ITCS 6114 – Algorithms and Data Structure

Dewan T. Ahmed, Ph.D.

Project 2

Graph Algorithms: Singles-source shortest path algorithm and Minimum Spanning Tree

Project Report submitted by:-

Nisha Ramrakhyani (801208678)

Zalak Panchal (801196881)

University of North Carolina at Charlotte

Problem 1: Single-source Shortest Path Algorithm

Find the shortest path tree in both directed and undirected weighted graphs for a given source vertex. Assume there is no negative edge in your graph. You will print each path and path cost for a given source.

Dijkstra Algorithm:

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V,E) for the case in which all edge weights are nonnegative. Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.

The algorithm repeatedly selects the vertex u ∈ V _S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.

The algorithm proceeds as follows:

1. While Q is not empty, pop the node v, that is not already in S, from Q with the smallest dist (v). In the first run, source node s will be chosen because dist(s) was initialized to 0. In the next run, the next node with the smallest dist value is chosen.

2. Add node v to S, to indicate that v has been visited.

3. Update dist values of adjacent nodes of the current node v as follows: for each new adjacent node u,If dist (v) + weight(u,v) < dist (u), there is a new minimal distance found for u, so update dist (u) to the new minimal distance value.

The algorithm has visited all nodes in the graph and found the smallest distance to each node. Dist now contains the shortest path tree from source s.

<u>1) Pseudocode and detailed analysis of algorithm:</u>

Pseudocode:

DIJKSTRA (G, w, s)

1 INITIALIZE-SINGLE-SOURCE (G, s)

```
2 S = ∅
3 Q =G.V
4 while Q ≠ ∅
5     u = EXTRACT-MIN(Q)
6     S = S ∪ {u}
7     for each vertex v ∈ G.Adj[u]
8         RELAX (u,v ,w)
```

Detail analysis of algorithm:

- Initialize-single-source would take O(n) time. A graph has n vertices, thus the ‘for’ loop in the initialization step would run n times i.e. ‘for’ loop would cost O(n).
- As heap has n items, Q= G.V, constructing a heap would take O(n logn) time.
- In the ‘while’ loop, Extract-MIN(Q) will be called one time for each node as each node gets removed from the Queue. This runs till the Queue becomes empty As the Q has n items, ‘while’ loop would be called n times and cost for each call would be O(log n). Therefore, the total cost would be O(n logn)
- The key-value would be changed based upon the total number of edges in the graph i.e. m. Whenever we change the key-value, the order of the nodes in the heap might change too. So, we’ll need to modify the heap which costs O(log n) time for each call. Therefore, the total cost would be O(m logn) Relax operation takes O(logn) time and for loop containing relax runs the number of edges time,means m times,so the total runtime for relax operation will be O(m logn)
- So, the total runtime time of this algorithm would be O(n logn + m logn) i.e. O((n+m)logn) which is O(m logn)

2) <u>Data Structures used in the Pseudocode and how it affects runtime of algorithm:</u>

Data Structures used:

- Heap
- Priority Queue

How do these data structures affect the runtime of the algorithm?

- Performance depends on implementation of priority queue:
    - binary heap implementation: O(m log n)
    - Fibonacci heap implementation: O(log n + m)
- Option 1: implement priority queue with binary heap
    - Cost of initializing Q and first for loop: O(n log n)
    - Cost of decreasing key of r: O(log n)
    - While loop:
        - V calls to front (=extract min) → O(n log n)
        - E (or fewer) calls to decrease key → O(m log n)
        - Total for loop: O((n + m) log n)
    -

      Overall Total: O((n + m) log n) = O(m log n),
        - Since m ≥ n+1, since G is connected
- Option 2: implement priority queue with Fibonacci heap
    - Decrease key can be done in O(1) (amortized) time
        - Amortized means the cost is spread out
        - Average time is O(1).
        - Other times may sometimes be larger.
    - E Calls to decrease key take O(m) * O(1) = O(m) time
    - Total: O(n log n + m)

3)Implementation of shortest path finding algorithm in directed/undirected graphs

Implementation code of Dijkstra’s algorithm:

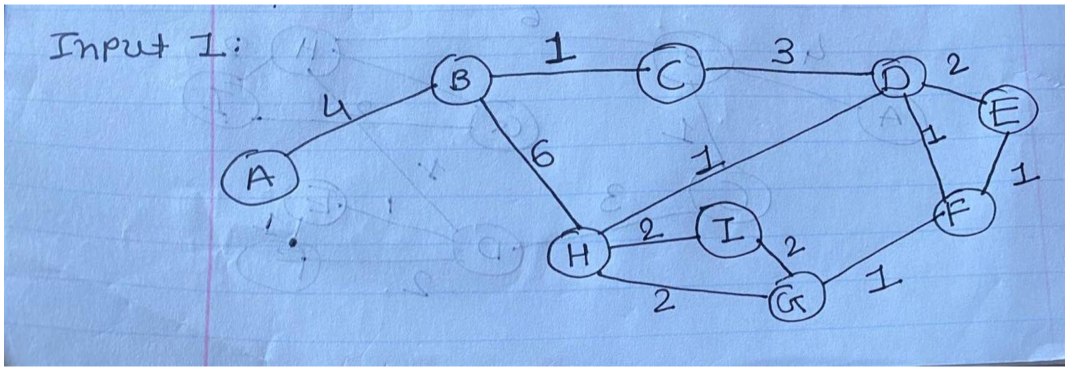
```
import heapq
import collections

def shortestpath(edges,startnode,destnode,graph_check):
    graph = collections.defaultdict(set)

    if graph_check == 'D':
        for l,r,c in edges:
            graph[l].add((c,r))
    else:
        for l,r,c in edges:
            graph[l].add((c,r))
            graph[r].add((c,l))



    queue, visited = [(0, startnode, [])], set()

    heapq.heapify(queue)

    while queue:
        (cost, node, path) = heapq.heappop(queue)
        if node not in visited:
            visited.add(node)
            path = path + [node]
            if node == destnode:
                return (cost, path)
            for c, neighbour in graph[node]:

                if neighbour not in visited:
                    heapq.heappush(queue, (cost+int(c), neighbour, path))

    return float("inf")
```

```
def main():
    edges = []
    nodes=[]
    file=open("Dijkstra_Directed_Graph/Input3.txt", "r+")
    lines=file.readlines()
    for line in lines:
        edges.append(line.split())

    first = list(edges[0])
    last= list(edges[-1])
    vertices = first[0]
    numofedges=first[1]
    g_check= first[2]
    edges.pop(0)


    if(len(last)==1):
        startnode=last[0]
        edges.pop()
    else:
        firstnode=list(edges[0])
        startnode=firstnode[0]

    print('\nSource vertex: '+str(startnode)+'\n')
    print('Nodes: '+str(vertices)+'\n')
    print('Edges: '+str(numofedges)+'\n')

    if g_check == 'U':
        print('Undirected Graph')
    else:
        print('Directed Graph')


    for i in edges:
        nodes.append(i[0])
    nodes.extend(y[1] for y in edges)
    destnode=list(set(nodes))
    for i in destnode:
        print ("\n"+str(startnode)+" -> "+str(i)+":")
        result = shortestpath(edges, startnode, i,g_check)
        print('Path Cost: '+str(result[0]))
        print('Path: '+str(result[1]))

if __name__== "__main__":
    main()
```

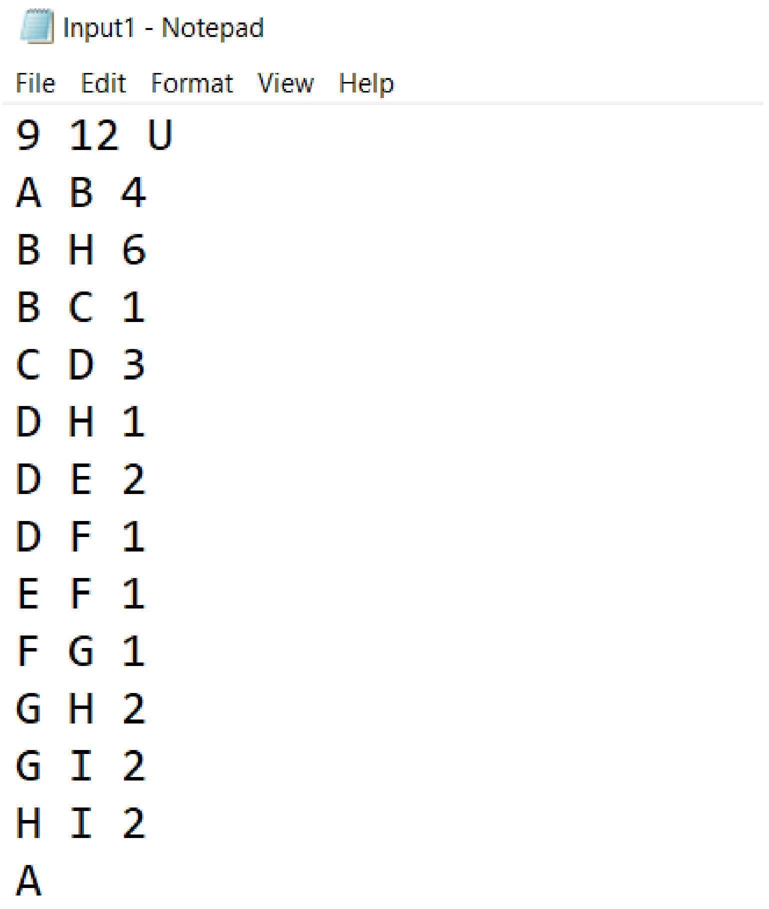
4)Test on four different graphs for both directed and undirected graphs and print path and path cost for each node from a given source

Test Cases of Undirected graph:

Test case 1:

