Adjacenecy List

*#include*<iostream>

using *namespace* std;

*class* Graph

{

*public:*

*int* V;

*int* E;

*int* \*\*Adj;

};

Graph*\** Matrix()

{

*int* Vertices,Edges;

    cout<<"Enter number of Vertices"<<endl;

    cin>>Vertices;

    cout<<"Enter number of Edges"<<endl;

    cin>>Edges;

    Graph\* G= new Graph;

    G->V = Vertices;

    G->E = Edges;

    G->Adj = new *int*\*[Vertices];

*for*(*int* i=0;i<Vertices;i++)

        G->Adj[i]=new *int*[Vertices];

*for*(*int* i=0;i<G->V;i++)

    {

*for*(*int* j=0;j<G->V;j++)

        {

*if*(i==j)

                G->Adj[i][j]=1;

*else*

                G->Adj[i][j]=0;

        }

    }

    cout<<"Enter the edges"<<endl;

*int* u,v;

*for*(*int* i=0;i<G->E;i++)

    {

        cin>>u>>v;

        cout<<endl;

        G->Adj[u][v]=1;

        G->Adj[v][u]=1;

    }

*return* G;

}

*void* print(Graph*\** G)

{

*for*(*int* i=0;i<G->V;i++)

    {

*for*(*int* j=0;j<G->V;j++)

        {

            cout<<G->Adj[i][j]<<" ";

        }

        cout<<endl;

    }

}

*int* main()

{

    Graph\* object = Matrix();

    print(object);

}

Adjacency Matrix

*#include*<iostream>

using *namespace* std;

*class* Graph

{

*public:*

*int* V;

*int* E;

*int* \*\*Adj;

};

Graph*\** Matrix()

{

*int* Vertices,Edges;

    cout<<"Enter number of Vertices"<<endl;

    cin>>Vertices;

    cout<<"Enter number of Edges"<<endl;

    cin>>Edges;

    Graph\* G= new Graph;

    G->V = Vertices;

    G->E = Edges;

    G->Adj = new *int*\*[Vertices];

*for*(*int* i=0;i<Vertices;i++)

        G->Adj[i]=new *int*[Vertices];

*for*(*int* i=0;i<G->V;i++)

    {

*for*(*int* j=0;j<G->V;j++)

        {

*if*(i==j)

                G->Adj[i][j]=1;

*else*

                G->Adj[i][j]=0;

        }

    }

*int* u,v;

*for*(*int* i=0;i<G->E;i++)

    {

        cout<<"Enter the edges"<<endl;

        cin>>u>>v;

        cout<<endl;

        G->Adj[u][v]=1;

        G->Adj[v][u]=1;

    }

*return* G;

}

*void* print(Graph*\** G)

{

*for*(*int* i=0;i<G->V;i++)

    {

*for*(*int* j=0;j<G->V;j++)

        {

            cout<<G->Adj[i][j]<<" ";

        }

        cout<<endl;

    }

}

*int* main()

{

    Graph\* object = Matrix();

    print(object);

}

Relaxation

The single - source shortest paths are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight. For each vertex v ∈ V, we maintain an attribute d [v], which is an upper bound on the weight of the shortest path from source s to v. We call d [v] the **shortest path estimate**.

Bellman’s Algorithm

Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.   
We have discussed [Dijkstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O((V+E)LogV) (with the use of Fibonacci heap). *Dijkstra doesn’t work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

For directed Graph

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

using *namespace* std;

*class* edge

{

*public:*

*int* src;

*int* dest;

*int* weight;

};

*class* graph

{

*public:*

*int* V;

*int* E;

    edge\* edges;

};

graph*\** CreateGraph(*int* edges, *int* vertices)

{

    graph\* g = new graph;

    g->E = edges;

    g->V = vertices;

    g->edges = new edge[edges];

*return* g;

}

*void* bellman(graph*\** g, *int* source)

{

*bool* answer=true;

*int* V = g->V;

    vector<*int*> distance(1);

    distance[V];

*for*(*int* i=0;i<V;i++)

        distance[i]=INT\_MAX;

    distance[source]=0;

*for*(*int* i=0;i<V-1;++i)

    {

*for*(*int* j=0;j<g->E;++j)

        {

*int* u=g->edges[j].src;

*int* v=g->edges[j].dest;

*int* wht=g->edges[j].weight;

*if*(distance[j]!=INT\_MAX && distance[u]+wht<distance[v])

                distance[v]=distance[u]+wht;

        }

    }

*for*(*int* j=0;j<g->E;++j)

        {

*int* u=g->edges[j].src;

*int* v=g->edges[j].dest;

