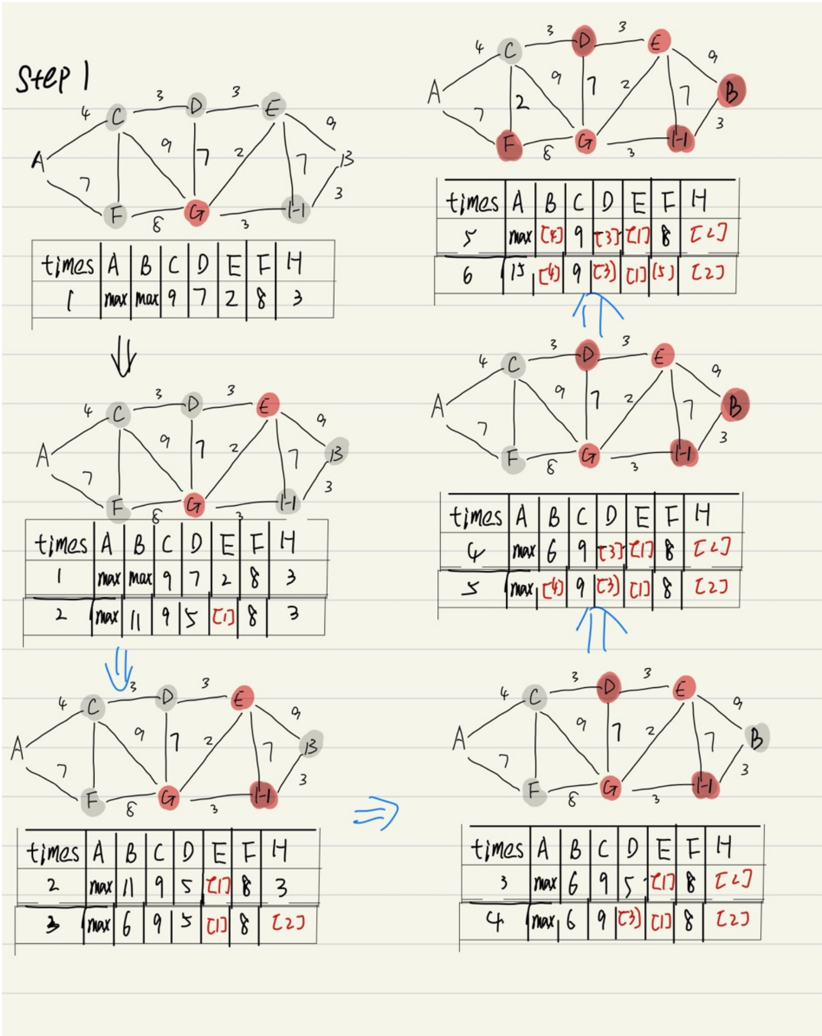
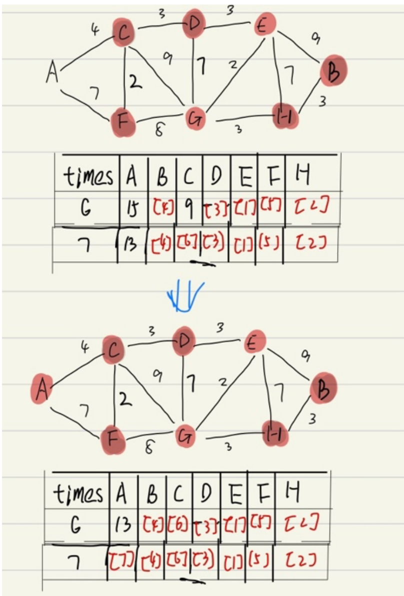
Problem 1.

