Department of
Computer Science and Engineering

Title: Finding Shortest Path using Dijkstra's Algorithm

Algorithms Lab
CSE 206

Submitted by
Shamim Ahmed
ID:201902067

Green University of Bangladesh

Objective(s)

- To understand the shortest path of a graph.

• To implement Dijkstra’s algorithm for finding the shortest path of a graph and analyse it.

Problem analysis

Given a graph and a source vertex in the graph, we have to find shortest paths from source to all vertices in the given graph. Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source. Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

1. Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While *sptSet* does not include all vertices
   - Pick a vertex *u* which is not there in *sptSet* and has the minimum distance value among all nodes in the set.
   - Include *u* to *sptSet*.
   - Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices each is denoted as *v*. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge u-v, is less than the distance value of *v*, then update the distance value of *v*. This step is known as *Relaxation*.

Time Complexity

Let *N* and *E* be the numbers of vertices and edges respectively. The *while* loop in line 6 of the algorithm will run *N* times, giving O(N). Extracting the minimum distance node in line 7 gives O(log N). The *for* loop running in line 9 and for the update of distance value gives O(E log N). So total running time is O(N log N + E log N) = O(E log N).

Algorithm

```
Algorithm 1: Dijkstra (Graph G, all weights w, source vertex s)
  Input: Directed or undirected weighted graph
  /* Dijkstra's algorithm for single source shortest path */ 1 Initialize sptSet[] as empty
2 for each vertex of G do
3 distance, d[v] = infinite
4 end
5 d[s] = 0
6 while sptSet[] does not include all vertex from G do
7 select u not in sptSet[] and has minimum distance from source s
8 insert u into sptSet[]
9 for each vertex v as adjacent of u do
10 if current distance, d[v] > distance, d[u] + weight, w(u,v) then
11 d[v] = d[u] + weight, w(u,v)
12 end
13 end
14 end
```

Implementation in C++

```
#include <bits/stdc++.h>



using namespace std;



vector<int> dijkstra(vector<vector<pair<int,int>>> graph, int start)

{

    vector<int> dist(graph.size(), INT_MAX);



    priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;



    pq.push(make_pair(0, start));

    dist[start] = 0;



    while(!pq.empty())

    {

        int u = pq.top().second;

        pq.pop();
```