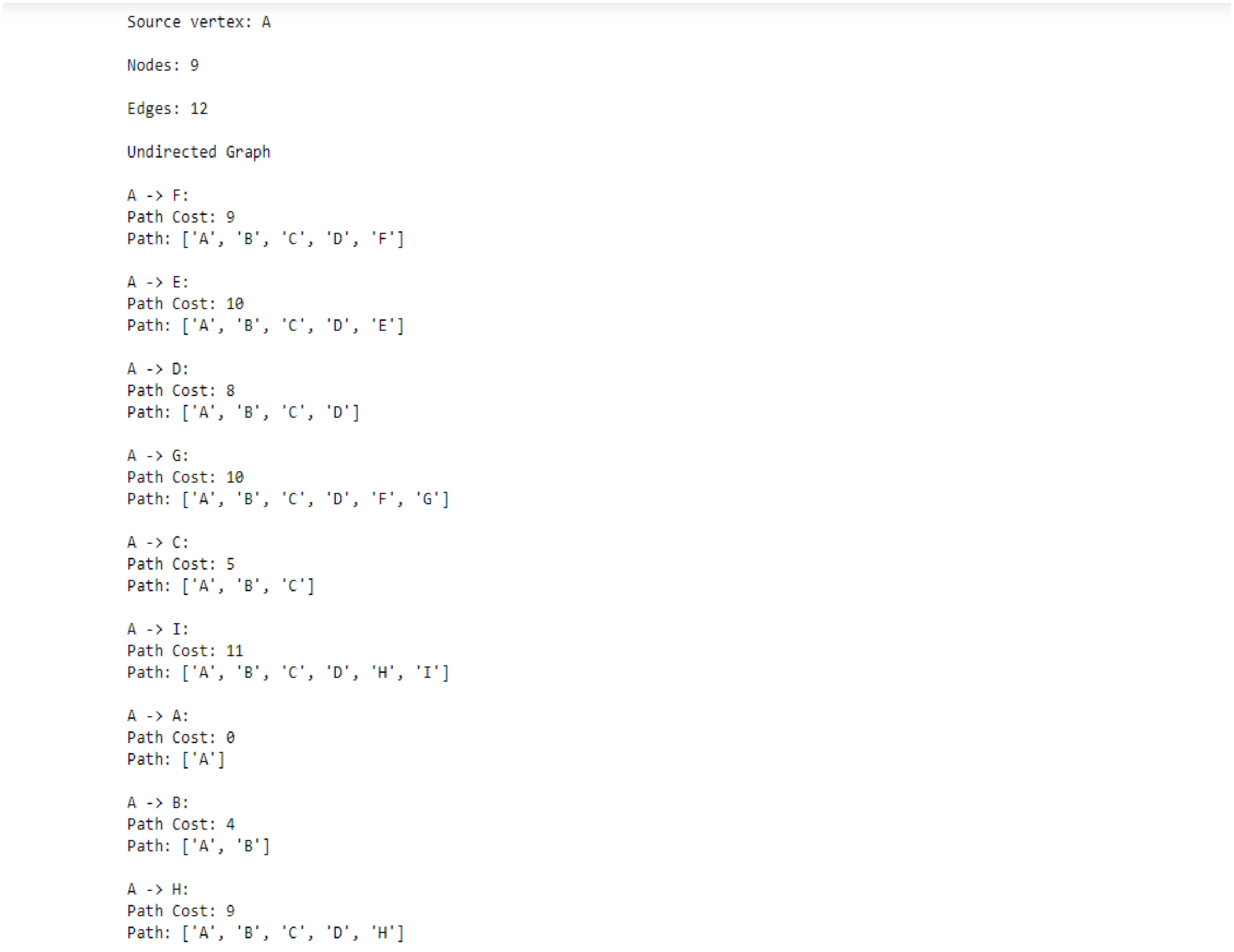


Input



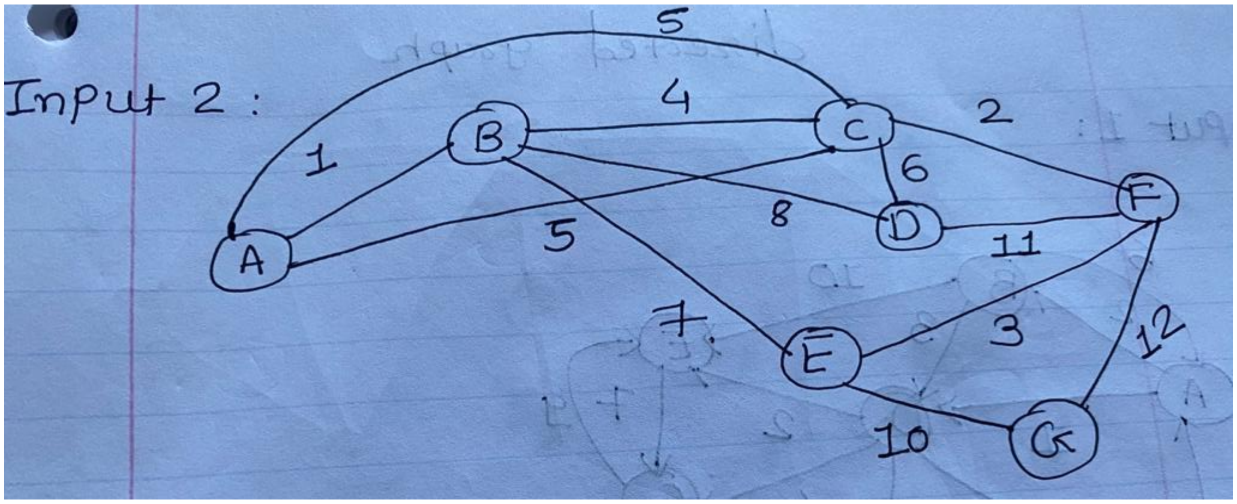
```
9 12 U
A B 4
B H 6
B C 1
C D 3
D H 1
D E 2
D F 1
E F 1
F G 1
G H 2
G I 2
H I 2
A
```

Output:

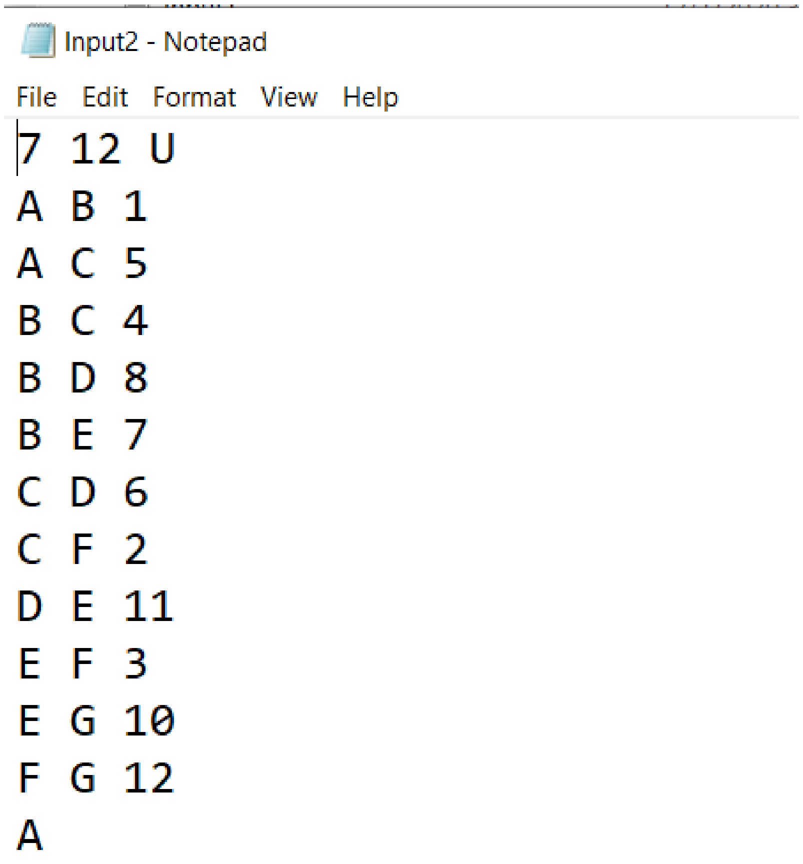


```
Source vertex: A

Nodes: 9

Edges: 12

Undirected Graph

A -> F:
Path Cost: 9
Path: ['A', 'B', 'C', 'D', 'F']

A -> E:
Path Cost: 10
Path: ['A', 'B', 'C', 'D', 'E']

A -> D:
Path Cost: 8
Path: ['A', 'B', 'C', 'D']

A -> G:
Path Cost: 10
Path: ['A', 'B', 'C', 'D', 'F', 'G']

A -> C:
Path Cost: 5
Path: ['A', 'B', 'C']

A -> I:
Path Cost: 11
Path: ['A', 'B', 'C', 'D', 'H', 'I']

A -> A:
Path Cost: 0
Path: ['A']

A -> B:
Path Cost: 4
Path: ['A', 'B']

A -> H:
Path Cost: 9
Path: ['A', 'B', 'C', 'D', 'H']
```

Test case 2



Input

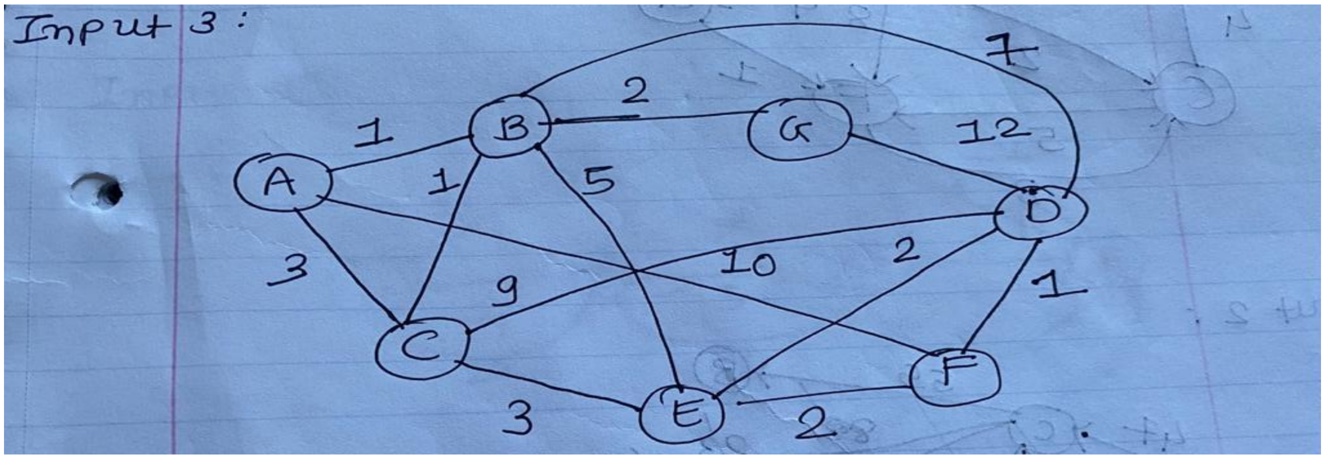


```
7 12 U
A B 1
A C 5
B C 4
B D 8
B E 7
C D 6
C F 2
D E 11
E F 3
E G 10
F G 12
A
```

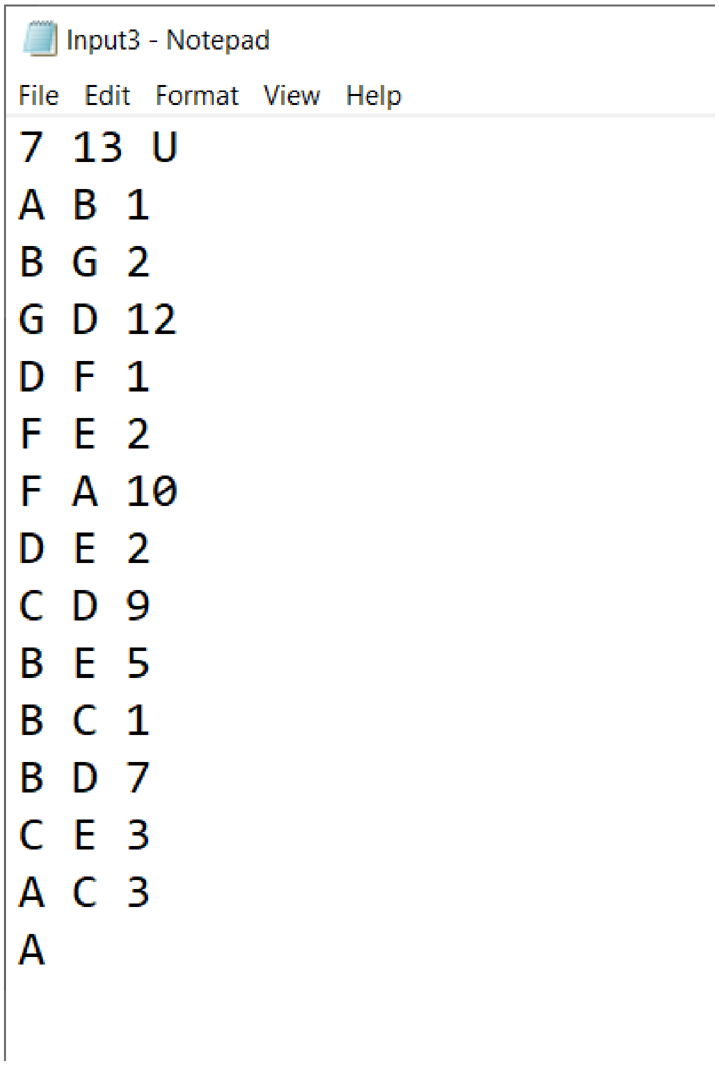
Output:

```
Source vertex: A

Nodes: 7

Edges: 12

Undirected Graph

A -> F:
Path Cost: 7
Path: ['A', 'C', 'F']

A -> E:
Path Cost: 8
Path: ['A', 'B', 'E']

A -> D:
Path Cost: 9
Path: ['A', 'B', 'D']

A -> G:
Path Cost: 18
Path: ['A', 'B', 'E', 'G']

A -> C:
Path Cost: 5
Path: ['A', 'C']

A -> A:
Path Cost: 0
Path: ['A']

A -> B:
Path Cost: 1
Path: ['A', 'B']
```

Test case 3



Input

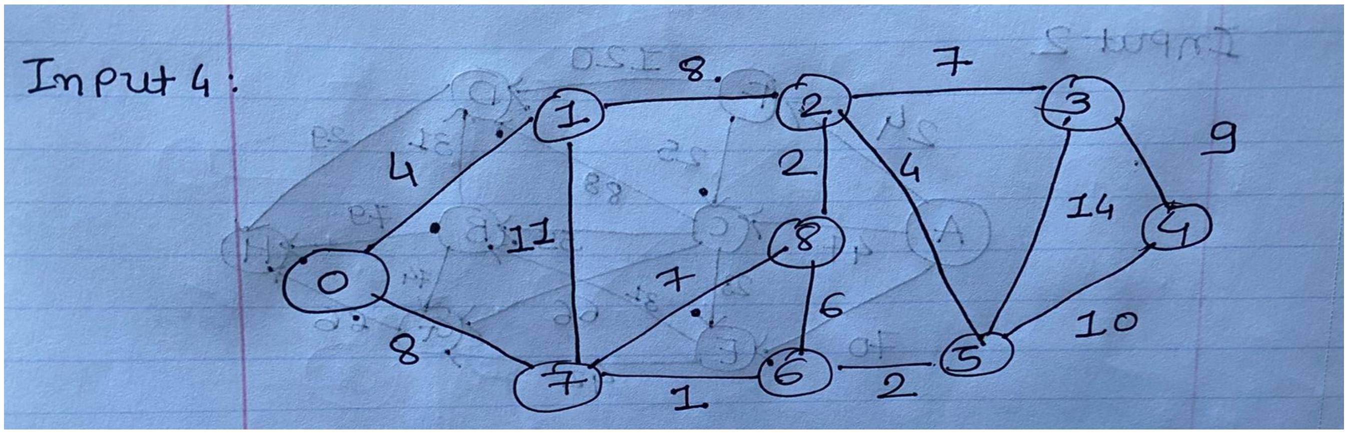


```
7 13 U
A B 1
B G 2
G D 12
D F 1
F E 2
F A 10
D E 2
C D 9
B E 5
B C 1
B D 7
C E 3
A C 3
A
```

