OCTOBER 27,2020

DESIGN AND ANALYSIS OF ALGORITHMS

(QUESTION: 8.1, BELLMAN-FORD)
EXERCISE 8

SUBMITTED BY-
NAME - RAJ KRISHNA
ROLL NO - 1805233
BATCH - CSE-G1
GROUP - B

1. The objective of the Experiment

The objective of the experiment is to find the shortest path from a vertex to all other vertices of a weighted graph via **Bellman-Ford**.

2. Solution Code

```
#include<iostream>
#define MAX 28
using namespace std;
typedef struct edge
{
  int src;
  int dest;
  int wt;
}edge;
void path(int parent[], int j)
{

        if (parent[j] == - 1)
        return;

        path(parent, parent[j]);

        cout<<" - "<<j;

}
void bellman_ford(int nv,edge e[],int src_graph,int ne)
{
  int u,v,weight,i,j=0;
  int dis[MAX];
  int parent[10];
  for(i=0;i<nv;i++)
  {
        dis[i]=999;
```

```
}
dis[src_graph]=0;
for(i=0;i<nv-1;i++)
{
	for(j=0;j<ne;j++)
	{
	u=e[j].src;
	v=e[j].dest;
	parent[v];
	weight=e[j].wt;

	if(dis[u]!=999 && dis[u]+weight < dis[v])
	{
	dis[v]=dis[u]+weight;
	parent[v]=u;
	}
	}
}
for(j=0;j<ne;j++)
{
	u=e[j].src;
	v=e[j].dest;
	weight=e[j].wt;

	if(dis[u]+weight < dis[v])
	{
	cout<<"\nNegative Cycle Present! \n";
	return;
	}
}
	for (int i = 0; i < v; i++)
	parent[0] = -1;
cout<<"\nVertex "<<"Distance "<<"Path ";
```

```
  for(i=0;i<nv;i++)
  {
        cout<<"\n"<<i<<"\t"<<dis[i]<<"\t"<<0;
        path(parent,i);
        cout<<endl;
  }
}
int main()
{
  int nv=9,ne=28,src_graph=0;
  edge e[MAX];
  cout<<"Enter Src,Desc,Wt-"<<endl;
  for(int i=0;i<ne;i++)
  {
        cin>>e[i].src >>e[i].dest >>e[i].wt;
  }
  bellman_ford(nv,e,src_graph,ne);
  return 0;
}
```

3. Summary of the program

The Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph. It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights. Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point. Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

Bellman Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes

those estimates by finding new paths that are shorter than the previously overestimated paths.
By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

Step-1: Start with the weighted graph.
Step-2: Choose a vertex and assign infinity values to all other vertices.
Step-3: Visit each edge and relax the path distance if they are inaccurate.
Step-4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times.
Step-5: Notice how the vertex at the top right corner had its path length adjusted.
Step-6: After all the vertices have their path lengths, we check if a negative cycle is present.

Time Complexity

Best Case	O(E)
Average Case	O(V)
Worst Case	O(V)

Space Complexity

Space complexity is O(V).

4. Sample Output

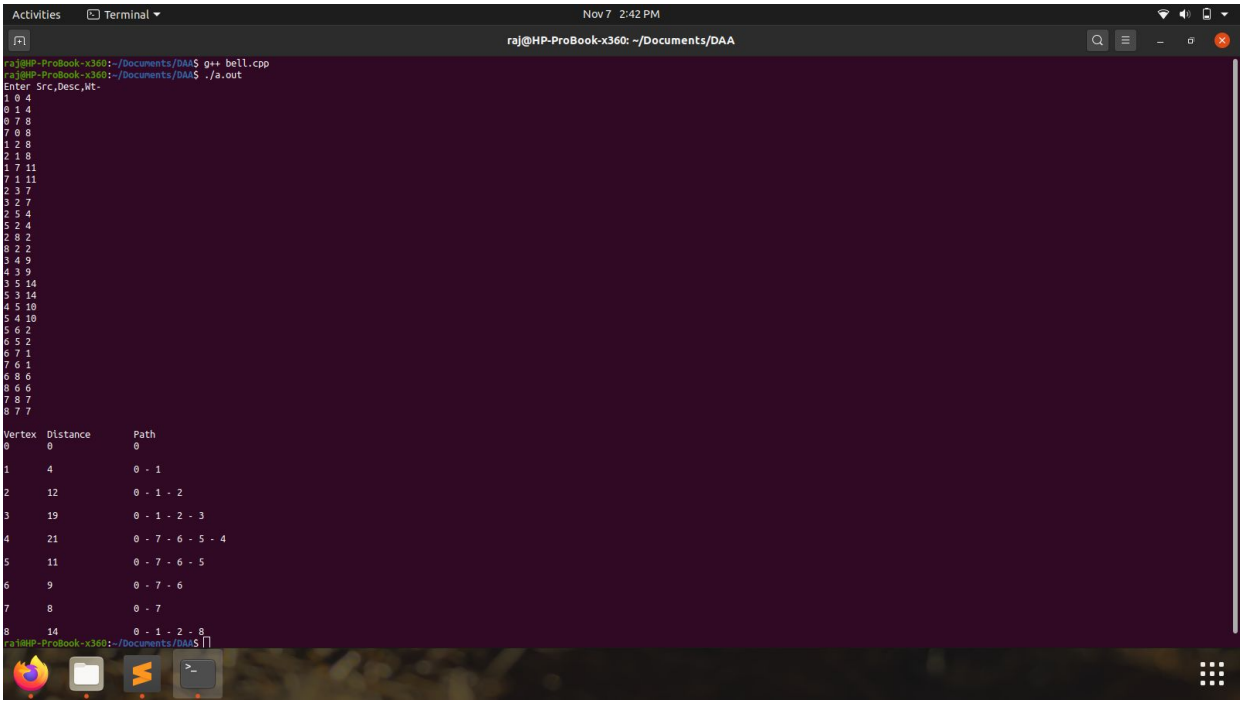
Input

```
1 0 4
0 1 4
0 7 8
7 0 8
1 2 8
2 1 8
1 7 11
7 1 11
```

