Walchand College of Engineering, Sangli
Computer Science & Engineering
Third Year

Course: Design and analysis of algorithm Lab

Lab course coordinator:
Mrs A M Chimanna- Batch: - T1, T2, T3,T4,T5,T8
Mr.R R Patil- Batch: - T6,T7

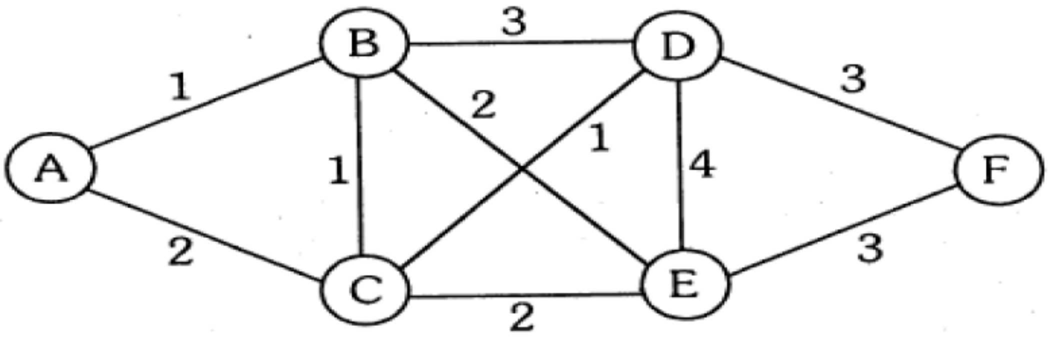
# Week 8 Assignment

## Greedy Method

**Name:** Trupti Rajendra Patil

**Prn:** 2020BTECS00051

1. **From a given vertex in a weighted connected graph, implement shortest path finding Dijkstra's algorithm.**