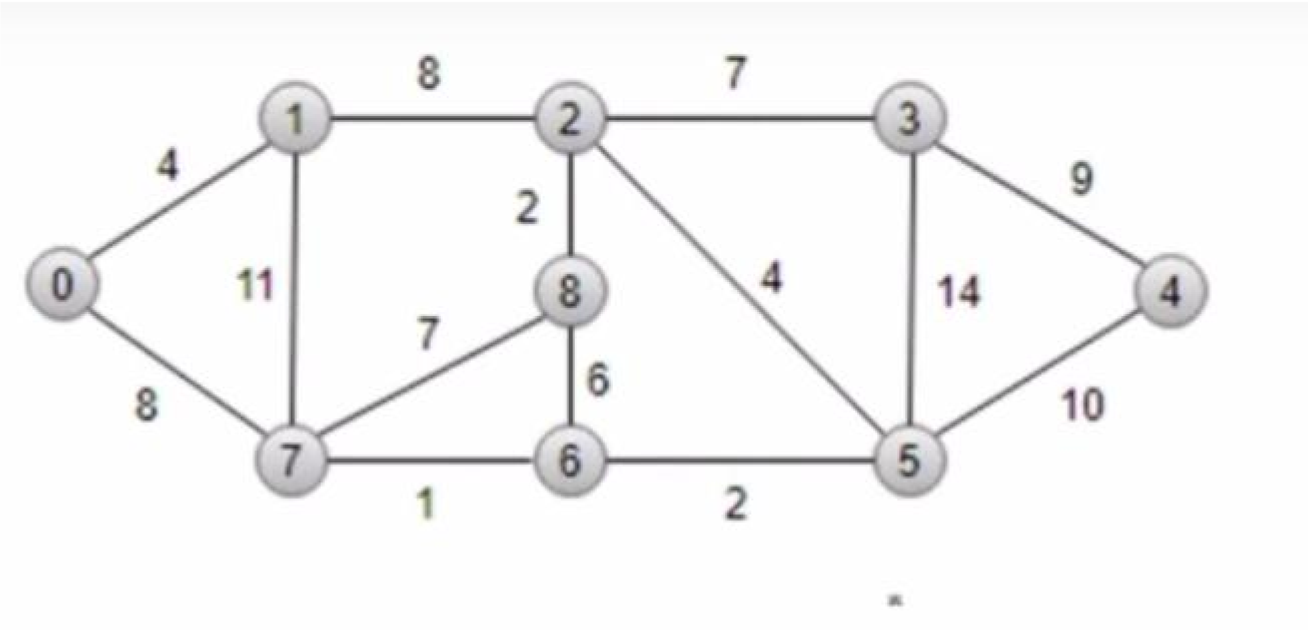
```
        for(int i = 0; i < graph[u].size(); i++)
        {
            int v = graph[u][i].first;
            int weight = graph[u][i].second;

            if (dist[v] > dist[u] + weight)
            {
                dist[v] = dist[u] + weight;
                pq.push(make_pair(dist[v], v));
            }
        }
    }
    return dist;
}

void addEdge(vector<vector<pair<int,int>>>& graph, int u, int v, int w)
{
    graph[u].push_back(make_pair(v, w));
```

```
    graph[v].push_back(make_pair(u, w));
}

int main()
{

    vector<vector<pair<int,int>>> graph(9, vector<pair<int,int>>(9));

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

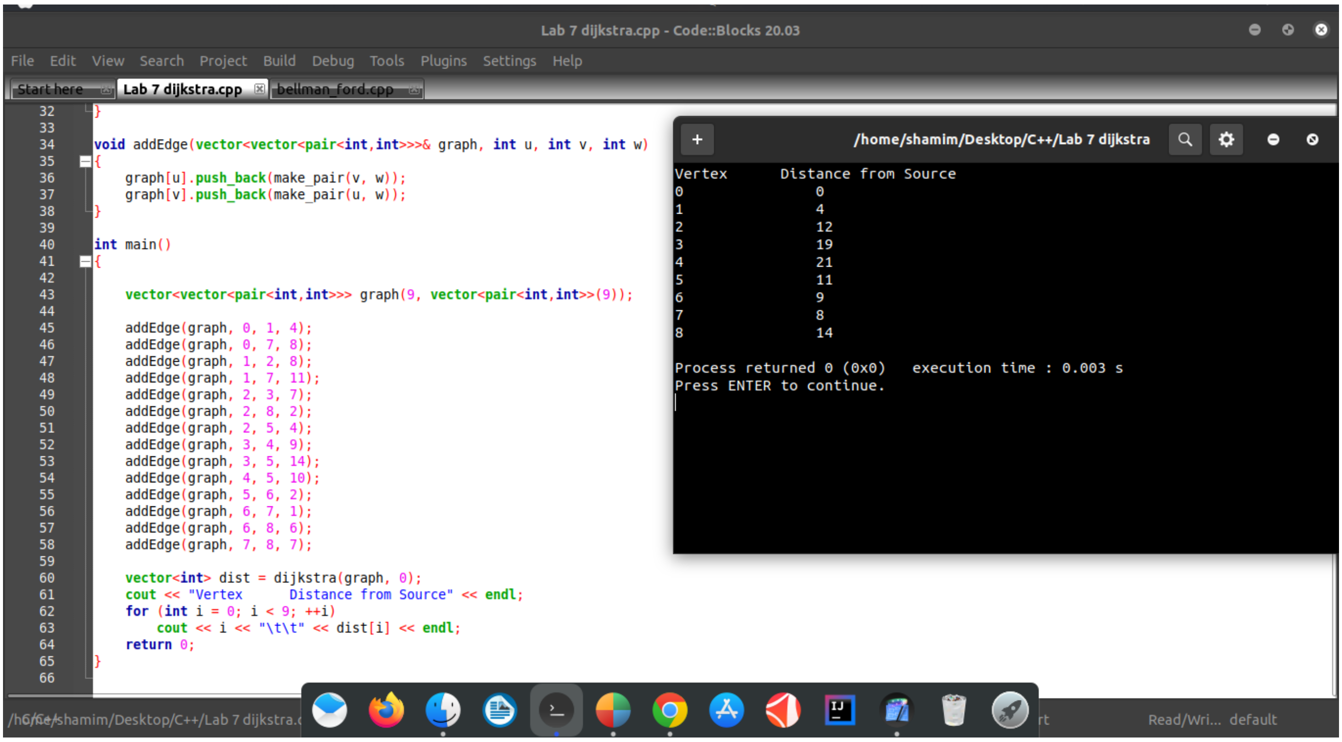
```
    addEdge(graph, 6, 7, 1);

