Project 3

Algorithm Aspects of Telecommunication Networks
CS 6385

Submitted by:

Niveditha Sasalapura Puttaiah (Net ID: nxs210042)

Table of Contents:

Introduction:

This project is to determine the lowest total cost of a network topology. Total cost is the geometric distance between the nodes and not the hop count. The Network topology must satisfy certain conditions before calculating its total cost. Two different heuristic algorithms must be implemented in order to get optimised total cost of the network.

A heuristic algorithm is one that sacrifices optimality, accuracy, precision, or completeness for speed in order to solve a problem faster and more efficiently than standard approaches. NP-complete problems, a type of decision problem, are frequently solved using heuristic algorithms. When approximate solutions are sufficient but exact solutions are computationally expensive, heuristic techniques are frequently used. There are multiple heuristics algorithms like Tabu search, Simulated annealing, Greedy local search and more.

Problem Description:

Finding lowest total cost of the network using two different heuristic algorithms. The network topology will have 'n' (>=15) nodes in a plane with undirected edges between them. This topology must satisfy below conditions,

1. The graph must be a connected one. i.e., should be able to reach a node from any other node in the network.
2. Degree of each node must be at least 3.
3. Diameter (hop count) from any node in the network to any other node must be at most 4.

Once the network satisfies these conditions then the heuristic algorithms are applied on the graph to get as low total cost as possible. This will be done for multiple topologies and then the results will be compared between both the algorithms.

Input: g, graph with >15 number of nodes with randomly generated co-ordinates.

Output: The total cost of the network

Below are the two heuristic algorithms used in this program in order to achieve the above.

1. Greedy Local Search Algorithm – Basic idea is calculating the total cost of the network at the initial stage and then iteratively removing the edges with highest cost and each time check if the resulted graph still satisfies the above mentioned three conditions. If it does, then again calculate the total cost. This will be done till graph no longer satisfies the conditions.

   The output contains list of costs as each weighted edge gets removed. Last output is the lowest cost of the network.

2. Original Heuristic Algorithm – Here a sub-graph which contains a minimum cost path to reach a node from other node by traversing through all the nodes in the inputted graph will be constructed. This sub-graph will be checked if it satisfies the basic conditions. If it does then its total cost will be calculated which becomes the lowest total cost of the network. If not, then the next lowest cost edge will be added to the topology and check the conditions again and so on till a sub-graph gets formed which satisfies the conditions.

   The output contains final lowest total cost of the network since the algorithm keeps on adding edges till graph satisfies all the conditions.

Heuristic Optimization Algorithms

1. Constructing a graph of 16 nodes and randomly selecting two nodes and generating the edges between them.

```
g = Graph()
g = Graph(directed=False)
for v in list1:
    g.add_vertex(v)
list1 = list(range(0, 16))
for node in list1:
    while True:
        candidate = random.choice(list1)
        if (candidate != node) and (candidate not in candidate_list):
            candidate_list.append(candidate)

            g.add_edge(node, candidate,)
            g.add_edge(candidate, node)

            adj_matrix[node][candidate] = 1
            adj_matrix[candidate][node] = 1
```

2. Checking if the generated graph satisfies the given conditions. If it does not, then re-generate the graph. First, check if the nodes are connected. This is done using BFS algorithm. Then check if every node has degree >=3. This is done by counting number of 1's in each row of adjacency matrix.

```
# Degree of every node is 3
for item in adj_matrix:
    if item.count(1) < 3:
        if not optimize_flag:
            generate_graph()
        else:
            return False
```

Then check the diameter of all the nodes which can be at most 4. This is checked using inbuilt python 'dijkstra' function.

```
# Maximum 4 diameter
for node in list1:
    dist, prev = g.dijkstra(node)
    if any(4 < val for val in dist.values()):
```

3. Weights of each edge will be calculated using Euclidean distance inbuilt function in python for randomly selected coordinates of each node. Total cost of the network is calculated by adding the weights of all the edges.

```
list2 = list(range(0, 50))
master_coord_list = []
while(len(master_coord_list) < 16):
    candidate1 = random.choice(list2)
    candidate2 = random.choice(list2)
    point = [candidate1, candidate2]
    if point not in master_coord_list:
        master_coord_list.append(point)
```

```
dist = round(np.linalg.norm(np.array(item1[0]) -
                            np.array(item[1])), 3)
```

Now this total cost needs to be optimised. Two heuristic algorithms are used in order to achieve this.

4. Greedy Local Search Algorithm

Greedy algorithm is by removing maximum cut edge from the structure and try to obtain a more optimal solution.

a. Initial graph will be considered as current solution S and total cost of the solution as C(S).
b. Select an edge with maximum weight and try to remove it.
c. Check if new solution S' still satisfies all the mentioned conditions.
d. If yes, then replace C(S) by C(S') as current total cost.
e. Repeat this till the solution no longer satisfies the conditions.