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

Output

Vertex	Distance	Path
0	0	0
1	4	0 - 1
2	12	0 - 1 - 2
3	19	0 - 1 - 2 - 3
4	21	0 - 7 - 6 - 5 - 4
5	11	0 - 7 - 6 - 5
6	9	0 - 7 - 6
7	8	0 - 7
8	14	0 - 1 - 2 - 8


Activities
Terminal
Nov 7 2:42 PM
raj@HP-ProBook-x360: ~/Documents/DAA
raj@HP-ProBook-x360:~/Documents/DAA$ g++ bell.cpp
raj@HP-ProBook-x360:~/Documents/DAA$ ./a.out
Enter Src,Desc,Wt-
1 0 4
0 1 4
0 7 8
7 0 8
1 2 8
2 1 8
1 7 11
7 1 11
2 3 7
3 2 7
2 5 4
5 2 4
2 8 2
8 2 2
3 4 9
4 3 9
3 5 14
5 3 14
4 5 10
5 4 10
5 6 2
6 5 2
6 7 1
7 6 1
6 8 6
8 6 6
7 8 7
8 7 7
Vertex Distance Path
0 0 0
1 4 0 - 1
2 12 0 - 1 - 2
3 19 0 - 1 - 2 - 3
4 21 0 - 7 - 6 - 5 - 4
5 11 0 - 7 - 6 - 5
6 9 0 - 7 - 6
7 8 0 - 7
8 14 0 - 1 - 2 - 8
raj@HP-ProBook-x360:~/Documents/DAA$

NOVEMBER 04,2020

DESIGN AND ANALYSIS OF ALGORITHMS

(QUESTION: 8.2, DIJKSTRA)
EXERCISE 8

SUBMITTED BY-
NAME - RAJ KRISHNA
ROLL NO - 1805233
BATCH - CSE-G1
GROUP - B

1. The Objective of experiment

The objective of the experiment is to find the shortest path between any two vertices of a graph via **Dijkstra**.

2. Solution Code

```
#include<bits/stdc++.h>
using namespace std;
# define INF 0x3f3f3f3f
typedef pair<int, int> iPair;
class Graph
{
    int V;

    list< pair<int, int> > *adj;

public:
    Graph(int V);
    void addEdge(int u, int v, int w);
    void shortestPath(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
}

void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}
void path(int parent[], int j)
```

```
{

        if (parent[j] == - 1)
        return;
        path(parent, parent[j]);
        cout<<"  "<<j;
}

void Graph::shortestPath(int src)
{ int parent[V];
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;
    vector<int> dist(V, INF);
    pq.push(make_pair(0, src));
    dist[src] = 0;
    while (!pq.empty())
    {
         int u = pq.top().second;
         pq.pop();
         list< pair<int, int> >::iterator i;
         for (i = adj[u].begin(); i != adj[u].end(); ++i)
         {
                 int v = (*i).first;
                 int weight = (*i).second;
                 if (dist[v] > dist[u] + weight)
                 {  parent[v]=u;
                         // Updating distance of v
                         dist[v] = dist[u] + weight;
                         pq.push(make_pair(dist[v], v));

                 }
         }
    }
   for (int i = 0; i < V; i++)
```

```
        parent[0] = -1;
    cout<<"Dijkstra, all shortest paths from 0"<<endl;
    cout<<"Vertex\t\t Distance\tPaths"<<endl;
    for (int i = 0; i < V; ++i) {
        cout<< i<<"\t\t"<<dist[i]<<"\t\t"<<0;
        path(parent,i);
        cout<<endl;
    }
}
int main()
{
    int V = 9;
    Graph g(V);

    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 5, 14);
    g.addEdge(4, 5, 10);
    g.addEdge(5, 6, 2);
    g.addEdge(6, 7, 1);
    g.addEdge(6, 8, 6);
    g.addEdge(7, 8, 7);
    g.shortestPath(0);
    return 0;
}
```

3. Summary of Program

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.
Dijkstra's Algorithm works on the basis that any subpath B -> D of the shortest path A -> D between vertices A and D is also the shortest path between vertices B and D.
Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Step-1: Start with a weighted graph.
Step-2: Choose a starting vertex and assign infinity path values to all other devices.
Step-3: Go to each vertex and update its path length.
Step-4: If the path length of the adjacent vertex is lesser than new path length, don't update it.
Step-5: Avoid updating path lengths of already visited vertices.
Step-6: After each iteration, we pick the unvisited vertex with the least path length.
Step-7: Notice how the rightmost vertex has its path length updated twice.
Step-8: Repeat until all the vertices have been visited.