*int* wht=g->edges[j].weight;

*if*(distance[j]!=INT\_MAX && distance[u]+wht<distance[v])

                answer=false;

        }

*if*(answer)

        {

            cout<<"Distance of source "<< source<<" from "<<endl;

*for*(*int* i=0;i<g->V;++i)

            {

                cout<<i<<" is "<<distance[i]<<endl;

            }

        }

*else*

        {

            cout<<"Negative cycle";

        }

}

*int* main()

{

*int* vertices, edges;

    cout<<"Enter the number of vertices"<<endl;

    cin>>vertices;

    cout<<"Enter the number of edges"<<endl;

    cin>>edges;

    graph\* g = CreateGraph(edges,vertices);

*int* source, destination, weight;

*for*(*int* i=0;i<edges;i++)

    {

        cout<<"source destination and weight"<<endl;

        cin>>source>>destination>>weight;

        g->edges[i].src=source;

        g->edges[i].dest = destination;

        g->edges[i].weight=weight;

    }

    bellman(g,0);

*return* 0;

}

Dijkstras

Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the nodes.

Time Complexity of the implementation is O(V^2). If the input [graph is represented using adjacency list](https://www.geeksforgeeks.org/graph-and-its-representations/), it can be reduced to O(E log V) with the help of a binary heap

Same Code can be used for directed graph.

Not used for graphs with negative weight cycles.

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

using *namespace* std;

*#define* V 5

*int* minDistance(*int* MinimumPath[], *bool* Visited[])

{

*int* min=INT\_MAX, min\_index;

*for*(*int* v=0; v<V;v++)

    {

*if*(Visited[v]==false && MinimumPath[v]<=min)

        {

            min = MinimumPath[v];

            min\_index=v;

        }

    }

*return* min\_index;

}

*void* dijktras(*int* graph[][V],*int* src)

{

*int* MinimumPath[V];

*bool* Visited[V];

*for*(*int* i=0;i<V;++i)

    {

        MinimumPath[i]=INT\_MAX;

        Visited[i]=false;

    }

    MinimumPath[src]=0;

*for*(*int* count=0;count<V-1;++count)

    {

*int* u=minDistance(MinimumPath,Visited);

        Visited[u]=true;

*for*(*int* v=0;v<V;++v)

        {

*if*(!Visited[v] && graph[u][v] && MinimumPath[u]!=INT\_MAX && (MinimumPath[u]+graph[u][v])<MinimumPath[v])

                {

                    MinimumPath[v]=MinimumPath[u]+graph[u][v];

                }

        }

    }

    cout<<"Distance of source "<<src<<" from"<<endl;

*for*(*int* i=0;i<V;i++)

    {

        cout<<i<<" is "<<MinimumPath[i]<<endl;

    }

}

*int* main()

{

*int* graph[V][V]={{0,2,0,0,5},

                    {2,0,1,2,10},

                    {0,1,0,6,3},

                    {0,2,6,0,4},

                    {5,10,3,4,0}};

    dijktras(graph,2);

}

[Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm)

The [Floyd Warshall Algorithm](http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Time Complexity: O(V3)

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

*#define* V 4

*#define* INF 99999999

using *namespace* std;

*void* Floyd(*int* graph[][V])

{

*int* distance[V][V];

*for* (*int* i = 0; i < V; ++i)

    {

*for* (*int* j = 0; j < V; ++j)

        {

            distance[i][j] = graph[i][j];

        }

    }

*for* (*int* intermidiates = 0; intermidiates < V; ++intermidiates)

    {

*for* (*int* source = 0; source < V; ++source)

        {

*for* (*int* vertex = 0; vertex < V; ++vertex)

            {

*if* ((distance[source][intermidiates] + distance[intermidiates][vertex]) < distance[source][vertex])

                {

                    distance[source][vertex] = distance[source][intermidiates] + distance[intermidiates][vertex];

                }

            }

        }

    }

*for* (*int* i = 0; i < V; i++)

    {

*for* (*int* j = 0; j < V; j++)

        {

            cout << distance[i][j] << " ";

        }

        cout << endl;

    }

}

*int* main()

{

*int* graph[V][V] = {{0, 5, INF, 10},

                       {INF, 0, 3, INF},

                       {INF, INF, 0, 1},

                       {INF, INF, INF, 0}};

    Floyd(graph);

