PROJECT-2

NETWORK TOPOLOGY DESIGN

ALGORITHMIC ASPECTS OF TELECOMMUNICATION NETWORKS

Submitted By:

PALLAVI DASARAPU

PXD210008

CONTENTS:

OBJECTIVE:

Given n (>=15) nodes on a plane and their coordinates, the key objective of this project is to create and implement two heuristic algorithms (different) of a network topology design, and experiment with it.

The network topology generated has the following properties:

1. The graph is connected, i.e. it contains all the given nodes.
2. The degree of every vertex in the graph is at least 3, i.e. each node is connected to 3 other nodes
3. The diameter of the graph is at most 4, i.e. any node can be reached from any other node in at most 4 hops. It is to be noted that this does not depend on the geometric distance.
4. The network topology has the lowest potential overall cost. The entire geometric length of all the links is used to compute this overall cost.

Only the fourth property is achieved by the heuristic algorithm. Every other characteristic is a constraint on the graph, requiring that it be present in the resultant network topology.

"Heuristic algorithms which prioritize speed over optimality and precision to achieve a solution that is quicker and efficient than the trivial techniques, are commonly used to tackle the decision issue class known as NP-complete problems. Heuristic approaches are typically employed when approximate answers are adequate but exact solutions are computationally expensive."[2] Numerous general purpose heuristic optimization algorithms are available, including Tabu search, Simulated Annealing, Greedy local search, and others.

HEURISTIC ALGORITHM & PSEUDOCODE – I

Greedy Local Search Algorithm:

Iteratively eliminating the edges with the highest cost while periodically determining if the resulting graph still meets the three (aforementioned) constraints is the basic notion behind this method. Calculate the whole cost once again if it does. This will continue until the conditions are no longer met by the graph.[1]

The program keeps eliminating the heavy weighted edges and ultimately gives the final optimum (least) total cost as the output.

a. The original graph will serve as the basis for the current solution, S, and the total cost of the solution, C(S).
b. Pick an edge that has the most weight and attempt to remove it.
c. Verify that the new solution S' continues to fulfill all the constraints.
d. If "yes," change C(S) to C(S') as the current total cost.
e. Loop until the conditions are no longer met by the answer.

Pseudocode:

```
Cal_cost = sum(costs)
Sort(costs)
pointr = 0
while(len(costs)){
  Eliminate costs [pointr]
  if constraints satisfy
  Calculate Opt_cost
  if opt_cost<cal_cost
    then cal_cost = opt_cost
    pointr++
}
Else
  then add back removed edges
  Iterate from the next max weighed edge
```

HEURISTIC ALGORITHM & PSEUDOCODE – II

Heuristic Search Algorithm:

A sub-graph with a minimal cost path to reach a node from another node by traveling via every node in the inputted graph will be built in this case. If this sub-graph satisfies the constraints, the network's lowest overall cost will be determined based on its total cost. Otherwise, the topology will be updated with the subsequent lowest cost edge, and the conditions will be checked once more, and so forth until a sub-graph is created that meets the requirements.[2]

a. Creating a sub-graph from the main graph by finding the minimum path between a source and destination that passes through every node in the network.
b. If all the 3 conditions against the sub-graph are fulfilled, then the overall cost of the sub-graph will be determined.
c. Otherwise, add the subsequent minimum-weighted edges to the sub-graph repeatedly until all the requirements are met.
d. Substitute the newly calculated cost for the optimized cost now.

Pseudocode:

```
Visited = [0]
Unvisited = [1, 2, ..., 18];
Sub = [];
while ( Unvisited != [] ){
   Find edge e = (x, y) such that:
      1. x in Visited
      2. y in Unvisited
      3. ec edge with min cost
   Sub = Sub.append(ec)
   Visited = Visited.append(y)
   Unvisited = Unvisited.remove(y)
}
Iterate{
   if (all constraints):
      calculate total cost
```

```
  Else:
     sub_graph.append(next min. wgt)
}
```

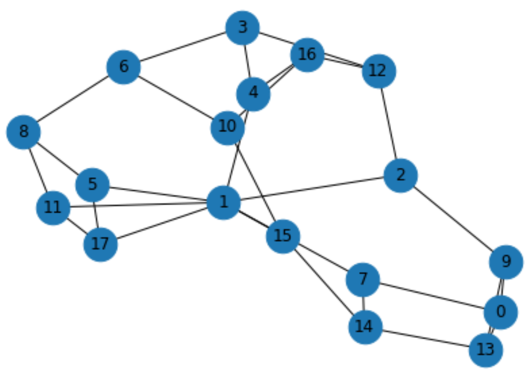
APPROACH:

1. To achieve, we construct a network of 18 nodes, and two nodes are chosen at random to create the edges connecting them.
2. We then determine if the resulting graph complies with the constraints and regenerate the graph otherwise.
3. Verify the nodes' connectivity first using the BFS algorithm.
4. Then verify if each node has a degree greater than 3 by counting the # of ones in each row of the adjacency matrix.
5. Then measure the diameter of each node, which should not exceed 4 using the built-in Python "dijkstra" function.
6. For randomly chosen coordinates of each node, the weights of each edge will be determined using the built-in Python function for calculating Euclidean distance. The weights of all the edges are added to determine the network's total cost.
7. We then implement Greedy Search and Heuristic Search algorithms to find the optimum cost for the randomly generated coordinates for five iterations.
8. Coordinates are generated randomly in a range of 80 and all these values are displayed.
9. For every set of 18 nodes, the optimum costs and runtimes for both the heuristic algorithms are calculated and displayed along with the respective network topologies.

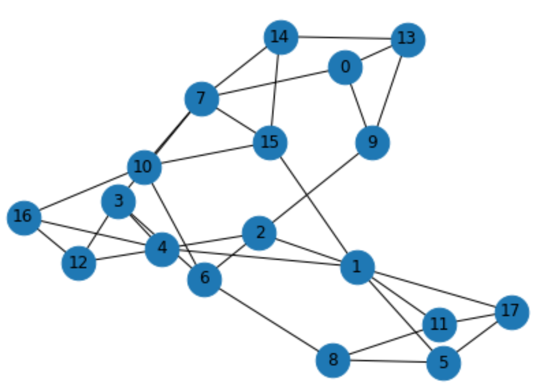
OUTPUTS/ GRAPHS:

Iteration-1:

```
Coordinates are [[2, 56], [32, 9], [78, 10], [37, 38], [46, 50], [55, 33], [55, 0], [4, 51], [40, 51], [73, 40], [38, 32], [33, 0], [32, 47], [7, 9], [49, 63], [37, 35], [5, 52], [18, 9]]
```



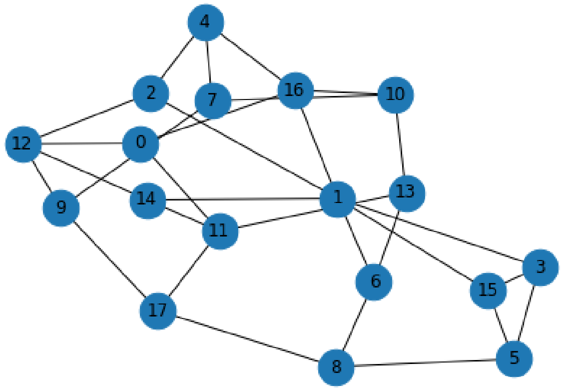
Greedy Search Optimum-cost 1053.017
Greedy Search Runtime 3.297ms



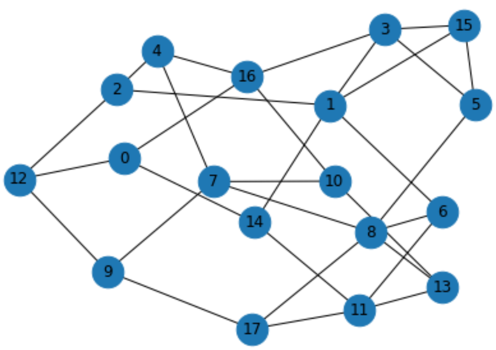
Heuristic Search Optimum-cost 1208.358
Heuristic Search Runtime 0.075ms

Iteration -2:

Coordinates are [[75, 42], [19, 25], [54, 27], [31, 48], [79, 36], [51, 9], [23, 39], [49, 23], [63, 9], [49, 6], [4, 72], [2, 8], [32, 33], [60, 38], [52, 8], [18, 12], [34, 32], [34, 15]]