1. Dijkstra's Algorithm can solve this problem. This algorithm could work on both directed and undirected graphs. By using greedy way find the shortest paths in the graph

   Demonstrate:





Resulting Routes:

Destination	Distance	Path
G -> A:	12	G->E->D->C->A
G -> B	6	G->H->B
G -> C	8	G->E->D->C
G -> D	5	G->E->D
G -> E	2	G -> E
G -> F	8	G -> F
G -> H	3	G -> H

2. Find one “optimal” location:

For this question I will use Dijkstra’s Algorithm. Firstly, I need to find distance than each point as a starting point to the farthest point. Secondly, use this starting point as a key, place the longest distance as a value in a tuple, and then add the tuple to the list. Until joining so the point. Finally, the tuples are sorted by value, returning the minimum key, which is the optimal point.

Algorithmic implementation:

```python
MAX= float('inf')
matrix = [[MAX, MAX, 4, MAX, MAX, 7, MAX, MAX],
          [MAX, MAX, MAX, MAX, 9, MAX, MAX, 3],
          [4, MAX, MAX, 3, MAX, 2, 9, MAX],
          [MAX, MAX, 3, MAX, 3, MAX, 7, MAX],
          [MAX, 9, MAX, 3, MAX, MAX, 2, 7],
          [7, MAX, 2, MAX, MAX, MAX, 8, MAX],
          [MAX, MAX, 9, 7, 2, 8, MAX, 3],
          [MAX, 3, MAX, MAX, 7, MAX, 3, MAX]]

def dijkstra(matrix, start_node):
    matrix_length = len(matrix)
    used_node = [False] * matrix_length
    distance = [MAX] * matrix_length
    distance[start_node] = 0
    while used_node.count(False):
        min_value = float('inf')
        min_value_index = 999
        for index in range(matrix_length):
            if not used_node[index] and distance[index] < min_value:
                min_value = distance[index]
                min_value_index = index

        used_node[min_value_index] = True
        for index in range(matrix_length):
            distance[index] = min(distance[index], distance[min_value_index] + matrix[min_value_index][index])

    return distance

list1 = []
for i in range(8):
    result = dijkstra(matrix,i)
    result.sort()
    list1.append((chr(65+i), result[7]))

list1.sort(key = lambda x : x[1])
print(list1)
print(list1[0])
```

The Dijkstra's Running time is O(E log V)

If the n is the number of vertices, the time complexity is n * O(E log V)

3. According to the output of the code, the optimal town is E

```
[('E', 10), ('D', 11), ('G', 12), ('C', 14), ('F', 14), ('H', 15), ('A', 18), ('B', 18)]
('E', 10)
```

4. For this problem, I still use Dijkstra's Algorithm. First, we need to find all the cases where any two-point combination is. Second, in the current combination, the Dijkstra's algorithm is used to get their distance to each point. Then compare the distance between the two points

to the same point and take out a smaller distance to join the list. Third, take the maximum value from the list as the distance from this combination to the furthest. Finally, compare the distance from all combinations to the farthest point. The smallest combination is the optimal solution.

Algorithmic implementation:

```
MAX= float('inf')
matrix = [[MAX, MAX, 4, MAX, MAX, 7, MAX, MAX],
          [MAX, MAX, MAX, MAX, 9, MAX, MAX, 3],
          [4, MAX, MAX, 3, MAX, 2, 9, MAX],
          [MAX, MAX, 3, MAX, 3, MAX, 7, MAX],
          [MAX, 9, MAX, 3, MAX, MAX, 2, 7],
          [7, MAX, 2, MAX, MAX, MAX, 8, MAX],
          [MAX, MAX, 9, 7, 2, 8, MAX, 3],
          [MAX, 3, MAX, MAX, 7, MAX, 3, MAX]]

def dijkstra(matrix, start_node):
    matrix_length = len(matrix)
    used_node = [False] * matrix_length
    distance = [MAX] * matrix_length
    distance[start_node] = 0
    while used_node.count(False):
        min_value = float('inf')
        min_value_index = 999
        for index in range(matrix_length):
            if not used_node[index] and distance[index] < min_value:
                min_value = distance[index]
                min_value_index = index

        used_node[min_value_index] = True
        for index in range(matrix_length):
            distance[index] = min(distance[index], distance[min_value_index] + matrix[min_value_index][index])

    return distance
```

```python
def comparelist(tuple1, tuple2):
    resultlist = []
    for i in range(len(tuple1)):
        for j in range(len(tuple2)):
            if (tuple1[i][0] == tuple2[j][0]):
                if (tuple1[i][1] < tuple2[j][1]):
                    resultlist.append(tuple1[i])
                else:
                    resultlist.append(tuple2[j])
    return resultlist

def creatlist(n):
    list1 = []
    for i in range(n):
        list1.append(i)
    return list1
```

```python
def findpoints(matrix):
    alldis_list = []
    x,y = 0, 0
    for i in range(len(matrix)-1):
        y = x+1
        for _ in range(len(matrix)-1-i):
            val_listx = creatlist(len(matrix))
            del val_listx[y]
            listx = dijkstra(matrix, x)
            del listx[y]
            tuplex = list(zip(val_listx,listx))


            val_listy = creatlist(len(matrix))
            del val_listy[x]
            listy = dijkstra(matrix, y)
            del listy[x]
            tupley = list(zip(val_listy,listy))


            finallist = comparelist(tuplex, tupley)
            finallist.sort(key = lambda x : x[1])
            maxd = finallist[5][1]
            alldis_list.append((chr(x+65), chr(y+65), maxd))
            y += 1
        x += 1
    alldis_list.sort(key= lambda x : x[2])
    return alldis_list

print(findpoints(matrix))
```

