Greedy heuristics for TSP problem

Julia Lorenz 156066, Marcel Rojewski 156059

1 Problem Description

The objective of the task was to optimize the cost of traversing through a cycle of nodes using different greedy heuristic methods. Given a set of 200 nodes, our task was to choose exactly half of the nodes that create a Hamiltonian cycle, where the total sum of Euclidean distances between nodes and the total costs of selected nodes was to be minimized.

2 Methodology

The task was carried out using four methods: Random Solution, Nearest Neighbor with node insertion only at the end of the current path, Nearest Neighbor allowing insertion at any position in the path, and Greedy Cycle. Their performances were later compared with one another.

2.1 Random solution

The first implemented procedure was the random cycle construction. It iteratively selected random, non-repeating nodes from the set of all available nodes and added them to the cycle until the desired number of nodes comprising the cycle was reached.

Algorithm 1: Random Construction of a Hamiltonian Cycle

Result: Randomly constructed Hamiltonian Cycle.

Main procedure:;
$targetCount \leftarrow \lceil |N|/2 \rceil$;
Initialize $visited$ array with **false** for all nodes;
Initialize empty array $cycle$ of size $targetCount + 1$;
$nodeCount \leftarrow 0$;
while $nodeCount < targetCount$ **do**
  Select $randomIndex$ randomly from $[0, |D| - 1]$;
  if $visited[randomIndex] =$ ***false*** **then**
    $cycle[nodeCount] \leftarrow randomIndex$;
    $visited[randomIndex] \leftarrow$ **true**;
    $nodeCount \leftarrow nodeCount + 1$;
  end
end
$cycle[targetCount] \leftarrow cycle[0]$;
return `Solution`$(cycle)$;

2.2 Nearest Neighbor added only at the end

Another method used to find the optimal solution was the Nearest Neighbor approach, where a new node could only be added at the end of the current path. The idea was simple: we first set the first and last elements of the path as the starting node. Then, until the path was complete, we repeatedly selected the next node as the neighbor of the current end node that added the smallest possible sum of (distance + cost) to the total score.

Algorithm 2: Nearest Neighbor considering adding new node at the end

Data: Objective matrix O, start node s
Result: Found Hamiltonian Cycle minimizing total length and total cost
Function `FindNearestNeighbor` ($currentNode$, $visited$) **:**
    $nearestNeighbor \leftarrow -1$, $minScore \leftarrow \infty$;
    for $i \leftarrow 0$ **to** $|O| - 1$ **do**
        if $\neg visited[i]$ ***and*** $O[currentNode][i] < minScore$ **then**
            $minScore \leftarrow O[currentNode][i]$;
            $nearest \leftarrow i$;
        end
    end
    return $nearest$;

Main procedure:;
$targetCount \leftarrow \lceil (|D| - 1)/2 \rceil$, $path[0] \leftarrow s$, $path[targetCount] \leftarrow s$,
$visited[s] \leftarrow$ **true**, $current \leftarrow s$, $nodeCount \leftarrow 1$;
while $nodeCount < targetCount$ **do**
    $next \leftarrow$ `FindNearestNeighbor` ($current$, $visited$) ;
    $path[nodeCount] \leftarrow next$;
    $visited[next] \leftarrow$ **true**;
    $current \leftarrow next$;
    $nodeCount \leftarrow nodeCount + 1$;
end
return `Solution`($path$)

2.3 Nearest Neighbor added anywhere

Additionally, we considered a variation of the previous method that allowed new nodes to be added not only at the end of the current path but at any position within it, including the start or end.

Algorithm 3: Nearest Neighbor considering adding new node at any place

Data: Objective matrix O, start node s
Result: Found Hamiltonian Cycle minimizing total length and total cost
Function `FindNeighborAndPosition(`$path$, $visited$`)` **:**
    $bestNode \leftarrow -1$, $bestPos \leftarrow -1$, $minScore \leftarrow \infty$;
    for $node \leftarrow 0$ **to** $|O| - 1$ **do**
        if $\neg visited[node]$ **then**
            for $pos \leftarrow 0$ **to** $|path| - 1$ **do**
                $tempScore \leftarrow O[pos-1][node] + O[node][pos]$;
                $tempScore \leftarrow tempScore - O[pos-1][pos]$;
                if $tempScore < minScore$ **then**
                    $minScore \leftarrow tempScore$;
                    $bestNode \leftarrow node$;
                    $bestPos \leftarrow pos$;
                end
            end
        end
    end
    return (`index=` $bestPos$, `nodeId=` $bestNode$);

