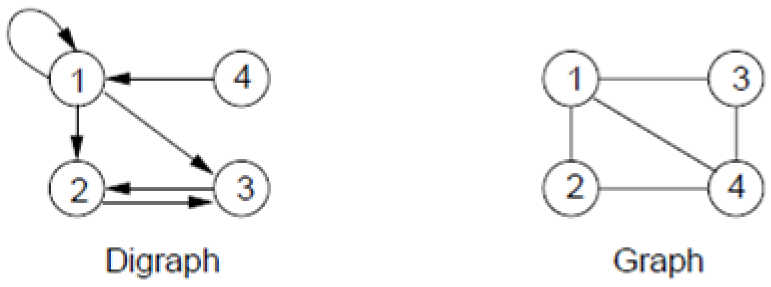
Chapter 6

Graph Algorithms

Graph is a collection of vertices or nodes, connected by a collection of edges. Graphs are extremely important because they are a very flexible mathematical model for many application problems. Basically, any time you have a set of objects, and there is some "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of graphs in application include communication and transportation networks, VLSI and other sorts of logic circuits, surface meshes used for shape description in computer-aided design and geographic information systems, precedence constraints in scheduling systems etc.

A directed graph (or digraph) G = (V,E) consists of a finite set V , called the vertices or nodes, and E, a set of ordered pairs, called the edges of G.

An undirected graph (or graph) G = (V,E) consists of a finite set V of vertices, and a set E of unordered pairs of distinct vertices, called the edges.



Digraph and graph example.

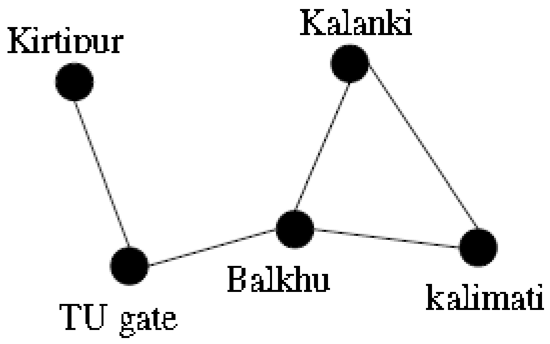
We say that vertex v is adjacent to vertex u if there is an edge (u; v). In a directed graph, given the edge e = (u; v), we say that u is the origin of e and v is the destination of e. In undirected graphs u and v are the endpoints of the edge. The edge e is incident (meaning that it touches) both u and v.

In a digraph, the number of edges coming out of a vertex is called the out-degree of that vertex, and the number of edges coming in is called the in-degree. In an undirected graph we just talk about the degree of a vertex as the number of incident edges. By the degree of a graph, we usually mean the maximum degree of its vertices.

Notice that generally the number of edges in a graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be sparse if $E = \Theta(V)$, and dense, otherwise. When giving the running times of algorithms, we will usually express it as a function of both V and E, so that the performance on sparse and dense graphs will be apparent.

Graph Representation

Graph is a pair G = (V,E) where V denotes a set of vertices and E denotes the set of edges connecting two vertices. Many natural problems can be explained using graph for example modeling road network, electronic circuits, etc. The example below shows the road network.



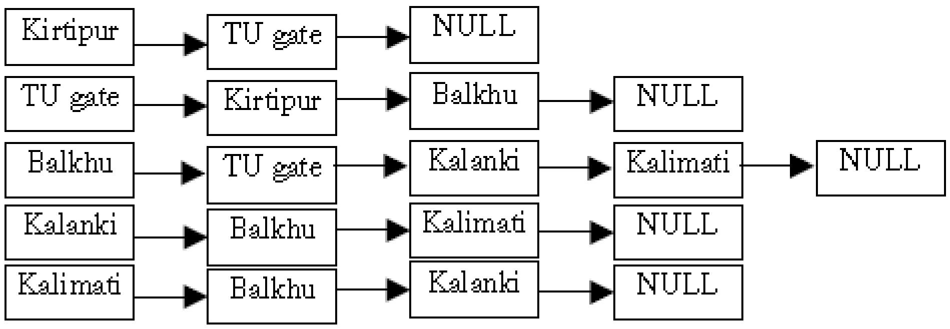
Representing Graphs

Generally we represent graph in two ways namely adjacency lists and adjacency matrix. Both ways can be applied to represent any kind of graph i.e. directed and undirected. An