Pseudo code,

Total_cost<-Add(cost_list)
Sort(cost_list)
Pointer=0
While(len(cost_list))
{
Remove cost_list[pointer]
Check if conditions satisfy
Calculate Optimized_cost
if optimized_cost<total_cost

then total_cost = optimised_cost
Increment the pointer on cost_list
}
If fails, then revert removed edges
Start the loop from the second max weighed edge.

```
#Removing the maximum weighed edge from the graph
sorted_ec_list = sorted(master_edge_cost_list, key = itemgetter(2), reverse=True)
for ec_idx1 in range(len(sorted_ec_list)):
    point1 = sorted_ec_list[ec_idx1][0]
    point2 = sorted_ec_list[ec_idx1][1]
    node1 = master_coord_list.index(point1)
    node2 = master_coord_list.index(point2)
    adj_matrix[node1][node2] = 0
    adj_matrix[node2][node1] = 0

    #Checking if the updated graph satisfies all the 3 conditions
    if check_conditions(adj_matrix, optimize_flag=True):
        cost = calc_total_cost(adj_matrix, master_coord_list, optimize_flag=True)
```

```
#Iterating through all the edges and checking if edges can be removed
for ec_idx2 in range(len(sorted_ec_list)):
    point11 = sorted_ec_list[ec_idx2][0]
    point12 = sorted_ec_list[ec_idx2][1]
    node11 = master_coord_list.index(point11)
    node12 = master_coord_list.index(point12)
    adj_matrix[node11][node12] = 0
    adj_matrix[node12][node11] = 0
    if check_conditions(adj_matrix, optimize_flag=True):
        cost = calc_total_cost(adj_matrix, master_coord_list, optimize_flag=True)
        if cost < optimized_cost:
            optimized_cost = cost
            print('Minimum cost optained from Greedy Local Search Algorithm {}' .format(optimized_cost))
        else:
            pass
    else: # Reverting the changes if the subsequent edge removal does not yeild lowest cost
        adj_matrix[node11][node12] = 1
        adj_matrix[node12][node11] = 1
```

5. Original Heuristic Algorithm –

a. Constructing a sub-graph from the original graph by calculating minimum path from a source to destination which traverses through all the nodes in the network.

b. Now this sub-graph will be check against all 3 conditions. If all conditions pass, then total cost of the sub-graph will be calculated.
c. If not, then iteratively add the next minimum weighted edges to the sub-graph till all the conditions satisfy.
d. Now replace the optimised cost with newly calculated cost.

Pseudo code,

```
ReachSet = {0};                 // any node can be used
UnReachSet = {1, 2, ..., 15};
Sub_graph = {};
while ( UnReachSet ≠ empty )
{
        Find edge e = (x, y) such that:
        1. x ∈ ReachSet
        2. y ∈ UnReachSet
        3. e has smallest cost
        Sub_graph = Sub_graph ∪ {e};
        ReachSet   = ReachSet ∪ {y};
        UnReachSet = UnReachSet - {y};
 }
Loop
{
if 3 conditions satisfy:
        then calculate the total cost
Else: add next min weight edge to sub_graph
}
```

```
for i in range(16):
    if node1[i]:
        for j in range(16):
            # If the analyzed node have a path to the ending node AND its not included in resulting matrix
            if (not node1[j] and cost_matrix[i][j]>0):
                if cost_matrix[i][j] < min_weight:
                    min_weight = cost_matrix[i][j]
                    start, end = i, j

node1[end] = True

result_matrix[start][end] = min_weight

if min_weight == highest_weight:
    result_matrix[start][end] = 0

count += 1

# This matrix will have minimum cost path from source to destination
result_matrix[end][start] = result_matrix[start][end]
```

```
#checking if resulted graph satisfies all the conditions
while not flag_check == check_conditions2(adj_matrix2):
    chk = list(map(sum, adj_matrix2))

    for i in range(0,16):
            for j in range(0,16):
                if adj_matrix[i][j] != adj_matrix2[i][j] and chk[i]<3:
                    adj_matrix2[i][j]=1
                    chk = list(map(sum, adj_matrix2))
```

Outputs and Graphs:

Note: I have displayed Adjacency matrix of only few values for comparison to limit the size of the document.

Output of Trial 1,

PS C:\users\nxs210042\anaconda3\lib\site-packages> & "C:/Program Files/Python39/python.exe" c:/Users/nxs210042/Anaconda3/Lib/site-packages/atn3_project.py

The coordinates of the nodes are [[18, 22], [26, 44], [12, 18], [33, 15], [23, 40], [15, 5], [3, 5], [21, 23], [20, 46], [16, 31], [23, 6], [6, 12], [31, 31], [28, 31], [35, 11], [29, 5]]

[[0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0]
 [1 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0]
 [1 1 0 0 1 1 0 1 0 0 1 0 1 0 1 1]
 [1 0 0 0 0 0 1 0 0 1 0 0 1 1 0 1]
 [0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 1]
 [0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0]
 [0 0 0 1 0 1 0 1 0 0 0 0 0 1 1 0]
 [0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0]
 [1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0]
 [1 1 1 0 0 1 0 0 1 0 0 0 1 0 0 1]
 [1 1 0 0 0 0 0 0 0 1 0 0 0 1 1 0]
 [0 1 1 1 1 0 0 0 0 0 1 0 0 1 0 0]
 [0 1 0 1 0 0 1 0 1 0 0 1 1 0 1 0]
 [0 0 1 0 1 0 1 0 0 0 0 1 0 1 0 0]
 [0 0 1 1 1 0 0 0 0 0 1 0 0 0 0 0]]

Minimum cost obtained from Greedy Local Search Algorithm 667.467

Minimum cost obtained from Greedy Local Search Algorithm 636.451

Minimum cost obtained from Greedy Local Search Algorithm 605.435

Minimum cost obtained from Greedy Local Search Algorithm 574.419

Minimum cost obtained from Greedy Local Search Algorithm 548.17

Minimum cost obtained from Greedy Local Search Algorithm 521.921

Minimum cost obtained from Greedy Local Search Algorithm 499.294

Minimum cost obtained from Greedy Local Search Algorithm 478.678

Minimum cost obtained from Greedy Local Search Algorithm 460.678

Minimum cost obtained from Greedy Local Search Algorithm 442.678

Minimum cost obtained from Greedy Local Search Algorithm 427.058

Minimum cost obtained from Greedy Local Search Algorithm 417.058

Minimum cost obtained from Greedy Local Search Algorithm 407.571

Minimum cost obtained from Greedy Local Search Algorithm 399.955

Minimum cost obtained from Greedy Local Search Algorithm 392.339

Minimum cost obtained from Greedy Local Search Algorithm 384.723

Minimum cost obtained from Greedy Local Search Algorithm 380.48

[[0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0]
 [0 0 0 0 1 1 0 1 0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1]
 [0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0]
 [0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0]
 [1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1]
 [1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0]
 [0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0]
 [0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0]
 [0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0]]

