Problem Set 7

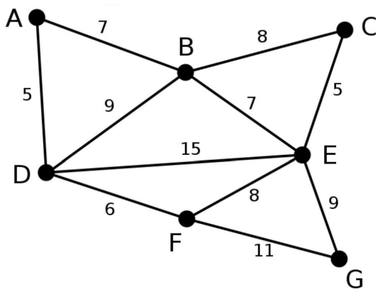
CS 5800

Submitted by : Hardik Bishnoi

NUID : 002807991

Problem 1

(a) Consider the following weighted undirected graph with 7 vertices and 11 edges.



Apply Dijkstra's Algorithm on the graph above, to determine the shortest distance from vertex G to each of the other six vertices (A, B, C, D, E, F). Clearly show all of your steps.

Solution 1(a):

Dijkstra's Algorithm is a pathfinding algorithm used to find the shortest path from a source vertex to a destination vertex in a weighted graph.

Here is how Dijkstra's Algorithm will be applied on the graph above:

Step #1

First, we pick a source node, which is G. It is from G that we will map the distances of all the other nodes which are A,B,C,D,E and F.

Then, we will initialize the distance of all the nodes from the source nodes. Since we only know the distance of the source node from itself, that is 0, and do not know the distances of the other nodes from the source node, we will initialize the unknown distances as ∞.

We can produce a table to demarcate the values of the distances of each node from the source node:

Vertices	**Distance from G**
G	0
A	∞
B	∞
C	∞
D	∞
E	∞
F	∞

Next, we will initialize a visited pool of nodes and an unvisited pool of nodes which can be represented by the arrays, V and U respectively.

$$V = []$$
$$U = [A, B, C, D, E, F, G]$$

<u>Step #2</u>

Now, as long as U is not empty, we will find the node with the smallest tentative distance from the source node that is also unvisited in the table and mark it as current. This node is G in the first iteration.

<u>Step #3</u>

Next, we explore the neighbors of G. This means calculating the tentative distance of the neighboring nodes, which is equal to the sum of the distance of the current node from the source node and the weight of the edge connecting the current node and the neighbors.

The unvisited neighbors of G are F and E. If the found distance of the neighbors are less than the current tentative distance, we update the tentative distance in the table. Since GE = 9 and GF = 11, we have :

Vertices	**Distance from G**
G	0
A	∞
B	∞
C	∞
D	∞
E	9
F	11

<u>Step #4</u>

After visiting the nodes and updating the table, we mark the current node as visited.

The current node is G and hence we will add it to the array V and remove it from the array U:

$$V = [G]$$
$$U = [A, B, C, D, E, F]$$

<u>Step #5</u>

Next, we will again find the unvisited node with the smallest tentative distance in the table. We find that the node is E and we select E as our current node.

<u>Step #6</u>

We then explore the neighbors of the current node, E. The neighbors are F, D, B, C.

We then calculate the tentative distances of the neighbors from the source nodes through E:

$$F = 9 + 8 = 17$$
$$D = 9 + 15 = 24$$
$$B = 9 + 7 = 16$$
$$C = 9 + 5 = 14$$

We will now find if the distances are less than the current distance in the table. The distances of D, B and C are less than the current tentative distance in the table but not of F. Hence, we update the values of D, B, C in the table.

Vertices	**Distance from G**
G	0
A	∞
B	16
C	14
D	24
E	9
F	11

<u>Step #5</u>

Since we have explored all the neighbors of E which is the current node, we will mark it as visited. This means we will remove it from U and add it to V:

$$V = [G, E]$$
$$U = [A, B, C, D, F]$$

<u>Step#6</u>

Next, we will select the next current node by picking the unvisited node that has the smallest tentative distance from the source nodes out of all the other nodes. This node is F and we set it as our current node.

<u>Step #7</u>

We then explore the unvisited neighbors of F which is only D (since E and G are already visited). We then find the tentative distance of D from the source node G through F:

$$D = 11 + 6 = 17$$

Since this is smaller than the current tentative distance 24, we will update the table with the distance:

Vertices	**Distance from G**
G	0
A	∞
B	16
C	14
D	17

Vertices	Distance from G
E	9
F	11

Step #8

We will mark the current node F as visited since we have explored all of its neighbors. This means we will remove it from U and add it to V:

$$V = [G, E, F]$$
$$U = [A, B, C, D]$$

Step #9

We then find the unvisited node with the shortest tentative distance from the source node. This node is C. We select it as the current node.

Step #10

We then find the unvisited neighbors of C. It has only one unvisited neighbor, B (since E is already visited). We then find the tentative distance of B from G through C:

$$B : 14 + 8 = 22$$

Since this is larger than the current tentative distance, 16, we will not update the table.

The current table stays the same :

Vertices	Distance from G
G	0
A	∞
B	16
C	14
D	17
E	9
F	11

Step #11

Since all the neighboring nodes of C are explored, we mark C as visited. This means we remove it from U and add it to V:

$$V = [G, E, F, C]$$
$$U = [A, B, D]$$

Step #12

We then pick the next current node by finding the unvisited node with the smallest tentative distance in the table. This node is B. We mark B as the current node.

Step #13

Next, we explore the unvisited neighbors of the current node B. We find that the neighbors are A and D (since C and E are already visited).