adjacency matrix is an *n*□*n* matrix *M* where *M*[*i*,*j*] = 1 if there is an edge from vertex *i* to vertex *j* and *M*[*i*,*j*]=0 if there is not. Adjacency matrices are the simplest way to represent graphs. This representation takes $O(n^2)$ space regardless of the structure of the graph. So, if we have larger number of nodes say 100000 then we must have spcae for 100000^2 = 10,000,000,000 and this is quite a lot space to work on. The adjacency matrix representation of a graph for the above given road network graph is given below. Take the order {Kirtipur,TU gate, Balkhu, Kalanki, Kalimati}

0	1	0	0	0
1	0	1	0	0
0	1	0	1	1
0	0	1	0	1
0	0	1	1	0

It is very easy to see whether there is edge from a vertex to another vertex (O(1) time), what about space? Especially when the graph is sparse or undirected. If **adjacency list** representation of a graph contains and array of size *n* such that every vertex that has edge between the vertex denoted by the vertex with array position is added as a list with the corresponding array element. The example below gives the adjacency list representation of the above road network graph.



Searching for some edge (i,j) required O(d) time where d is the degree of i vertex.

Some points:

- To test if (*x*, *y*) is in graph adjacency matrices are faster.
- To find the degree of a vertex adjacency list is good
- For edge insertion and deletion adjacency matrix takes O(1) time where as adjacency list takes O(d) time.

Graph Traversals

There are a number of approaches used for solving problems on graphs. One of the most important approaches is based on the notion of systematically visiting all the vertices and edge of a graph. The reason for this is that these traversals impose a type of tree structure (or generally a forest) on the graph, and trees are usually much easier to reason about than general graphs.

Breadth-first search

This is one of the simplest methods of graph searching. Choose some vertex arbitrarily as a root. Add all the vertices and edges that are incident in the root. The new vertices added will become the vertices at the level 1 of the BFS tree. Form the set of the added vertices of level 1, find other vertices, such that they are connected by edges at level 1 vertices. Follow the above step until all the vertices are added.

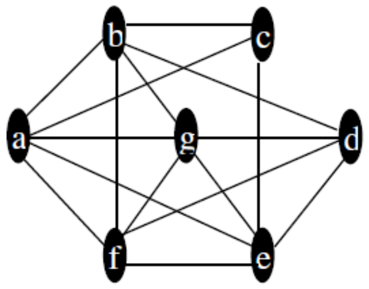
Algorithm:

```
BFS(G,s) //s is start vertex
{
    T = {s};
    L =Φ; //an empty queue
    Enqueue(L,s);
    while (L != Φ )
    {
        v = dequeue(L);
        for each neighbor w to v
            if ( w∉ L and w ∉ T )
```

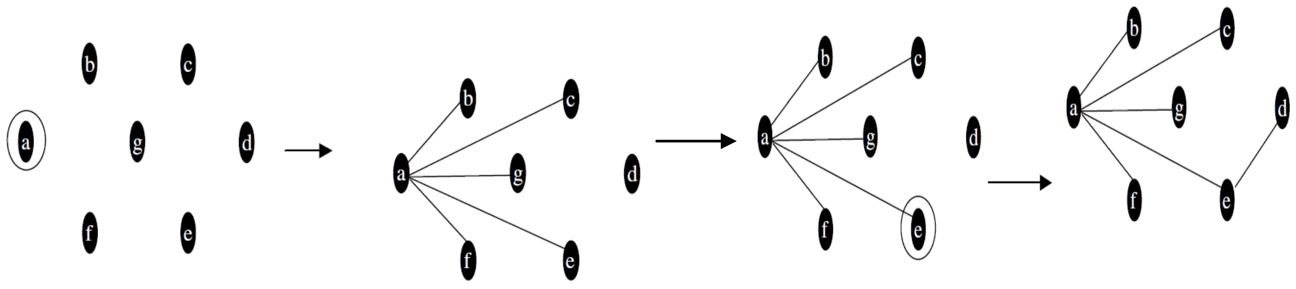
```
                {
                    enqueue( L,w);
                    T = T U {w}; //put edge {v,w} also
                }
        }
    }
```

Example:

Use breadth first search to find a BFS tree of the following graph.



