CSE 624/424
Heuristic Optimization

All Homeworks

Muharrem Ozan Yeşiller
171044033

Problem Definition

- Minimum Spanning Tree (MST) Problem:

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, **without any cycles and with the minimum possible total edge weight**. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

- Sorting Problem:

Sorting is the process of placing elements from a collection in **some kind of order.** Like searching, the efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it is worth.

Solution Format

- Minimum Spanning Tree (MST) Problem:

The solution is an array. Contains the edges of the spanning tree with the minimum cost in the array.

For instance:

```
[
    [sourceY, destinationY, costY],
    [sourceX, destinationX, costX],
    ...
]
```

- Sorting Problem:

The solution is an array in which all numbers are **ordered from least to greatest.**

Decision Variables:

- Minimum Spanning Tree (MST) Problem:

$$\left[E_1, E_2, E_3, E_4 \dots E_n\right]$$

n = Total Edge Number of Graph.

- Sorting Problem:

$$\left[x_1, x_2, x_3, x_4 \dots x_n\right]$$

n = Total Size of Array

Constraints:

- Minimum Spanning Tree (MST) Problem:

The feasible solution is that it contains a non cyclic subgraph and the feasible subgraph's edges are in the real graph. The number of edges should be at least as much as (vertex-1).

A spanning tree G' of G

$$G' = (V, E') \quad \forall E' \subseteq E$$

- Sorting Problem:

All arrays in the result space must have the same size as the original array. Because the selected decision variables can be ordered, they must contain the entire original sequence.

For instance:

Assume that the original array is [3, 4, 5, 8, 1]

The solution must be [1, 3, 4, 5, 8]. Cause of [3, 4, 5] is a sorted subarray but it is not a feasible solution therefore the subarray does not contain all original array decision variables.

Objective Function:

- Minimum Spanning Tree (MST) Problem:

Sum of all edge costs in G'

$$C(G') = \sum_{e \in E}(C(e))$$

The optimum solution minimizes this objective function equation, so minimum cost spanning tree.

- Sorting Problem:

According to my design, the objective function should be: The more ordered the array, the more optimal the solution. That is, the sum of all numbers that break the order should be minimized.

$$count(\forall((array_i < array_j \Leftrightarrow (i > j))) + count(\forall((array_i > array_j \Leftrightarrow (j > i)))$$

(Inversion Count)

This sum should be minimized to reach an optimal solution.

Particle Swarm Optimization:

Each individual who is looking for a solution in the pso is called a particle, and the population in which the particles are located is called a herd.
To understand how close an individual is to the solution, the fitness function is used. This function can be a function that evaluates the suitability of the solution taking into account the total value of the selected loads, if our example of a freighter is taken into account. The main purpose of this function is to measure how close we are to the actual solution.
The best state of a particle at the moment when it is closest to the solution during the dec solution search is called pbest, and the current state of the particle that is closest to the solution during the entire search is called gbest in the entire version.

In the PSO, first of all, the herd that will search for the solution and the necessary parameters are determined. With the help of the conformity function, the proximity of the particles to the solution is measured and the pbest and gbest values are updated according to these values. Then, with the change rate function, the movement of each particle is determined and its new states are set. It is checked how close the solution is to the solution with the repeat fit function. This cycle is repeated until the desired conditions are reached.

The formula for the rate of change of particles:
x: particle value
v : the rate of change of the particle,
c1,c2 : constant values,
rand1, rand2: randomly generated values,
pbest : the situation where the particle comes closest to the solution
gbest: it is calculated by the following formula, which is the most approximate state of decolonization among all particles.

$$v_{i+1} = v_i + c_1 * rand_1 * (pbest - x) + c_2 * rand_2 * (gbest - x)$$

The particles reach their new position as follows:

1) First of all, the parameters are determined and the rate of change for each particle is found. (Because I use Discrete PSO, I use the objective function output of these solutions instead of gbest, pbest and x)
2) The swap operation takes place until the speed values.

Algorithm:

Init parameters
Inıt population
Loop:
    Each Particle Loop:
        Update Local Best
        Update Global Best
        Update Velocity Vector
        Calculate fitness function
    Update Each Particle Position

Ant Colony Optimization:
Tabu Search Solution:

First, a random feasible solution is determined. A candidate solution space is created, which are neighbors of the current solution. According to my design, this neighborhood is a swapping of an element (sorting) and an edge swapping (MST). The candidate list does not contain the elements in the tabu list, but it can contain a solution that will improve the solution with aspiration.

$$s' \in N(s) = \{N(s) - T(s)\} + A(s)$$

Then the best one from the list of candidates we have is selected, then the tabu list and the current solution are updated. As the tabu tenure I take a static number as the square root of the number of elements and decrease by 1 the value mapped by each element in a tabu list in each iteration. If there is a solution that stays in iterations as many as the number of tenures, I delete it from my tabu list, that is, in the hashmap structure. (solution —> tenure)

Algorithm:

```
Current Solution <- Random construct
Iteration <- 100
Tabu_tenure <- sqrt(len(Current Solution)
tabuList <- empty Hashmap
Best Solution <- Current Solution
Loop i in 0 to Iteration:
        Candidate List <- return all neihgboor (Current Solution)
        New Solution <- []
        Candidate List <- (Candidate List - Tabu List) + Aspiration Set
        New Solution <- Best of Candidate List
        improve(New Solution)? :- then Best Solution set with New Solution
        Current Solution <- New Solution
        Update Tabu List

return Best Solution
```

Genetic Algorithm Solution:

When using this algorithm, a feasible random solution is built first. It is then put into a loop, which is the generation cycle. In each new generation cycle, new populations take place. First, the set of several feasible solutions we construct is a population. For sorting among this population, the best solution is looked for and if the cost is 0, this array is returned. This is purely for optimizing complexity. In general, the most cost-effective solution is found so that if my generation cycle ends, I can return the best solution. After the best solution is found, the next generation population is created and the following steps are followed:

1) First of all, they contribute to the population for each parent crossover operation. Here I divided the population in half and performed the crossing over operation so that the first element is associated with the last element and the second element is associated with the second element from the end.

2) After generating as many child results as the number of parents from here, I decided whether there will be a mutation or not. According to my design, there will be 25% mutation. My mutation rate is twenty-five percent.

3) Then I created a gene pool with mutated/non mutated children and parents. In total, they take up as much space in this pool as the number of members in the 2 * population.

4) Then I eliminated half of them by natural selection, moved to the 2nd generation with the 6 best-cost ones, and my loop went like this.

Algorithm:

Population size = 6
Current solution <- random construct * Population size
Iteration <- len(Current solution) * 50
Loop (i to Iteration):
    Find best solution in the current population space
    Set and save best solution
    Crossing Over population (n point crossing over) (i used 2 instead of n)
    Mutation new children population (%25 chances) (swap mutation)
    Natural selection of new children and parent population (generate new population)
Return best solution

Simulated Annealing Solution:

There is a T value that we have at the beginning, this value means the tolerance that we will show for the error. The existence of this DEC value is due to the following reason: when we search locally, we can find local minimum or maximum points. Sometimes we have to ignore the margin of error to reach the global. While this value is a positive value at the beginning, this value approaches zero with each step.

Neighborhood Design:
For MST: here's how to find a possible solution by changing the edge (edges that we don't use) from a neighborly feasible solution (edge array).
For Array: an element swapped with an element is a neighbor. (It is the narrowest neighbor interval.)

First, we select a random neighbor, and compare the current solution and this neighbor solution. If the current solution is worse than the neighbor, the current solution will now be the neighbor. If the current solution is good, but we have a certain margin of error (T Value), this time we will change the current solution again. The probability of realization of the second state varies depending on the fact that the value of e^((Delta)/T) is greater than a random value that will come between 0 and 1. Decoupling the value of e^((Delta)/T) is possible.

This loop continues until a certain iteration counter. And we call it shit criteria.