We then find the tentative distance of A and D from G through B:

$$A : 16 + 7 = 23$$
$$D : 16 + 9 = 25$$

Since the distance for A is less than the current tentative distance in the table, we update it. Since the distance for D is more than the current tentative distance in the table, we do not update it.

Hence the updated table will look like this:

Vertices	**Distance from G**
G	0
A	23
B	16
C	14
D	17
E	9
F	11

Step #14

Since we have explored all the neighbors of B, we will mark it as a visited node. This means we will remove it from U and add it to V:

$$V = [G, E, F, C, B]$$
$$U = [A, D]$$

Step #15

Next, we will choose the unvisited node with the lowest tentative distance in the table from the source node as the current node. This node is D.

Step #16

We will now explore the neighbors of D. The neighbor is A. We will now calculate the tentative distance of A from the source node G through D:

$$A : 17 + 5 = 22$$

Since the calculated distance is lower than the tentative distance of A from G in the table, we will update the table as follows:

Vertices	**Distance from G**
G	0

Vertices	**Distance from G**
A	22
B	16
C	14
D	17
E	9
F	11

Step #17

Since we have explored all the neighbors of D, we will mark the node D as visited. Hence we will remove it from U and add it to V:

$$V = [G, E, F, C, B]$$
$$U = [A, D]$$

Step #18

Next we will pick a new current node which is $\boldsymbol{A}$. We will visit the neighbors of A which are not visited yet. Since there are no neighbors, we will mark A as visited and remove it from U and add it to V.

Hence the final table looks like this:

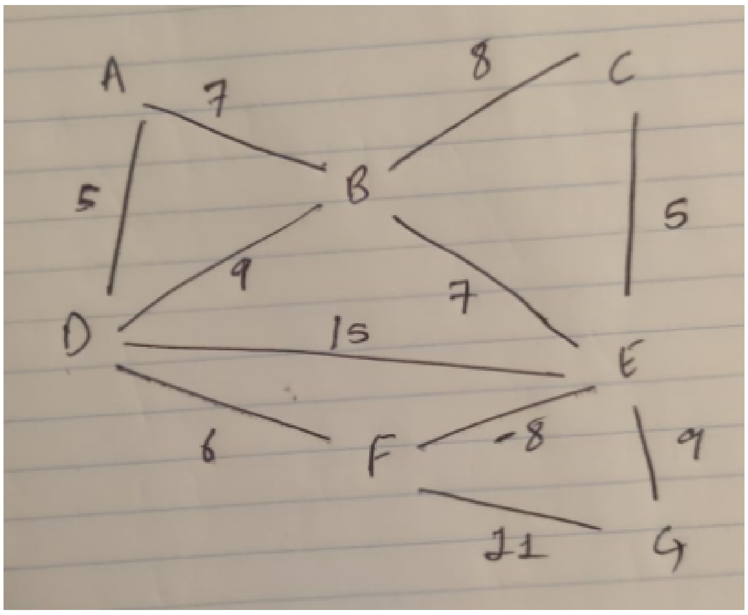
Vertices	**Distance from G**
G	0
A	22
B	16
C	14
D	17
E	9
F	11

These are the shortest distances of all the nodes from the source. Hence we have successfully applied Dijkstra's algorithm.

(b) Now suppose we change the weight of edge EF from +8 to −8. What happens? Using this example, explain why Dijkstra's Algorithm can produce incorrect outputs when one or more edges is negative.

Solution 1(b):

After changing the weight, here is how the graph will look like:



If we apply Dijkstra's algorithm to this graph, we will first explore the source node G, then move to its neighbors E and F. Since E has the smallest tentative weight we will use E as the next current node and explore its neighbors. This will result in a table as follows:

Vertices	Distance from G
G	0
A	∞
B	16
C	14
D	24
E	9
F	1

After this, we will mark the node E visited. However, this will produce an incorrect final output once Dijkstra's algorithm is done with exploring all the nodes because the minimum distance of E from the source node is not $EG = 9$, but is equal to $FG + FE = 11 + (-8) = 3$. This will cause all the shortest paths going through E to also have the incorrect distance. Since this algorithm marks the nodes as visited without finding the optimal path to the node itself through its neighbors, we will never find all the shortest paths for a graph with some negative weights.

(c) Determine a precise Loop Invariant for the Dijkstra's Algorithm, clearly stating your Initialization, Maintenance, and Termination statements. Prove that your loop invariant holds, clearly and carefully justifying each step in your proof.

Solution 1(c):

First we will define Dijkstra's algorithm through the help of a pseudocode:

```
function Dijkstra(graph, source):
    dist = array of distances initialized to infinity for all vertices
    dist[source] = 0
    S = empty set to keep track of visited vertices

    while S does not include all vertices:
        u = vertex not in S with the minimum dist[u]


        S.add(u)

        for each neighbor 'v' of 'u' not in S:
            new_dist = dist[u] + weight(u, v)
            if new_dist < dist[v]:
                dist[v] = new_dist
```

Explanation of Pseudocode:

First the algorithm initializes all distances except the source node to infinity since they are not known. We set the distance of source node from source node as 0. We initialize an empty set S to keep track of visited vertices. Then, we find the vertex 'u' not in S with the minimum dist[u], and we add u to S. We then run a for loop scanning the neighbors of u and update their distance values in dist by calculating the potential new distance through u. If new distance is shorter than the old distance we replace it. When the loop terminates, dist contains the minimum distances from all vertices to the source.