    addEdge(graph, 6, 8, 6);

    addEdge(graph, 7, 8, 7);


    vector<int> dist = dijkstra(graph, 0);

    cout << "Vertex      Distance from Source" << endl;

    for (int i = 0; i < 9; ++i)

        cout << i << "\t\t" << dist[i] << endl;

    return 0;

}
```

Sample Input/Output (Compilation, Debugging & Testing)

My graph



Output:



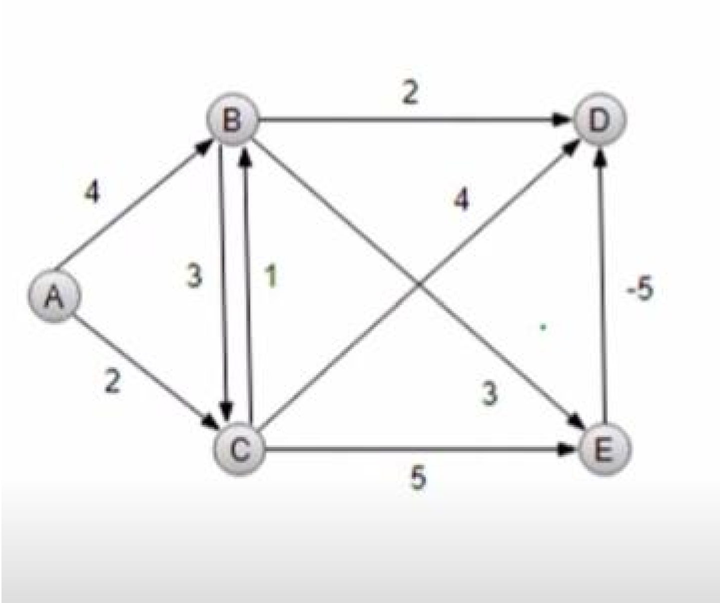
Lab Task

Bellman Ford in C++

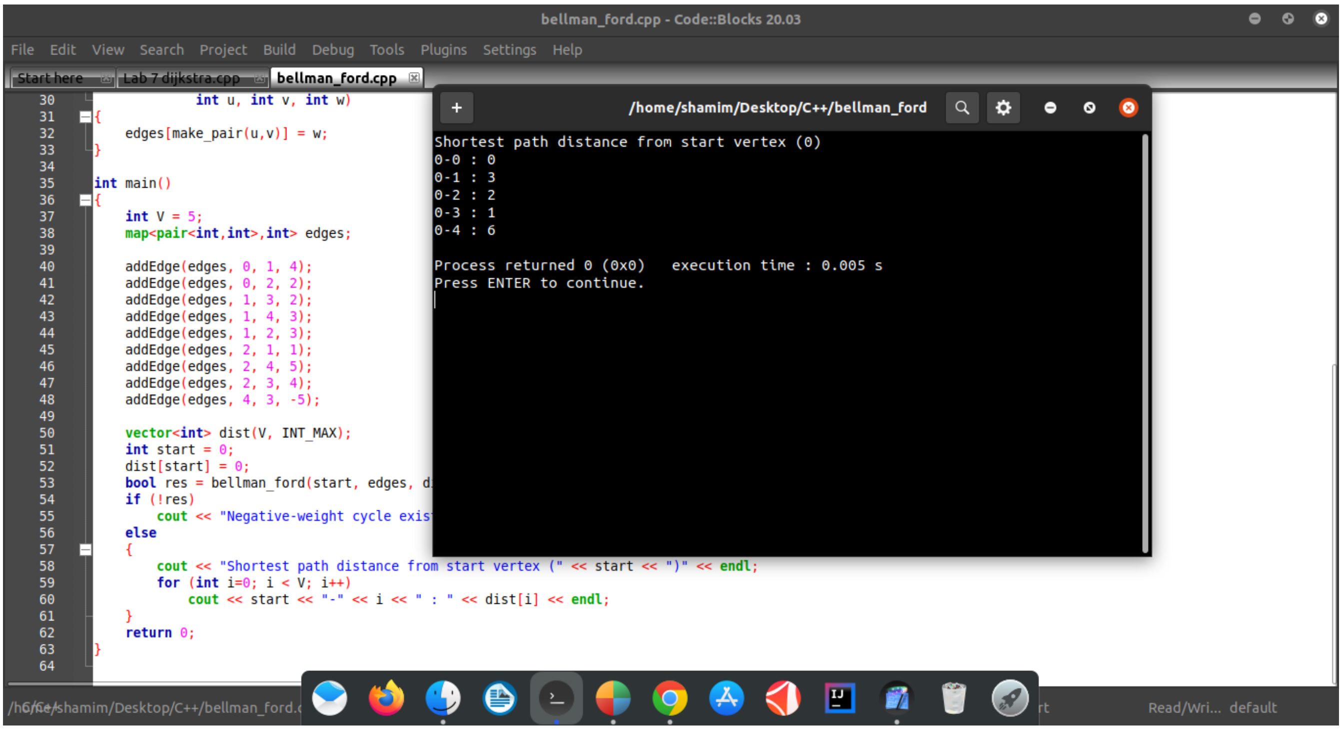
```
#include <bits/stdc++.h>

