Shortest Path

Dijkstra Algorithm (SSSP)

Dijkstra's algorithm finds shortest paths from the starting node to all nodes of the graph. Dijkstra's **algorithm** runs with a **time complexity** of O(E+VlogV). Dijkstra's algorithm is efficient,because it only processes each edge in the graph once, using the fact that there are no negative edges.

The graph is stored as adjacency lists so that adj[u] contains a pair (v,w) always when there is an edge from node u to node v with weight w.

```
vector<pair<int,int>>adj[n+1];
```

At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is smallest. An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time. The priority queue contains a pair to store the vertex and it's corresponding weight.

```
priority_queue<pair<int,int>>pq;
```

Initially the distance is 0 to src and 1 to all other nodes.

```
    //Make all the distance Infinity
    for(int i=0;i<=n;i++){
        dist[i]=INF;
    }
    //Starting Node Distance is Zero
    dist[src]=0;
```

Using this pair of vertex and weight we process the adjacent vertices.

A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. So we use a boolean vector to keep a track of processed node.

```
    //Track of Processed Vertex
    vector<bool>visited(n+1,0);
```

The complete implementation is

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

void dijkstra(int src,vector<pair<int,int>>adj[],int n){
    priority_queue<pair<int,int>,vector<pair<int,int>>,
greater<pair<int,int>>>pq;
    int dist[n+1];
    //Make all the distance Infinity
    for(int i=0;i<=n;i++){
```

```
        dist[i]=INF;
    }
    //Starting Node Distance is Zero
    dist[src]=0;
    pq.push({0,src});

    //Track of Processed Vertex
    vector<bool>visited(n+1,0);

    while(!pq.empty()){

        pair<int,int> pick = pq.top();
        pq.pop();
        int src = pick.second;
        int wt = pick.first;
        if(visited[src])continue;
        //This obtained distance is minimal distance
        dist[src]=wt;
        visited[src]=1;
        //See the Adjacent Edge and Update if distance is minimal
        for(auto v:adj[src]){
            if(dist[src]+v.second<dist[v.first]){
                dist[v.first]=dist[src]+v.second;
                pq.push({dist[v.first],v.first});
            }
        }
    }
    for(int i=1;i<=n;i++){
        cout<<"1"<<" to "<<i<<"-->"<<dist[i]<<endl;
    }
}

int main(){
    int n;
    cin>>n;
    int e;
    cin>>e;
    bool bidir;
    cin>>bidir;
    vector<pair<int,int>>adj[n+1];
    for(int i=0;i<e;i++){
        int u,v,w;
        cin>>u>>v>>w;
        adj[u].push_back({v,w});
        adj[v].push_back({u,w});
    }
    dijkstra(1,adj,n);
}
```

Note

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. Since it is a greedy approach it may fail or pass.

Bellman-Ford Algorithm(SSSP)

We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(E+VLogV) (with the use of min heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.*

The Bellman–Ford algorithm finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to all other nodes in infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Algorithm

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as INT_MAX except dist[src]=0 where src is source vertex.

```
for(int i=1;i<=v;i++){
        dist[i]=INT_MAX;
    }
dist[src]=0;
```

2) Create a adjList which contain all the edges of the graph . I used here a map data structure with pair of pair to store src,dest,wt.

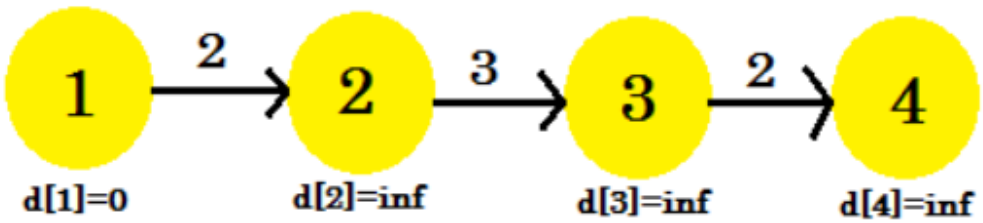
```
map<int,pair<int,pair<int,int>>>adjList;

void addEdge(int count,int u,int v,int w){
    adjList[count]=make_pair(u,make_pair(v,w));
}
```

3) This step calculates shortest distances. Perform relaxation on all the edges |N|-1 times where |N| is the number of vertices in given graph.