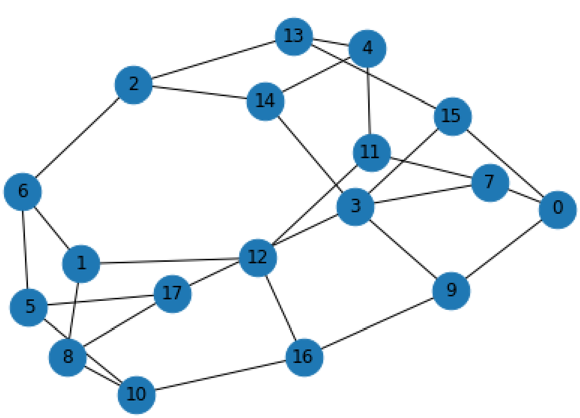
Greedy Search Optimum-cost 1022.027
Greedy Search Runtime 4.905ms



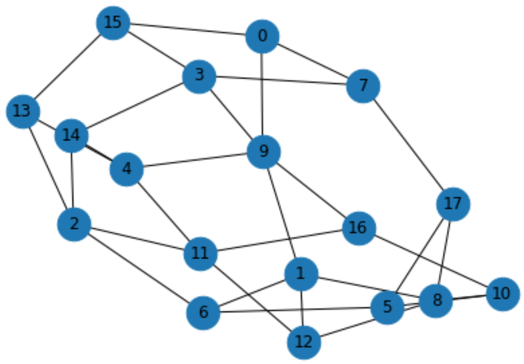
Heuristic Search Optimum-cost 1341.484
Heuristic Search Runtime 0.099ms

Iteration – 3:

Coordinates are [[21, 52], [62, 58], [27, 54], [6, 0], [26, 54], [39, 12], [78, 66], [12, 51], [32, 31], [51, 24], [72, 53], [31, 0], [49, 67], [73, 12], [26, 76], [39, 50], [6, 44], [36, 40]]



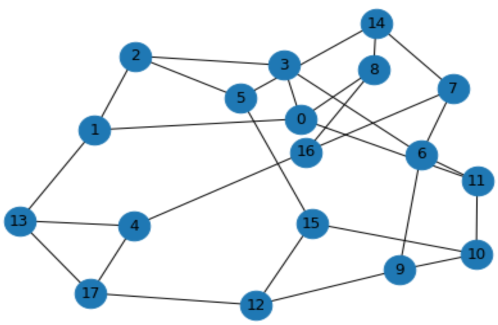
```
Greedy Search Optimum-cost 1120.041
Greedy Search Runtime 5.358ms
```



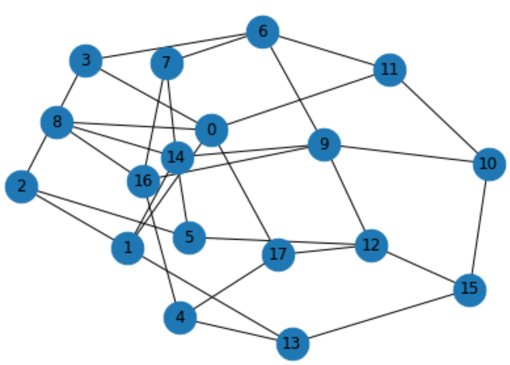

```
Heuristic Search Optimum-cost 1148.809
Heuristic Search Runtime 0.093ms
```

Iteration – 4:

```
Coordinates are [[17, 40], [8, 47], [41, 71], [65, 66], [44, 1], [78, 72],
[29, 10], [51, 59], [79, 45], [53, 42], [62, 17], [48, 65], [24, 27], [77,
45], [17, 31], [55, 33], [69, 11], [3, 29]]
```



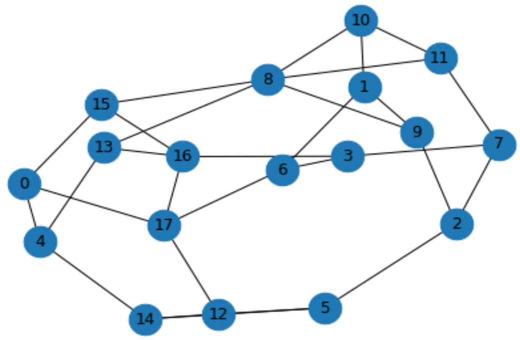

```
Greedy Search Optimum-cost 1243.921
Greedy Search Runtime 9.81ms
```