Proof:

Loop Invariant: At the beginning of each iteration of the main loop of Dijkstra's algorithm, $dist[v]$ is the minimum distance from the source vertex to vertex v, either directly or through a path of visited vertices.

Initialization:

$dist[source] = 0$, which is the correct minimum distance from the source vertex to itself. For all other vertices v, $dist[v]$ is initially set to infinity, which is a conservative estimate of the distance.

So, the loop invariant holds before the first iteration.

Maintenance:

In each iteration, we select a vertex u not in S with the minimum $dist[u]$ value. By selecting `u` in this way, we ensure that $dist[u]$ is the minimum distance to u from the source vertex through the current set of visited vertices.

For each neighbor v of u not in S, we update $dist[v]$ by taking the minimum between its current value and the distance from the source to u, plus the weight of the edge (u, v). This update ensures that $dist[v]$ is the minimum distance to vertex `v` from the source vertex either directly or through the current set of visited vertices.

Therefore, the loop invariant is maintained during each iteration.

To prove the correctness of Dijkstra's algorithm using a loop invariant, we can follow these steps:

Termination:

When the main loop terminates, all vertices are included in S, and $dist[v]$ contains the minimum distance from the source vertex to vertex v for all vertices.

Conclusion:

Since the loop invariant holds before the first iteration, is maintained during each iteration, and holds after the loop terminates, we can conclude that Dijkstra's algorithm correctly computes the minimum distances from the source vertex to all other vertices in the graph.

Problem 2

(a) Let G be a graph with V vertices and E edges. One can implement Kruskal's Algorithm to run in O(E log V) time, and Prim's Algorithm to run in O(E + V log V) time. If G is a dense graph with an extremely large number of vertices, determine which algorithm would output the minimum-weight spanning tree more quickly. Clearly justify your answer.

Solution 2(a):

If we are given a large <u>dense</u> graph G, this means that the number of vertices V are very high but the number of edges E are significantly higher than the vertices because each vertex is connected to multiple other vertices. Hence,

$$E >> V$$

Now, with said information in mind let us consider how the runtimes of Prim's and Kruskal's algorithm are affected:

<u>Kruskal's Algorithm: </u>$O(E\, logV)$

In a dense graph, E is large, and $logV$ is relatively smaller than E. However, since the $logV$ term is multiplied by E, the overall complexity stays the same.

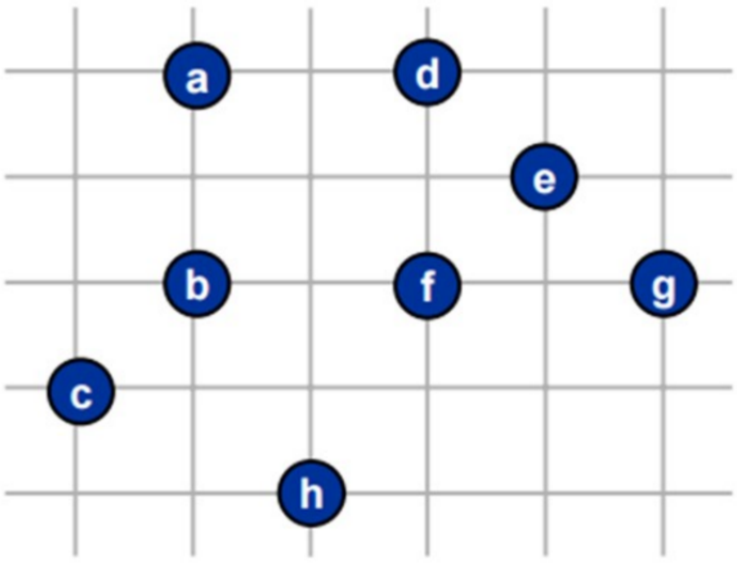
<u>Prim's Algorithm:</u> $O(E + VlogV)$

In a dense graph, E is already large, and $VlogV$ is relatively smaller than E. So, the time complexity is dominated by the E factor. Hence we can discard the smaller term and the time complexity approaches $O(E)$.

In this analysis, we can see that for a dense graph with a large number of vertices, the time complexity of both Kruskal's and Prim's algorithms is primarily determined by the number of edges, E. However, since we ignore the lower order term in Prim's algorithm's complexity, it will be faster then Kruskal's algorithm for such a case because n Edges, $O(n)$ is faster than $O(nlogV)$ assuming a large value of V, $logV$ is positive.

<u>Hence Prim's algorithm will be always faster.</u>

(b) Consider eight points on the Cartesian two-dimensional x-y plane.



For each pair of vertices u and v, the weight of edge uv is the Euclidean (Pythagorean) distance between those two points.

Using the algorithm of your choice, determine one possible minimum-weight spanning tree and compute its total distance, rounding your answer to one decimal place. Clearly show your steps.

Solution 2(b):

I will use Prim's Algorithm to produce a minimum spanning tree.

To initialize Prim's algorithm, we first select an arbitrary point in a graph. Let this point be the point A on the Cartesian. Next, we will initialize a array V to keep track of visited nodes.