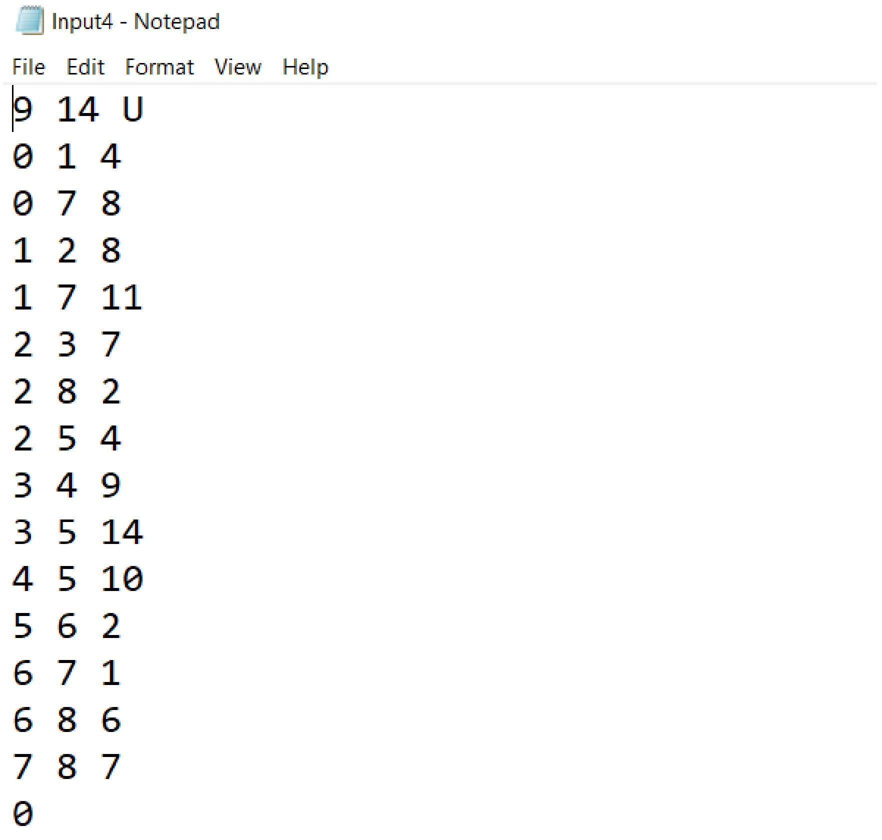
Output:

```
Source vertex: A

Nodes: 7

Edges: 13

Undirected Graph

A -> F:
Path Cost: 7
Path: ['A', 'B', 'C', 'E', 'F']

A -> E:
Path Cost: 5
Path: ['A', 'B', 'C', 'E']

A -> D:
Path Cost: 7
Path: ['A', 'B', 'C', 'E', 'D']

A -> G:
Path Cost: 3
Path: ['A', 'B', 'G']

A -> C:
Path Cost: 2
Path: ['A', 'B', 'C']

A -> A:
Path Cost: 0
Path: ['A']

A -> B:
Path Cost: 1
Path: ['A', 'B']
```

Test case 4



Input



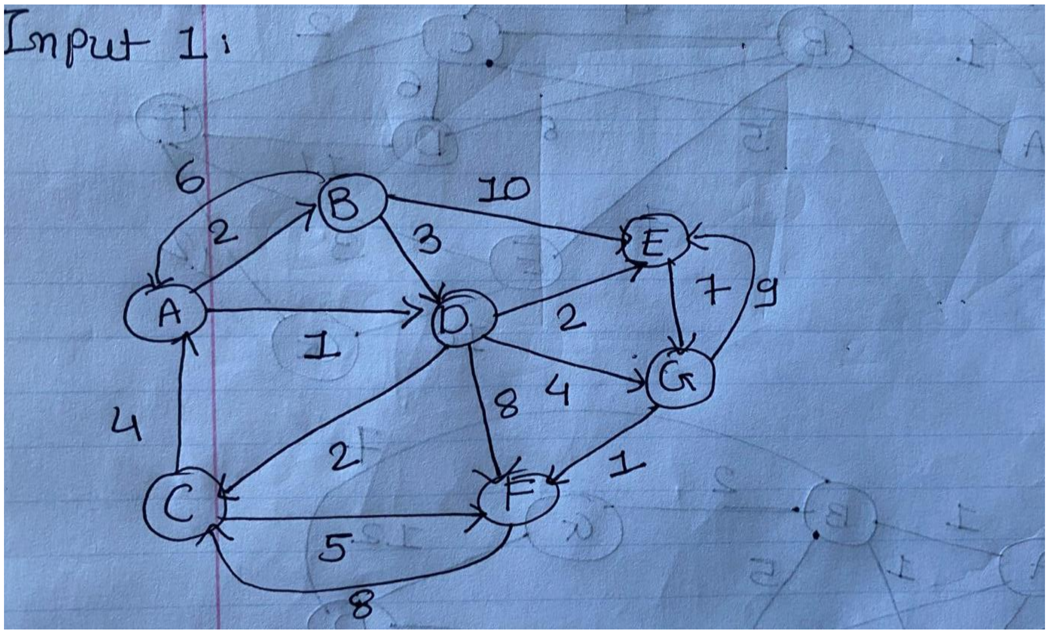
```
9 14 U
0 1 4
0 7 8
1 2 8
1 7 11
2 3 7
2 8 2
2 5 4
3 4 9
3 5 14
4 5 10
5 6 2
6 7 1
6 8 6
7 8 7
0
```

Output:

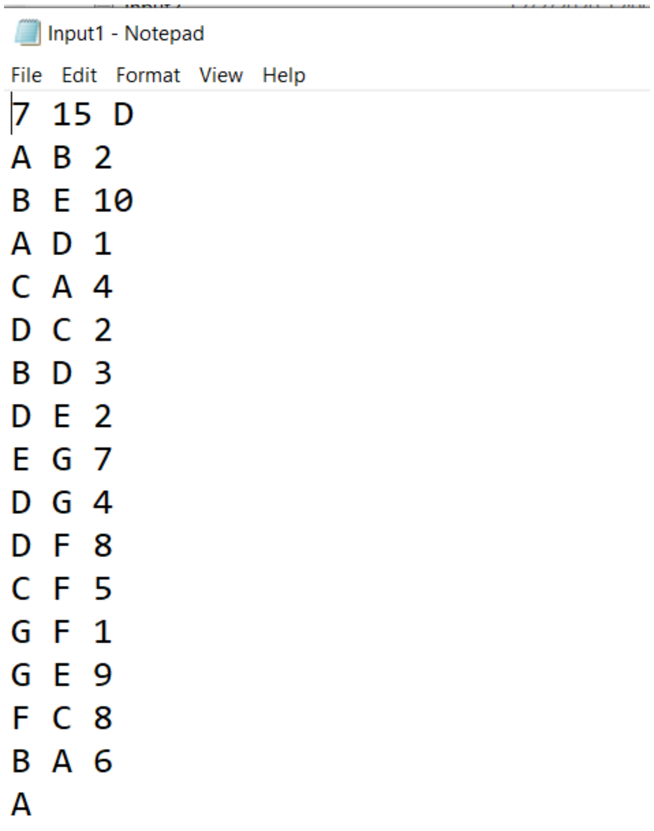
```
Source vertex: 0

Nodes: 9

Edges: 14

Undirected Graph

0 -> 3:
Path Cost: 19
Path: ['0', '1', '2', '3']

0 -> 8:
Path Cost: 14
Path: ['0', '1', '2', '8']

0 -> 5:
Path Cost: 11
Path: ['0', '7', '6', '5']

0 -> 1:
Path Cost: 4
Path: ['0', '1']

0 -> 7:
Path Cost: 8
Path: ['0', '7']

0 -> 2:
Path Cost: 12
Path: ['0', '1', '2']

0 -> 0:
Path Cost: 0
Path: ['0']

0 -> 4:
Path Cost: 21
Path: ['0', '7', '6', '5', '4']

0 -> 6:
Path Cost: 9
Path: ['0', '7', '6']
```

Test Cases of Directed graph:

Test case 1:

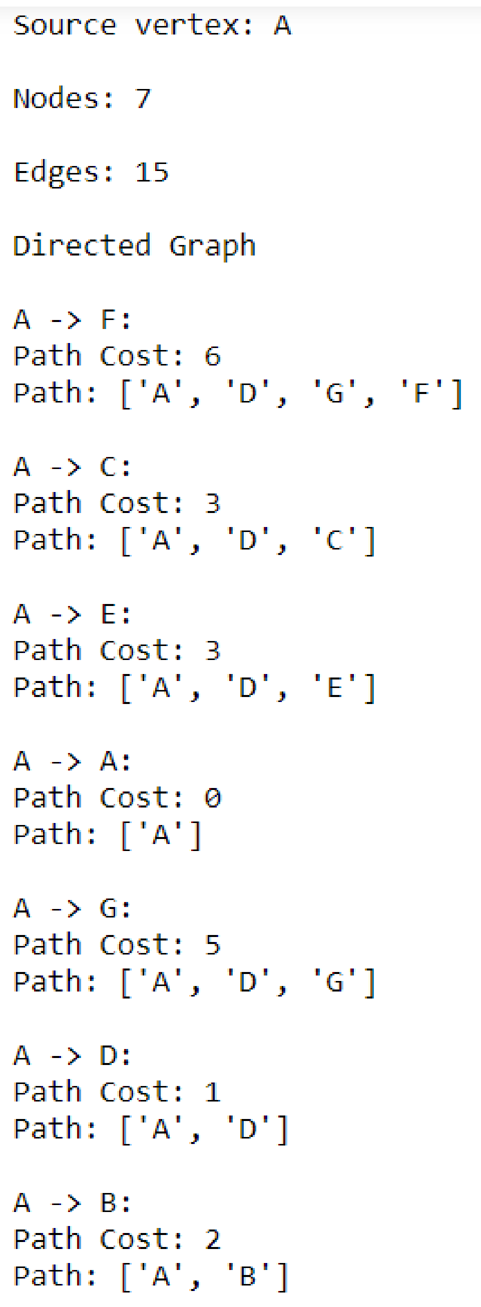


Input:



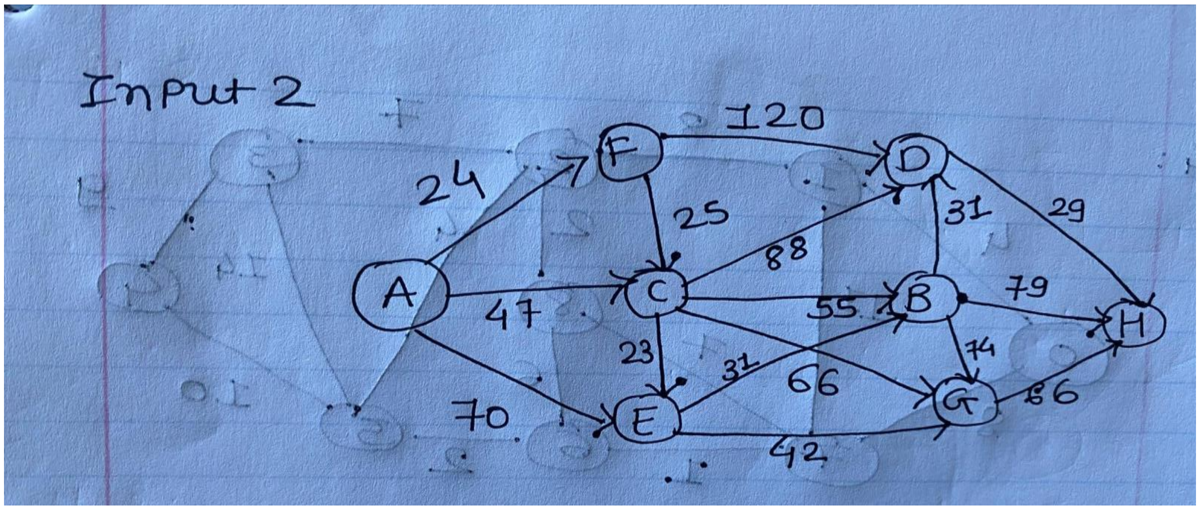
```
7 15 D
A B 2
B E 10
A D 1
C A 4
D C 2
B D 3
D E 2
E G 7
D G 4
D F 8
C F 5
G F 1
G E 9
F C 8
B A 6
A
```

Output:

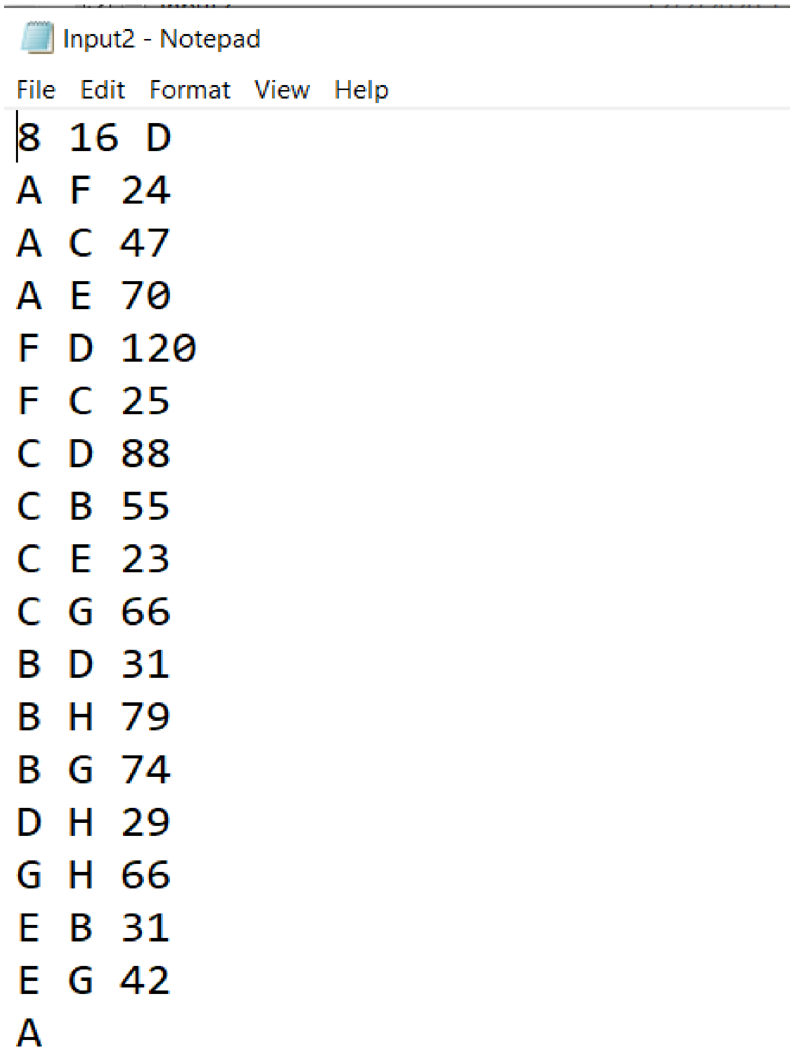


```
Source vertex: A

Nodes: 7

Edges: 15

Directed Graph

A -> F:
Path Cost: 6
Path: ['A', 'D', 'G', 'F']

A -> C:
Path Cost: 3
Path: ['A', 'D', 'C']

A -> E:
Path Cost: 3
Path: ['A', 'D', 'E']

A -> A:
Path Cost: 0
Path: ['A']

A -> G:
Path Cost: 5
Path: ['A', 'D', 'G']

A -> D:
Path Cost: 1
Path: ['A', 'D']

A -> B:
Path Cost: 2
Path: ['A', 'B']
```

Test case 2:

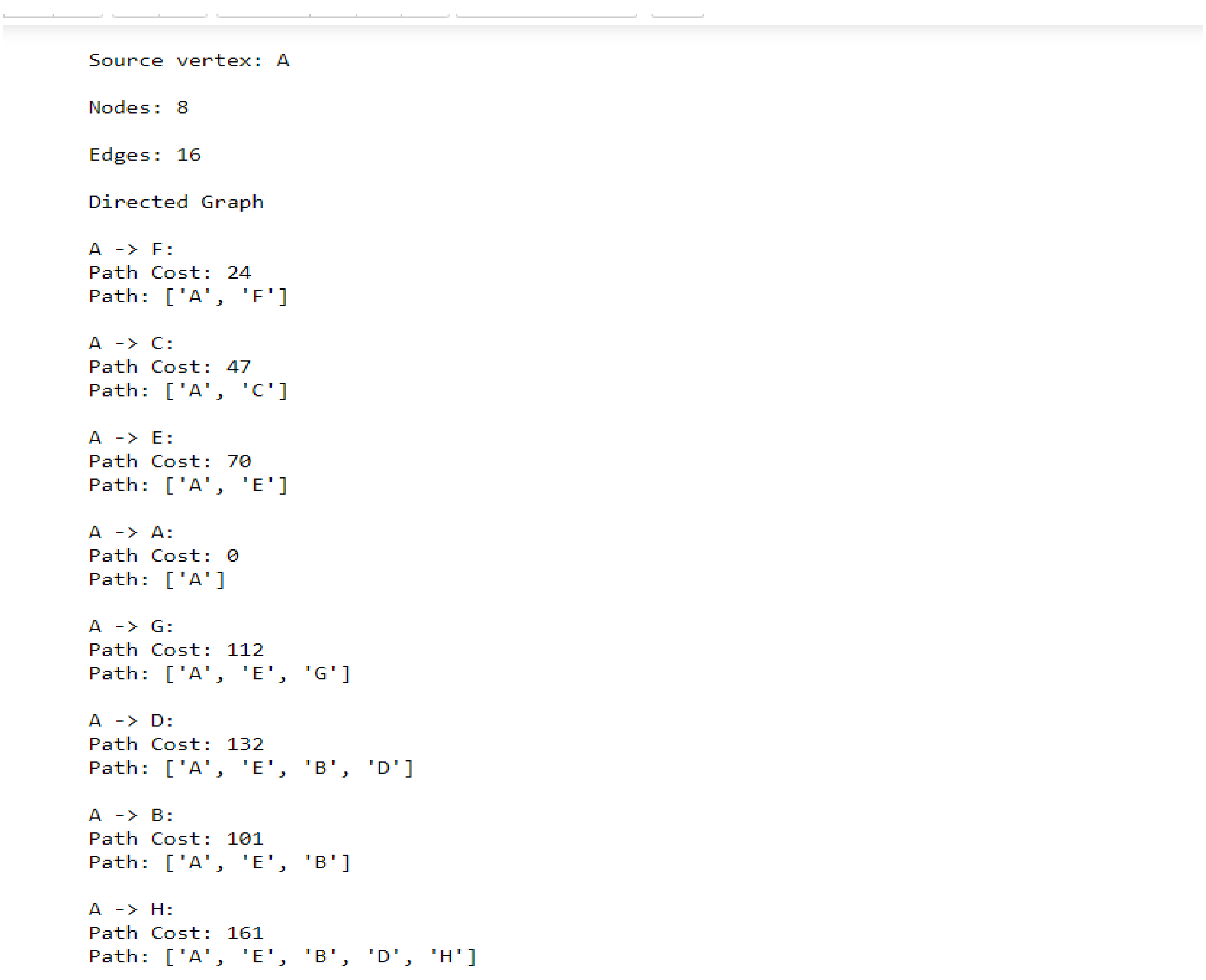


Input:



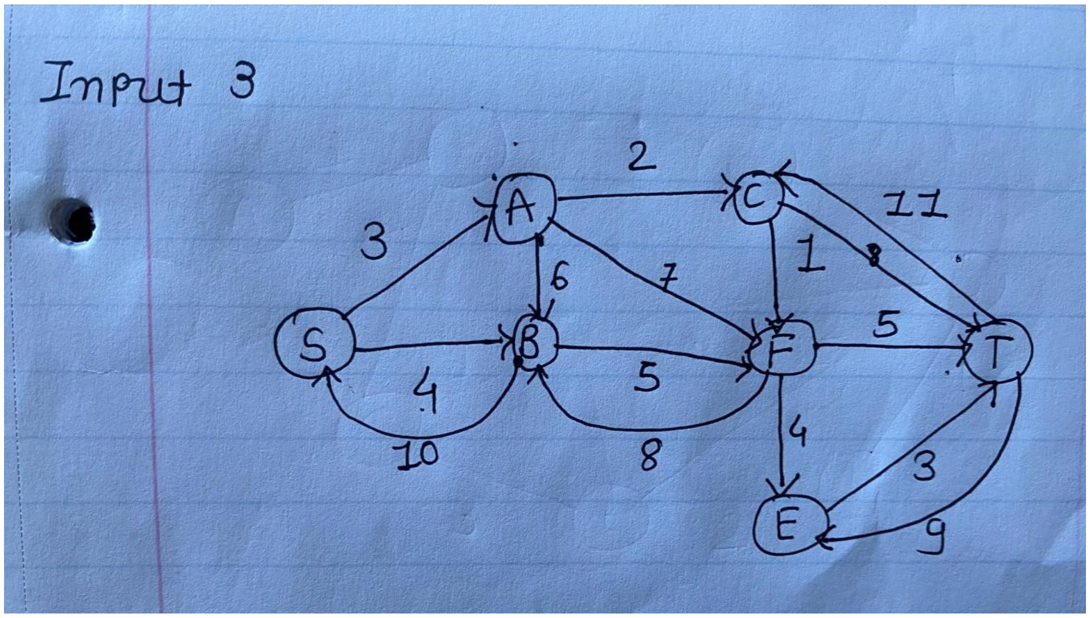
```
8 16 D
A F 24
A C 47
A E 70
F D 120
F C 25
C D 88
C B 55
C E 23
C G 66
B D 31
B H 79
B G 74
D H 29
G H 66
E B 31
E G 42
A
```

Output:

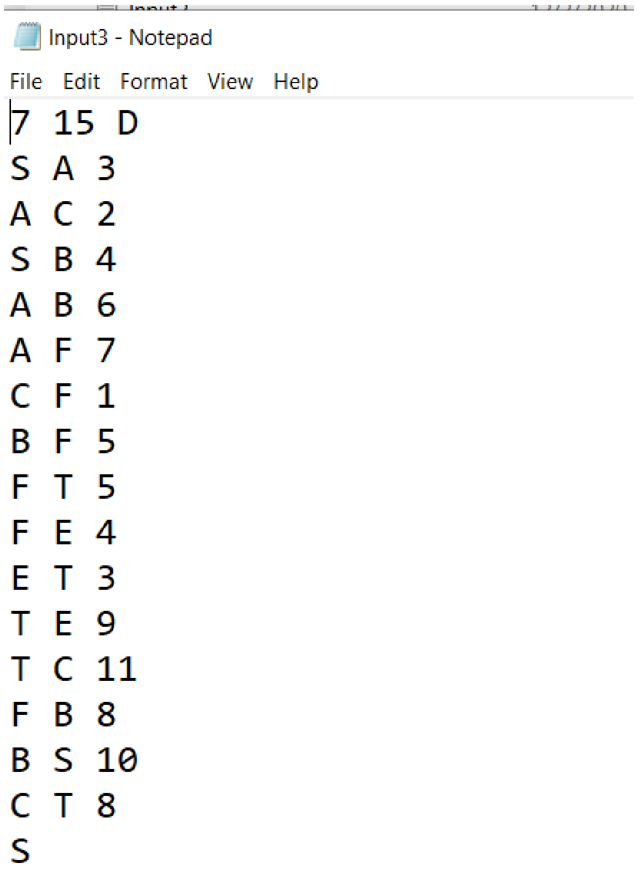


```
Source vertex: A

Nodes: 8

Edges: 16

Directed Graph

A -> F:
Path Cost: 24
Path: ['A', 'F']

A -> C:
Path Cost: 47
Path: ['A', 'C']

A -> E:
Path Cost: 70
Path: ['A', 'E']

A -> A:
Path Cost: 0
Path: ['A']

A -> G:
Path Cost: 112
Path: ['A', 'E', 'G']

A -> D:
Path Cost: 132
Path: ['A', 'E', 'B', 'D']

A -> B:
Path Cost: 101
Path: ['A', 'E', 'B']

A -> H:
Path Cost: 161
Path: ['A', 'E', 'B', 'D', 'H']
```

Test case 3:



Input:

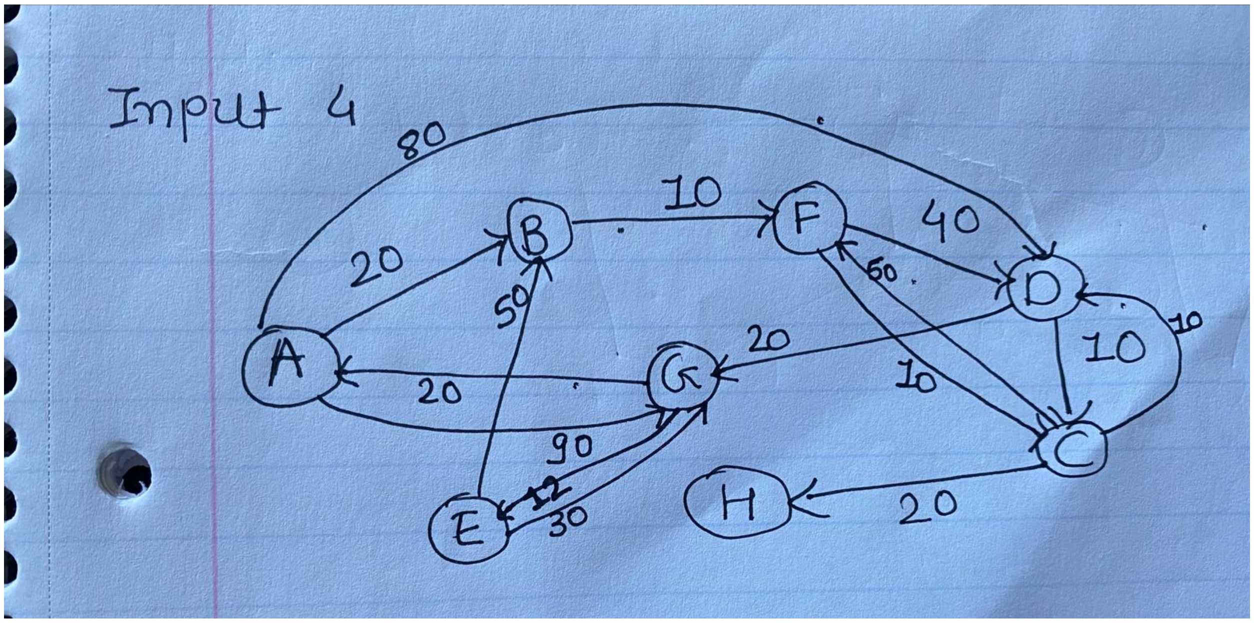


```
7 15 D
S A 3
A C 2
S B 4
A B 6
A F 7
C F 1
B F 5
F T 5
F E 4
E T 3
T E 9
T C 11
F B 8
B S 10
C T 8
S
```

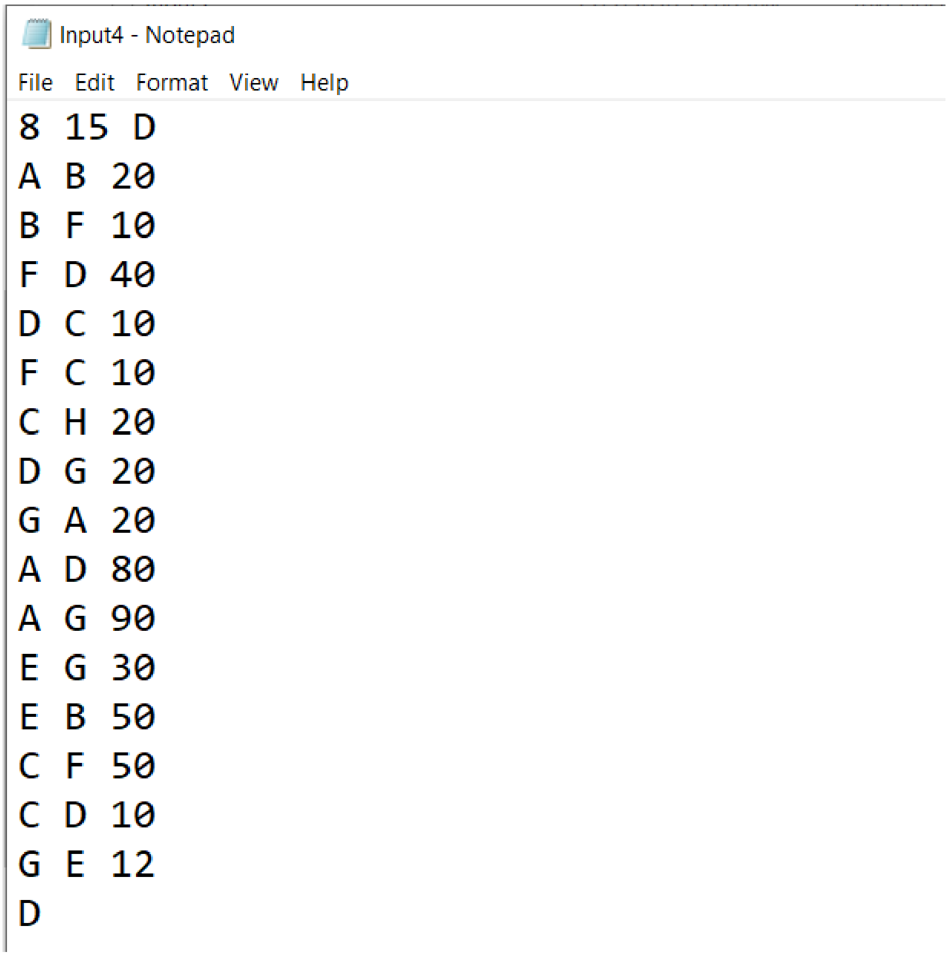
Output:

```
Source vertex: S

Nodes: 7

Edges: 15

Directed Graph

S -> F:
Path Cost: 6
Path: ['S', 'A', 'C', 'F']

S -> C:
Path Cost: 5
Path: ['S', 'A', 'C']

S -> S:
Path Cost: 0
Path: ['S']

S -> E:
Path Cost: 10
Path: ['S', 'A', 'C', 'F', 'E']

S -> A:
Path Cost: 3
Path: ['S', 'A']

S -> T:
Path Cost: 11
Path: ['S', 'A', 'C', 'F', 'T']

S -> B:
Path Cost: 4
Path: ['S', 'B']
```

Test case 4:

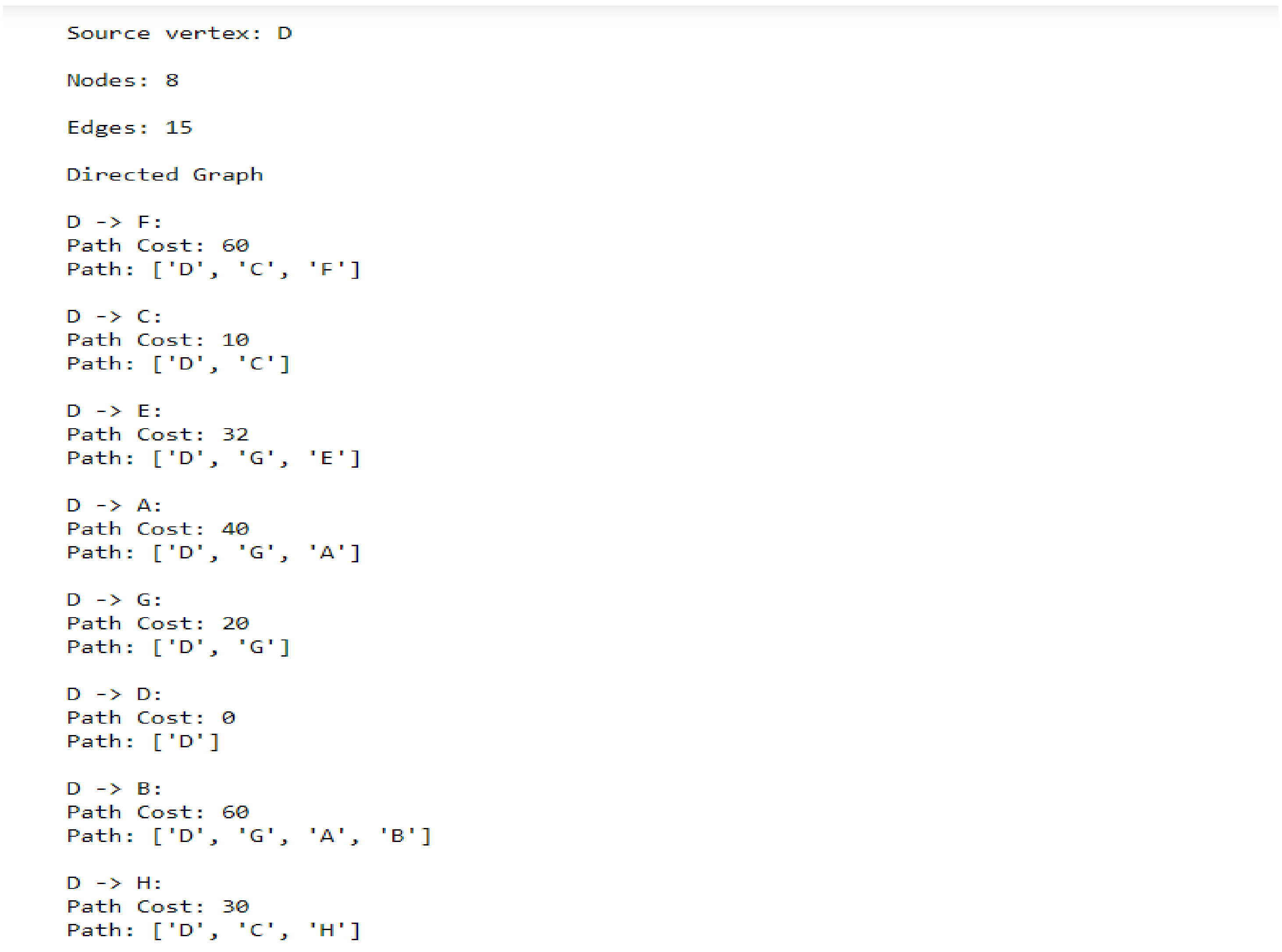


Input:



```
8 15 D
A B 20
B F 10
F D 40
D C 10
F C 10
C H 20
D G 20
G A 20
A D 80
A G 90
E G 30
E B 50
C F 50
C D 10
G E 12
D
```

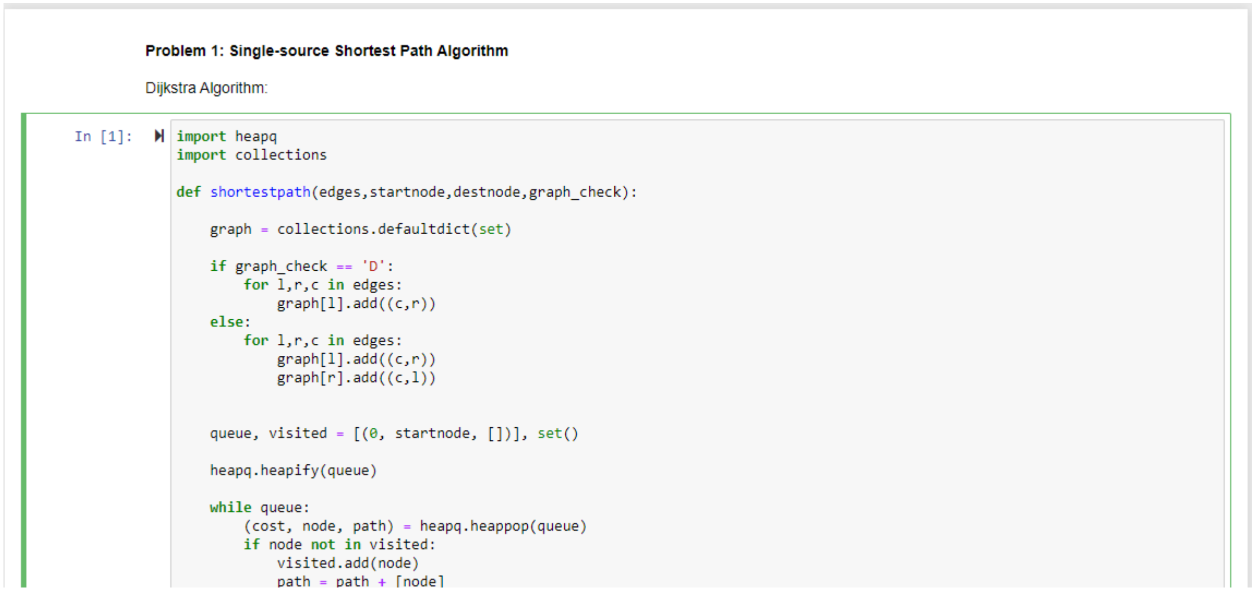
Output:



```
Source vertex: D

Nodes: 8

Edges: 15

Directed Graph

D -> F:
Path Cost: 60
Path: ['D', 'C', 'F']

D -> C:
Path Cost: 10
Path: ['D', 'C']

D -> E:
Path Cost: 32
Path: ['D', 'G', 'E']

D -> A:
Path Cost: 40
Path: ['D', 'G', 'A']

D -> G:
Path Cost: 20
Path: ['D', 'G']

D -> D:
Path Cost: 0
Path: ['D']

D -> B:
Path Cost: 60
Path: ['D', 'G', 'A', 'B']

D -> H:
Path Cost: 30
Path: ['D', 'C', 'H']
```

INSTRUCTIONS TO RUN THE SINGLE SOURCE PATH ALGORITHM IMPLEMENTATION CODE

1. **Open the python notebook “Project2_SSSP_MST.ipynb ”.**
2. **Run the problem 1.**