Theoretical running time: O(V^3)

5.

```
[('C', 'H', 5), ('A', 'G', 6), ('B', 'C', 6), ('C', 'G', 6), ('F', 'G', 6), ('F', '
H', 6), ('A', 'H', 7), ('B', 'D', 7), ('D', 'G', 7), ('D', 'H', 7), ('A', 'B', 8),
('A', 'E', 8), ('B', 'F', 8), ('C', 'E', 8), ('D', 'E', 8), ('E', 'F', 8), ('B', 'E
', 10), ('E', 'G', 10), ('E', 'H', 10), ('A', 'D', 11), ('C', 'D', 11), ('D', 'F',
11), ('B', 'G', 12), ('G', 'H', 12), ('A', 'C', 14), ('A', 'F', 14), ('C', 'F', 14)
, ('B', 'H', 15)]
```

In the output of the code, we can see the “C” and “H” are the optimal towns to place the two distribution centres.

Problem 2.

1. **Verbal description:**

   I used Kruskal Algorithm to solve this MST problem. This algorithm is seeming like greedy algorithms, which find the optimum in the hopes of finding a global optimum. In the Kruskal algorithm, 1. need sort each edge according to the distance. 2. Take the minimum distance and add it to the spanning tree. If adding the edge created a cycle, then reject this edge. 3. Keep adding edges until all the points are connected. In the main function, we can just need add the distance of edge which in the spanning tree together.

2. **Pseudocode:**

```
def findParent(parent, n):
    if parent[n] == n:
        return n
    return findParent(parent, parent[n])

def union(parent, x, y):
    parent[findParent(parent,x)] = findParent(parent,y)

def kruskal_algo(g, n):
    g <- sort according x[2]
    for node in range(n):
        parent[] <- node
    for i in range(len(g)):
        u, v, w = g[i]
        x = findParent(parent, u)
        y = findParent(parent, v)
        if (x != y):
            result += w
            edge += 1
            union(parent, x, y)
        if (edge == n-1):
            return result

def handlefile(filename):
    f = open(filename)
    lines <= f.readlines()
    test_cases <-lines[0]
    for i in range(test_cases):
        n_vertex <- lines[lines_idx]
        for _ in range(n_vertex):
            point[] <- lines[lines_idx]
            all_points.append(point)
            lines_idx += 1
        g = creat_graph(all_points)
        print("Test case {0}: MST weight {1}".format(i+1, kruskal_algo(g, n_vertex)))
```

3. Theoretical running time:

   O(E log V)

   1. For loop each vertex, put them into a list: O(V)

   2. Sort the edge according to the weight: O(E log E)

   3. O(E) calls to Findparent() and union(). Each take O(log V)

   4. Because |E| < V*V => log E = 2logV

      The total running time we can write O(E log V)