Time Complexity: O(E Log V) where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

4. Sample Output

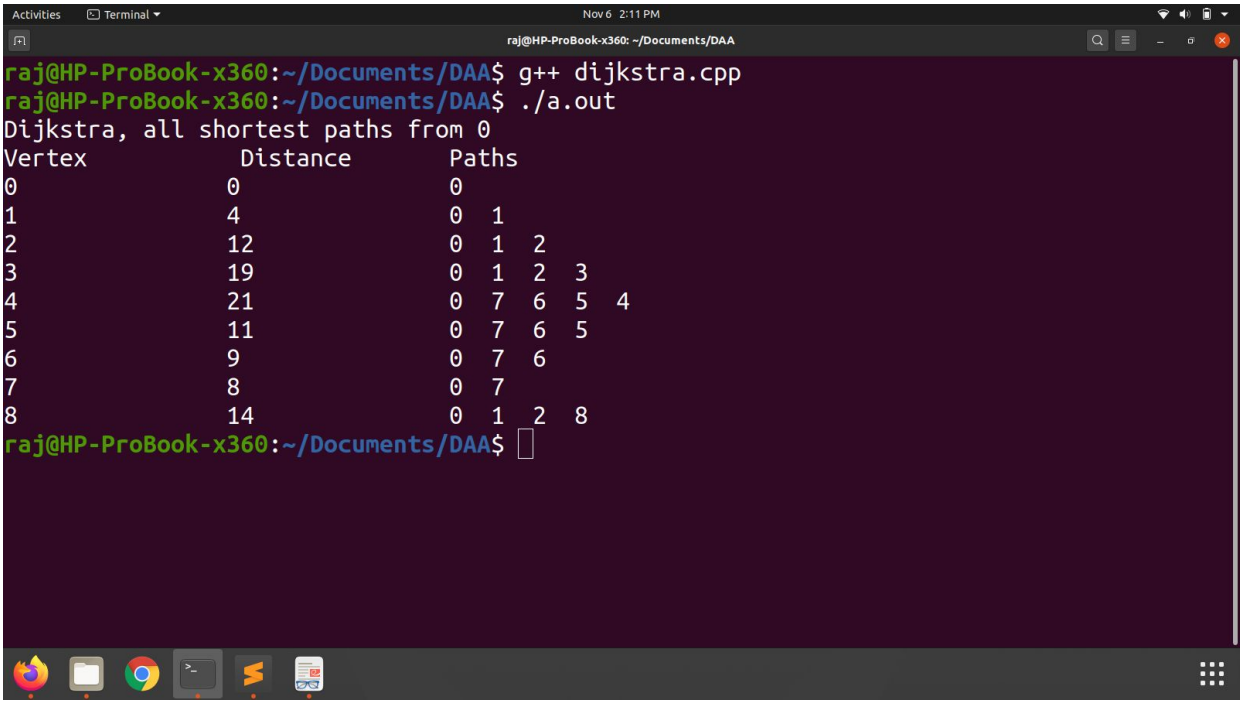
Input-

```
g.addEdge(0, 1, 4);
g.addEdge(0, 7, 8);
g.addEdge(1, 2, 8);
g.addEdge(1, 7, 11);
g.addEdge(2, 3, 7);
g.addEdge(2, 8, 2);
g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);
```

Output-

Dijkstra, all shortest paths from 0

Vertex	Distance	Paths
0	0	0
1	4	0 1
2	12	0 1 2
3	19	0 1 2 3
4	21	0 7 6 5 4
5	11	0 7 6 5
6	9	0 7 6
7	8	0 7
8	14	0 1 2 8


Activities
Terminal
Nov 6 2:11 PM
raj@HP-ProBook-x360: ~/Documents/DAA
raj@HP-ProBook-x360:~/Documents/DAA$ g++ dijkstra.cpp
raj@HP-ProBook-x360:~/Documents/DAA$ ./a.out
Dijkstra, all shortest paths from 0
Vertex Distance Paths
0 0 0
1 4 0 1
2 12 0 1 2
3 19 0 1 2 3
4 21 0 7 6 5 4
5 11 0 7 6 5
6 9 0 7 6
7 8 0 7
8 14 0 1 2 8
raj@HP-ProBook-x360:~/Documents/DAA$