Solution:



Analysis

From the algorithm above all the vertices are put once in the queue and they are accessed. For each accessed vertex from the queue their adjacent vertices are looked for and this can be done in O(n) time(for the worst case the graph is complete). This computation for all the possible vertices that may be in the queue i.e. n, produce complexity of an algorithm as O(n^2). Also from aggregate analysis we can write the complexity as O(E+V) because inner loop executes E times in total.

Depth First Search

This is another technique that can be used to search the graph. Choose a vertex as a root and form a path by starting at a root vertex by successively adding vertices and edges. This process is continued until no possible path can be formed. If the path contains all the vertices then the tree consisting this path is DFS tree. Otherwise, we must add other edges and vertices. For this move back from the last vertex that is met in the previous path and find whether it is possible to find new path starting from the vertex just met. If there is such a path continue the process above. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called **backtracking**.

Algorithm:

```
DFS(G,s)
{
    T = {s};
    Traverse(s);
}
Traverse(v)
{
    for each w adjacent to v and not yet in T
    {
        T = T U {w}; //put edge {v,w} also
        Traverse (w);
    }
}
```

Analysis:

The complexity of the algorithm is greatly affected by **Traverse** function we can write its running time in terms of the relation T(n) = T(n-1) + O(n), here O(n) is for each vertex at most all the vertices are checked (for loop). At each recursive call a vertex is decreased. Solving this we can find that the complexity of an algorithm is $O(n^2)$.

Also from aggregate analysis we can write the complexity as O(E+V) because traverse function is invoked V times maximum and for loop executes O(E) times in total.

for each execution the block inside the loop takes O(V) times . Hence the total running time is $O(V^2)$.

Minimum Spanning Tree

A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices). Assume there is an undirected, connected graph G. A spanning tree is a sub-graph of G that is tree and contains all the vertices of G. A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Application of MST

- Practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. MST can be used to determine the least costly paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- Another useful application of MST would be finding airline routes. MST can be applied to optimize airline routes by finding the least costly paths with no cycles

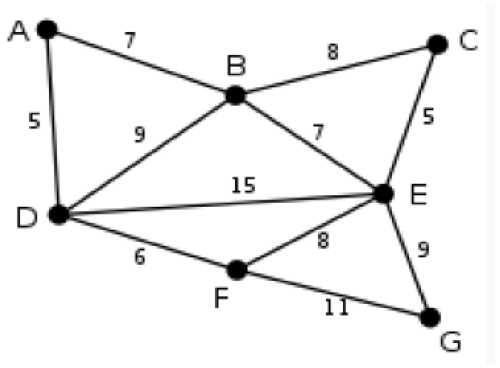
Kruskal's algorithm

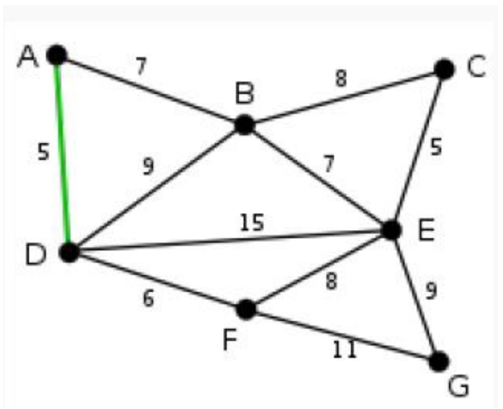
It is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm. It works as follows:

- create a forest *F* (a set of trees), where each vertex in the graph is a separate tree
- create a set *S* containing all the edges in the graph
- while *S* is nonempty and F is not yet spanning
- remove an edge with minimum weight from *S*
- if that edge connects two different trees, then add it to the forest, combining two trees into a single tree (i.e does not creates cycle)
- Otherwise discard that edge.