Step #1

We first visit A since it is our source node and mark it as visited:

$$V = [A]$$

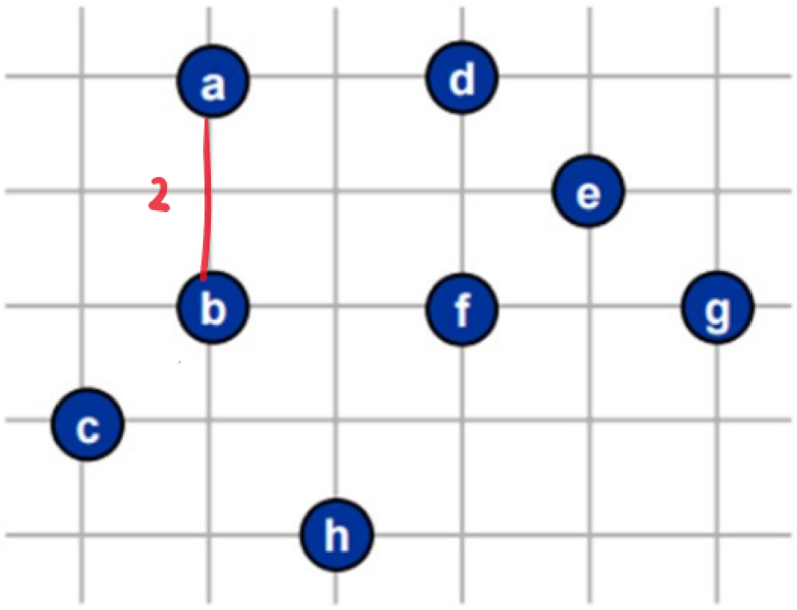
Let us compute the Euclidean distance of all the nodes from A:

$$AB = \sqrt{2^2 + 0^2} = 2$$
$$AC = \sqrt{3^2 + 1^2} = 3.2$$
$$AD = \sqrt{0^2 + 2^2} = 2$$
$$AE = \sqrt{0^2 + 3^2} = 3.2$$
$$AF = \sqrt{2^2 + 2^2} = 2.8$$
$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$

Hence, AB and AD take precedence. We will use alphabetical order as a tie-breaker. Hence we select AB.

We hence visit B and add it to the list of visited nodes.

$$V = [A, B]$$



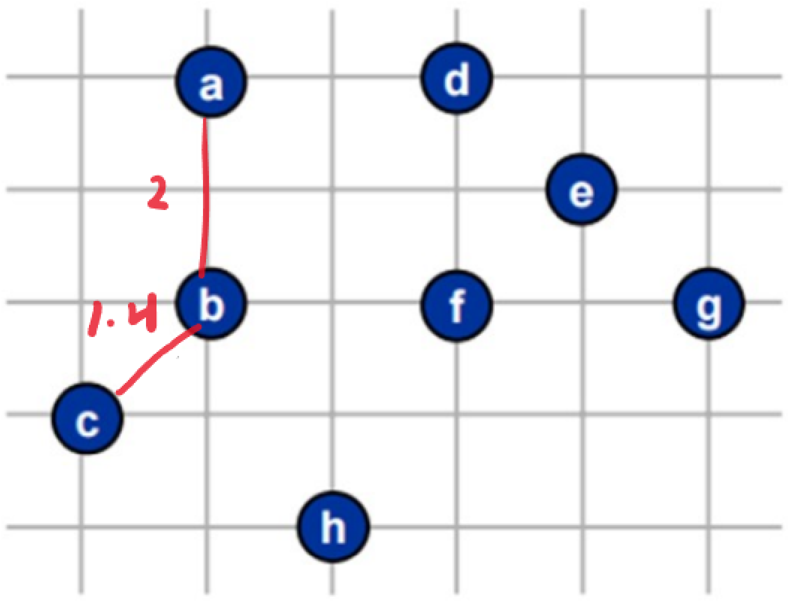
Step #2

We then check for any unvisited node that can be connected from A or B and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AC = \sqrt{3^2 + 1^2} = 3.2$$
$$AD = \sqrt{0^2 + 2^2} = 2$$
$$AE = \sqrt{0^2 + 3^2} = 3.2$$
$$AF = \sqrt{2^2 + 2^2} = 2.8$$
$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BC = \sqrt{1^2 + 1^2} = 1.4$$
$$BD = \sqrt{2^2 + 2^2} = 2.8$$
$$BE = \sqrt{1^2 + 3^2} = 3.2$$
$$BF = \sqrt{0^2 + 2^2} = 2$$
$$BG = \sqrt{0^2 + 4^2} = 4$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$

Since BC is the smallest path, we select C as the next node to visit and add it to V:

$$V = [A, B, C]$$



Step #3

We then check for any unvisited node that can be connected from A, B or C and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AD = \sqrt{0^2 + 2^2} = 2$$
$$AE = \sqrt{0^2 + 3^2} = 3.2$$
$$AF = \sqrt{2^2 + 2^2} = 2.8$$
$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BD = \sqrt{2^2 + 2^2} = 2.8$$
$$BE = \sqrt{1^2 + 3^2} = 3.2$$
$$BF = \sqrt{0^2 + 2^2} = 2$$
$$BG = \sqrt{0^2 + 4^2} = 4$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$
$$CD = \sqrt{3^2 + 3^2} = 4.2$$
$$CE = \sqrt{2^2 + 4^2} = 4.5$$
$$CF = \sqrt{1^2 + 3^2} = 3.2$$
$$CG = \sqrt{1^2 + 5^2} = 5.1$$
$$CH = \sqrt{1^2 + 2^2} = 2.2$$

We find that BF and AD are the two smallest paths. Hence we choose AD through alphabetical tiebreaking.