Main procedure:;
$targetCount \leftarrow \lceil (|D| - 1)/2 \rceil$, $path \leftarrow [s]$, $visited[s] \leftarrow$ **true**, $nodeCount \leftarrow 1$;
while $nodeCount < targetCount$ **do**
    $(pos, nextNode) \leftarrow$ `FindNeighborAndPosition(`$path$, $visited$`)` ;
    `Insert` $nextNode$ at position pos in $path$;
    $visited[nextNode] \leftarrow$ **true**;
    $nodeCount \leftarrow nodeCount + 1$;
end
`Insert` $path[0]$ at end of $path$;
return `Solution(`$path$`)`

2.4 Greedy cycle

The fourth and final method we implemented was the Greedy Cycle Construction Heuristic, where the cycle is iteratively increased by adding a node in a place where it results in the smallest increment to the objective value function. The construction is finished when there is a desired number of nodes in the cycle.

Algorithm 4: Greedy Cycle Construction

Result: Greedy constructed partial Hamiltonian cycle
Input: Objective matrix O, nodes N, start node $start$;

Main procedure:;
$targetCount \leftarrow \lceil |N|/2 \rceil$;
Initialize $visited$ array with **false** for all nodes;
Initialize $unvisitedIds \leftarrow$ set of all node IDs;

Mark $start$ as visited: $visited[start] \leftarrow$ **true**;
Remove $start$ from $unvisitedIds$;
$nearestNode \leftarrow$ `FindNearestNeighbor`($start$, $visited$, O);
Mark $nearestNode$ as visited and remove from $unvisitedIds$;
Initialize $cycle \leftarrow [start, nearestNode, start]$;
$nodeCount \leftarrow 2$;

while $nodeCount < targetCount$ **do**
    $bestNode \leftarrow$ `null`;
    $bestPosition \leftarrow$ `null`;
    $bestIncrement \leftarrow +\infty$;
    foreach *candidate in* $unvisitedIds$ **do**
        for $i \leftarrow 0$ **to** $|cycle| - 2$ **do**
            $curr \leftarrow cycle[i]$;
            $next \leftarrow cycle[i+1]$;
            $increment \leftarrow O[curr][candidate] + O[candidate][next] - O[curr][next]$;
            if $increment < bestIncrement$ **then**
                $bestIncrement \leftarrow increment$;
                $bestNode \leftarrow candidate$;
                $bestPosition \leftarrow i + 1$;
            end
        end
    end
    Insert $bestNode$ into $cycle$ at $bestPosition$;
    Mark $bestNode$ as visited and remove from $unvisitedIds$;
    $nodeCount \leftarrow nodeCount + 1$;
end
return $cycle$

3 Results

For each method, we measured the time required to find the optimum and evaluated the quality of the obtained solution. Additionally, we visualized the best solutions found by each method to enable visual comparison of their quality. Each best solution was extracted as a list of nodes and then verified using the solution checker to confirm the correctness of the obtained optimum value.

3.1 TSPA

Method	Min (s)	Max (s)	Avg (s)
Random	0.00000	0.00612	0.00004
NN end	0.00001	0.00377	0.00006
NN any	0.00427	0.07050	0.00574
Greedy	0.00136	0.03132	0.00216

Table 1: Comparison of computation times to solve TSPA problem for all methods

Method	Min	Max	Avg
Random	237845	297860	263795.97
NN end	83182	89433	85108.51
NN any	71179	75450	73178.55
Greedy	71488	74410	72646.375