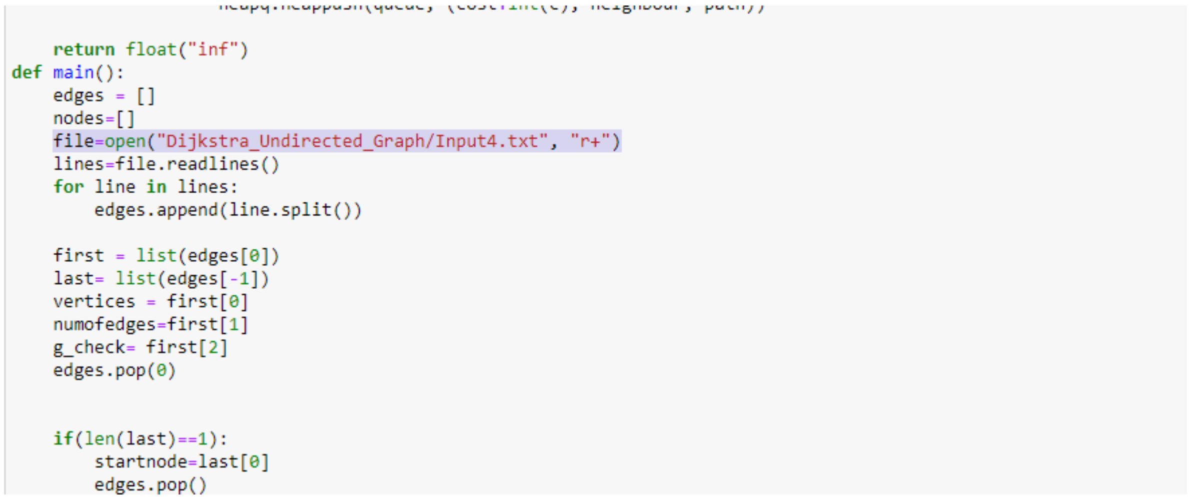


Problem 1: Single-source Shortest Path Algorithm

Dijkstra Algorithm:

```
In [1]:  import heapq
         import collections

         def shortestpath(edges,startnode,destnode,graph_check):

             graph = collections.defaultdict(set)

             if graph_check == 'D':
                 for l,r,c in edges:
                     graph[l].add((c,r))
             else:
                 for l,r,c in edges:
                     graph[l].add((c,r))
                     graph[r].add((c,l))


             queue, visited = [(0, startnode, [])], set()

             heapq.heapify(queue)

             while queue:
                 (cost, node, path) = heapq.heappop(queue)
                 if node not in visited:
                     visited.add(node)
                     path = path + [node]
```

3. **To change the Input, Change file path in the main function:**



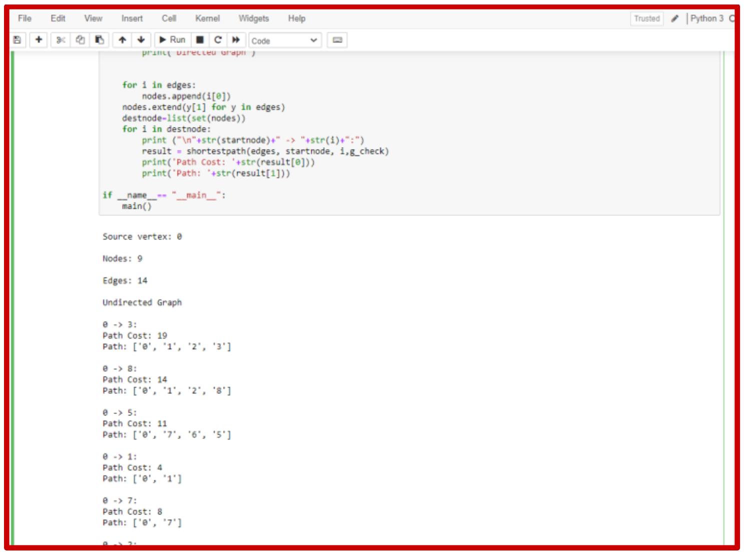
```
    return float("inf")
def main():
    edges = []
    nodes=[]
    file=open("Dijkstra_Undirected_Graph/Input4.txt", "r+")
    lines=file.readlines()
    for line in lines:
        edges.append(line.split())

    first = list(edges[0])
    last= list(edges[-1])
    vertices = first[0]
    numofedges=first[1]
    g_check= first[2]
    edges.pop(0)


    if(len(last)==1):
        startnode=last[0]
        edges.pop()
```

----------->file=open("Dijkstra_Undirected_Graph/Input1.txt", "r+")

For Undirected Graph: “Dijkstra_Undirected_Graph/Input1.txt”
For Directed Graph: “Dijkstra_Directed_Graph/Input1.txt”

(Change “Input1.txt” to Input2.txt,Input3.txt and Input4.txt with appropriate folder name,to check four different test cases,for both the graphs).

4. **You will see output right after the code block.**

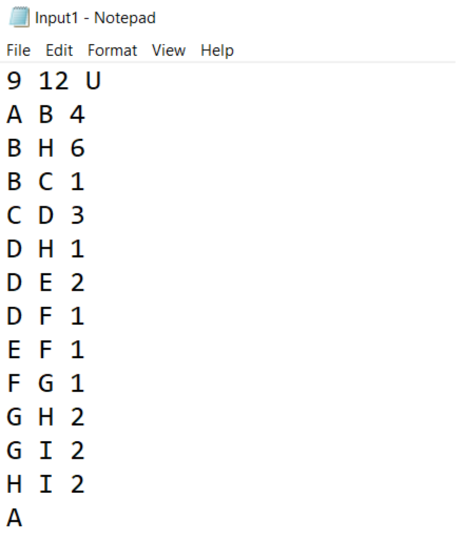


```
    for i in edges:
        nodes.append(i[0])
    nodes.extend(y[1] for y in edges)
    destnode=list(set(nodes))
    for i in destnode:
        print ("\n"+str(startnode)+" -> "+str(i)+":")
        result = shortestpath(edges, startnode, i,g_check)
        print('Path Cost: '+str(result[0]))
        print('Path: '+str(result[1]))

if __name__== "__main__":
    main()
```

```
Source vertex: 0

Nodes: 9

Edges: 14

Undirected Graph

0 -> 3:
Path Cost: 19
Path: ['0', '1', '2', '3']

0 -> 8:
Path Cost: 14
Path: ['0', '1', '2', '8']

0 -> 5:
Path Cost: 11
Path: ['0', '7', '6', '5']

0 -> 1:
Path Cost: 4
Path: ['0', '1']

0 -> 7:
Path Cost: 8
Path: ['0', '7']
```

The input file format:

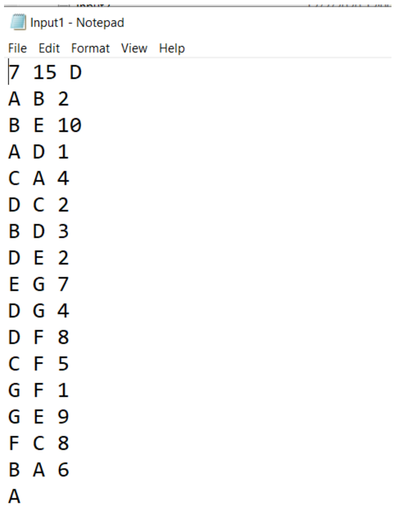
Here, the first two numbers represent the number of vertices and edges third is Graph type(Directed/Undirected).
U represents Undirected Graph.D represent Directed Graph
From the second line, it mentions all edges and its weight.



```
9 12 U
A B 4
B H 6
B C 1
C D 3
D H 1
D E 2
D F 1
E F 1
F G 1
G H 2
G I 2
H I 2
A
```

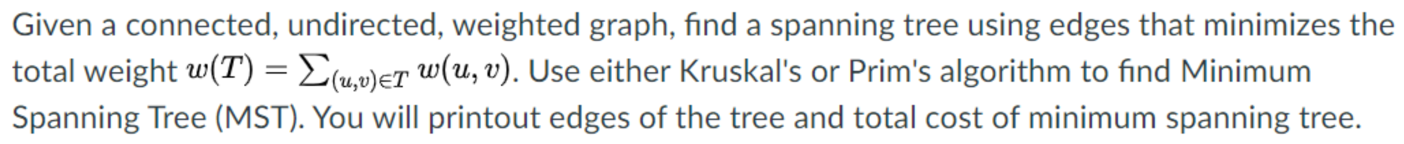
E.g. In the above input file, there are 9 vertices and 12 edges and Undirected Graph. Edge (A, B) has weight 4

and in the below input file, there are 7 vertices and 15 edges and Directed Graph. Edge (A, B) has weight 2.



```
7 15 D
A B 2
B E 10
A D 1
C A 4
D C 2
B D 3
D E 2
E G 7
D G 4
D F 8
C F 5
G F 1
G E 9
F C 8
B A 6
A
```

Problem 2: Minimum Spanning Tree Algorithm


Given a connected, undirected, weighted graph, find a spanning tree using edges that minimizes the total weight $w(T) = \sum_{(u,v)\in T} w(u, v)$. Use either Kruskal's or Prim's algorithm to find Minimum Spanning Tree (MST). You will printout edges of the tree and total cost of minimum spanning tree.

Prim’s Algorithm

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm may informally be described as performing the following steps:

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

1) Pseudocode and detail analysis of runtime

Pseudocode of Minimum Spanning Tree:

```
MST-PRIM(G, w, r)

1  for each $u \in G.V$
2       $u.key = \infty$
3       $u.\pi = NIL$
4  $r.key = 0$
5  $Q = G.V$
6  while $Q \neq \emptyset$
7        $u = EXTRACT\text{-}MIN(Q)$
8        for each $v \in G.Adj[u]$
```

```
9        if $v \in Q$ and $w(u,v) < v.key$
10           $v.\pi = u$
11           $v.key = w(u,v)$
```

Detail analysis of runtime

- A graph has n vertices, thus the ‘for’ loop in the first line would run n times i.e. ‘for’ loop would cost O(n).
- As heap has n items, Q= G.V, constructing a heap would take O(n logn) time.
- In the ‘while’ loop, Extract-MIN(Q) will be called one time for each node. As the Q has n items, ‘while’ loop would be called n times and cost for each call would be O(log n). Therefore, the total cost would be O(n logn)
- The key-value would be changed based upon the total number of edges in the graph i.e. m. Whenever we change the key-value, the order of the nodes in the heap might change too. So, we’ll need to modify the heap which costs O(log n) time for each call. Therefore, the total cost would be O(m logn) Relax operation takes o(logn) time and for loop containing relax runs the number of edges time,means m times,so the total runtime for relax operation will be O(mlogn)
- So, the total runtime time of this algorithm would be O(n logn + m logn) i.e. O((n+m)logn) which is O(m logn).

2) <u>Data Structures used in the Pseudocode and how it affects runtime of algorithm:</u>

Data Structures

- Heap
- Priority Queue

How do these data structures affect the runtime of the algorithm?