Minimum cost obtained from Greedy Local Search Algorithm 376.237

Run time of Greedy Local Search Algorithm is 1.9281711ms

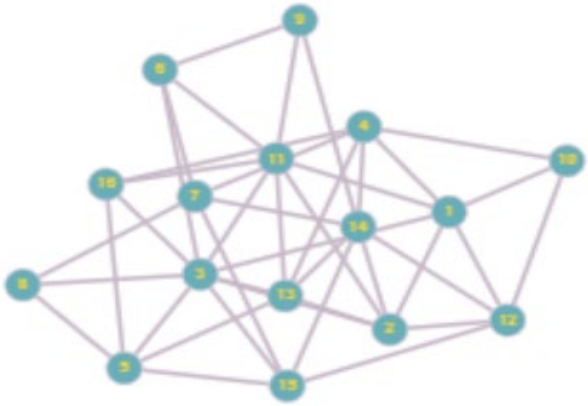
[[0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0]
 [0 0 0 0 1 1 0 1 0 0 0 0 0 0 1 1]
 [0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1]

[0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1]
[0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0]
[0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0]
[1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0]
[1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[1 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0]
[0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0]
[0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]]

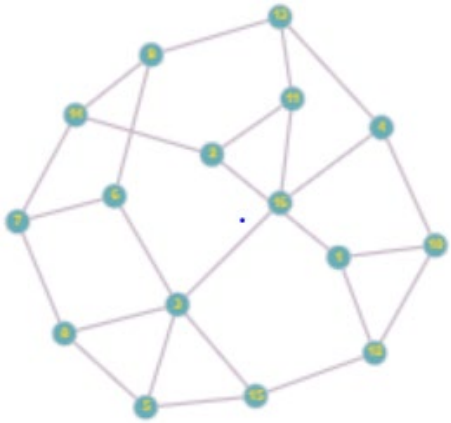
Minimum cost obtained from Original Heuristic Algorithm is 450.03

Run time of Original Heuristic Algorithm is 0.011492200000000174ms

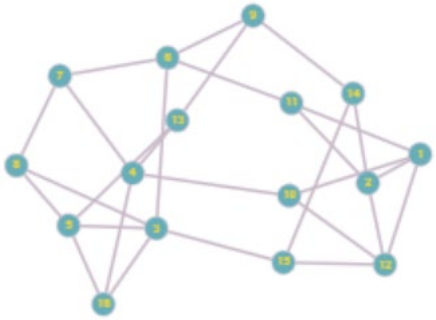
Original graph when total cost is at its max i.e., 667.467,



Graph when lowest cost obtained from Greedy Local Search Algorithm is 376.237,



Graph when lowest cost obtained from Original Heuristic Algorithm is 450.03,



Output of Trial 2,

PS C:\users\nxs210042\anaconda3\lib\site-packages> & "C:/Program Files/Python39/python.exe" c:/Users/nxs210042/Anaconda3/Lib/site-packages/atn3_project.py

The coordinates of the nodes are [[44, 7], [25, 35], [49, 15], [49, 8], [24, 26], [42, 11], [46, 18], [30, 34], [45, 22], [44, 12], [7, 47], [44, 1], [42, 25], [32, 40], [39, 42], [6, 17]]

[[0 0 1 1 1 0 1 0 0 0 1 1 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0]
 [1 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0]
 [1 0 1 0 0 1 1 0 1 0 0 0 1 0 0 0]
 [1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0]

[0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 0]
[1 0 0 1 0 0 0 1 0 1 1 0 0 0 0 0]
[0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1]
[0 0 1 1 1 1 0 0 0 0 1 0 0 1 0 0]
[0 0 0 0 1 1 1 0 0 0 0 1 0 1 0 0]
[1 0 0 0 0 0 1 0 1 0 0 0 0 1 1 1]
[1 0 1 0 0 0 0 0 0 1 0 0 1 1 0 1]
[0 1 1 1 0 0 0 0 0 0 0 1 0 1 0 0]
[0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1]
[1 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 1 1 0 1 0 0]]
Minimum cost obtained from Greedy Local Search Algorithm 1279.733

Minimum cost obtained from Greedy Local Search Algorithm 1230.815

Minimum cost obtained from Greedy Local Search Algorithm 1181.897

Minimum cost obtained from Greedy Local Search Algorithm 1132.979

Minimum cost obtained from Greedy Local Search Algorithm 1084.061

Minimum cost obtained from Greedy Local Search Algorithm 1045.422

Minimum cost obtained from Greedy Local Search Algorithm 1007.567

Minimum cost obtained from Greedy Local Search Algorithm 973.552

Minimum cost obtained from Greedy Local Search Algorithm 939.537

[[0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0]
[0 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0]
[1 0 1 0 0 1 1 0 1 0 0 0 1 0 0 0]
[0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1]
[0 0 1 1 1 1 0 0 0 0 1 0 0 1 0 0]
[0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 1 1 1 0 0 0 0 0 0 0 1 0 1 0 0]
[0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 1]
[0 1 1 0 0 1 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0]]