We add D to the list of visited nodes:

$$V = [A, B, C, D]$$

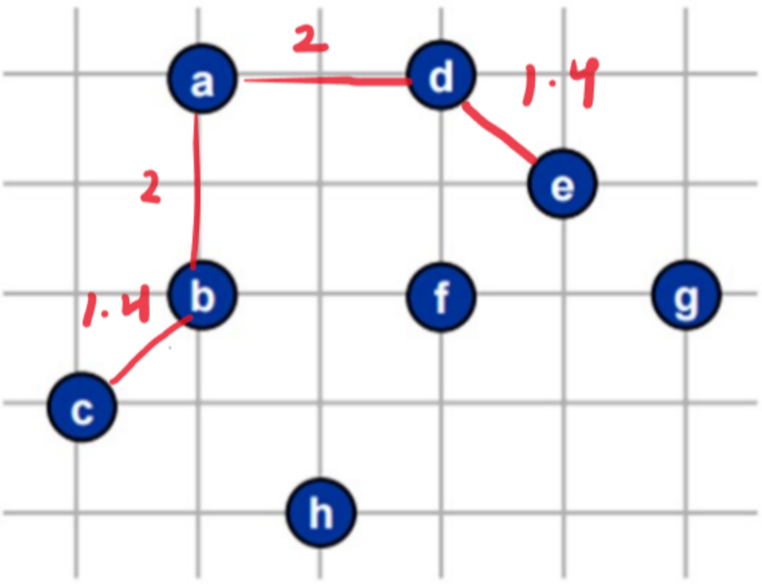
Step #4

We then check for any unvisited node that can be connected from A, B, C or D and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AE = \sqrt{0^2 + 3^2} = 3.2$$
$$AF = \sqrt{2^2 + 2^2} = 2.8$$
$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BE = \sqrt{1^2 + 3^2} = 3.2$$
$$BF = \sqrt{0^2 + 2^2} = 2$$
$$BG = \sqrt{0^2 + 4^2} = 4$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$
$$CE = \sqrt{2^2 + 4^2} = 4.5$$
$$CF = \sqrt{1^2 + 3^2} = 3.2$$
$$CG = \sqrt{1^2 + 5^2} = 5.1$$
$$CH = \sqrt{1^2 + 2^2} = 2.2$$
$$DE = \sqrt{1^2 + 1^2} = 1.4$$
$$DF = \sqrt{2^2 + 0^2} = 2$$
$$DG = \sqrt{2^2 + 2^2} = 2.8$$
$$DH = \sqrt{4^2 + 1^2} = 4.1$$

We find that DE is the smallest distance, hence we visit E and mark it as visited by entering it into V:

$$V = [A, B, C, D, E]$$



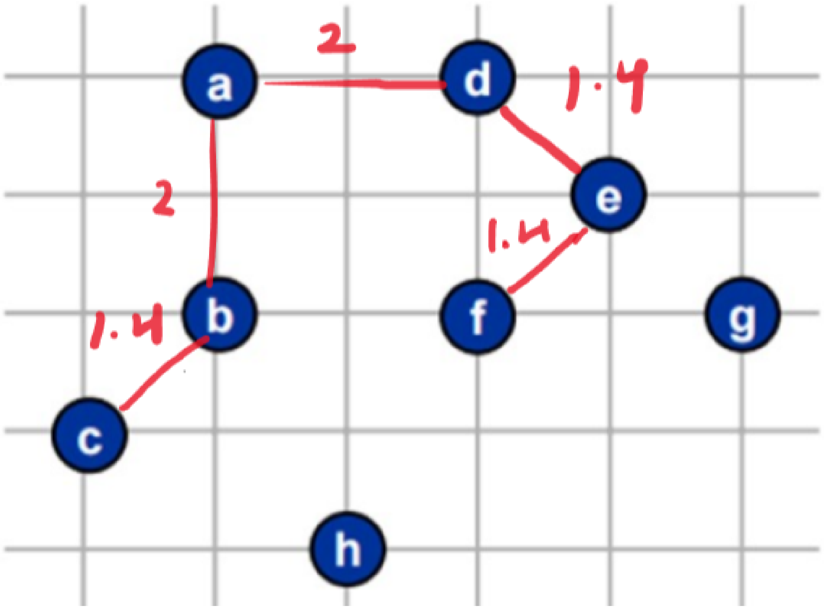
Step #5

We then check for any unvisited node that can be connected from A, B, C, D or E and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AF = \sqrt{2^2 + 2^2} = 2.8$$
$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BF = \sqrt{0^2 + 2^2} = 2$$
$$BG = \sqrt{0^2 + 4^2} = 4$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$
$$CF = \sqrt{1^2 + 3^2} = 3.2$$
$$CG = \sqrt{1^2 + 5^2} = 5.1$$
$$CH = \sqrt{1^2 + 2^2} = 2.2$$
$$DF = \sqrt{2^2 + 0^2} = 2$$
$$DG = \sqrt{2^2 + 2^2} = 2.8$$
$$DH = \sqrt{4^2 + 1^2} = 4.1$$
$$EF = \sqrt{1^2 + 1^2} = 1.4$$
$$EG = \sqrt{1^2 + 1^2} = 1.4$$
$$EH = \sqrt{3^2 + 2^2} = 3.6$$