Table 2: Comparison of value of objective function for all methods

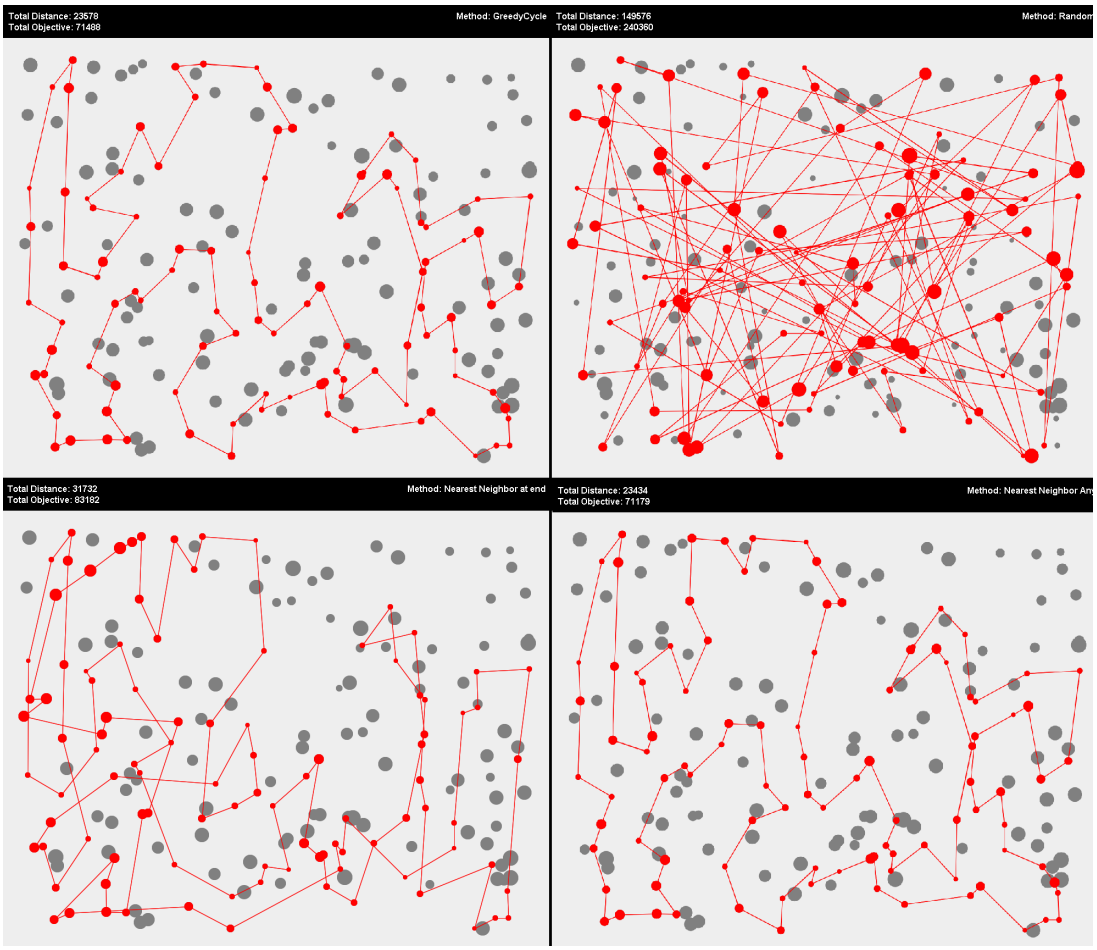

Figure 1: **Best Solution**. The figure depicts best path found by each method for the TSPA problem. From the left to right, and from up to down the methods are: Greedy Cycle, Random solution, Nearest Neighbor, where new node is only added at the end of current path, Nearest Neighbor, where new node can be added at all positions

3.2 TSPB

Method	Min (s)	Max (s)	Avg (s)
Random	0.00000	0.00394	0.00003
NN end	0.00002	0.00502	0.00008
NN any	0.00424	0.07200	0.00563
Greedy	0.00138	0.03287	0.00213

Table 3: Comparison of computation times to solve TSPB problem for all methods

Method	Min	Max	Avg
Random	190037	235290	213579.86
NN end	52319	59030	54390.43
NN any	44417	53438	45870.254
Greedy	49001	57324	51400.64