**Dijkstra's Algorithm:**

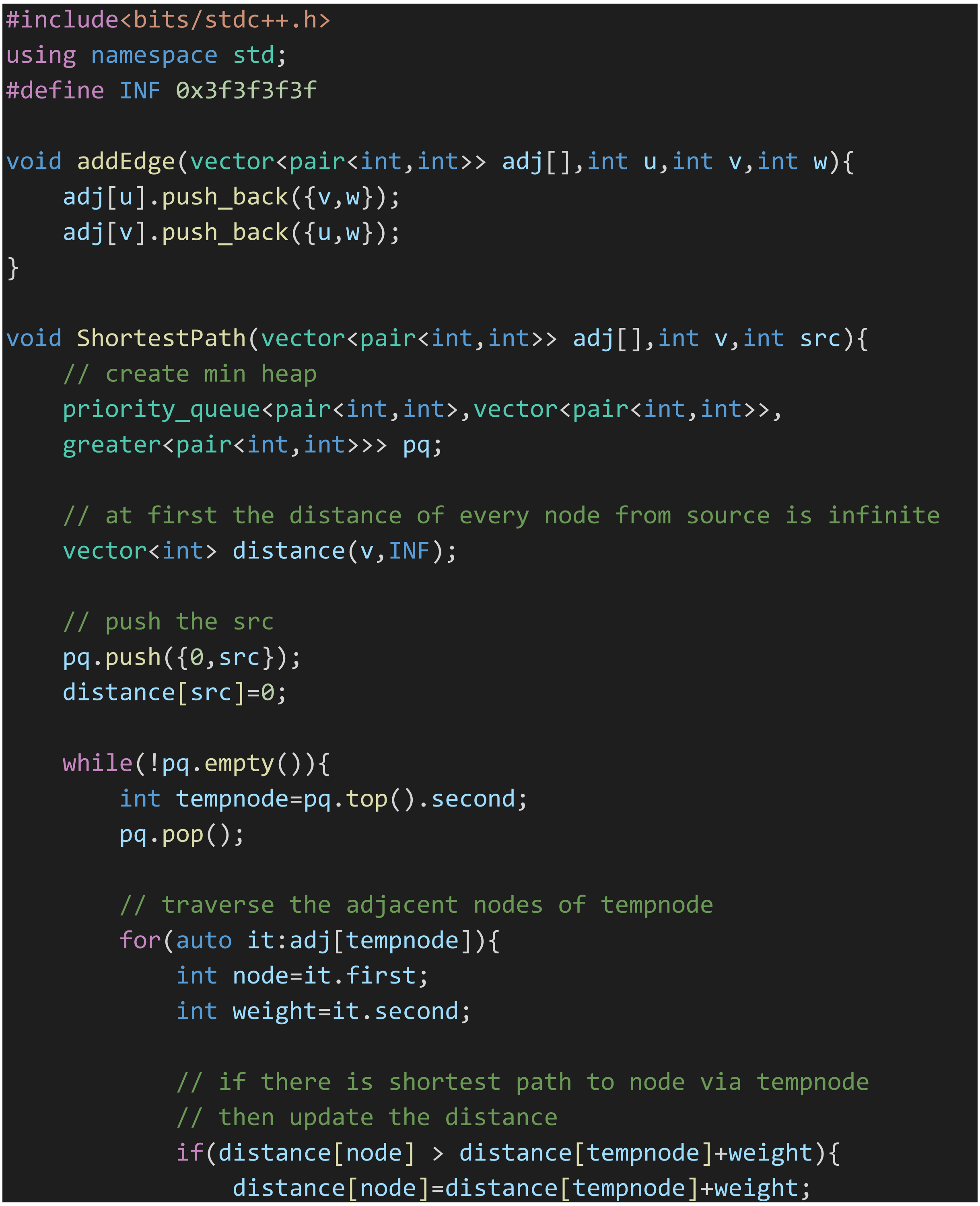
**Algorithm:**

1) Initialize distances of all vertices as infinite.

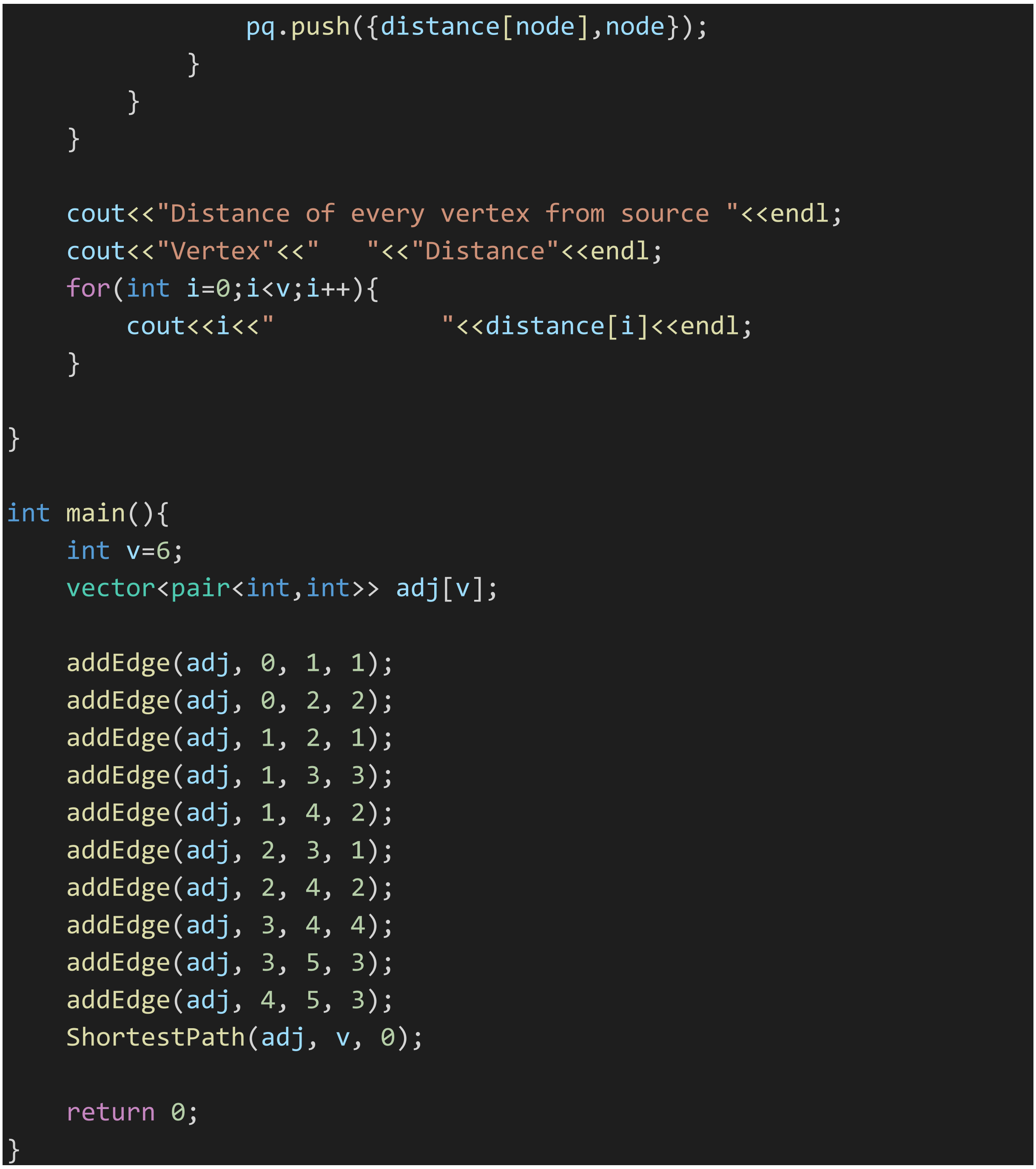
2) Create an empty priority queue pq. Every item of pq is a pair (weight, vertex). Weight (or distance) is used as first item of pair as first item is by default used to compare two pairs
3) Insert source vertex into pq and make its distance as 0.
4) While either pq doesn't become empty
   a) Extract minimum distance vertex from pq. Let the extracted vertex be tempnode.
   b) Loop through all adjacent of tempnode, do following for every vertex node.

```
        // If there is a shorter path to node through tempnode.
        If dist[node] > dist[tempnode] + weight(tempnode, node)
            (i) Update distance of node, i.e., do
                    dist[node] = dist[tempnode] + weight(tempnode, node)
            (ii) Insert node into the pq (Even if node is already there)
5) Print distance array dist[] to print all shortest paths.
```