We find that EF and EG are the smallest distances. Hence we select EF through alphabetical tiebreaking and visit the node F. We enter it into the array V:

$$A = [A, B, C, D, E, F]$$



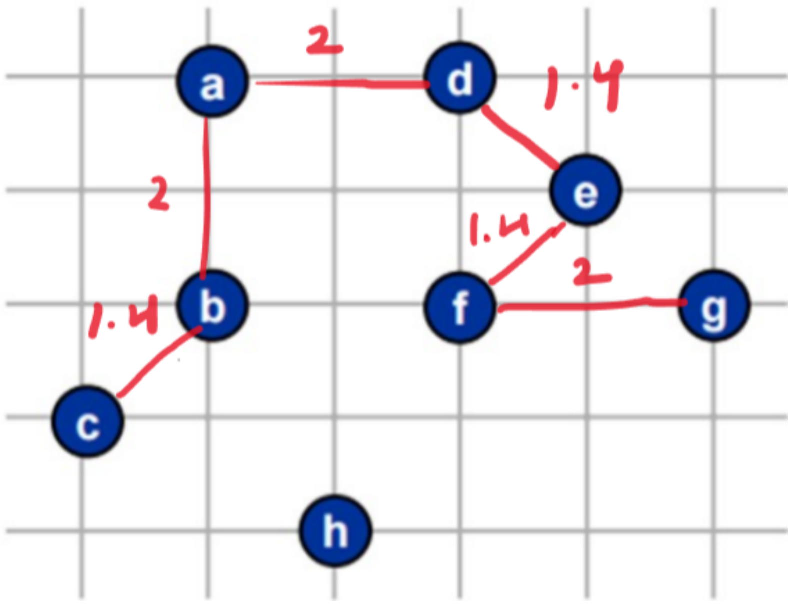
<u>**Step#6**</u>

We then check for any unvisited node that can be connected from A, B, C, D, E or F and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AG = \sqrt{2^2 + 4^2} = 4.5$$
$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BG = \sqrt{0^2 + 4^2} = 4$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$
$$CG = \sqrt{1^2 + 5^2} = 5.1$$
$$CH = \sqrt{1^2 + 2^2} = 2.2$$
$$DG = \sqrt{2^2 + 2^2} = 2.8$$
$$DH = \sqrt{4^2 + 1^2} = 4.1$$
$$EG = \sqrt{1^2 + 1^2} = 1.4$$
$$EH = \sqrt{3^2 + 2^2} = 3.6$$
$$FG = \sqrt{0^2 + 2^2} = 2$$
$$FH = \sqrt{2^2 + 1^2} = 2.2$$

Since FG is the smallest distance out of all the distances measured, we select G to visit and add it to V:

$$V = [A, B, C, E, F, G]$$



Step #7

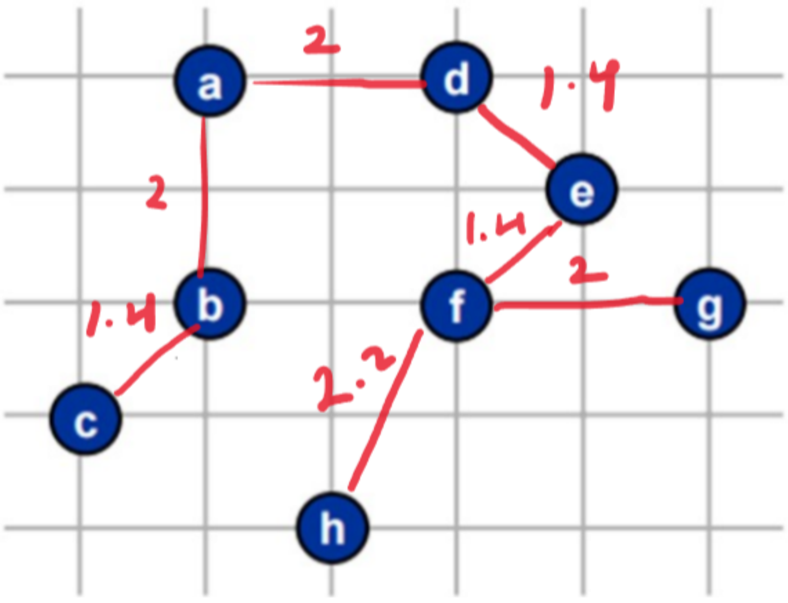
We then check for any unvisited node that can be connected from A, B, C, D, E, F or G and calculate their Euclidean distances. For the previous calculations we recall them from memory:

$$AH = \sqrt{4^2 + 1^2} = 4.1$$
$$BH = \sqrt{2^2 + 1^2} = 2.2$$
$$CH = \sqrt{1^2 + 2^2} = 2.2$$
$$DH = \sqrt{4^2 + 1^2} = 4.1$$
$$EH = \sqrt{3^2 + 2^2} = 3.6$$
$$FH = \sqrt{2^2 + 1^2} = 2.2$$
$$GH = \sqrt{2^2 + 4^2} = 4.5$$

We find that BH, FH or CH is the smallest. We arbitrarily pick FH. Hence we visit the final node H and add it to the visited list V:

$$V = [A, B, C, D, E, H]$$

The final graph looks like this, and is a minimal spanning tree generated from source node A :



(c) Because many pairs of points have identical distances (e.g. dist(h, c) = dist(h, b) = dist(h, f) = √ 5), the above diagram has more than one minimum-weight spanning tree. Determine the total number of minimum-weight spanning trees that exist in the above diagram. Clearly justify your answer.