Minimum cost obtained from Greedy Local Search Algorithm 905.996

Minimum cost obtained from Greedy Local Search Algorithm 872.455

Minimum cost obtained from Greedy Local Search Algorithm 846.745

Minimum cost obtained from Greedy Local Search Algorithm 821.566

Minimum cost obtained from Greedy Local Search Algorithm 800.542

Minimum cost obtained from Greedy Local Search Algorithm 779.518

Minimum cost obtained from Greedy Local Search Algorithm 758.494

Minimum cost obtained from Greedy Local Search Algorithm 745.078

Minimum cost obtained from Greedy Local Search Algorithm 740.606

Minimum cost obtained from Greedy Local Search Algorithm 702.751

Minimum cost obtained from Greedy Local Search Algorithm 698.279

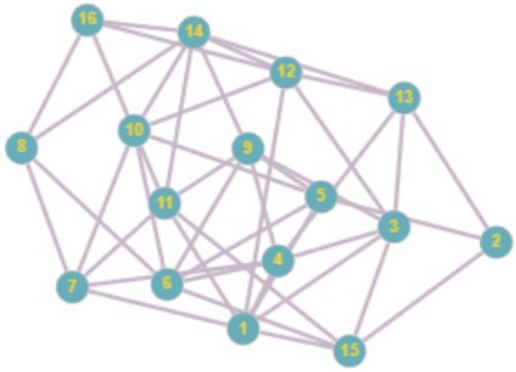
Run time of Greedy Local Search Algorithm is 2.8331144000000004ms

[[0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0]
 [0 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0]
 [1 0 1 0 0 1 1 0 1 0 0 0 0 0 0 0]
 [1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0]
 [1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1]
 [0 0 1 1 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 1 1 0 1]
 [0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 1]
 [1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0]]

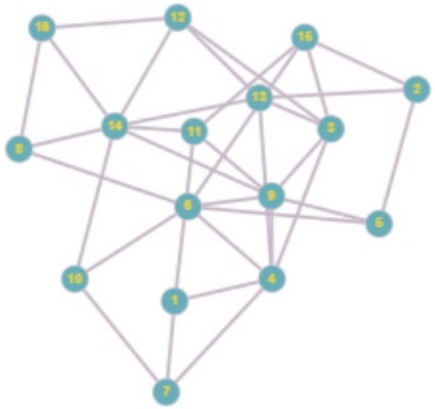
Minimum cost obtained from Original Heuristic Algorithm is 821.928

Run time of Original Heuristic Algorithm is 0.014366899999999738ms

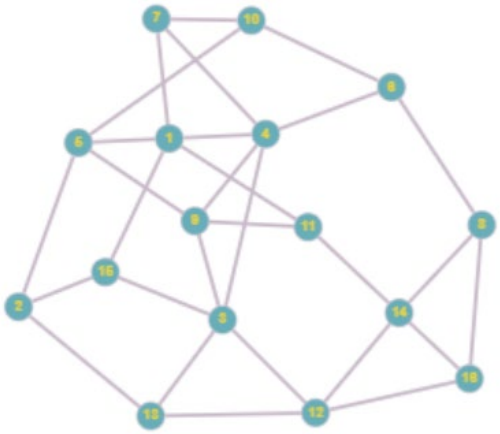
Original graph when total cost is at its max i.e., 1279.733,



Graph when cost obtained from Greedy Local Search Algorithm is 905.996,



Graph when lowest cost obtained from Original Heuristic Algorithm is 821.928,



Output of Trial 3,

PS C:\users\nxs210042\anaconda3\lib\site-packages> & "C:/Program Files/Python39/python.exe" c:/Users/nxs210042/Anaconda3/Lib/site-packages/atn3_project.py

The coordinates of the nodes are [[12, 48], [19, 2], [45, 27], [11, 13], [48, 23], [35, 2], [32, 41], [18, 35], [8, 35], [46, 12], [2, 22], [13, 35], [36, 0], [5, 30], [41, 7], [25, 44]]

[[0 0 0 0 0 0 0 0 1 1 0 1 0 0 1 0]
 [0 0 0 1 0 0 1 1 1 0 0 0 1 0 0 1]
 [0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 1]
 [0 1 0 0 0 0 1 1 0 0 0 1 1 0 1 0]
 [0 0 0 0 0 1 1 1 0 0 1 0 0 0 1 1]
 [0 0 0 0 1 0 1 0 0 1 0 0 0 1 0 1]
 [0 1 1 1 1 1 0 1 1 0 0 0 0 0 1 0]
 [0 1 0 1 1 0 1 0 0 0 0 0 0 0 1 0]
 [1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0]
 [1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0]
 [0 0 1 0 1 0 0 0 1 1 0 1 1 0 0 0]
 [1 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0]
 [0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 1]
 [0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0]
 [1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 1]
 [0 1 1 0 1 1 0 0 0 0 0 0 1 0 1 0]]
Minimum cost obtained from Greedy Local Search Algorithm 816.397

