**PRACTICAL – 6A**

**AIM:** Implement Dijkstra’s Algorithm.

**TOOLS USED:** Sublime Text

**THEORY:** Dijkstra’s algorithm is used to find the shortest distance to all the nodes from the source node. Here, we cannot have negative weights and negative cycles.

**ALGORITHM:**

Dijkstra's Algorithm (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. S←∅

3. Q←V [G]

4. while Q ≠ ∅

5. do u ← EXTRACT - MIN (Q)

6. S ← S ∪ {u}

7. for each vertex v ∈ Adj [u]

8. do RELAX (u, v, w)

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

#define pb push\_back

using namespace std;

void dijkstra(int src, int V, vector<pair<int,int>>adj[])

{

vector<int>distance(V+1,INT\_MAX);

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;

pq.push({0,src});

distance[src] = 0;

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first;

int nextdist = it.second;

if(distance[next] > distance[prev] + nextdist)

{

distance[next] = distance[prev] + nextdist;

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

}

}

}

//printing the distance array

cout << "Vectex" << " " << "distance for source" << "\n";

for(int i=1; i<=V; i++)

{

cout << i << " " << distance[i] << "\n";

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

ios\_base::sync\_with\_stdio(0);

cin.tie(0); cout.tie(0);

int tc = 1;

for (int t = 1; t <= tc; t++)

{

int V,E;

cin>>V>>E;

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

for(int i=1; i<=E; i++)

{

int u,v,wt;

cin>>u>>v>>wt;

adj[u].pb({v,wt});

//adj[v].pb({u,wt});

}

dijkstra(1,V,adj);

}

}

**Output:**

**Text

Description automatically generated**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

Dijkstra algorithm is implemented using binary heap as a priority queue to implement the Extract-Min function. Since the time complexity of min-heap is O(logV) and the number of edges is E, therefore the time complexity of the algorithm is O(ElogV).

**RESULT:**

Time Complexity for Dijkstra’s Algorithm problem using greedy approach is O(ElogV).

**PRACTICAL – 6B**

**AIM:** Implement Bellman Ford Algorithm.

**TOOLS USED:** Dev-C++

**THEORY:** Bellman Ford algorithm is used to find the shortest path from the source nodes to all the nodes. Here, the edges can have negative weights, but there cannot be a negative cycle in the graph.

**ALGORITHM:**

BELLMAN -FORD (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s)

2. for i ← 1 to |V[G]| - 1

3. do for each edge (u, v) ∈ E [G]

4. do RELAX (u, v, w)

5. for each edge (u, v) ∈ E [G]

6. do if d [v] > d [u] + w (u, v)

7. then return FALSE.

8. return TRUE.

**PROGRAM:**

**Code:**

#include <bits/stdc++.h>

#define pb push\_back

using namespace std;

void bellmanFord(int src, int V, vector<pair<int,int>>adj[])

{

vector<int>distance(V+1,INT\_MAX);

priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>pq;

pq.push({0,src});

distance[src] = 0;

for(int i = 1; i<V; i++) {

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first; //that node

int nextdist = it.second; //its weight

if(distance[next] > distance[prev] + nextdist)

{

distance[next] = distance[prev] + nextdist;

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

}

}

}

}

while(!pq.empty())

{

int dist = pq.top().first;

int prev = pq.top().second;

pq.pop();

for(auto it : adj[prev])

{

int next = it.first; //that node

int nextdist = it.second; //its weight

if(distance[next] > distance[prev] + nextdist)

{

cout<<"There is a negative cycle";

exit;

}

}

}

//printing the distance array

cout << "Vectex" << " " << "distance from source" << "\n";

for(int i=1; i<=V; i++)

{

cout << i << " " << distance[i] << "\n";

}

}

int main() {

#ifndef ONLINE\_JUDGE

freopen("input.txt","r",stdin);

freopen("output.txt","w",stdout);

#endif

int tc = 1;

for (int t = 1; t <= tc; t++)

{

int V,E;

cin>>V>>E;

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

for(int i=1; i<=E; i++)

{

int u,v,wt;

cin>>u>>v>>wt;

adj[u].pb({v,wt});

}

bellmanFord(1,V,adj);

}

}

**Output:**

**A screenshot of a computer

Description automatically generated with medium confidence**

**COMPLEXITY ANALYSIS OF ALGORITHM:**

BELLMAN -FORD (G, w, s)

1. INITIALIZE - SINGLE - SOURCE (G, s) ---------- O(V)

2. for i ← 1 to |V[G]| - 1

3. do for each edge (u, v) ∈ E [G]

4. do RELAX (u, v, w) ---------- O(EV)

5. for each edge (u, v) ∈ E [G]

6. do if d [v] > d [u] + w (u, v) ---------- O(E)

7. then return FALSE.

8. return TRUE.

Therefore, the overall time complexity of Bellman Ford Algorithm is O(EV).

**RESULT:**

Time Complexity for Bellman Ford Algorithm = O(EV).

**COMPARISON TABLE:**

|  |  |
| --- | --- |
| Dijkstra Algorithm | Bellman Ford Algorithm |
| Dijkstra’s Algorithm doesn’t work when there is negative weight edge. | Bellman Ford’s Algorithm works when there is negative weight edge, it also detects the negative weight cycle. |
| It is less time consuming. Its complexity is O(ElogV). | It is more time consuming than Dijkstra. Its complexity is O(EV). |
| Greedy approach is taken to implement the algorithm. | Dynamic Programming approach is taken to implement the algorithm. |