**Code:**

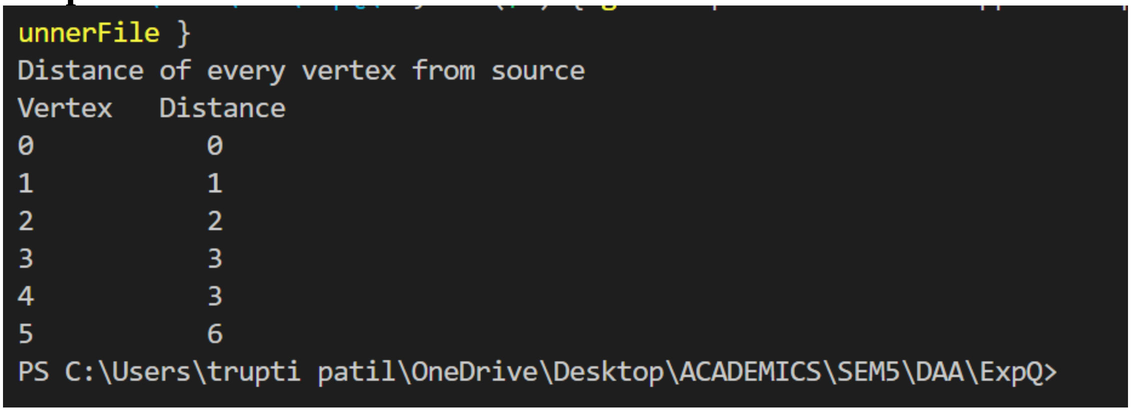


```cpp
#include<bits/stdc++.h>
using namespace std;
#define INF 0x3f3f3f3f

void addEdge(vector<pair<int,int>> adj[],int u,int v,int w){
    adj[u].push_back({v,w});
    adj[v].push_back({u,w});
}

void ShortestPath(vector<pair<int,int>> adj[],int v,int src){
    // create min heap
    priority_queue<pair<int,int>,vector<pair<int,int>>,
    greater<pair<int,int>>> pq;

    // at first the distance of every node from source is infinite
    vector<int> distance(v,INF);

    // push the src
    pq.push({0,src});
    distance[src]=0;

    while(!pq.empty()){
        int tempnode=pq.top().second;
        pq.pop();

        // traverse the adjacent nodes of tempnode
        for(auto it:adj[tempnode]){
            int node=it.first;
            int weight=it.second;

            // if there is shortest path to node via tempnode
            // then update the distance
            if(distance[node] > distance[tempnode]+weight){
                distance[node]=distance[tempnode]+weight;
```



```
                pq.push({distance[node],node});
            }
        }
    }

    cout<<"Distance of every vertex from source "<<endl;
    cout<<"Vertex"<<"   "<<"Distance"<<endl;
    for(int i=0;i<v;i++){
        cout<<i<<"           "<<distance[i]<<endl;
    }

}

int main(){
    int v=6;
    vector<pair<int,int>> adj[v];

    addEdge(adj, 0, 1, 1);
    addEdge(adj, 0, 2, 2);
    addEdge(adj, 1, 2, 1);
    addEdge(adj, 1, 3, 3);
    addEdge(adj, 1, 4, 2);
    addEdge(adj, 2, 3, 1);
    addEdge(adj, 2, 4, 2);
    addEdge(adj, 3, 4, 4);
    addEdge(adj, 3, 5, 3);
    addEdge(adj, 4, 5, 3);
    ShortestPath(adj, v, 0);

    return 0;
}
```