Solution 2(c):

Since we can compute the distances of all the points in the graph from each other, this setting of points can be represented as a complete graph.

Since if there are no variations in the spanning tree, the same nodes will be picked each time. However, if we have an option to pick one of many nodes at a given step, the number of possible trees increases. The number of total spanning trees should equate to the number of combinations possible by picking identical distances. However, the total weight of all the edges in a minimum spanning tree must remain the same, which is 11.9 in this case.

To tackle this we can inspect the various times we needed to select one edge from multiple edges in solution 2(b):

In **Step #7**, we select 1 out of 3 nodes, BH, CH or FH : $\binom{3}{1} = 3$

In **Step #5**, we select 1 out of 2 nodes, EG and EG: $\binom{2}{1} = 2$

In **Step #3**, we select 1 out of 2 nodes, BF and AD: $\binom{2}{1} = 2$

In **Step #1**, we select 1 out of 2 nodes, AD and AB: $\binom{2}{1} = 2$

Total combinations : 3 + 2 + 2 + 2 = 9 combinations.

Hence 9 possible minimum spanning trees exist for this complete graph.

(d) Suppose the n points are situated so that each of the $\binom{n}{2} = n(n-1)/2$ distances are distinct positive numbers. Prove that graph G has only one minimum-weight spanning tree. Clearly explain each step in your proof.

Solution 2(d):

I will formulate this proof through contradiction.

Conjecture: Prove that graph G has only one minimum-weight spanning tree. Clearly explain each step in your proof.

Assumption: We assume that there exist 2 Minimal Spanning Trees that can be formed out of a given graph G with e $\binom{n}{2} = n(n-1)/2$ edges.

Through our assumption, we know that these two MSTs can not share the exact same set of edges. Hence two trees are different spanning trees even if one edge is different between them.

Hence, the two graphs will share the same $n-2$ edges (since each MST has $n-1$ edges in total). Let us assume that the weight of the last edge in one tree is w_1 and the other tree has the weight w_2. Hence, by assumption, $w_1 \neq w_2$.

Since all MSTs share have the same sum of edge weights, the total weight of both the spanning trees must be the same.

$$w_{(n-2)} + w_1 = w_{(n-2)} + w_2$$
$$w_1 = w_2$$

where $w_{(n-2)}$ is the weight of $n-2$ edges shared in the MSTs.

Hence, we arrive at a contradiction to our assumption since w_1 and w_2 must be equal.

Therefore, our assumption was false.

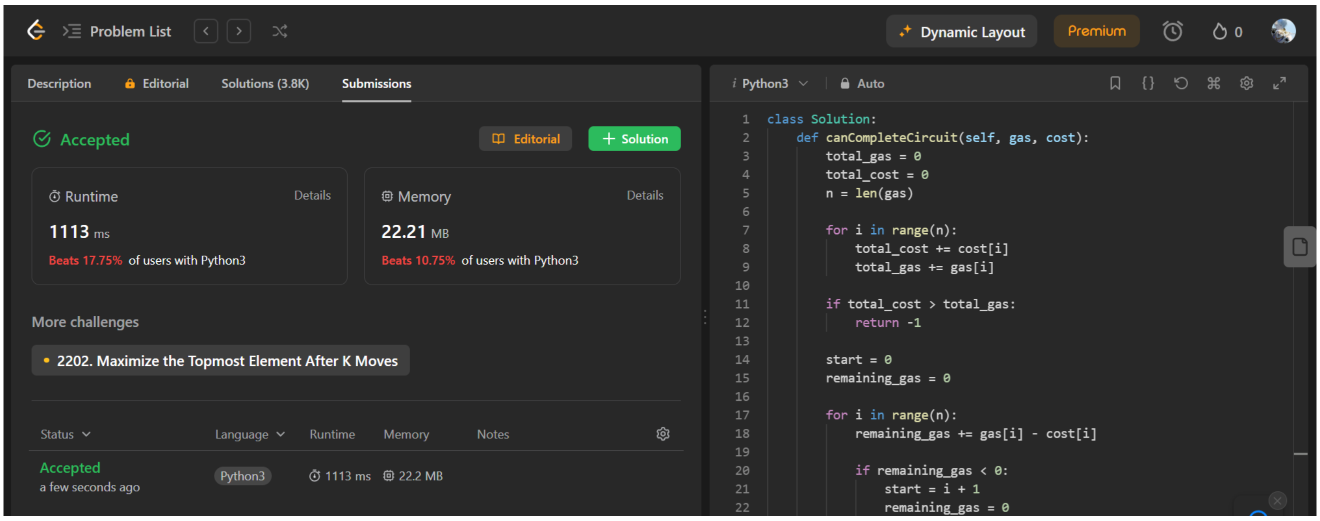
Hence, it proves that graph G has only one minimum-weight spanning tree, where the graph G has n points so that each of the $\binom{n}{2} = n(n-1)/2$ edges which are all distinct positive numbers.