Algorithm:

```
Current solution <- random construct
T = (0.3) * len(solution)
Iteration = 100
Loop (i to Iteration):
        Algorithm Loop:
                Temp Solution <- get random neighbor
                if cost(Current Solution) > cost(Temp Solution)
                        Current Solution <- Temp Solution
                else if randnumber in [0, 1)  <  exp((cost diff current and temp) / T)
                        Current Solution <- Temp Solution
                If iterator == len(CurrentSolution) break
                Else Iterator++
        T *= 0.99 and check iterator is 100 halt the algorithm
```

Complexity

Total Complexity (For sorting) =
O(iter count) x O(len(solution))^2

Total Complexity (For MST) =
O(iter count) x O(len(solution))^2 x O(Vertex)^3 (Circle Detect Algorithm Extras)

VNS Solution:

At the beginning, we randomly create a feasible set of solutions and have a max level value. This max level value is half of the total array.

A loop iterates from 0 to max level. It undergoes a shake process at every step. This process is actually the process of randomly choosing a neighbor. I mentioned my neighbor's design in the Simulated Annealing section. Another feature of the shake process is that it takes the current level information and expands the neighboring environment to the level information. For example, when level 1 is 1, the difference in 1 element in the solution set is level 2, and the difference in 2 elements in the solution set is returned. This step gives us a Stochastic solution. With this Stochastic solution, we obtain a Deterministic solution by working with our local Search algorithm. If our Deterministic solution is more optimized (if its cost is less), it is now considered our current solution and undergoes the same operations again (up to the level). After that, the existing solution is returned.

Algorithm:

```
Current Solution <- random construct
Level Max Value <- len(Current Solution) / 2
Loop Level from 0 to Level Max Value:
        Stochastic Solution <-  shake (Current Solution, level)
        Deterministic Solution <- local search (Stochastic Solution)
        If (cost(Stochastic Solution) > cost(Deterministic Solution):
                Current Solution = Deterministic Solution
                Level = 0
Return Current Solution
```

Complexity

Total Complexity (For sorting) = O(N^3) x O(Level Max)
Total Complexity (For MST) = O(E^3) x O(Level Max)

Brute Force Solution:

With this technique, all feasible possibilities are calculated.

- Minimum Spanning Tree (MST) Problem:

For the MST problem, all (vertex size) elements up to the permutation of the edges are found in a result space. Cyclic substrates are removed from this result space.

Algorithm:

Step 1 - Assume that vertex size is n
Step 2 - Insert all n elements permutation of edges in solution space if the sub tree does not contain a cyclic edge.
Step 3 - Calculate and compare all subgraphs in solution space and return subgraphs that have minimum cost.

Complexity

N = Edge size
Theta(N!) -> permutation of N elements array does not contains circle
Theta(n) -> insert solution to the solution space
Theta(n) -> traverse and calculate cost array for minimum cost
Total Complexity = Theta(n) + Theta(N! X n) = Theta(N! X n)

- Sorting Problem:

For the sorting problem, all n elements permutation are found in a space space. Any array that disrupts the order, does not insert solution space by program.

Algorithm

Step 1 - Insert all n elements permutation of sub array in solution space if the subarray does not disrupt the order.
Step 2 - Calculate and compare sub arrays in solution space and return subarray that minimized sorted cost.

Complexity

N = Array size

Theta(N!) -> permutation of N elements array

Theta(n) -> insert solution to the solution space

Theta(n) -> traverse and calculate cost array for sorted situation.

Total Complexity = Theta(n) + Theta(n x N!) = Theta(n x N!)

Backtracking Solution:

- Sorting Problem:

Unlike the Bruteforce approach; Bruteforce fills and checks the solution array, but backtracking checks for the correct path before each fill. If it is in a wrong step, that is, it will put 1 at the end of a sequence with [2, 3], it goes to the previous steps with backtracking and corrects the error. The important detail here is that the elements of the visited array are kept in a boolean table. If the number we are on is wrong, we indicate that we came here, but we will not come again for this subproblem and subproblems that extend from this subproblem.

Algorithm:

Step 1 - Loop each element array

Step 2 - Check this element feasible or not for solution

Step 3 - If the element is feasible for solution recursive calls to go to next subproblem

If the element is not feasible, interrupt this subproblem and will extend any subproblem and go to the other subproblem via backtracking.

Step 4 - If the solution is a n element array then return this solution.

Complexity

N = Array size

O(N!) -> permutation of N elements array

O(1) -> check the element non feasible for solution

```
        if(len(solution) > 0):
            if(solution[len(solution) - 1] > arr[idx]):
                interrupt_subproblems = True
                continue
```

O(n) -> traverse and calculate cost array for sorted situation.

Total Complexity = O(n) + O(N!) = O(N!) (This is worst case situation)

Best case if the array is sorted -> O(n)

Branch and Bound Solution:

- **Minimum Spanning Tree (MST) Problem:**

Greedy Solution:

- **Minimum Spanning Tree (MST) Problem:**

While solving the MST problem with the greedy approach, the edge with the least cost is selected at each step, but while this edge is selected, the union-find algorithm is used to check the feasible state (i.e. whether there is a circle or not). It is implemented just like the Kruskal Algorithm, which we examined in the lecture.

Find Union Algorithm:

```
# find the vertex from disjoint set
def find_VertexSet(disjoint_set, idx):
   while (disjoint_set[idx] != idx):
       idx = disjoint_set[idx]
   return idx
# union source index and dest index in disjoint set
def union(disjoint_set, s_idx, d_idx):
   source = find_VertexSet(disjoint_set, s_idx)
   dest = find_VertexSet(disjoint_set, d_idx)
   disjoint_set[source] = dest
```

Algorithm:

Step 1 - Sort all the edges in non-decreasing order of their weight.
Step 2 - Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
Step 3 - Repeat Step 2 until there are (V-1) edges in the spanning tree.

Complexity

E = Edge size
V = Vertex size
O(E x logE) -> sorting of edges
O(E x logV) -> iterate all sorted edges with find-union algorithm
Total Complexity = O(E x logE) + O(E x logV).

- **Sorting Problem:**

While solving the sorting problem as Greedy, the first smallest element was searched greedily at each step. The smallest value obtained with this greedy approach is replaced at each step with the referenced index. Just like Selection Sort. Selection sort is also an algorithm implemented with the Greedy Approach.

Algorithm:

Step 1 - All indices are considered by going from the first element to the last element.
Step 2 - Going to the end of all these considered reference indices, the smallest element is found and replaced with the number in this index.

Complexity

N = Array size
O(N) -> iterate on all elements in array
    O(N) -> choose the minimum element forward from previous step indices
Total Complexity = O(N^2).

Local Search:

- Minimum Spanning Tree (MST) Problem:

Edges that do not draw at least (vertex-1) circles are collected in a set and the remaining elements are kept in a buffer. Replacing each element in our main set with each element held in the buffer will be a neighbor. Among these neighbors, the most optimized result is return, and in the next step, the same application is made to this side. If the total cost of the rotating edge set is greater than the cost of the previous set of edges, the previous set of edges is considered a solution.

Algorithm:

Step 1 - Randomly min(vertex-1) element edge set construct
Step 2 - Solution is not feasible then again construct until solution is a feasible
Step 2 - Calculate Cost.
Step 3 - Find a minimum neighbor and compare cost. If the returned set cost greater than previous set cost then return previous array as a solution

Complexity

V = Vertex size
E = Edge size (solution set contains)
O(E) -> Generate random edge set with an element. Worst Case O(E), Best caseO(V-1)
O(E^3) -> Control the feasible solution.
O(E) -> Calculate Cost
O(E^2) -> Find minimum neighbor
Total Complexity = O(E^3).

- Sorting Problem:

When solving the problem with local search, the array is shuffled (randomly) first. Then, assuming that each element in this shuffled array is replaced by an element, a set of neighbors is found. It takes the one with the lowest cost (calculates the cost function) from this set of neighbors and does the same operation again. The important part here is that the loop is either broken when the value is the least, or if the least found neighbor increases the cost.