**Output:**

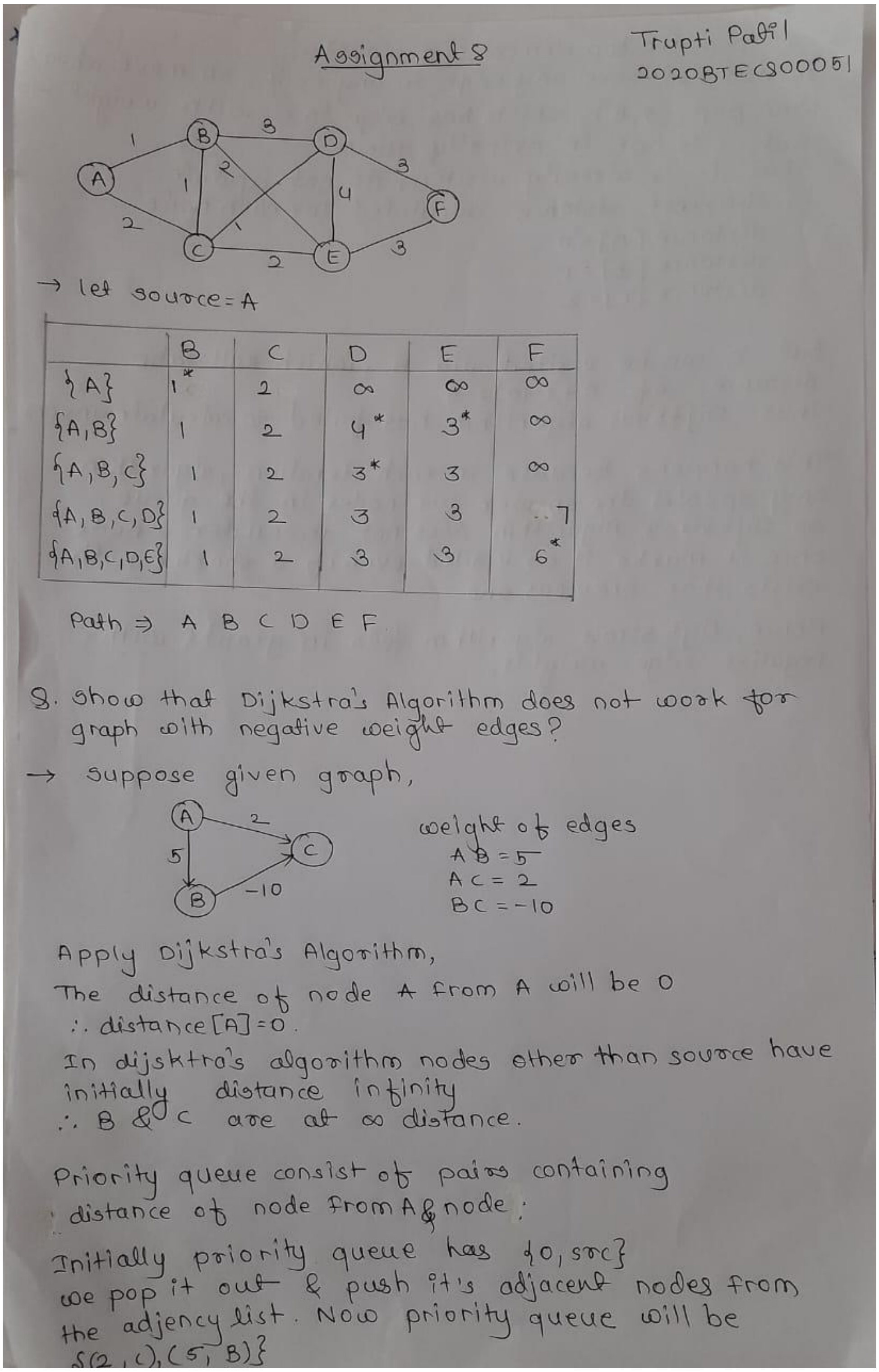


```
unnerFile }
Distance of every vertex from source
Vertex   Distance
0           0
1           1
2           2
3           3
4           3
5           6
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
```