Problem 3

Problem Number : 134

Problem Title: [134. Gas Station]

Difficulty Level: Medium



Accepted

Runtime
1113 ms
Beats 17.75% of users with Python3

Memory
22.21 MB
Beats 10.75% of users with Python3

More challenges

• 2202. Maximize the Topmost Element After K Moves

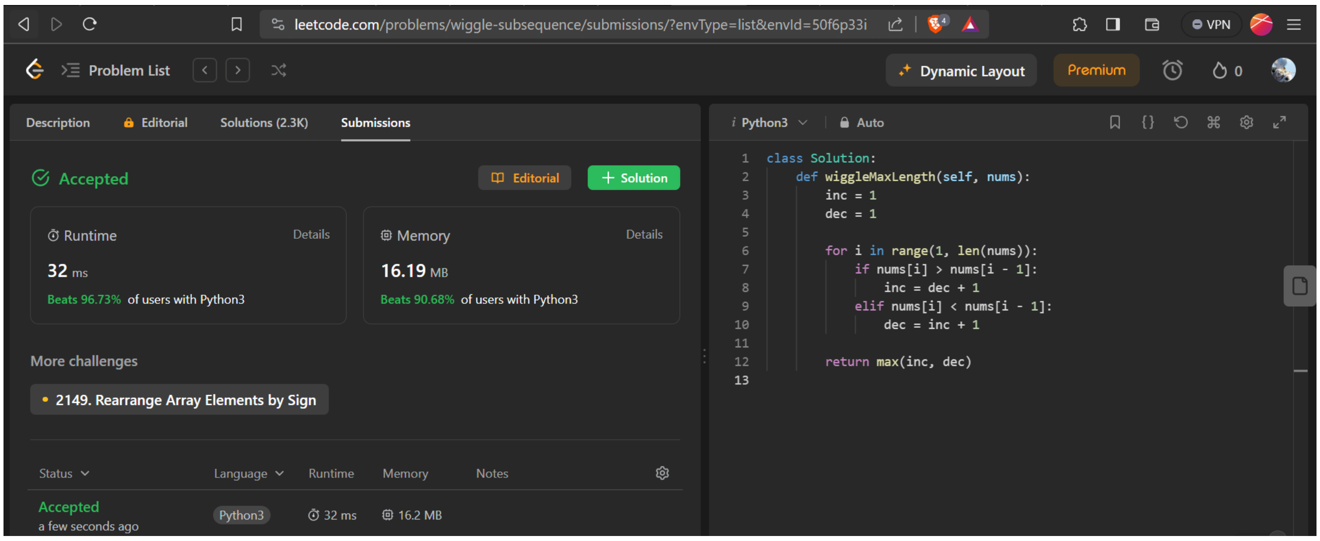
Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	Python3	1113 ms	22.2 MB	

```python
class Solution:
    def canCompleteCircuit(self, gas, cost):
        total_gas = 0
        total_cost = 0
        n = len(gas)

        for i in range(n):
            total_cost += cost[i]
            total_gas += gas[i]

        if total_cost > total_gas:
            return -1

        start = 0
        remaining_gas = 0

        for i in range(n):
            remaining_gas += gas[i] - cost[i]

            if remaining_gas < 0:
                start = i + 1
                remaining_gas = 0
```

Problem Number : 376

Problem Title: [376. Wiggle Subsequence]

Difficulty Level: Medium



Accepted

Runtime
32 ms
Beats 96.73% of users with Python3

Memory
16.19 MB
Beats 90.68% of users with Python3

More challenges

• 2149. Rearrange Array Elements by Sign

Status	Language	Runtime	Memory	Notes
Accepted a few seconds ago	Python3	32 ms	16.2 MB	

```python
class Solution:
    def wiggleMaxLength(self, nums):
        inc = 1
        dec = 1

        for i in range(1, len(nums)):
            if nums[i] > nums[i - 1]:
                inc = dec + 1
            elif nums[i] < nums[i - 1]:
                dec = inc + 1

        return max(inc, dec)
```

3(b) For one of the problems you are including in your mini-portfolio, explain the various ways you tried to solve this problem, telling us what worked and what did not work. Describe what insights you had as you eventually found a correct solution. Reflect on what you learned from struggling on this problem, and describe how the struggle itself was valuable for you.

Solution 3(b):

In my mini portfolio, I solved the "Gas Station #134" problem, where you are given two lists: ***gas*** representing the amount of gas at each gas station, and ***cost*** representing the amount of gas required to travel from one station to the next. The task is to find the starting gas station from which you can travel the entire circuit without running out of gas.

I first tried to optimize the gas cost directly without a greedy approach. I realized that if the total cost of the journey was greater than the total gas available, it would not be possible to complete the circuit. So, I added a check to calculate the total gas and total cost upfront and return -1 if the total cost exceeded the total gas. This was a good optimization and significantly reduced the time complexity but did not pass all the test cases.

The key insight for solving the problem efficiently came from using a greedy approach. I maintained a ***remaining_gas*** variable and iterated through the gas stations while tracking the remaining gas after each station. If at any point, the remaining gas became negative, I updated the ***start*** variable to the next station and reset the ***remaining_gas*** to 0. This approach efficiently identified the starting station, if it existed, in a single pass.