- Performance depends on implementation of priority queue:
  - binary heap implementation: O(m log n)
  - Fibonacci heap implementation: O(log n + m)
- Option 1: implement priority queue with binary heap

    - Cost of initializing Q and first for loop: O(n log n)
    - Cost of decreasing key of r: O(log n)
    - While loop:
        - V calls to front (=extract min) → O(n log n)
        - E (or fewer) calls to decrease key → O(m log n)
        - Total for loop: O((n + m) log n)
    - Overall Total: O((n + m) log n) = O(m log n),
        - Since m ≥ n+1, since G is connected
- Option 2: implement priority queue with Fibonacci heap
    - Decrease key can be done in O(1) (amortized) time
        - Amortized means the cost is spread out
        - Average time is O(1).
        - Other times may sometimes be larger.
    - E Calls to decrease key take O(m) * O(1) = O(m) time
    - Total: O(n log n + m)

3)Implementation of Minimum Spanning Tree Algorithm

Implementation code of Prim's Algorithm

#Program to find the Minimum Spanning Tree using Prim's Algorithm and Heap Data Structure

```
import sys
#Implementation of Heap
class Heap:
    s = 0
    a = []
    p = {}
    key = {}

    def __init__(self):
        self.s = -1
```

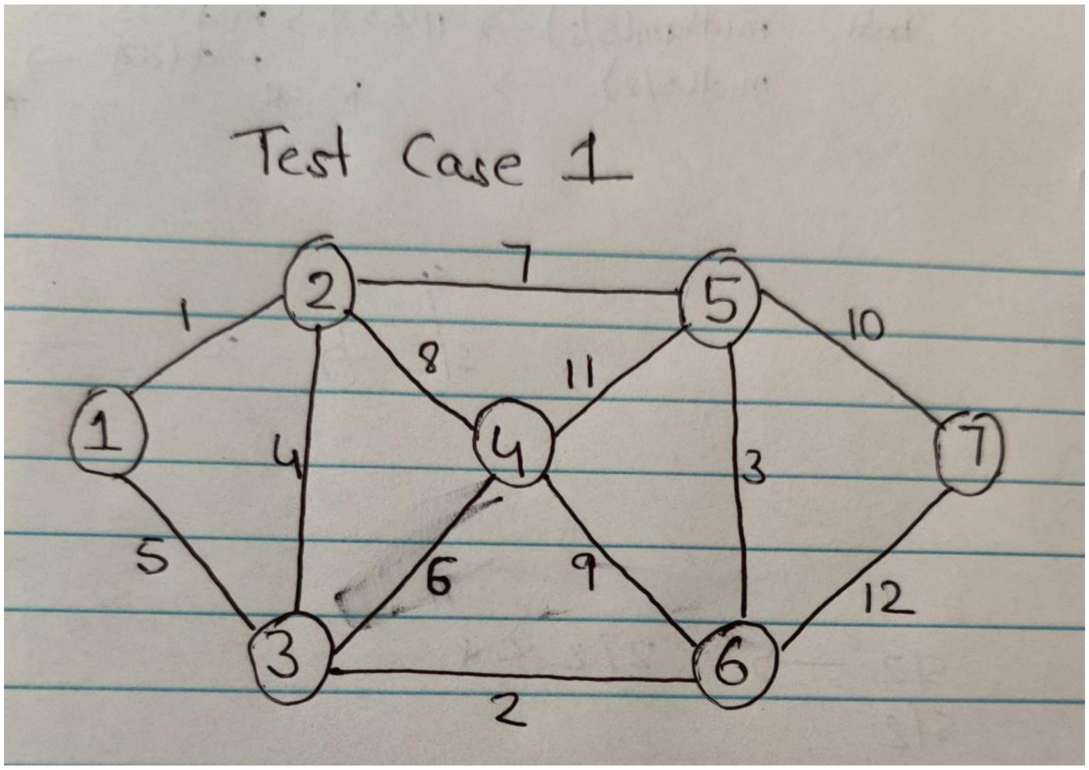
```
def heapify_up(self, i):
    while i>0:
        j = i//2
        if self.key[self.a[i]] < self.key[self.a[j]]:
            temp = self.a[i]
            self.a[i] = self.a[j]
            self.a[j] = temp
            self.p[self.a[i]] = i
            self.p[self.a[j]] = j
            i = j
        else:
            break

def heapify_down(self,i):
    j=-1
    while 2*i <= self.s:
        if 2*i == self.s or self.key[self.a[2*i]] <= self.key[self.a[2*i + 1]]:
            j = 2*i
        else:
            j = 2*i + 1
        if self.key[self.a[j]] < self.key[self.a[i]]:
            temp = self.a[i]
            self.a[i] = self.a[j]
            self.a[j] = temp
            self.p[self.a[i]] = i
            self.p[self.a[j]] = j
            i = j
        else:
            break

def decrease_key(self, v, key_value):
    self.key[v] = key_value
    self.heapify_up(self.p[v])
    def extract_min(self):
        ret = self.a[0]
        self.a[0]=self.a[self.s]
        self.p[self.a[0]] = 0
        self.s-=1
        if self.s>=0:
            self.heapify_down(0)
        return ret

    def insert(self, v, key_value):
        self.a.append(v)
        self.s +=1
        self.p[v] = self.s
        self.key[v] = key_value
        self.heapify_up(self.s)

    def printdata(self):
        print("Value of array a: ", self.a)
        print("Value of p: ", self.p)
        print("Value of key: ", self.key)

#Reading data from file
f = open("MST/input1_MST.txt")
lines = f.readlines()
f.close()

#Global variables for the Minimum Spanning Tree
Q = Heap()
n,m = map(int,lines[0].strip().split(" "))
edges = [[-1 for x in range(0,n+1)] for y in range(0,n+1)]
d = {}
pi = {}
S = []
V = []
total_weight = 0
```

```
edges_mst = [None]*(n-1)

#Add all nodes to the list V
for i in range(1,n+1):
    V.append(i)

#Add all edges to the edges matrix
for i in range(0,m):
    p,q,r = map(int,lines[i+1].strip().split(" "))
    edges[p][q] = r
    edges[q][p] = r #Adding the edges for both because it is an undirected graph

#Choosing the Arbitrary vertex as "Vertex 1"
d[1] = 0
Q.insert(1,d[1])

#Inserting the Infinity value for all other vertices
for i in range(1,n+1):
    d[i] = sys.maxsize
    Q.insert(i,d[i])

#Finding the Minimum Spanning Tree
while set(S)!=set(V):
    u = Q.extract_min()
    S.append(u)
    left = list(set(V) - set(S))
    for v in left:
        if edges[u][v] != -1:
            if edges[u][v] < d[v]:
                d[v] = edges[u][v]
                Q.decrease_key(v,d[v])
                pi[v] = u

#Adding the list of edges into a string array and calculating total weight
i = 0
```

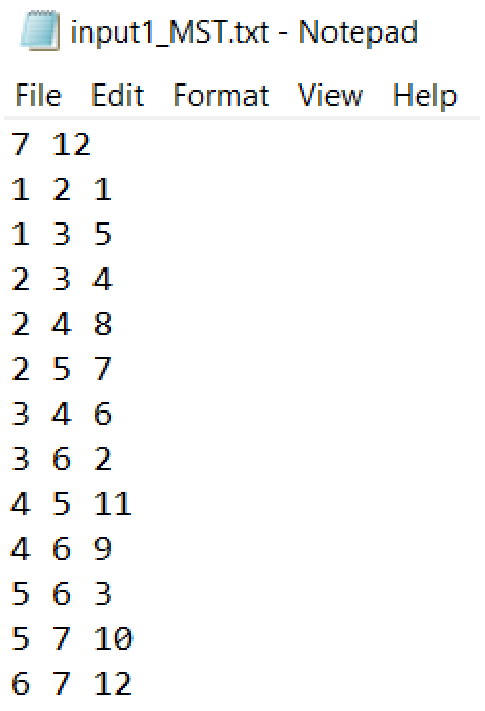
```
for v in list(set(V) - set({1})):
    total_weight+=edges[v][pi[v]]
    if v < pi[v]:
        edges_mst[i] = "    " +str(v) + " - " + str(pi[v]) + "    \t" + str(edges[v][pi[v]])
    else:
        edges_mst[i] = "    " +str(pi[v]) + " - " + str(v) + "    \t" + str(edges[v][pi[v]])
    i+=1

print("Total cost of Minimum Spanning Tree (MST):",str(total_weight)+"\n")
print("Edges of MST   Weight \n")
for i in range(0, n-1):
    print(edges_mst[i])
```

4)Test on four different graphs and printing tree edges and cost

Test Case 1



Input

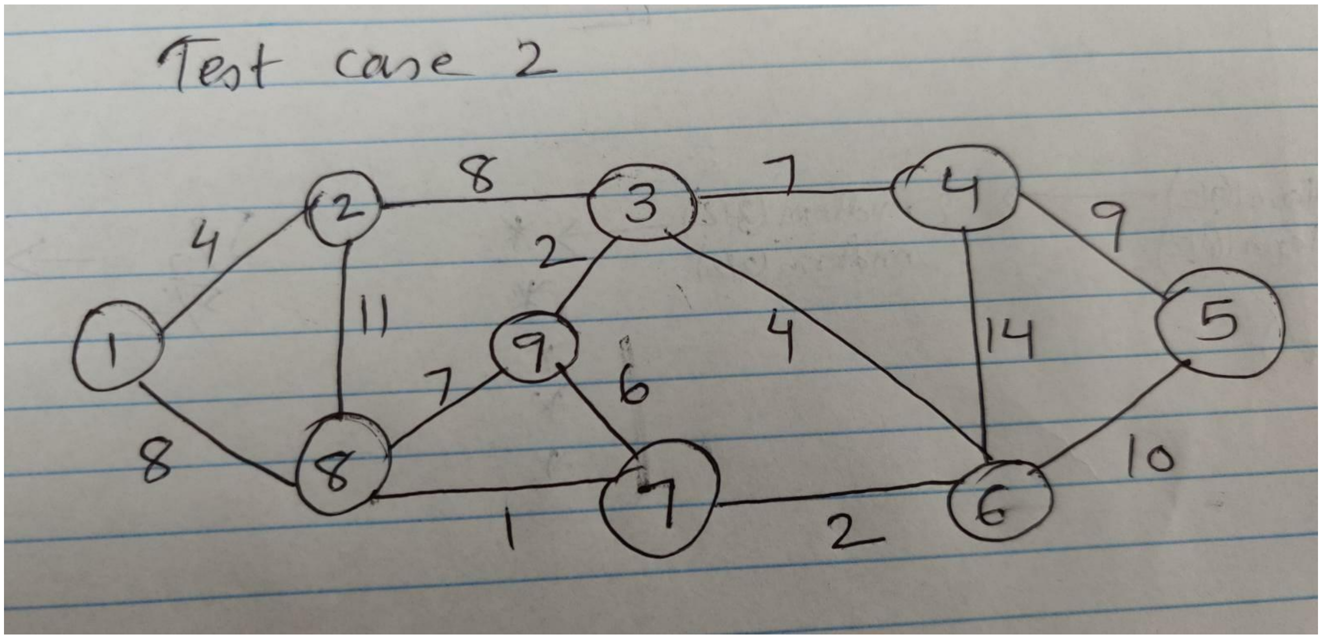


```
7 12
1 2 1
1 3 5
2 3 4
2 4 8
2 5 7
3 4 6
3 6 2
4 5 11
4 6 9
5 6 3
5 7 10
6 7 12
```

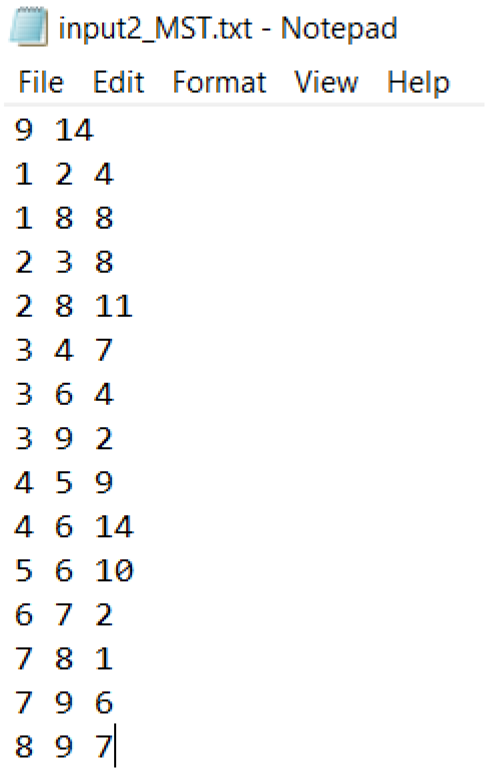
Output

```
Total cost of Minimum Spanning Tree (MST): 26

Tree Edges   Cost

   1 - 2       1
   2 - 3       4
   3 - 4       6
   5 - 6       3
   3 - 6       2
   5 - 7       10
```

Test Case 2



Input

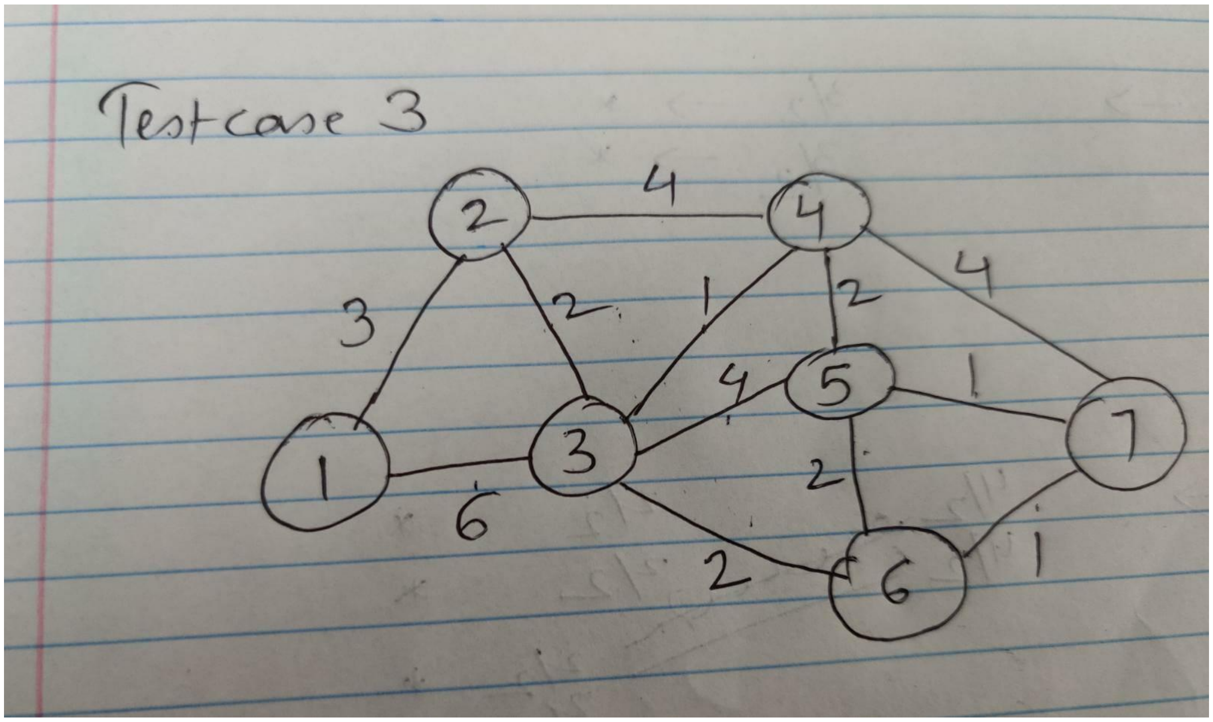


```
9 14
1 2 4
1 8 8
2 3 8
2 8 11
3 4 7
3 6 4
3 9 2
4 5 9
4 6 14
5 6 10
6 7 2
7 8 1
7 9 6
8 9 7
```