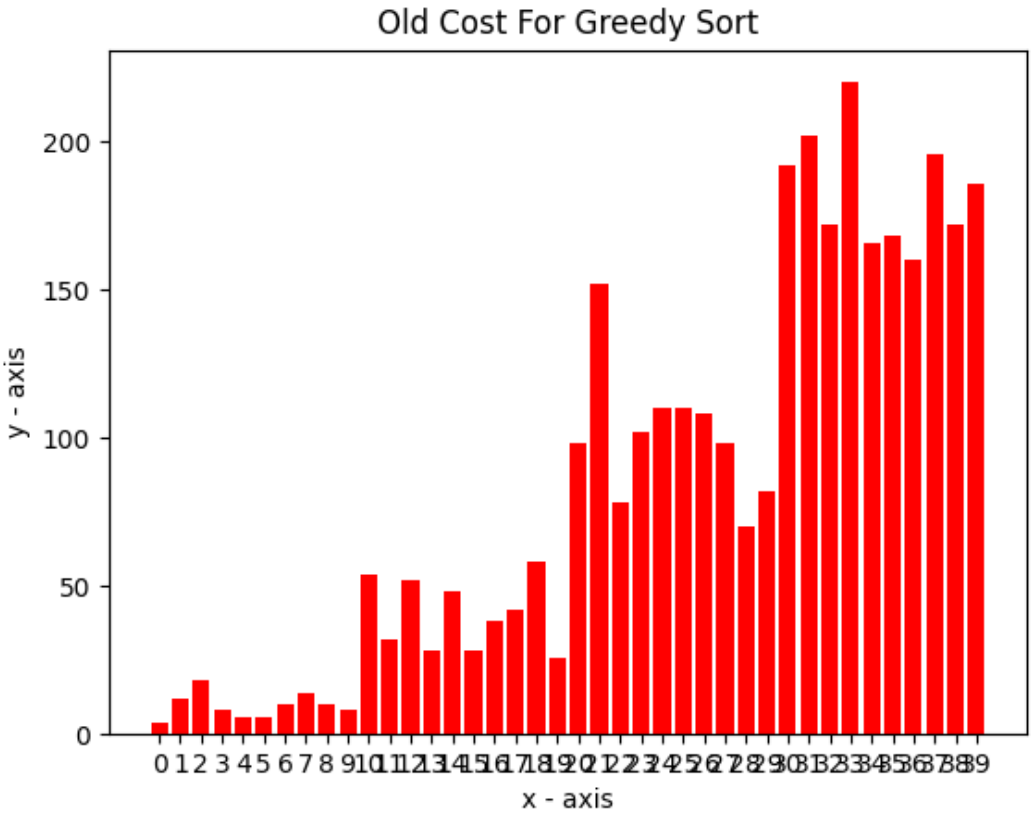
Algorithm:

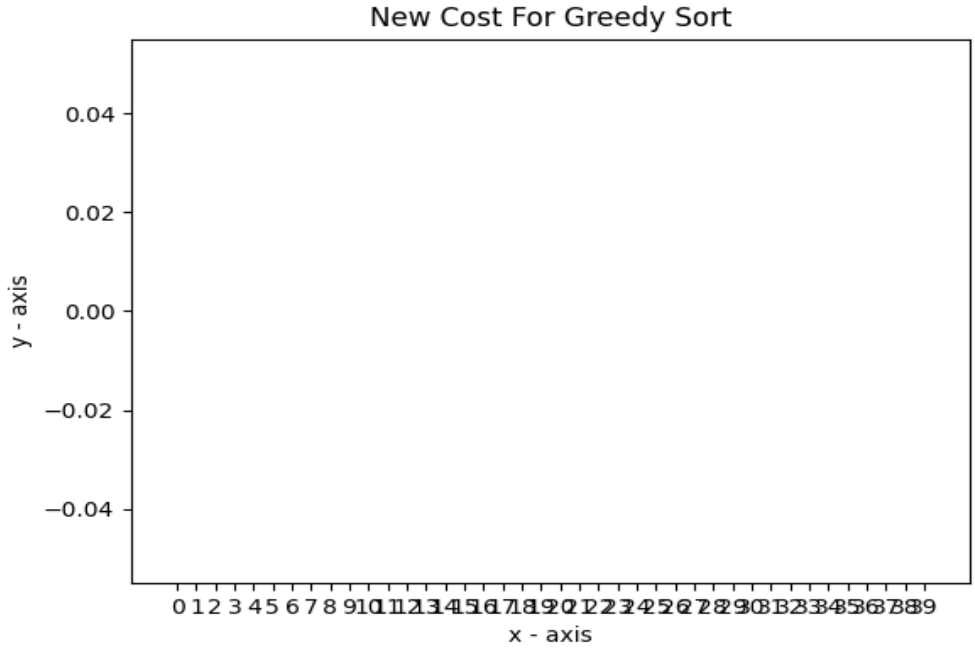
Step 1 - Shuffle array. Randomly construct.
Step 2 - Calculate Cost.
Step 3 - Find and Return minimized neighbor
Step 4 - If the returned array cost is zero then break loop and return this array
If the returned array cost greater than previous array cost then return previous array as a solution