```
if(dist[u]!=INT_MAX && dist[u]+w<dist[v]){
                dist[v]=dist[u]+w;
            }
```

Why relaxation |N| - 1 time ?

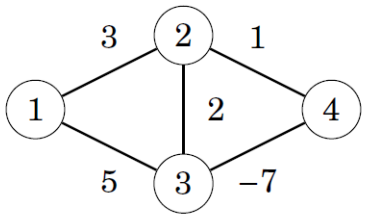
Let's take a look at an example:



Here, the **source** vertex is 1. We will find out the shortest distance between the **source** and all the other vertices. We can clearly see that, to reach **vertex 4**, in the worst case, it'll take **(V-1)** edges. Now depending on the order in which the edges are discovered, it might take **(V-1)** times to discover **vertex 4**.

Negative Cycles

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. For example the following graph :



contains a negative cycle 2 -> 3 -> 4 -> 2 with length -4.

> **Note :** The detection of cycle depends on the starting node we chose in directed graph.

The complete implementation of bellman-ford algorithm :

```
#include<bits/stdc++.h>
using namespace std;
map<int,pair<int,pair<int,int>>>adjList;

void addEdge(int count,int u,int v,int w){
    adjList[count]=make_pair(u,make_pair(v,w));
}

void BellmanFord(int src,int v,int e){
    int dist[v+1];
    for(int i=1;i<=v;i++){
        dist[i]=INT_MAX;
    }
    dist[src]=0;

    for(int i=1;i<v;i++){
        for(int j=0;j<e;j++){
            int u=adjList[j].first;
            int v=adjList[j].second.first;
```

```
            int w=adjList[j].second.second;
            if(dist[u]!=INT_MAX && dist[u]+w<dist[v]){
                dist[v]=dist[u]+w;
            }
        }
    }
    bool flag=0;
    for(int j=0;j<e;j++){
            int u=adjList[j].first;
            int v=adjList[j].second.first;
            int w=adjList[j].second.second;
            if(dist[u]!=INT_MAX && dist[u]+w<dist[v]){
                dist[v]=dist[u]+w;
                flag=1;
                break;
            }
    }
    if(flag==1){
        cout<<"negative edge encountered"<<endl;
        return ;
    }
    for(int i=2;i<=v;i++){
        cout<<dist[i]<<" ";
    }
    cout<<endl;
}

int main(){
    int n,m;
    cin>>n>>m;
    for(int i=0;i<m;i++){
        int u,v,w;
        cin>>u>>v>>w;
        addEdge(i,u,v,w);
    }
    BellmanFord(1,n,m);
}
```

Detect Negative Cycle in a Directed Graph G ?

As said earlier in Bellman Ford algorithm detection of negative cycle in Graph depends solely upon the starting node. It will detect negative cycles if they are reachable from the starting node. In undirected connected graph we don't have this problem because all nodes are reachable from any node. Hypothetically we can think to solve this problem in directed graph we can add one node and connect it to all the other nodes using directed edges with weight 0. In practice we can simply update the dist to be 0 . This will prevent us writing extra lines of code.

To Get The Nodes of Negative Cycle

Find an vertex which is affected by negative cycle using the above method. The vertex maybe the part of cycle or a vertex which is affected by the cycle. So to get the nodes of negative cycle, traverse the nth parent of that node. And we reach the cycle.

Shortest Path Faster Algorithm

The shortest path faster algorithm is based on Bellman-Ford algorithm where every vertex is used to relax its adjacent vertices but in SPF algorithm, is often more efficient because does not go through all the edges on each round it chooses the edges to be examined in a more intelligent way. To stop preventing duplicate node in queue we maintain a boolean array which keeps track of vertices in queue.