Table 4: Comparison of value of objective function for all methods

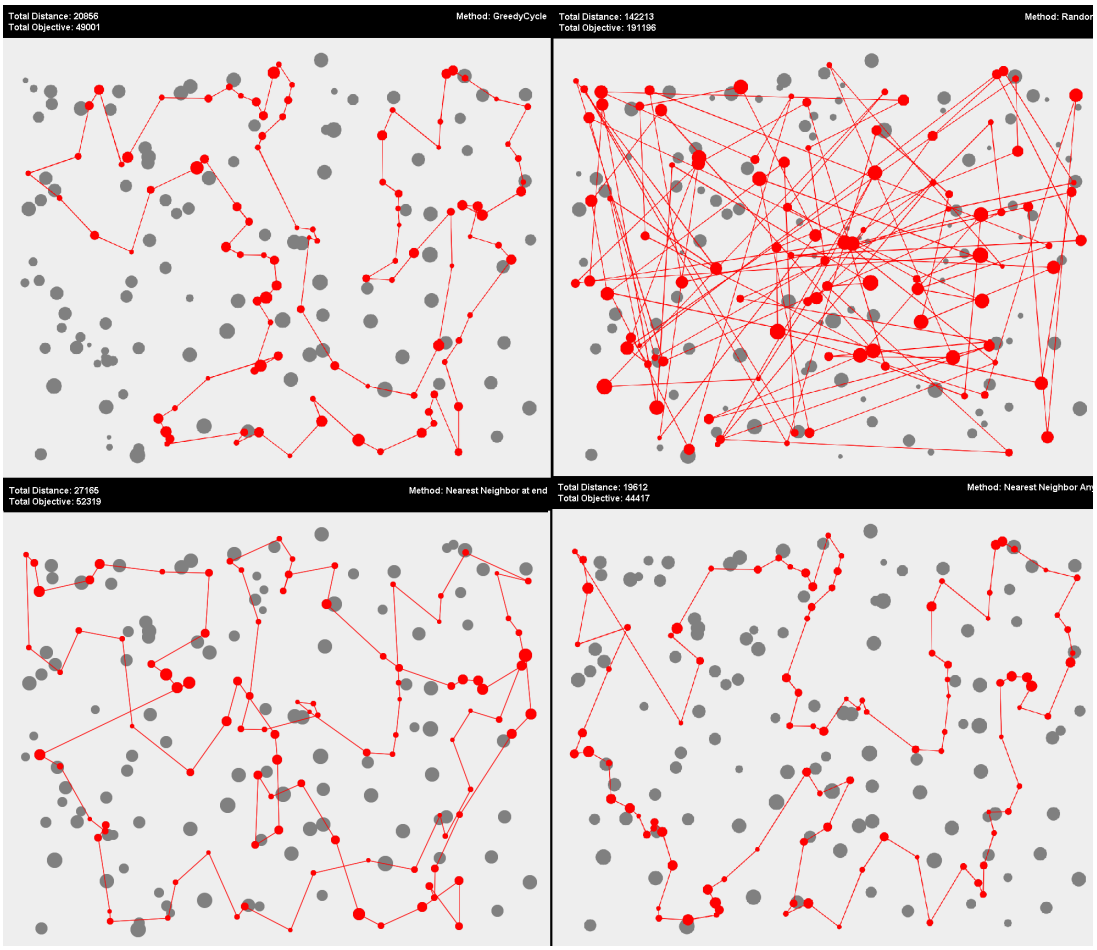

Figure 2: **Best Solution**. The figure depicts best path found by each method for the TSPB problem. From the left to right, and from up to down the methods are: Greedy Cycle, Random solution, Nearest Neighbor, where new node is only added at the end of current path, Nearest Neighbor, where new node can be added at all positions

4 Conclusion

We observed that implementing a simple greedy heuristic, such as the basic variant of the Nearest Neighbor algorithm, can significantly improve the objective metric. In general, more complex heuristics tended to yield better results. However, this came at the cost of increased inference time, though in our case, the impact was minimal due to the relatively small problem size. It is also worth noting that the performance of a method depends not only on its implementation but also on the dataset. For instance, the ”NN any” method performed best on average for the TSPB dataset, while the ”Greedy Cycle” heuristic achieved the best results on TSPA.

Code Availability

The source code for all experiments and heuristics presented in this report is available at:

`https://github.com/julkalorenz/EC`

Solution Checker

Best solutions have been checked with the solution checker.