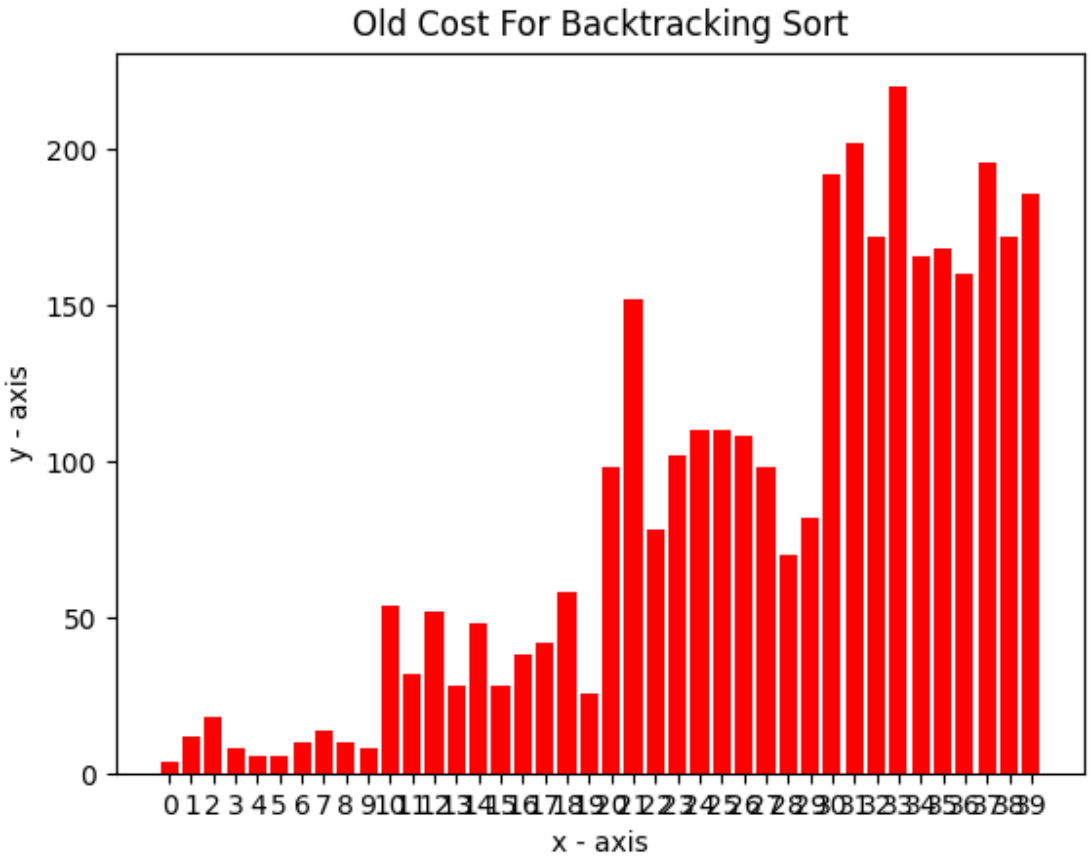
Complexity

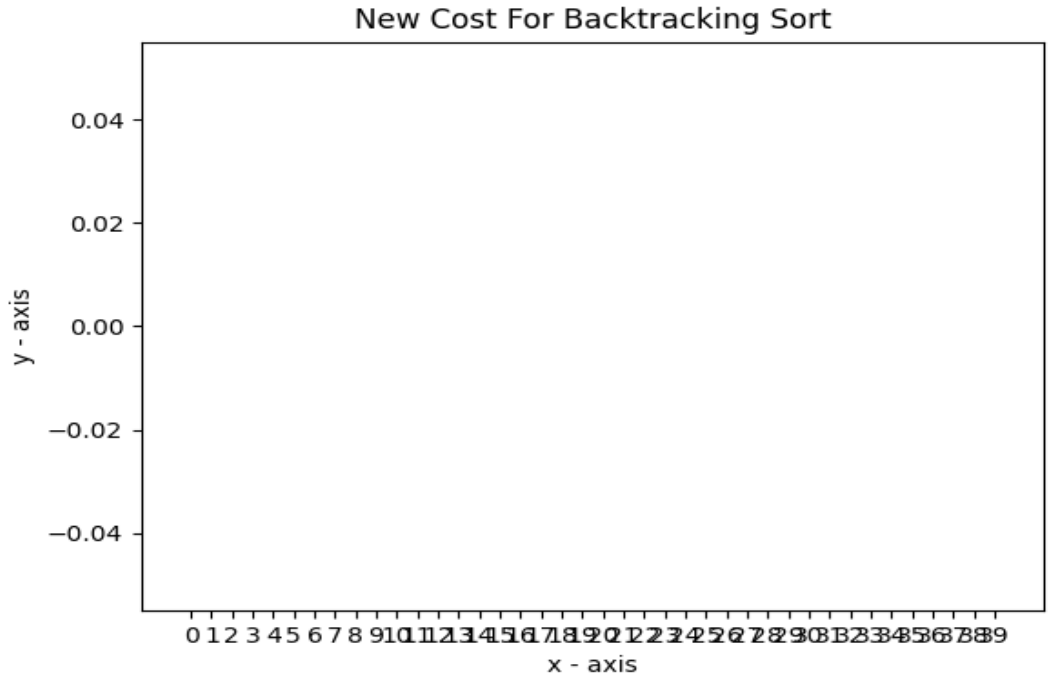
N = Array size
O(N) -> Shuffle array.
O(N^2) -> swap each front element from the selected element. (one by one)
O(N) -> Copy the new array to the neighbor set.
O(N^2) -> Calculate Cost
Total Complexity = O(N^2). (without copy)
O(N^3) (with copy)

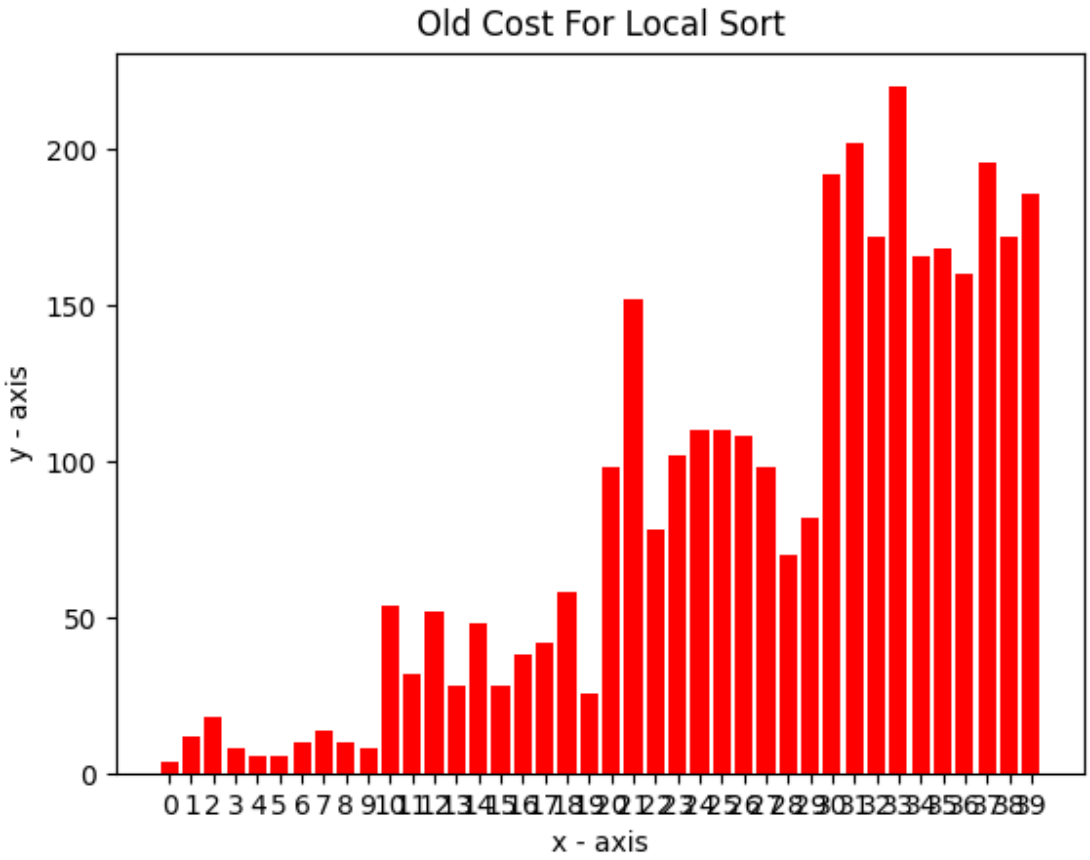
Experimental Tables:

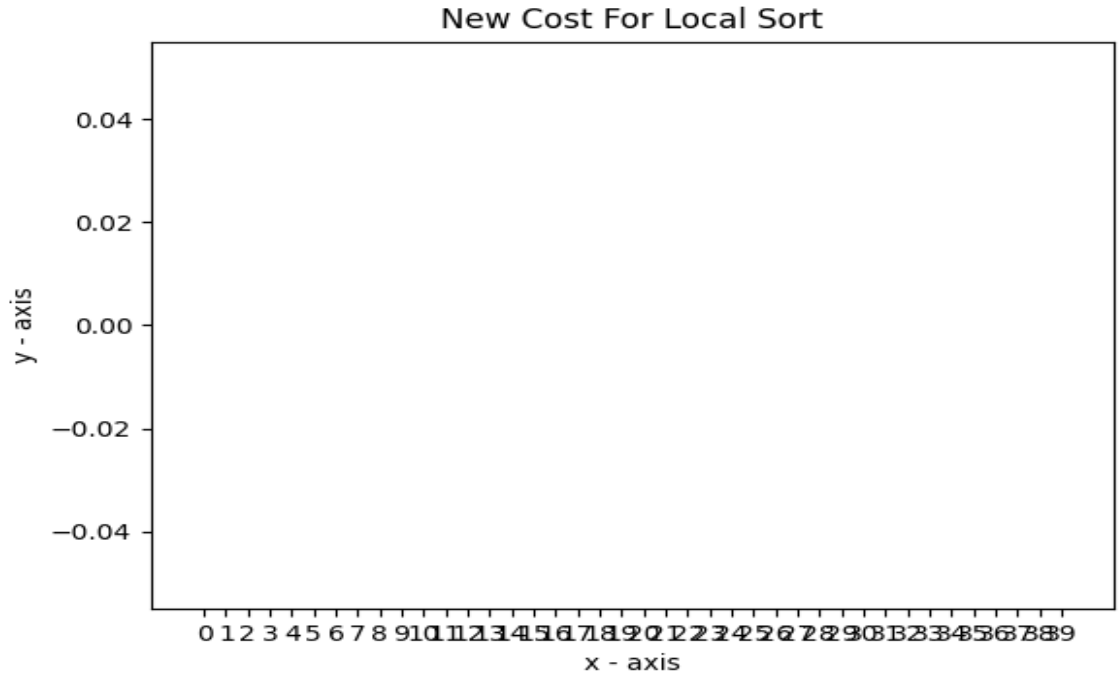


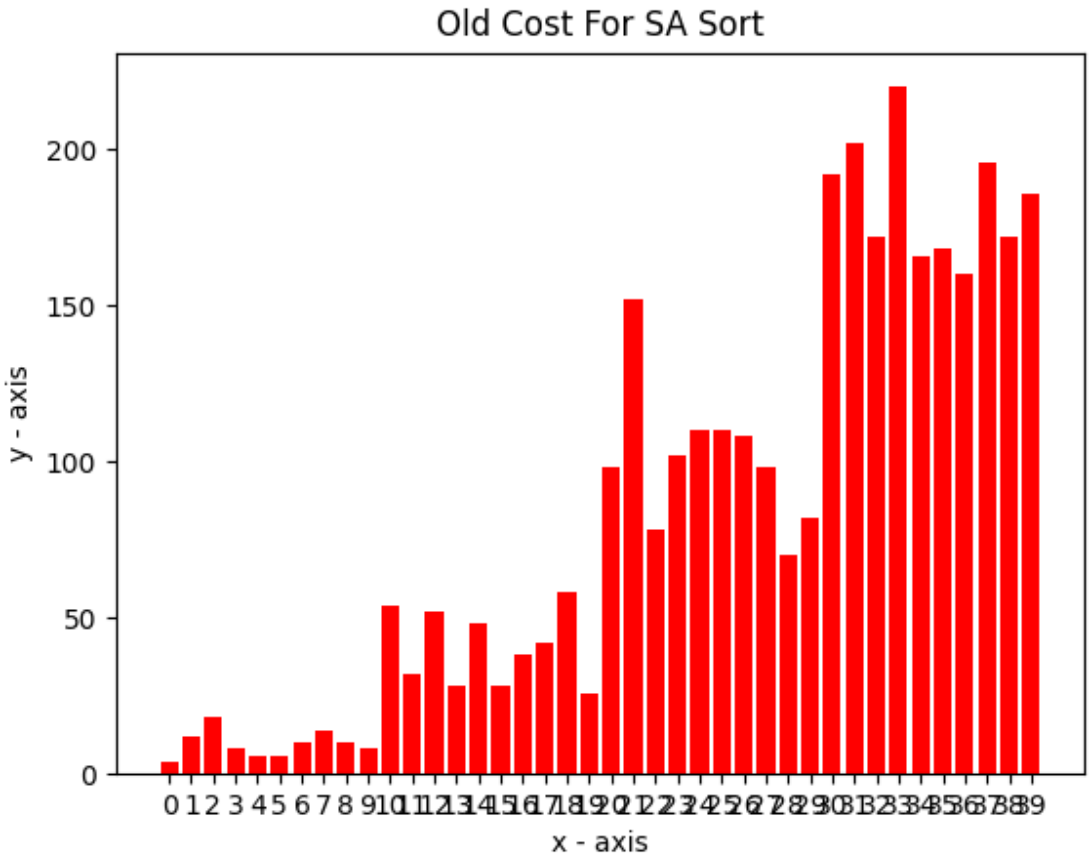


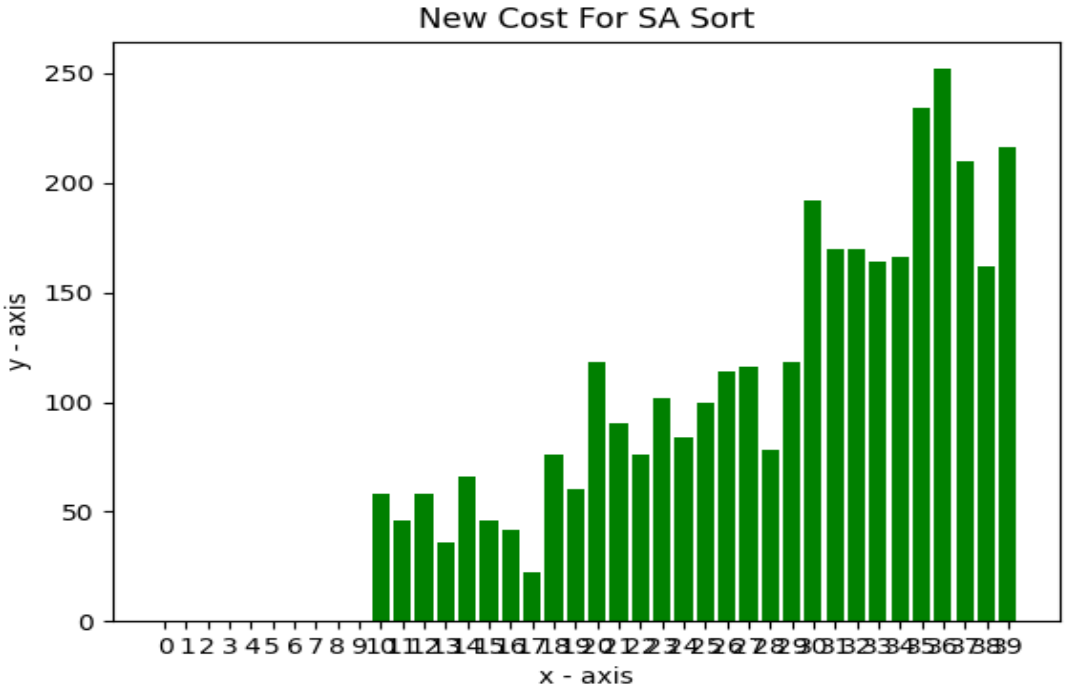

Old Cost For Backtracking Sort
y - axis
x - axis
0
50
100
150
200

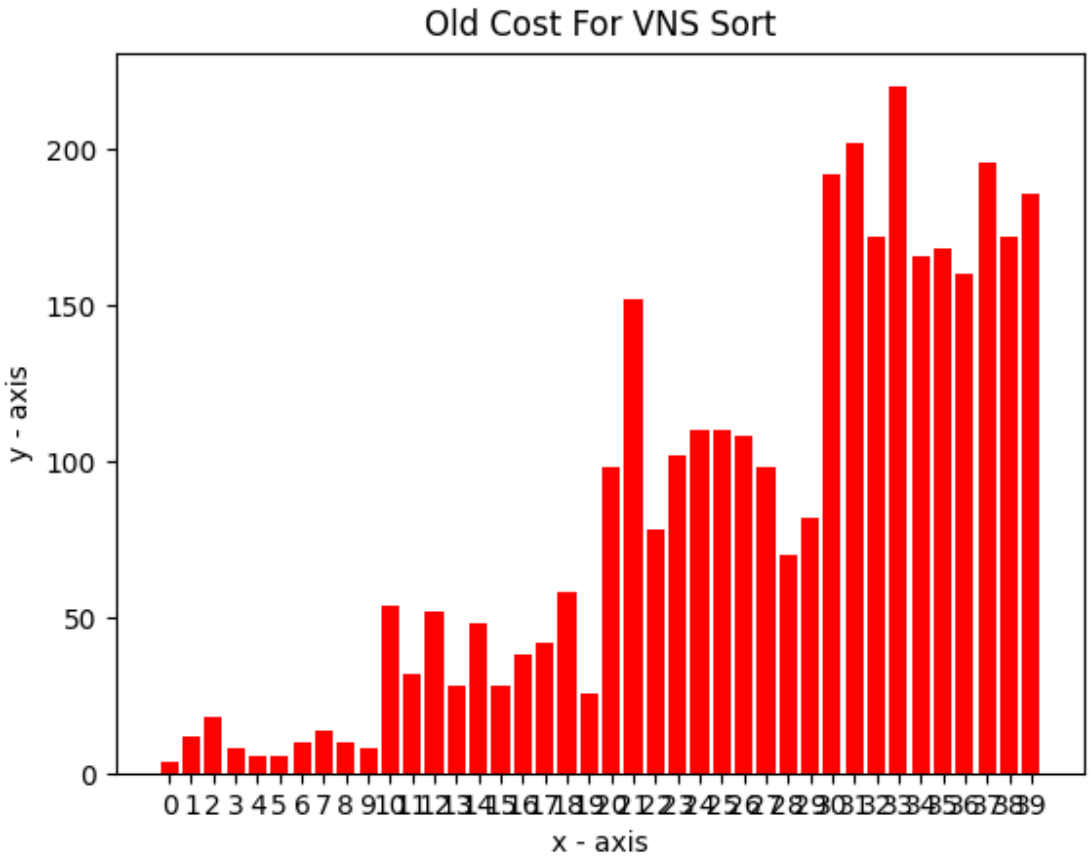

New Cost For Backtracking Sort
y - axis
x - axis
0.04
0.02
0.00
−0.02
−0.04

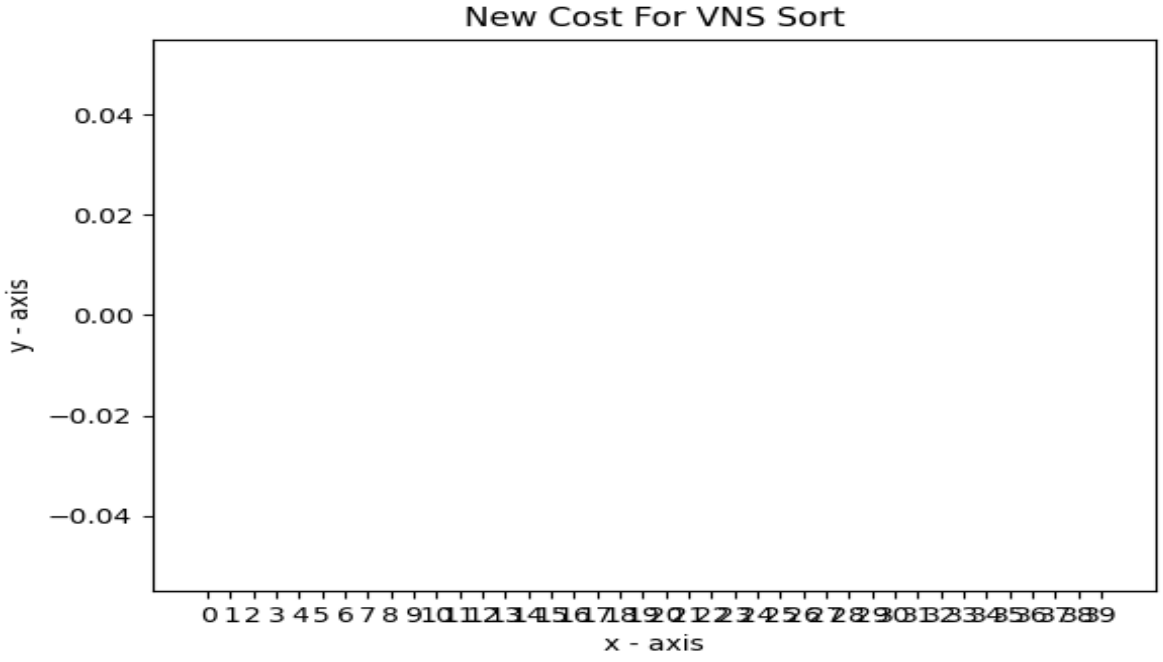

Old Cost For Local Sort
y - axis
x - axis
0
50
100
150
200