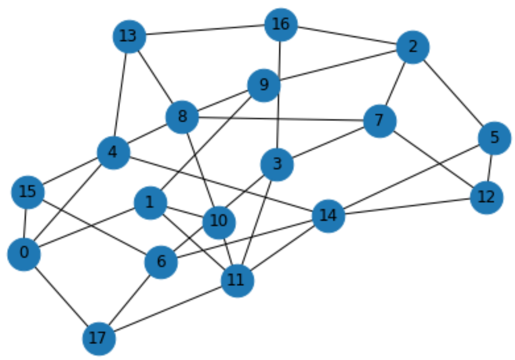

```
Heuristic Search Optimum-cost 1399.655
Heuristic Search Runtime 0.099ms
```

Iteration -5:

```
Coordinates are [[20, 29], [26, 34], [62, 25], [53, 16], [16, 40], [72,
31], [36, 56], [45, 78], [27, 6], [21, 74], [2, 10], [55, 63], [50, 36],
[15, 10], [36, 43], [1, 55], [4, 70], [26, 20]]
```



Greedy Search Optimum-cost 880.148
Greedy Search Runtime 7.84ms



Heuristic Search Optimum-cost 1413.136
Heuristic Search Runtime 0.099ms

CONCLUSION:

In Greedy Search algorithm, as the network iterates, its total cost reduces. Based on the graph, the number of iterations required to attain the lowest cost differs.[1] Some graphs may require more iterations than others to get the lowest total cost.

In the heuristic search algorithm, on the other hand, the graph is at its lowest total cost as soon as all three requirements are met.[2] Thus, only the lowest output is displayed.

As can be witnessed from the examples above, almost always, the greedy search algorithm yields a more optimized graph than the heuristic algorithm. Since the heuristic technique is not iterative, it has a substantially shorter run time than the greedy search algorithm. In other words, the number of iterations affects run time. One can see that the runtime required by the algorithm for a graph with a higher starting total cost is somewhat longer than for networks that have a slight bit lower initial cost.

REFERENCES

1. Greedy Search Algorithm: https://www.cs.jhu.edu/~mdinitz/classes/ApproxAlgorithms/Spring2019/Lectures/lecture5.pdf
2. Heuristic Algorithms: https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms
3. Display Graphs from adjacency matrix: https://stackoverflow.com/questions/29572623/plot-networkx-graph-from-adjacency-matrix-in-csv-file
4. BFSearch: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
5. Graph Tools: https://graph-tool.skewed.de/static/doc/quickstart.html
6. Graphs: https://plotly.com/python/network-graphs/#create-network-graph
7. https://python-course.eu/machine-learning/data-representation-and-visualization-data.php
8. https://www.programiz.com/dsa/graph-bfs
9. https://www.cambridge.org/core/books/abs/design-of-approximation-algorithms/greedy-algorithms-and-local-search/5CB0128EBCA19A51353C64A425B57FFE
10. Professor's learning modules

APPENDIX:

BFSearch.py

```
import numpy as np
import networkx as ntx
import matplotlib.pyplot as mplt

class BFSearch:
    def BFSearch(adj_mat):
        check = False
        visit_list = [False for i in range(18)]
        nodes_visited = []
        visited_count = 0
        for i in range(len(adj_mat)):
            for j in range(18):
                # to verify edge between any 2 vertices
                if (not(visit_list[j]) and adj_mat[i][j] == 1
                        and (j not in nodes_visited)):
                        nodes_visited.append(j)
                        visited_count += 1
                        visit_list[j] = True
        # if all nodes are visited
        if (visited_count == 18):
            check = True

        return check

    def display_graph(adj_mat):
        rows, cols = np.where(adj_mat == 1)
        edges = zip(rows.tolist(), cols.tolist())
        graph = ntx.Graph()
        all_rows = range(0, adj_mat.shape[0])
        for n in all_rows:
            graph.add_node(n)
        graph.add_edges_from(edges)
        ntx.draw(graph, node_size=600, with_labels=True)
        mplt.show()
```