```
bool inQueue[V + 1] = { false };
```

Algorithm

1. Create an array **dist[]** to store the shortest distance of all vertex from the source vertex. Initialize this array by infinity except for dist[src] = 0 where **src** is source vertex.
2. Create a queue **Q** and push starting source vertex in it
    - while Queue is not empty, do the following for each edge(u, v) in the graph
        - If d[v] > d[u] + weight of edge(u, v)
        - d[v] = d[u] + weight of edge(u, v)
        - If vertex v is not present in Queue, then push the vertex v into the Queue.

Floyd-Warshall Algorithm

The Floyd–Warshall algorithm provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms of this chapter, it finds all pair shortest paths between the nodes in a single run.

Algorithm

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Implementation

The following code constructs a distance matrix where distance[a] [b] is the shortest distance between nodes a and b. First, the algorithm initializes distance matrix and then fill given edges connection using the adjacency matrix adj of the graph.

```
    // dynamically create array of pointers of size M
    int** distance = new int*[N+1];

    // dynamically allocate memory of size N for each row
    for (int i = 1; i <=N; i++)
        distance[i] = new int[N+1];
```

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

Here one by one pick all vertices and updates all paths between i & j which include the picked vertex(k) as an intermediate vertex .

```
for(int k=1;k<=n;k++){
 for(int i=1;i<=n;i++){
    for(int j=1;j<=n;j++){
       if(distance[i][k]!=INF && distance[k][j]!=INF){
           distance[i][j]=min(distance[i][j],distance[i][k]+distance[k][j]);
        }
     }
  }
}
```

The complete implementation is :

```
#include<bits/stdc++.h>
using namespace std;

#define INF INT_MAX

int
main ()
{

  int n, m, q;
  cin >> n >> m >> q;

  // dynamically create array of pointers of size M
  int **distance = new int *[n + 1];

  // dynamically allocate memory of size N for each row
  for (int i = 1; i <= n; i++)
    {
      distance[i] = new int[n + 1];
    }

  for (int i = 1; i <= n; i++)
    {
      for (int j = 1; j <= n; j++)
    {
      if (i == j)
        distance[i][j] = 0;
      else
        distance[i][j] = INF;
    }
    }

  for (int i = 0; i < m; i++)
    {
      int u, v, w;
      cin >> u >> v >> w;
      distance[u][v] = w;
      distance[v][u] = w;
    }

  for (int k = 1; k <= n; k++)
    {
      for (int i = 1; i <= n; i++)
    {
```

```
      for (int j = 1; j <= n; j++)
        { //To Handle The OverFlow Error. Since INF = INT_MAX
          if (distance[i][k] != INF && distance[k][j] != INF)
        {
          distance[i][j] =
            min (distance[i][j], distance[i][k] + distance[k][j]);
        }
        }
    }
    }

  for (int i = 1; i <= n; i++)
    {
      for (int j = 1; j <= n; j++)
    {
      if (distance[i][j] == INT_MAX)
        {
          cout << "INF" << "\t";
        }
      else
        {
          cout << distance[i][j] << "\t";
        }
    }
      cout << endl;
    }
}
```

> A negative cycle is one in which the overall sum of the cycle comes negative. Distance of any node from itself is always zero. If distance of any node from itself is negative then it becomes negative cycle. To detect negative cycle,after applying **Floyd-Warshall** we just have to check the nodes distance from itself and if it comes out to be negative, we will detect the negative cycle.

Important

- **If there is a negative edge in undirected graph then no algorithm will find the shortest distance.**
- **If there is a negative edge in directed graph and there is no negative cycle then Floyd-Warshall & bellman-ford algorithm will find the minimum distance.**