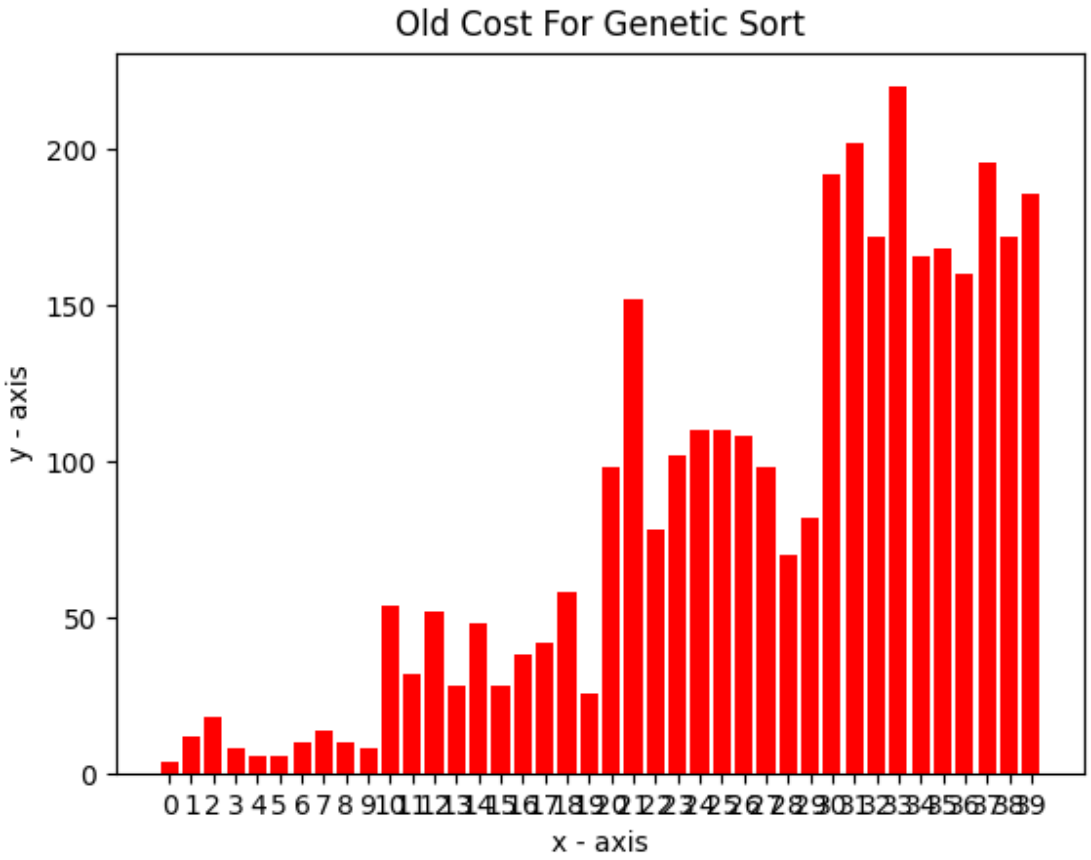

New Cost For Local Sort
y - axis
x - axis
0.04
0.02
0.00
−0.02
−0.04

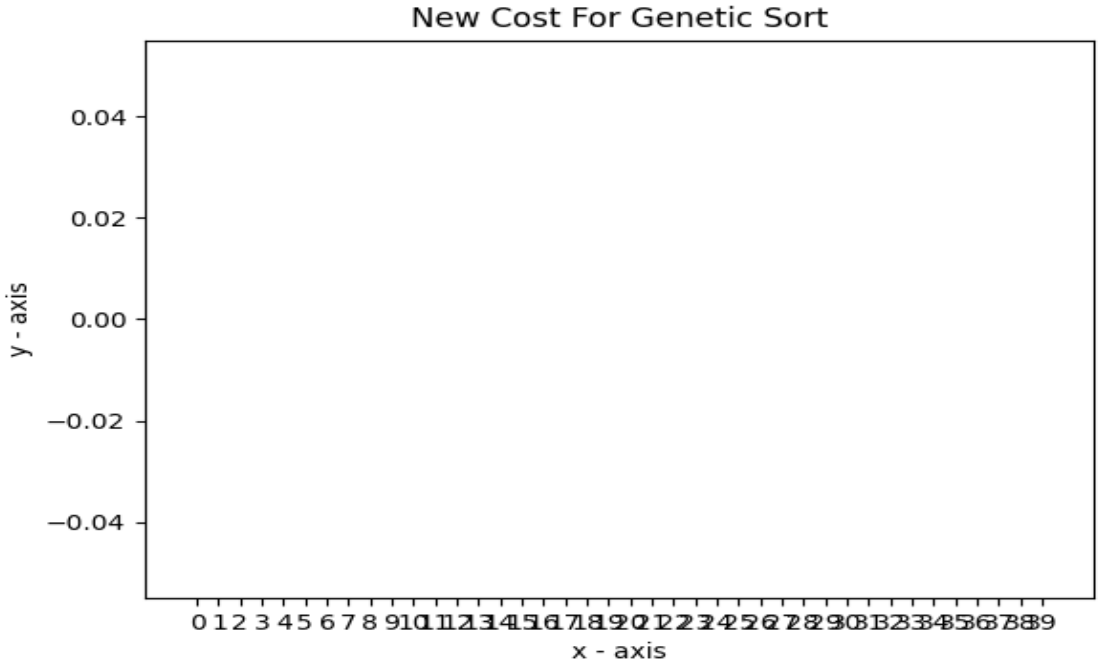

Old Cost For SA Sort
y - axis
x - axis
0
50
100
150
200

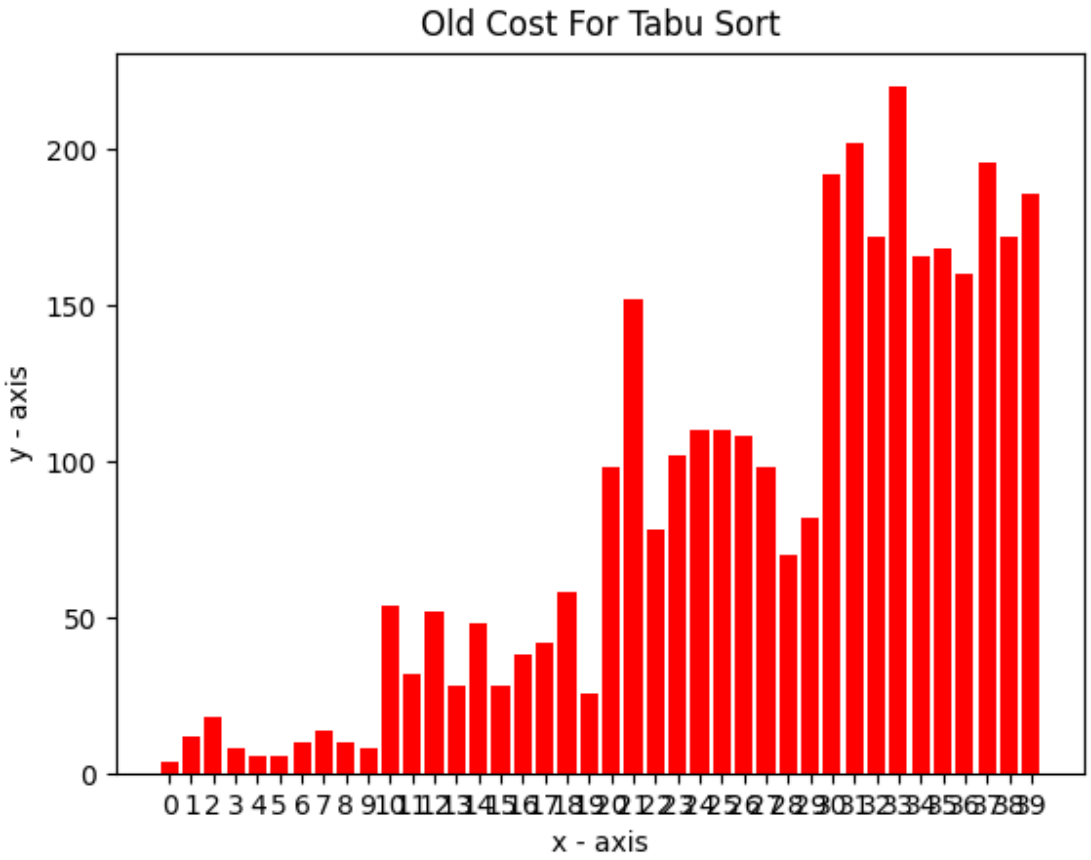

New Cost For SA Sort
y - axis
x - axis
0
50
100
150
200
250

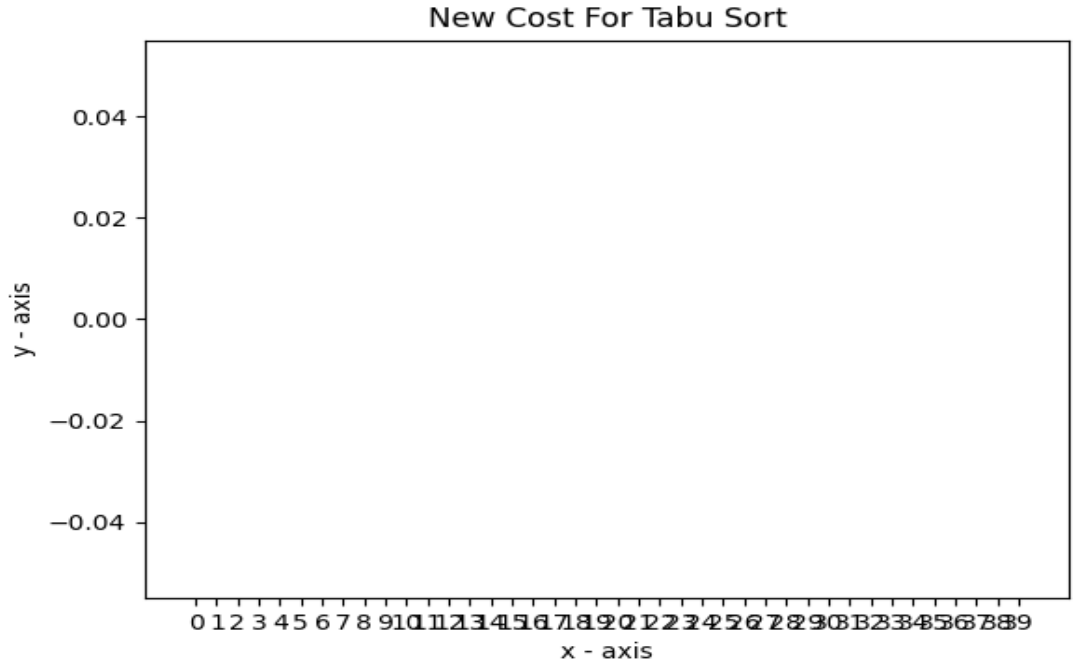

Old Cost For VNS Sort
y - axis
x - axis
0
50
100
150
200