**Complexity Analysis:**

Time Complexity: O (E log V) where E is number of edges and Vis number of vertices.

### Q) Show that Dijkstra’s algorithm does not work for graphs with negative weight edges



Assignment 8

Trupti Patil
2020BTECS00051

A
B
C
D
E
F
1
2
1
2
1
3
4
2
3
3

→ let source = A

| | B | C | D | E | F |
|---|---|---|---|---|---|
| {A} | 1* | 2 | ∞ | ∞ | ∞ |
| {A,B} | 1 | 2 | 4* | 3* | ∞ |
| {A,B,C} | 1 | 2 | 3* | 3 | ∞ |
| {A,B,C,D} | 1 | 2 | 3 | 3 | 7 |
| {A,B,C,D,E} | 1 | 2 | 3 | 3 | 6* |

Path ⇒ A B C D E F

Q. show that Dijkstra's Algorithm does not work for graph with negative weight edges?

→ suppose given graph,

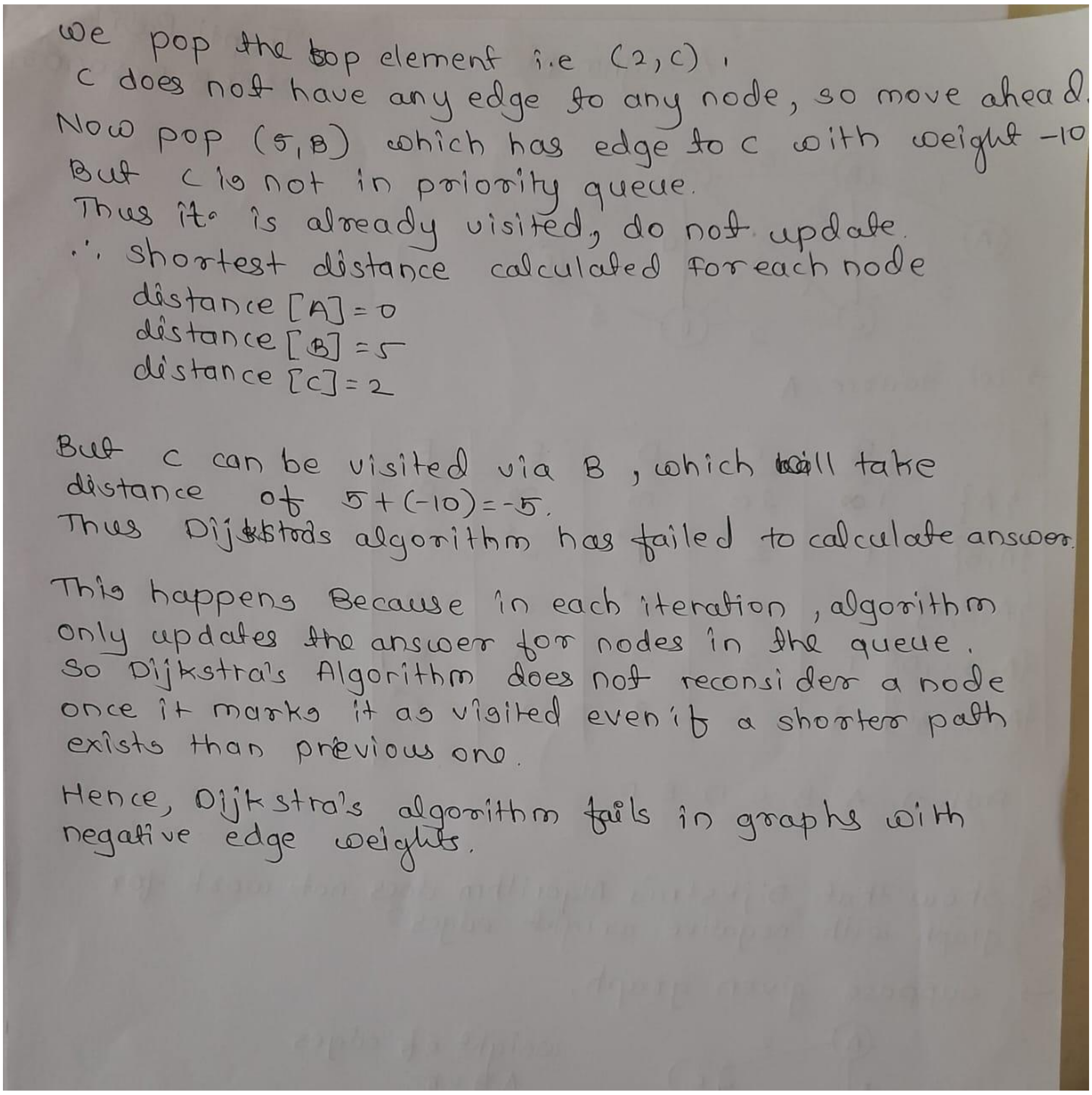
A
2
C
5
B
-10

weight of edges
AB = 5
AC = 2
BC = -10

Apply Dijkstra's Algorithm,
The distance of node A from A will be 0
∴ distance[A] = 0.
In dijsktra's algorithm nodes other than source have initially distance infinity
∴ B & C are at ∞ distance.

Priority queue consist of pairs containing distance of node from A & node;

Initially priority queue has {0, src}
we pop it out & push it's adjacent nodes from the adjency list. Now priority queue will be
{(2, C), (5, B)}



we pop the top element i.e (2,c).
c does not have any edge to any node, so move ahead.
Now pop (5,B) which has edge to c with weight -10
But c is not in priority queue.
Thus it is already visited, do not update.
∴ shortest distance calculated for each node

distance [A] = 0
distance [B] = 5
distance [C] = 2

But c can be visited via B, which will take distance of 5+(-10)=-5.
Thus Dijkstra's algorithm has failed to calculate answer.

This happens Because in each iteration, algorithm only updates the answer for nodes in the queue. So Dijkstra's Algorithm does not reconsider a node once it marks it as visited even if a shorter path exists than previous one.

Hence, Dijkstra's algorithm fails in graphs with negative edge weights.

**Q) Modify the Dijkstra's algorithm to find shortest path.**

Modification of Dijkstra's algorithm to find shortest path can be done by counting the edges of path travelled. If new shortest path is discovered which is of the same distance as the last shortest path, make an if statement asking whether or not the new path has less number of edges.

```
//The relaxation part of Dijkstra's algorithm
if (new_path == shortest_path && new_path_edges < shortest_path_edges)
   shortest_path= new_path
elseif (new_path < shortest_path)
```

```
function Dijkstra(Graph, source):
       for each vertex v in Graph:
           dist[v]   := infinity;
            dist_edges[v]:= 0;
           visited[v]   := false;
           previous[v]  := undefined;
       end for

       dist[source]  := 0;
        dist_edges[source] := 0;
       insert source into Q;

      while Q is not empty:
          u := vertex in Q with smallest distance in dist[] and has not been visited;
          remove u from Q;
          visited[u] := true

          for each neighbor v of u:
             alt := dist[u] + dist_between(u, v);
                alt_edges := dist_edges[u] + 1; //Note the increment by 1
                if (alt = dist[v] && alt_edges < dist_edges[v])
                    previous[v] := u;
                    dist_edges[v]= alt_edges
              if alt < dist[v]:
                  dist[v]  := alt;
                    dist_edges[v] := alt_edges;
                  previous[v]  := u;
                  if !visited[v]:
                       insert v into Q;
                  end if
              end if
          end for
      end while
      return dist;
  endfunction
```