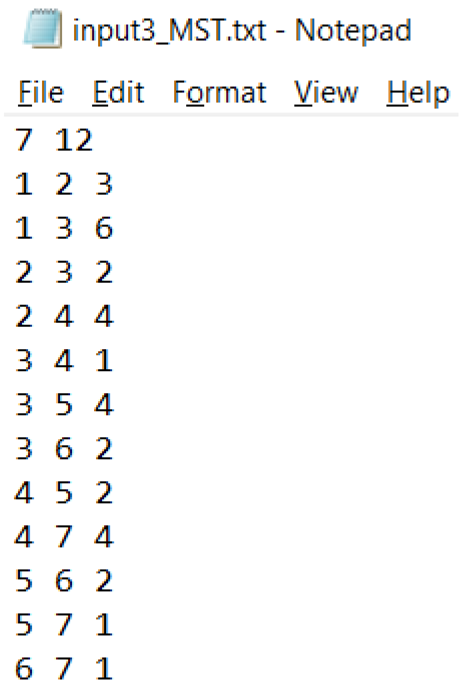
Output

```
Total cost of Minimum Spanning Tree (MST): 37

Tree Edges   Cost

   1 - 2        4
   3 - 6        4
   3 - 4        7
   4 - 5        9
   6 - 7        2
   7 - 8        1
   1 - 8        8
   3 - 9        2
```

Test Case 3

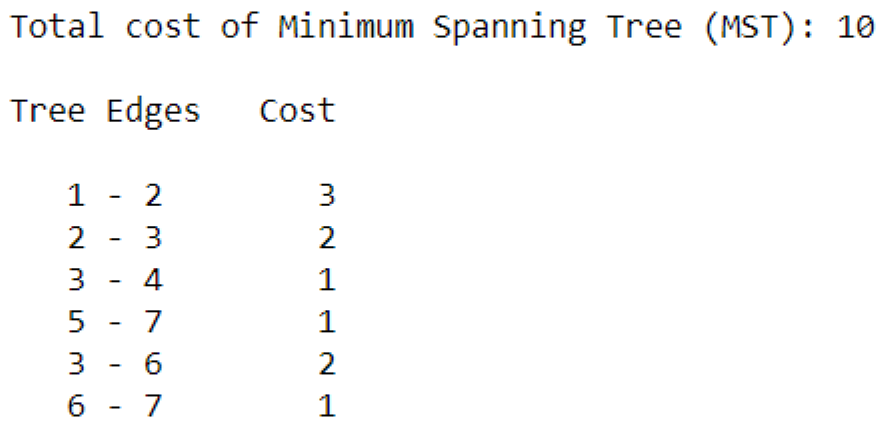


Input



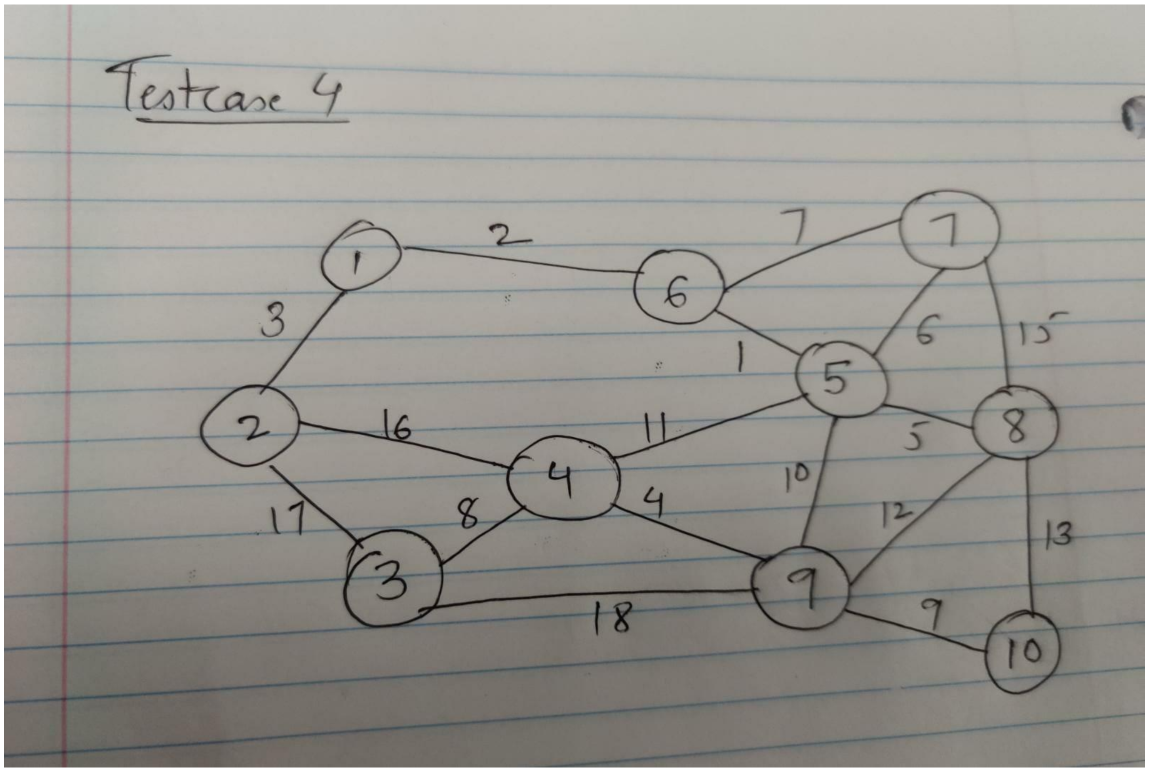
```
7 12
1 2 3
1 3 6
2 3 2
2 4 4
3 4 1
3 5 4
3 6 2
4 5 2
4 7 4
5 6 2
5 7 1
6 7 1
```

Output

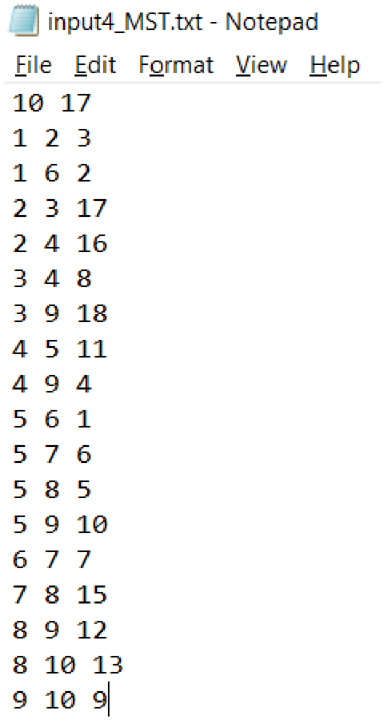


```
Total cost of Minimum Spanning Tree (MST): 10

Tree Edges    Cost

   1 - 2         3
   2 - 3         2
   3 - 4         1
   5 - 7         1
   3 - 6         2
   6 - 7         1
```

Test Case 4

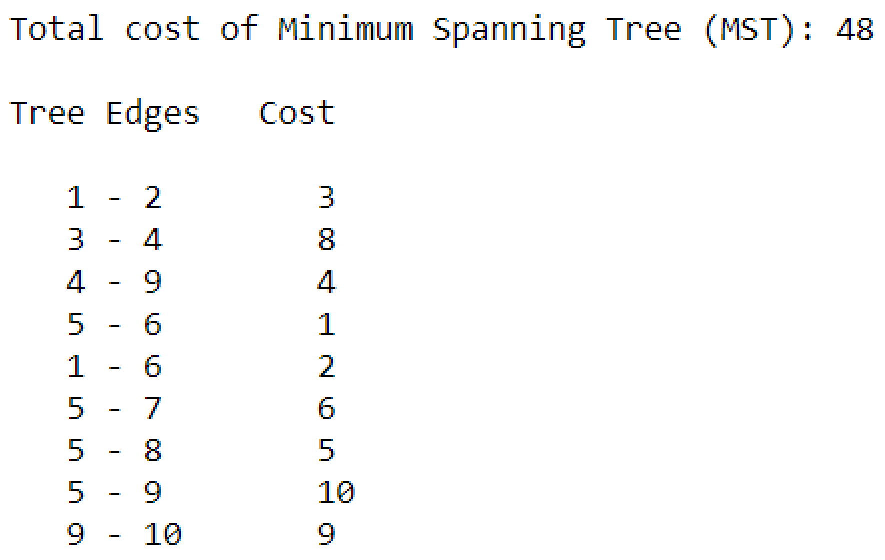


Input



```
10 17
1 2 3
1 6 2
2 3 17
2 4 16
3 4 8
3 9 18
4 5 11
4 9 4
5 6 1
5 7 6
5 8 5
5 9 10
6 7 7
7 8 15
8 9 12
8 10 13
9 10 9
```

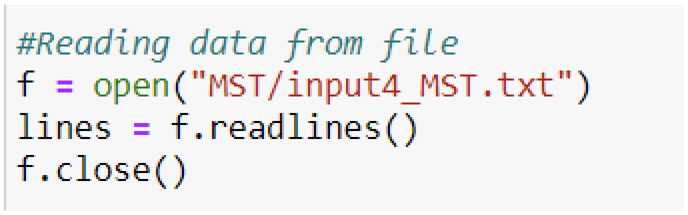
Output



```
Total cost of Minimum Spanning Tree (MST): 48

Tree Edges   Cost

   1 - 2       3
   3 - 4       8
   4 - 9       4
   5 - 6       1
   1 - 6       2
   5 - 7       6
   5 - 8       5
   5 - 9       10
   9 - 10      9
```

INSTRUCTIONS TO RUN THE MST IMPLEMENTATION CODE

1) **Open the python notebook(Project2_SSSP_MST.ipynb)**
2) **Run the block with code for MST**
3) **To test different test cases, change the name of the file in the path as seen in the below screenshot.**

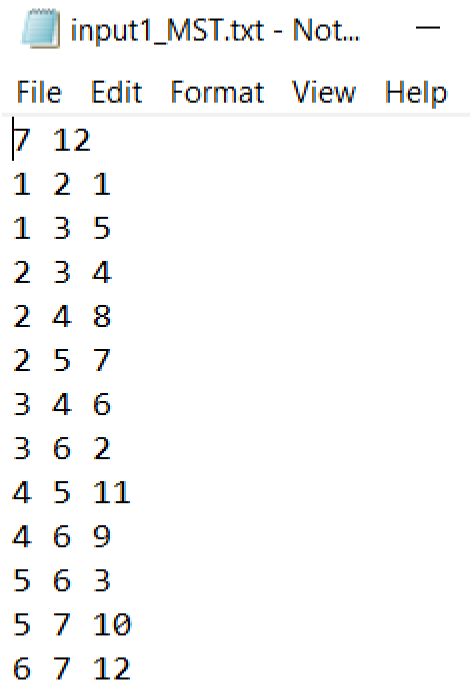


```
#Reading data from file
f = open("MST/input4_MST.txt")
lines = f.readlines()
f.close()
```

The different input file names are- input1_MST.txt, input2_MST.txt, input3_MST.txt and input4_MST.txt for the 4 test cases.

The input file format:

Here, the first two numbers represent the number of vertices and edges. From the second line, it mentions all edges and its weight.



```
7 12
1 2 1
1 3 5
2 3 4
2 4 8
2 5 7
3 4 6
3 6 2
4 5 11
4 6 9
5 6 3
5 7 10
6 7 12
```

E.g. In the above input file, there are 7 vertices and 12 edges. Edge (1, 2) has weight 1.