}

Prims Algorithm

**Spanning tree -** A spanning tree is the subgraph of an undirected connected graph.

**Minimum Spanning tree -** Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

**Prim's Algorithm** is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Working:

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

* First, we have to initialize an MST with the randomly chosen vertex.
* Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
* Repeat step 2 until the minimum spanning tree is formed.
* *#include* <iostream>
* *#include* <bits/stdc++.h>
* *#define* V 5
* *#define* INF 999999
* using *namespace* std;
* *int* minDistance(*int* distances[], *bool* visited[])
* {
* *int* min = INT\_MAX;
* *int* min\_index;
* *for* (*int* i = 0; i < V; i++)
* {
* *if* (distances[i] < min && visited[i] == false)
* {
* min = distances[i];
* min\_index = i;
* }
* }
* *return* min\_index;
* }
* *void* prims(*int* graph[][V])
* {
* *bool* visited[V];
* *int* distances[V];
* *int* parent[V];
* *for* (*int* i = 0; i < V; i++)
* {
* visited[i] = false;
* distances[i] = INF;
* }
* distances[0] = 0;
* parent[0] = -1;
* *for* (*int* i = 0; i < V - 1; ++i) *//Vertices*
* {
* *int* u = minDistance(distances, visited);
* visited[u] = true;
* *for* (*int* j = 0; j < V; ++j)
* {
* *if* (graph[u][j] && graph[u][j] != INF && visited[j] == false && graph[u][j] < distances[j])
* {
* distances[j] = graph[u][j];
* parent[j] = u;
* }
* }
* }
* *for* (*int* i = 1; i < V; i++)
* {
* cout << parent[i] << " to " << i << " cost is " << distances[i] << endl;
* }
* }
* *int* main()
* {
* *int* graph[V][V] = {{0, 8, INF, INF, 5},
* {8, 0, INF, 10, 3},
* {INF, INF, 0, 8, 6},
* {INF, 10, 8, 0, 2},
* {5, 3, 6, 2, 0}};
* prims(graph);
* }
* **Time Complexity**

|  |  |
| --- | --- |
| **Data structure used for the minimum edge weight** | **Time Complexity** |
| Adjacency matrix, linear searching | O(|V|2) |
| Adjacency list and binary heap | O(|E| log |V|) |
| Adjacency list and Fibonacci heap | O(|E|+ |V| log |V|) |

Kruskals Algorithm

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

* First, sort all the edges from low weight to high.
* Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
* Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

The applications of Kruskal's algorithm are -

* Kruskal's algorithm can be used to layout electrical wiring among cities.
* It can be used to lay down LAN connections

 TC: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most O(V2), so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

*#include* <iostream>

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

using *namespace* std;

*class* unionFind

{

*public:*

*int* \*parent;

*int* \*rank;

    unionFind(*int* n)

    {

        parent = new *int*[n];

        rank = new *int*[n];

*for* (*int* i = 0; i < n; i++)

        {

            parent[i] = i;

            rank[i] = 1;

        }

    }

*int* find(*int* i)

    {

*if* (parent[i] == i)

        {

*return* i;

        }

*return* parent[i] = find(parent[i]);

    }

*void* Union(*int* x, *int* y)

    {

*int* s1 = find(x);

*int* s2 = find(y);

*if* (s1 != s2)

        {

*if* (rank[s1] < rank[s2])

            {

                parent[s1] = s2;

                rank[s2] += rank[s1];

            }

*else*

            {

                parent[s2] = s1;

                rank[s1] += rank[s2];

            }

        }

    }

};

*class* graph

{

*public:*

*int* V;

    vector<vector<*int*>> edges;

    vector<pair<*int*, *int*>> srcDes;

    vector<*int*> distances;

    graph(*int* n)

    {

*this*->V = n;

    }

*void* addEdge(*int* src, *int* dest, *int* wht)

    {

        edges.push\_back({wht, src, dest});

    }

*void* kruskals()

    {

        sort(edges.begin(), edges.end());

        unionFind uf(V);

*for* (*int* i = 0; i < edges.size(); i++)

        {

*int* x = edges[i][1];

*int* y = edges[i][2];

*int* w = edges[i][0];

*if* (uf.find(x) != uf.find(y))

            {

                uf.Union(x, y);

                srcDes.push\_back(make\_pair(x, y));

                distances.push\_back(w);

*//cout << w << " ";*

            }

        }

    }

*void* printDistances()

    {

*for* (*int* i = 0; i < distances.size(); i++)

        {

            cout << srcDes[i].first << " to " << srcDes[i].second << " cost is " << distances[i] << endl;

        }

    }

};

*int* main()

{

    graph g(4);

    g.addEdge(0, 1, 1);

    g.addEdge(1, 3, 3);

    g.addEdge(3, 2, 4);

    g.addEdge(2, 0, 2);

    g.addEdge(0, 3, 2);

    g.addEdge(1, 2, 2);

    g.kruskals();

    g.printDistances();

*return* 0;

}