Minimum cost obtained from Greedy Local Search Algorithm 783.155

Minimum cost obtained from Greedy Local Search Algorithm 752.268

Minimum cost obtained from Greedy Local Search Algorithm 724.465

Minimum cost obtained from Greedy Local Search Algorithm 696.662

Minimum cost obtained from Greedy Local Search Algorithm 668.859

Minimum cost obtained from Greedy Local Search Algorithm 642.176

Minimum cost obtained from Greedy Local Search Algorithm 618.745

Minimum cost obtained from Greedy Local Search Algorithm 596.184

Minimum cost obtained from Greedy Local Search Algorithm 577.826

Minimum cost obtained from Greedy Local Search Algorithm 559.468

Minimum cost obtained from Greedy Local Search Algorithm 542.468

Minimum cost obtained from Greedy Local Search Algorithm 525.468

Minimum cost obtained from Greedy Local Search Algorithm 508.468

Minimum cost obtained from Greedy Local Search Algorithm 493.335

Minimum cost obtained from Greedy Local Search Algorithm 478.202

Minimum cost obtained from Greedy Local Search Algorithm 463.069

Minimum cost obtained from Greedy Local Search Algorithm 455.069

Minimum cost obtained from Greedy Local Search Algorithm 443.069

```
[[0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0]
 [0 0 0 1 0 0 1 1 0 0 0 0 1 0 0 1]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 1]
 [0 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1]
 [0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 1 0 0 1 0 0 0 0 0 0 0 0 1 0]
 [1 0 1 0 0 0 1 0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0]
 [1 0 0 1 0 0 0 0 0 1 0 0 0 0 0]
 [0 1 0 1 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 1 0 0 1 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 1 0 0 0 0 0 1 0]
 [0 1 1 0 0 1 0 0 0 0 0 0 0 0 0]]
```

Minimum cost obtained from Greedy Local Search Algorithm 419.638

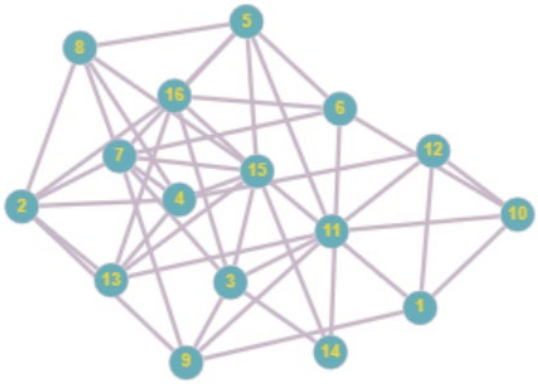
Run time of Greedy Local Search Algorithm is 1.759869ms

```
[[0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0]
 [0 0 0 1 0 0 1 1 1 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 1]
 [0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1]
 [0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1]
 [0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
 [0 1 0 0 1 0 0 0 0 0 0 0 0 0 1 0]
```

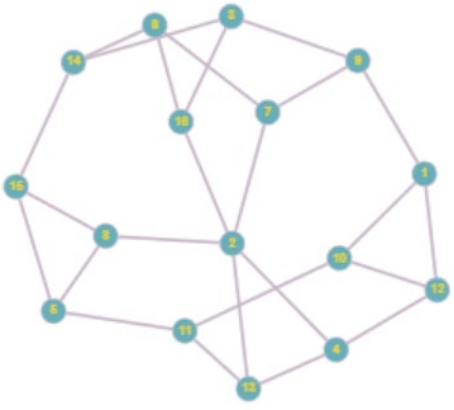
[1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 1 0 0 0 0 1 1 0 0 0 0]
[0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0]
[0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 1 0 0 0 0 1 1 0 0]
[0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0]]
Minimum cost obtained from Original Heuristic Algorithm is 445.508

Run time of Original Heuristic Algorithm is 0.007286500000000196ms

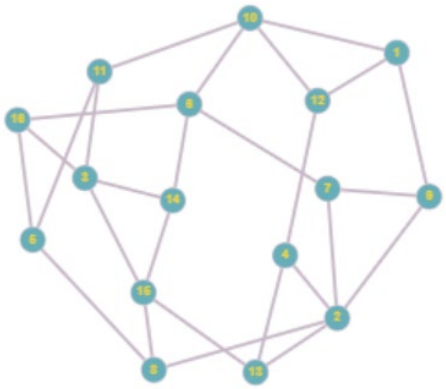
Original graph when total cost is at its max i.e., 816.397,



Graph when lowest cost obtained from Greedy Local Search Algorithm is 419.638,



Graph when lowest cost obtained from Original Heuristic Algorithm is 445.508,



Output of Trial 4,

PS C:\users\nxs210042\anaconda3\lib\site-packages> & "C:/Program Files/Python39/python.exe" c:/Users/nxs210042/Anaconda3/Lib/site-packages/atn3_project.py

The coordinates of the nodes are [[13, 47], [46, 16], [29, 19], [44, 19], [47, 25], [20, 17], [38, 17], [4, 16], [35, 28], [44, 5], [0, 12], [27, 23], [20, 11], [30, 11], [8, 33], [46, 43]]

[[0 1 1 1 0 0 1 1 1 1 0 1 0 0 0 0]
 [1 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1]
 [1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0]
 [1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 1]
 [0 0 1 1 0 1 0 1 1 0 1 0 0 1 0 0]
 [0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 1]
 [1 0 0 1 0 0 0 0 0 0 0 1 0 1 1 0]
 [1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1]
 [1 0 1 1 1 1 0 1 0 0 0 0 0 0 1 0]
 [1 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0]
 [0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 1]
 [1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
 [0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 0]
 [0 1 0 1 0 0 1 0 1 1 0 0 0 0 0 0]
 [0 1 0 1 0 1 0 1 0 0 1 0 1 0 0 0]]
Minimum cost obtained from Greedy Local Search Algorithm 1360.006

