearest neighbor algorithm which takes into account all possible positions for the construction of the path, we also realize its inefficiency (a little less important than the random one) because certain paths also intersect.

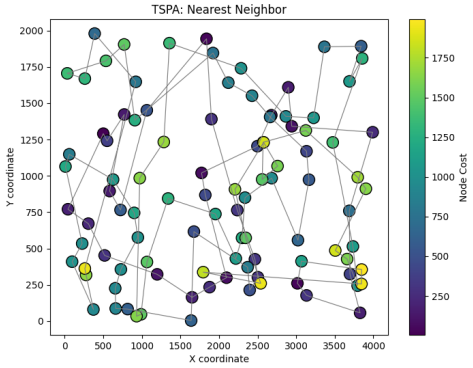

Figure 5: NN TSPA

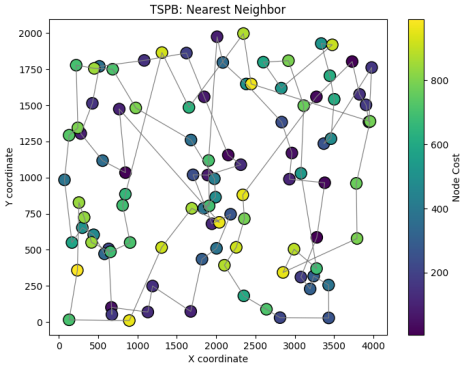

Figure 6: NN TSPB

For the normal nearest neighbor algorithm, the result is better, although there are still some crossing paths.

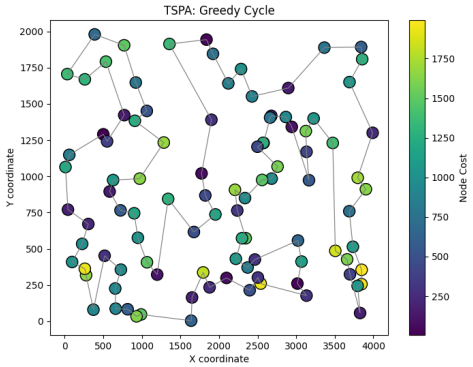

Figure 7: Greedy Cycle TSPA

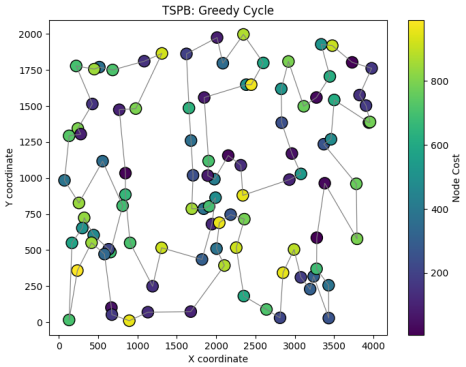

Figure 8: Greedy Cycle TSPB

For the Greedy Cycle, the result is better than the other algorithms with no crossing paths here.

Method	Avg (Min, Max)
Random Search	258228.79 (234698, 278190)
Nearest Neighbor End	139338.46 (136498, 143678)
Nearest Neighbor Flexible	161501.76 (151624, 183074)
Greedy Cycle	117655.33 (116083, 119349)

Table 1: Average, minimum, and maximum objective values for TSPA

Method	Avg (Min, Max)
Random Search	212741.68 (193021, 235886)
Nearest Neighbor End	88678.09 (85775, 90786)
Nearest Neighbor Flexible	116182.93 (103382, 136700)
Greedy Cycle	74507.47 (73640, 75453)

Table 2: Average, minimum, and maximum objective values for TSPB

Thanks to these tables, we can see that the Greedy Cycle is more accurate and does not move away towards distant values unlike others algorithms.

4 Conclusion

Among these algorithms, the one that is significantly stronger than the others is Greedy Cycle.

This assignment allowed us to see the difference between the different search algorithms in order to introduce the next concepts that we will see in class. By analyzing their performance, strengths, and limitations across different scenarios, we gained a deeper understanding of how algorithmic efficiency impacts problem-solving in computer science.

⋆ ⋆ ⋆