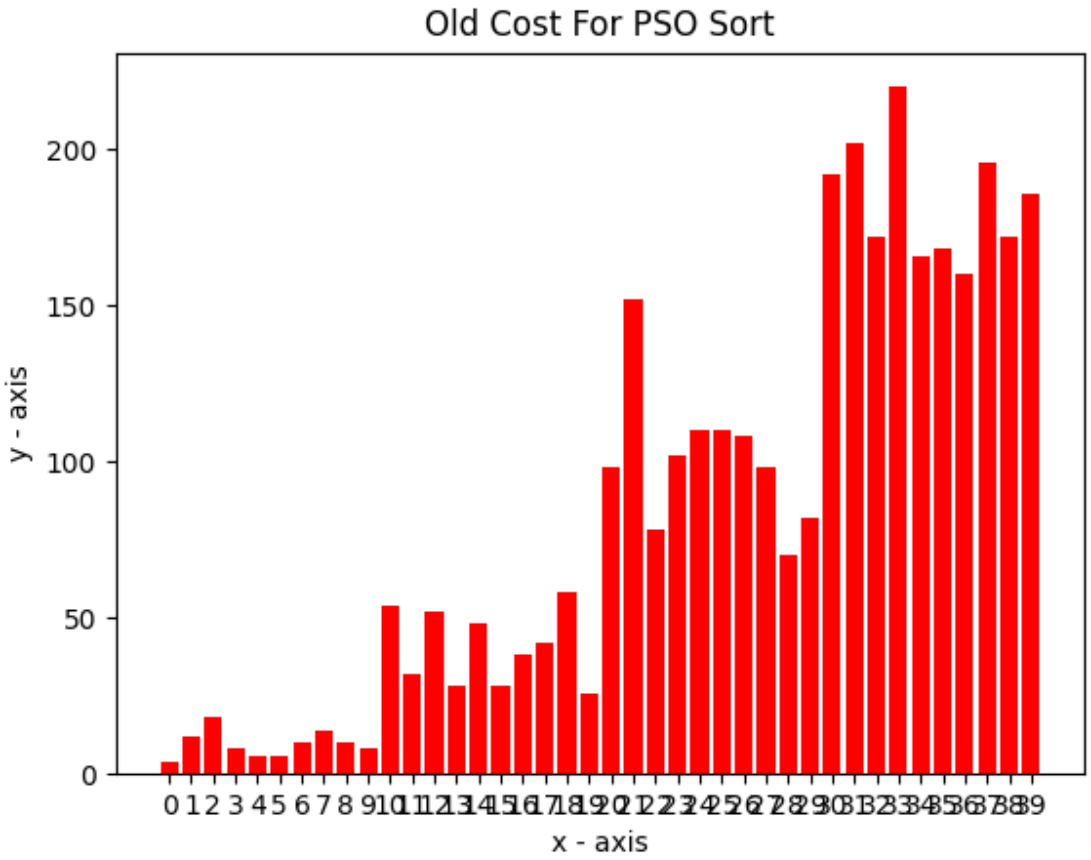

New Cost For VNS Sort
y - axis
x - axis
0.04
0.02
0.00
−0.02
−0.04

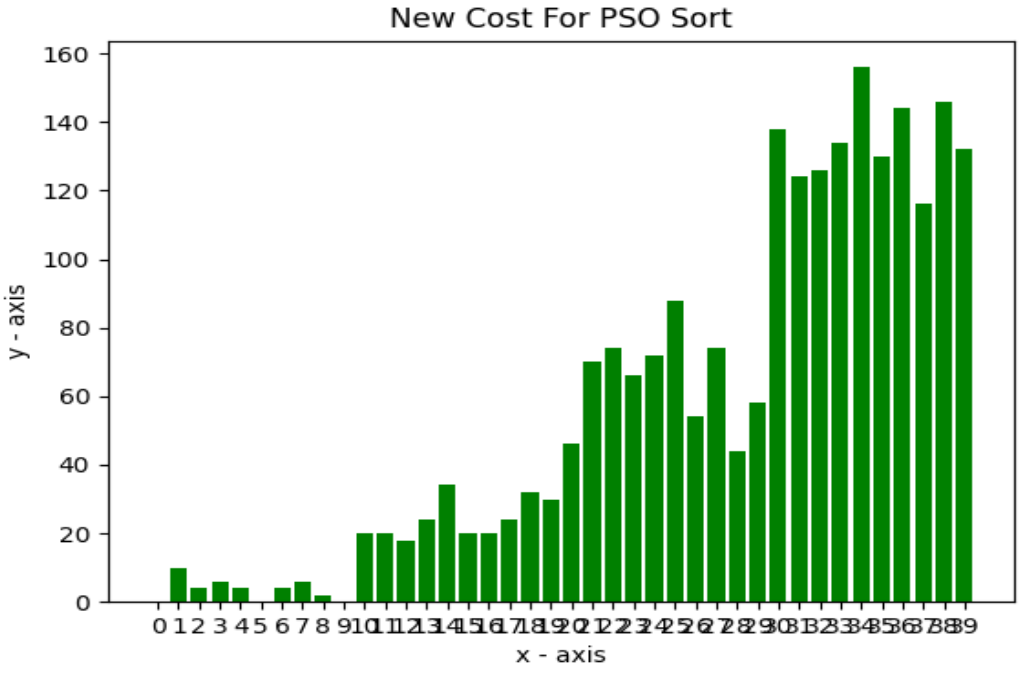

Old Cost For Genetic Sort
y - axis
x - axis
0
50
100
150
200


New Cost For Genetic Sort
y - axis
x - axis
0.04
0.02
0.00
−0.02
−0.04

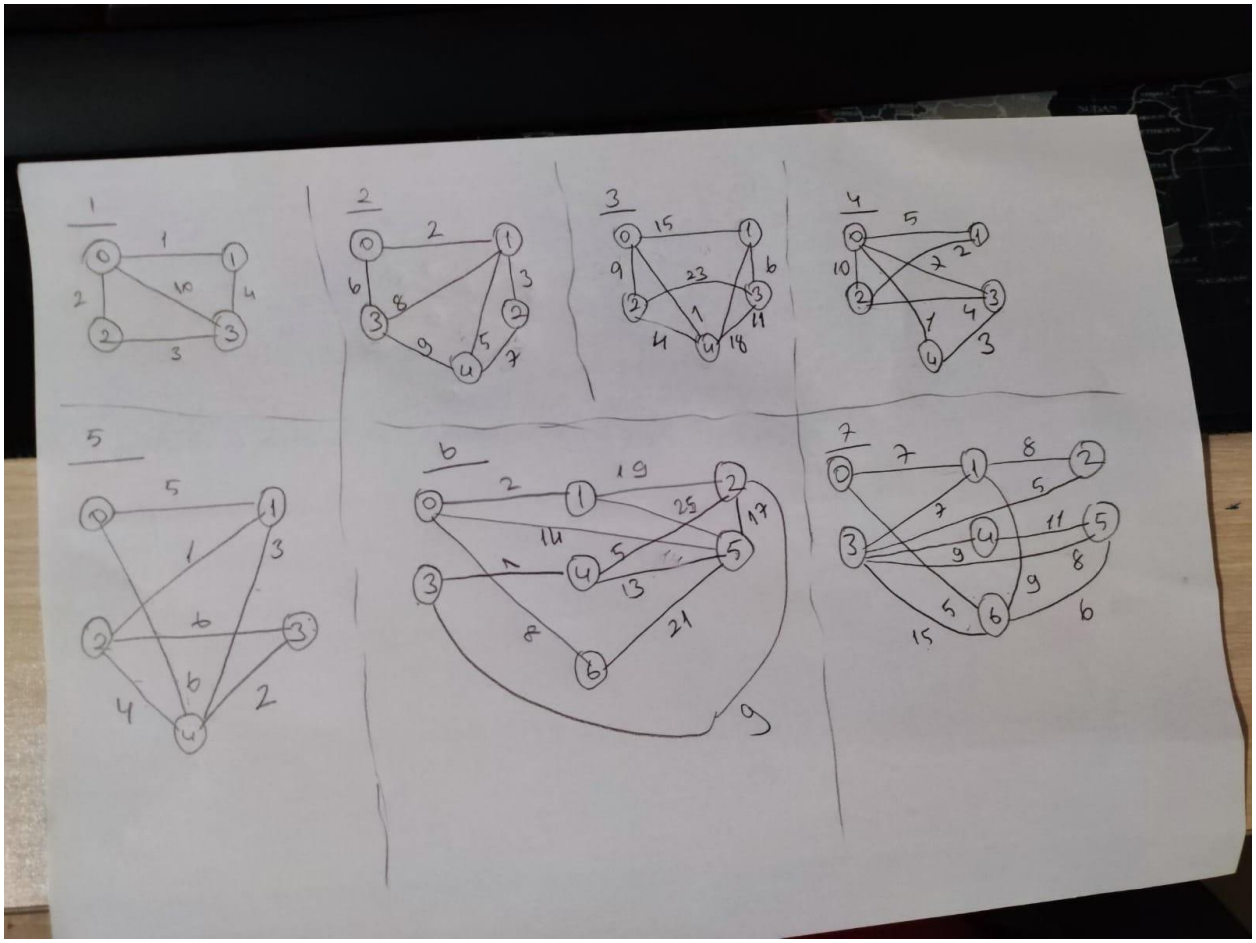

Old Cost For Tabu Sort
y - axis
x - axis
0
50
100
150
200


New Cost For Tabu Sort
y - axis
x - axis
0.04
0.02
0.00
−0.02
−0.04


Old Cost For PSO Sort
y - axis
x - axis
0
50
100
150
200


New Cost For PSO Sort
y - axis
x - axis
0
20
40
60
80
100
120
140
160

Some Graph Data Structures which i used for test algorithm



NOTE:

1) **The BruteForce test implementation is in the comment line in the main.py file because bruteforce is too slow.**
2) **I only have console output for the MST problem**.