Minimum cost obtained from Greedy Local Search Algorithm 1317.462

Minimum cost obtained from Greedy Local Search Algorithm 1274.918

Minimum cost obtained from Greedy Local Search Algorithm 1232.374

Minimum cost obtained from Greedy Local Search Algorithm 1189.83

Minimum cost obtained from Greedy Local Search Algorithm 1150.511

Minimum cost obtained from Greedy Local Search Algorithm 1111.192

Minimum cost obtained from Greedy Local Search Algorithm 1071.873

Minimum cost obtained from Greedy Local Search Algorithm 1032.554

Minimum cost obtained from Greedy Local Search Algorithm 993.235

Minimum cost obtained from Greedy Local Search Algorithm 954.402

Minimum cost obtained from Greedy Local Search Algorithm 916.759

Minimum cost obtained from Greedy Local Search Algorithm 879.116

Minimum cost obtained from Greedy Local Search Algorithm 841.473

Minimum cost obtained from Greedy Local Search Algorithm 803.83

Minimum cost obtained from Greedy Local Search Algorithm 770.229

Minimum cost obtained from Greedy Local Search Algorithm 738.908

Minimum cost obtained from Greedy Local Search Algorithm 715.201

[[0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
 [0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 1 0 1 0 0 0 0 0 0 0 0 1 0]
 [0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1]
 [1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0]
 [1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1]
 [0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0]
 [0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
 [0 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0]
 [0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0]

[0 1 0 0 0 1 0 1 0 0 0 0 1 0 0 0]]
Minimum cost obtained from Greedy Local Search Algorithm 706.257

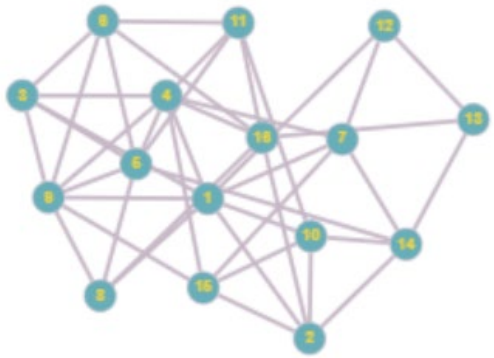
Minimum cost obtained from Greedy Local Search Algorithm 670.132

Run time of Greedy Local Search Algorithm is 2.3351680999999997ms

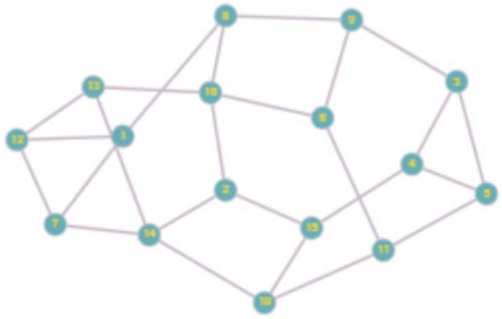
[[0 0 0 1 0 0 1 1 1 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1]
[0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 1 0 0 0 1 1 1 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1]
[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0]
[1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 1 0 1 1 0 1 0 0 0 0 0 0 1 0]
[0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0]
[0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
[0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0]
[0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 0]]
Minimum cost obtained from Original Heuristic Algorithm is 879.872

Run time of Original Heuristic Algorithm is 0.010114199999999407ms

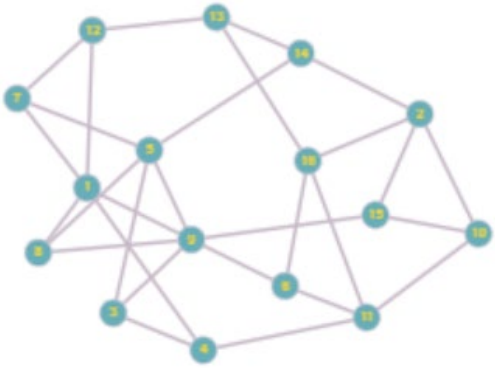
Original graph when total cost is at its max i.e., 1360.006,



Graph when cost obtained from Greedy Local Search Algorithm is 706.257,



Graph when lowest cost obtained from Original Heuristic Algorithm is 879.872,



Output of Trial 5,

Have not included adjacency matrix in this trial to show how randomly the node coordinates are selected and the difference of the total costs after each iteration and the run time of the algorithms.

```
PS C:\users\nxs210042\anaconda3\lib\site-packages> & "C:/Program Files/Python39/python.exe" c:/Users/nxs210042/Anaconda3/Lib/site-pac
kages/atn3_project.py
The coordinates of the nodes are [[13, 6], [23, 45], [44, 24], [8, 37], [35, 30], [30, 42], [8, 22], [4, 7], [49, 17], [10, 20], [15,
 23], [43, 8], [29, 29], [15, 7], [35, 15], [49, 4]]


Minimum cost obtained from Greedy Local Search Algorithm 996.466
Minimum cost obtained from Greedy Local Search Algorithm 955.969
Minimum cost obtained from Greedy Local Search Algorithm 915.472
Minimum cost obtained from Greedy Local Search Algorithm 874.975
Minimum cost obtained from Greedy Local Search Algorithm 835.05
Minimum cost obtained from Greedy Local Search Algorithm 795.125
Minimum cost obtained from Greedy Local Search Algorithm 755.73
Minimum cost obtained from Greedy Local Search Algorithm 720.73
Minimum cost obtained from Greedy Local Search Algorithm 685.73
Minimum cost obtained from Greedy Local Search Algorithm 650.902
Minimum cost obtained from Greedy Local Search Algorithm 616.074
Minimum cost obtained from Greedy Local Search Algorithm 599.521
Minimum cost obtained from Greedy Local Search Algorithm 585.521
Minimum cost obtained from Greedy Local Search Algorithm 571.521
Minimum cost obtained from Greedy Local Search Algorithm 557.521
Minimum cost obtained from Greedy Local Search Algorithm 552.136
Minimum cost obtained from Greedy Local Search Algorithm 546.751
Minimum cost obtained from Greedy Local Search Algorithm 542.628
Run time of Greedy Local Search Algorithm is 1.7062492999999999ms


Minimum cost obtained from Original Heuristic Algorithm is 630.81
Run time of Original Heuristic Algorithm is 0.0050098999999996785ms
```

Conclusion:

Greedy Local Search Algorithm is an iterative one. Total cost of the network decreases as it iterates. The number of iterations to reach the lowest cost varies based on the graph. Few graphs might reach lowest total cost in few iterations, and some might take more iterations.