main.py

```
import random
import numpy as np
import timeit
from operator import itemgetter
from BFSearch import BFSearch
```

```
from graph_tools import *
from enum import Flag
import math
import sys


sys.setrecursionlimit(1_000_000)

nodes_list = list(range(0, 18))
adj_matrx = [[0] * 18 for _ in range(18)]
# to create an adjacent matrix 18x18 and a graph of 18 nodes
global graph
graph = Graph(directed=True)
for i in nodes_list:
    graph.add_vertex(i)


def set_adj_matrix():
    nodes_list = list(range(18))
    adj_mat = [[0] * 18 for _ in range(18)]
    cand_list = []

    for curr_cand in nodes_list:
        while True:
            random_cand = random.choice(nodes_list)
            if (random_cand != curr_cand):
                if (random_cand not in cand_list):
                    cand_list.append(curr_cand)

                    # updating the adjacency matrix
                    adj_mat[curr_cand][random_cand] = 1
                    adj_mat[random_cand][curr_cand] = 1

                    # adding the edges generated randomely
                    graph.add_edge(curr_cand, random_cand)
                    graph.add_edge(random_cand, curr_cand)

            if (len(cand_list) == 3):
                cand_list = []
                break

    first_constraint_check(adj_mat)


def first_constraint_check(adj_mat, is_optimal=False):
    # constraint to check if graph is a complete digraph
    if not BFSearch.BFSearch(adj_mat):
        if not is_optimal:
            set_adj_matrix()
        else:
            return False

    # constraint to check if the diameter is atleast 4
    for i in nodes_list:
        dis, pre = graph.dijkstra(i)
        if any(4 < value for value in dis.values()):
            if not is_optimal:
```

```
                set_adj_matrix()
            else:
                return False

    # constraint to check if degree of graph is at least 3
    for i in adj_mat:
        if i.count(1) < 3:
            if not is_optimal:
                set_adj_matrix()
            else:
                return False

    if not is_optimal:
        gen_node_coordinates(adj_mat)
    else:
        return True


def second_constraint_check(adj_mat):
    # constraint to check if graph is a complete digraph
    if not BFSearch.BFSearch(adj_mat):
        return False

    # constraint to check if the diameter is atleast 4
    for i in nodes_list:
        dis, pre = graph.dijkstra(i)
        if any(4 < value for value in dis.values()):
            return False

    # constraint to check if degree of graph is at least 3
    for i in adj_mat:
        if i.count(1) < 3:
            return False

    return True


# this generates the (x,y) coordinates
# of every node on the graph
def gen_node_coordinates(adj_mat):
    # selects coordinates randomly in range of 0-80
    coord_range = list(range(0, 80))
    coord_list = []
    while (len(coord_list) < 18):
        x_coord = random.choice(coord_range)
        y_coord = random.choice(coord_range)
        x_y = [x_coord, y_coord]
        if x_y not in coord_list:
            coord_list.append(x_y)

    # prints all coordinates list
    print('Coordinates are {}\n'.format(coord_list))

    total_cost = total_costs(adj_mat, coord_list)


# greedy search
```

```python
def greedy_search(edges_cost, opt_cost,
                  adj_mat, coord_list):
    # the edges with max weight are removed from the graph
    edg_costs_sorted = sorted(edges_cost,
                              key=itemgetter(2), reverse=True)
    for i in range(len(edg_costs_sorted)):
        coord_1 = edg_costs_sorted[i][0]
        coord_2 = edg_costs_sorted[i][1]
        node_1 = coord_list.index(coord_1)
        node_2 = coord_list.index(coord_2)

        adj_mat[node_1][node_2] = 0
        adj_mat[node_2][node_1] = 0

        # the resultant graph is checked for constraints
        if first_constraint_check(adj_mat, True):
            # if yes, calculate the total cost
            o_cost = total_costs(adj_mat, coord_list, True)

            # check if this new cost is less than the earlier cost
            if o_cost < opt_cost:
                opt_cost = o_cost

                # Loop through all edges to remove any edges
            for j in range(len(edg_costs_sorted)):
                coord_11 = edg_costs_sorted[j][0]
                coord_22 = edg_costs_sorted[j][1]
                node_11 = coord_list.index(coord_11)
                node_22 = coord_list.index(coord_22)

                adj_mat[node_11][node_22] = 0
                adj_mat[node_22][node_11] = 0

                # the resultant graph is checked for constraints
                if first_constraint_check(adj_mat, True):
                    # if yes, calculate the total cost
                    o_cost = total_costs(adj_mat, coord_list, True)
                    if o_cost < opt_cost:
                        opt_cost = o_cost
                    else:
                        pass
                # if the total cost of resultant is not optimum
                # revert the removed edge
                else:
                    adj_mat[node_11][node_22] = 1
                    adj_mat[node_22][node_11] = 1
        else:
            adj_mat[node_1][node_2] = 1
            adj_mat[node_2][node_1] = 1
        adj_mat[node_1][node_2] = 1
        adj_mat[node_2][node_1] = 1

    # display the graph of the adjacent matrix
    mat_array = []
    mat_array = np.array(adj_mat)
    BFSearch.display_graph(mat_array)
    print('Greedy Search Optimum-cost {}'.format(opt_cost))
```

```python
# heuristic
def heuristic_search(edges_cost, final_total_cost,
                     adj_mat, cost_mat, coord_list):
    # Let the max weight
    max_wgt = float('inf')

    # list the selected nodes to avoid redundant selections
    node_1 = [False for i in range(18)]

    res_mat = [[0] * 18 for _ in range(18)]

    count = 0
    while (False in node_1):
        min_wgt = max_wgt
        begin = 0
        last = 0
        for x in range(18):
            if node_1[x]:
                for y in range(18):
                    # avoid cycles in the graph
                    if (not node_1[y] and cost_mat[x][y] > 0):
                        if (cost_mat[x][y] < min_wgt):
                            min_wgt = cost_mat[x][y]
                            begin, last = x, y

        node_1[last] = True

        res_mat[begin][last] = min_wgt

        if min_wgt == max_wgt:
            res_mat[begin][last] = 0

        count += 1

        # resultant matrix should have path with min cost possible
        res_mat[last][begin] = res_mat[begin][last]

    adj_mat_2 = [[0] * 18 for _ in range(18)]

    for i in range(18):
        for j in range(18):
            if res_mat[i][j] != 0:
                adj_mat_2[i][j] = 1

    flg = True
    # check for constraints on the resultant graph
    while not flg == second_constraint_check(adj_mat_2):
        thisflg = list(map(sum, adj_mat_2))

        for i in range(18):
            for j in range(18):
                if adj_mat[i][j] != adj_mat_2[i][j] and thisflg [i] < 3:
                    adj_mat_2[i][j] = 1
                    adj_mat_2[j][i] = 1
                    thisflg = list(map(sum, adj_mat_2))
```

```
    # now calculate the total cost of resultant graph
    o_cost = total_costs(adj_mat_2, coord_list, True)

    # display graph of resultant adjacent matrix
    mat_array = []
    mat_array = np.array(adj_mat_2)
    BFSearch.display_graph(mat_array)
    print('Heuristic Search Optimum-cost {}\n'.format(o_cost))


def total_costs(adj_mat, coord_list, is_optimal=False):
    cost_mat = [[0] * 18 for _ in range(18)]

    # coordinate list
    temp_coords = [[x_i, y_i] for x_i, row in enumerate(adj_mat)
                   for y_i, i in enumerate(row) if i == 1]
    temps = [tuple(sorted(x)) for x in temp_coords]
    distinct_coords = [list(x) for x in {*temps}]


    # to calculate Euclidean distance between any two nodes
    edges = []
    for item in distinct_coords:
        coord = []
        for j in item:
            coord.append(coord_list[j])
        edges.append(coord)

    # to get edge weight
    edges_cost = []
    for i in edges:
        a = np.array(i[0])
        b = np.array(item[1])
        dis = round(np.linalg.norm(a - b), 3)

        edges_cost.append([i[0], i[1], dis])


    # sum up the edge cost
    tot_cost = 0
    for i in edges_cost:
        tot_cost += i[-1]
    final_totalcost = round(tot_cost, 3)

    # matrix with edge costs
    x = 0
    for i in range(18):
        for j in range(18):
            if i < j and adj_mat[i][j] == 1:
                cost_mat[i][j] = edges_cost[x][2]
                cost_mat[j][i] = edges_cost[x][2]
                x += 1

    if not is_optimal:
        tot_cost1 = timeit.timeit(lambda: greedy_search(
            edges_cost, final_totalcost, adj_mat, coord_list),
```

```
                                    number=1)
        print('Greedy Search Runtime {}ms'.format(round(tot_cost1, 3)))

        tot_cost2 = timeit.timeit(lambda: heuristic_search(
            edges_cost, final_totalcost, adj_mat, cost_mat, coord_list),
                                    number=1)
        print('Heuristic Search Runtime {}ms'.format(round(tot_cost2, 3)))

    if is_optimal:
        return final_totalcost


if __name__ == "__main__":

    for i in range(5):
        set_adj_matrix()
```

READ ME:

install python 3.9.12

1. pip install graphtools
2. pip install networkx = = 2.7.1

Steps to run the program:

- Copy the code in appendix and open files in Terminal/IDE to paste and save them. Maintain the same mentioned names for the files and both the files are supposed to be in same folder.
- Run the "main.py" file and the results will be displayed.
- It takes a little while for all the iterations to complete. The results will be displayed sequentially.