using namespace std;

bool bellman_ford(int start, map<pair<int,int>,int>& edges,vector<int>& dist,int V)
{
    for (int i=0; i < V; i++)
    {
        for (auto e : edges)
        {
            int u = e.first.first;
            int v = e.first.second;
            int weight = e.second;
            if (dist[u] != INT_MAX && (dist[v] > dist[u] + weight))
              dist[v] = dist[u] + weight;
        }
    }
    for (auto e : edges)
    {
        int u = e.first.first;
        int v = e.first.second;
        int weight = e.second;
        if (dist[u] != INT_MAX && (dist[v] > dist[u] + weight))
            return false;
    }
    return true;
}

void addEdge(map<pair<int,int>,int>& edges,
             int u, int v, int w)
{
    edges[make_pair(u,v)] = w;
```

```
}

int main()
{
    int V = 5;
    map<pair<int,int>,int> edges;

    addEdge(edges, 0, 1, 4);
    addEdge(edges, 0, 2, 2);
    addEdge(edges, 1, 3, 2);
    addEdge(edges, 1, 4, 3);
    addEdge(edges, 1, 2, 3);
    addEdge(edges, 2, 1, 1);
    addEdge(edges, 2, 4, 5);
    addEdge(edges, 2, 3, 4);
    addEdge(edges, 4, 3, -5);

    vector<int> dist(V, INT_MAX);
    int start = 0;
    dist[start] = 0;
    bool res = bellman_ford(start, edges, dist, V);
    if (!res)
        cout << "Negative-weight cycle exists" << endl;
    else
    {
        cout << "Shortest path distance from start vertex (" << start << ")" << endl;
        for (int i=0; i < V; i++)
            cout << start << "-" << i << " : " << dist[i] << endl;
    }
    return 0;
}
```

My graph



Output:



Discussion & Conclusion

Dijkstra's algorithm use for find the shortest path from source node to all other node. Time Complexity of Dijkstra's Algorithm is O (V 2) but with min-priority queue it drops down to O (V + E l o g V) . But Dijkstra's algorithm does not work for negative weight . to handle that we used Bellman Ford Algorithm .