Original Heuristic Algorithm is non-iterative. As soon as the graph satisfies all three conditions, it is already at its lowest total cost. Hence only lowest output is shown.

As we can see from the examples, Greedy local search algorithm yields more optimised graph than the original heuristic algorithm almost everytime. The run time of the original algorithm is much lesser than that of local search algorithm, since it is not iterative. That is, run time depends on number of iterations. It can be observed that if the initial total cost is more, then the runtime taken by the algorithm for that graph is bit higher than the graphs with bit lesser initial cost.

References:

1. Lecture Notes by Professor Andras Farago
2. Wikipedia - https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms#:~:text=A%20heuristic%20algorithm%20is%20one,a%20class%20of%20decision%20problems.
3. Local search algorithm - https://www.cs.jhu.edu/~mdinitz/classes/ApproxAlgorithms/Spring2019/Lectures/lecture5.pdf
4. To generate graphs - https://graphonline.ru/en/create_graph_by_matrix

Appendix:

```
from enum import Flag
from msilib.sequence import AdminExecuteSequence
import networkx as netx
import random
import math
import numpy as np
from graph_tools import *
from operator import itemgetter
import sys
import timeit
sys.setrecursionlimit(10000)

adj_matrix = [[0]*16 for _ in range(16)]
list1 = list(range(0, 16))
#generating a graph with 16 nodes
g = Graph()
g = Graph(directed=True)
for v in list1:
    g.add_vertex(v)

def BFS(Adjmatrix):

    flag = False
    visited = [False for i in range(16)]
    Visitednode = []
```

```
    countVisited = 0
    for src in range(len(Adjmatrix)):
        for idx in range(16):
            #checking if edge present between 2 nodes
            if (not(visited[idx]) and Adjmatrix[src][idx] == 1 and (idx not in Visitednode)):
                Visitednode.append(idx)
                countVisited += 1
                visited[idx] = True
    # Verification and returning the decision
    if (countVisited == 16):
        flag = True

    return flag

def generate_graph():
    list1 = list(range(0, 16))

    candidate_list = []
    adj_matrix = [[0]*16 for _ in range(16)]

    for node in list1:
        while True:
            candidate = random.choice(list1)
            if (candidate != node) and (candidate not in candidate_list):
                candidate_list.append(candidate)

                #adding the edges generated randomely
```

```
            g.add_edge(node, candidate,)
            g.add_edge(candidate, node)

            #updating the adjacency matrix
            adj_matrix[node][candidate] = 1
            adj_matrix[candidate][node] = 1

        if (len(candidate_list) == 3):
            candidate_list = []
            break
    check_conditions(adj_matrix)


def check_conditions(adj_matrix, optimize_flag=False):

    flag = True
    # Fully conncted
    if not BFS(adj_matrix):
        if not optimize_flag:
            generate_graph()
        else:
            return False

    # Maximum 4 diameter
    for node in list1:
        dist, prev = g.dijkstra(node)
        if any(4 < val for val in dist.values()):
```

```
            if not optimize_flag:
                generate_graph()
            else:
                return False

    # Degree of every node is 3
    for item in adj_matrix:
        if item.count(1) < 3:
            if not optimize_flag:
                generate_graph()
            else:
                return False

    if flag and not optimize_flag:
        naming_vertex(adj_matrix)
    if flag and optimize_flag:
        return True

def check_conditions2(adj_matrix2):

    # Fully conncted
    if not BFS(adj_matrix2):
        return False

    # Maximum 4 diameter
    for node in list1:
        dist, prev = g.dijkstra(node)
```

```
        if any(4 < val for val in dist.values()):
                return False

    # Degree of every node is 3
    for item in adj_matrix2:
        if item.count(1) < 3:
                return False

    return True

#Generating unique pair of x,y coordinates for each node
def naming_vertex(adj_matrix):
    list2 = list(range(0, 50))
    master_coord_list = []
    while(len(master_coord_list) < 16):
        candidate1 = random.choice(list2)
        candidate2 = random.choice(list2)
        point = [candidate1, candidate2]
        if point not in master_coord_list:
            master_coord_list.append(point)

    print('The coordinates of the nodes are {}'.format(master_coord_list))
    print('\n')

    Total_cost = calc_total_cost(adj_matrix, master_coord_list)
```

```
def calc_total_cost(adj_matrix, master_coord_list, optimize_flag=False):

    cost_matrix = [[0]*16 for _ in range(16)]

    #Generating the coordinate list for nodes with edges between them
    coord_list = [[ix,iy] for ix, row in enumerate(adj_matrix)
                  for iy, i in enumerate(row) if i == 1]
    unique_coord_list = [list(i) for i in {*[tuple(sorted(i)) for i in coord_list]}]



    #adding the connected x1,y1 and x2,y2 coordinates to a master list in order to calculate
euclidean distance betweem the nodes
    master_edge_list = []
    for item in unique_coord_list:
        point = []
        for sub_item in item:
            point.append(master_coord_list[sub_item])
        master_edge_list.append(point)

    #Calculating euclidean distance to get the weight of each edge
    master_edge_cost_list = []
    for item1 in master_edge_list:
        dist = round(np.linalg.norm(np.array(item1[0]) -
                                    np.array(item[1])), 3)

        master_edge_cost_list.append([item1[0], item1[1], dist])
```

```
    #Adding the cost of each edge
    total_cost = 0
    for item in master_edge_cost_list:
        total_cost += item[-1]
    original_total_cost = round(total_cost, 3)


    #Forming a cost matrix out of adjacency matrix
    k=0
    for i in range(0,16):
            for j in range(0,16):
                if i<j and adj_matrix[i][j]==1:
                    cost_matrix[i][j] = master_edge_cost_list[k][2]
                    cost_matrix[j][i] = master_edge_cost_list[k][2]
                    k += 1




    if not optimize_flag:
        t = timeit.timeit(lambda: optimize1(master_edge_cost_list, original_total_cost, adj_matrix,
master_coord_list), number=1)
        print('Run time of Greedy Local Search Algorithm is {}ms' .format(t))


        t1= timeit.timeit(lambda: optimize2(master_edge_cost_list, original_total_cost,
adj_matrix, cost_matrix, master_coord_list), number=1)
        print('Run time of Original Heuristic Algorithm is {}ms' .format(t1))


    if optimize_flag:
        return original_total_cost
```

```
#Greedy Local Search Algorithm
def optimize1(master_edge_cost_list, optimized_cost, adj_matrix, master_coord_list):

    #Removing the maximum weighed edge from the graph
    sorted_ec_list = sorted(master_edge_cost_list, key = itemgetter(2), reverse=True)
    for ec_idx1 in range(len(sorted_ec_list)):
        point1 = sorted_ec_list[ec_idx1][0]
        point2 = sorted_ec_list[ec_idx1][1]
        node1 = master_coord_list.index(point1)
        node2 = master_coord_list.index(point2)
        adj_matrix[node1][node2] = 0
        adj_matrix[node2][node1] = 0

        #Checking if the updated graph satisfies all the 3 conditions
        if check_conditions(adj_matrix, optimize_flag=True):
            cost = calc_total_cost(adj_matrix, master_coord_list, optimize_flag=True)

            #Checking if newly calculated cost is lesser than the previous one
            if cost < optimized_cost:
                optimized_cost = cost
                list_as_array = []
                list_as_array = np.array(adj_matrix)
                #print(list_as_array)
                print('Minimum cost obtained from Greedy Local Search Algorithm {}'
.format(optimized_cost))

            #Iterating through all the edges and checking if edges can be removed
            for ec_idx2 in range(len(sorted_ec_list)):
```

```
            point11 = sorted_ec_list[ec_idx2][0]
            point12 = sorted_ec_list[ec_idx2][1]
            node11 = master_coord_list.index(point11)
            node12 = master_coord_list.index(point12)
            adj_matrix[node11][node12] = 0
            adj_matrix[node12][node11] = 0
            if check_conditions(adj_matrix, optimize_flag=True):
               cost = calc_total_cost(adj_matrix, master_coord_list, optimize_flag=True)
               if cost < optimized_cost:
                  optimized_cost = cost
                  list_as_array = []
                  list_as_array = np.array(adj_matrix)
                  #print(list_as_array)
                  print('Minimum cost obtained from Greedy Local Search Algorithm {}'
.format(optimized_cost))
               else:
                  pass
            else: # Reverting the changes if the subsequent edge removal does not yeild lowest
cost
               adj_matrix[node11][node12] = 1
               adj_matrix[node12][node11] = 1
      else:
         adj_matrix[node1][node2] = 1
         adj_matrix[node2][node1] = 1
      adj_matrix[node1][node2] = 1
      adj_matrix[node2][node1] = 1
```

```
#Original Heuristic Algorithm

def optimize2(master_edge_cost_list, original_total_cost, adj_matrix, cost_matrix,
master_coord_list):


    #highest weight in comparisons

    highest_weight = float('inf')


    # List showing which nodes are already selected so not to repeat the same node twice

    node1 = [False for node in range(16)]


    result_matrix = [[0]*16 for _ in range(16)]


    count = 0


    while(False in node1):

        min_weight = highest_weight

        start = 0

        end = 0

        for i in range(16):

            if node1[i]:

                for j in range(16):

                    # If the analyzed node have a path to the ending node AND its not included in
resulting matrix (to avoid cycles)

                    if (not node1[j] and cost_matrix[i][j]>0):

                        if cost_matrix[i][j] < min_weight:

                            min_weight = cost_matrix[i][j]

                            start, end = i, j
```

```
    node1[end] = True

    result_matrix[start][end] = min_weight

    if min_weight == highest_weight:
        result_matrix[start][end] = 0

    count += 1

    # This matrix will have minimum cost path from source to destination
    result_matrix[end][start] = result_matrix[start][end]

adj_matrix2 = [[0]*16 for _ in range(16)]

for i in range(16):
    for j in range(16):
        if result_matrix[i][j]!=0:
            adj_matrix2[i][j]=1

flag_check = True

#checking if resulted graph satisfies all the conditions
while not flag_check == check_conditions2(adj_matrix2):
    chk = list(map(sum, adj_matrix2))

    for i in range(0,16):
            for j in range(0,16):
```

```
                if adj_matrix[i][j] != adj_matrix2[i][j] and chk[i]<3:
                    adj_matrix2[i][j]=1
                    adj_matrix2[j][i]=1
                    chk = list(map(sum, adj_matrix2))


    #Calculating the total cost of the end graph
    cost = calc_total_cost(adj_matrix2, master_coord_list, optimize_flag=True)

    list_as_array = []
    list_as_array = np.array(adj_matrix2)
    #print(list_as_array)
    print('\n')
    print('Minimum cost obtained from Original Heuristic Algorithm is {}' .format(cost) )

if __name__ == "__main__":

    for i in range (0,5):
            generate_graph()
```