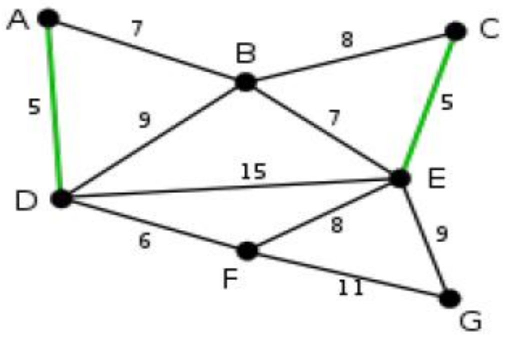
At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

Consider the following Example, This is our original graph. The numbers near the arcs indicate their weight. None of the arcs are highlighted.

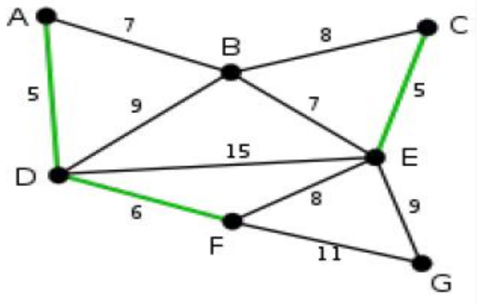




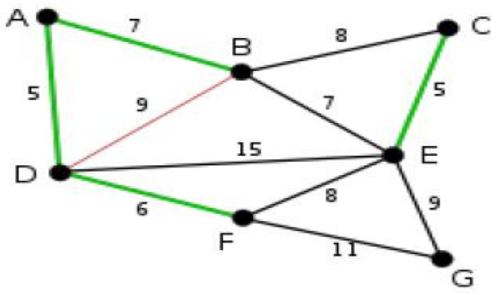
AD and **CE** are the shortest arcs, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.



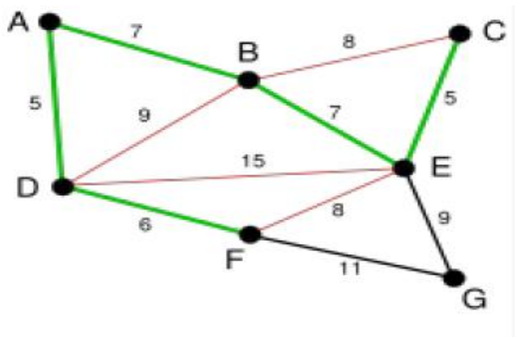
CE is now the shortest arc that does not form a cycle, with length 5, so it is highlighted as the second arc.



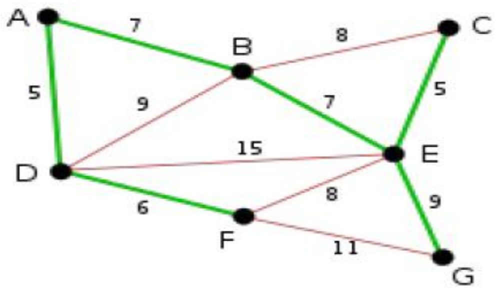
The next arc, **DF** with length 6, is highlighted using much the same method.



The next-shortest arcs are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The arc **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.



The process continues to highlight the next-smallest arc, **BE** with length 7. Many more arcs are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.



Finally, the process finishes with the arc **EG** of length 9, and the minimum spanning tree is found.

Algorithm:

```
KruskalMST(G)
{
        T = {V} // forest of n nodes
        S = set of edges sorted in nondecreasing order of weights
        while(|T| < n-1 and E !=Φ)
        {
                Select (u,v) from S in order
                Remove (u,v) from E
                if((u,v) doesnot create a cycle in T))
                        T = T U {(u,v)}
        }
```

}

Analysis

In the above algorithm creating n tree forest at the beginning takes (V) time, the creation of set S takes O(ElogE) time and while loop execute O(V) times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is O(ElogE) which is asymptotically equivalently O(ElogV).

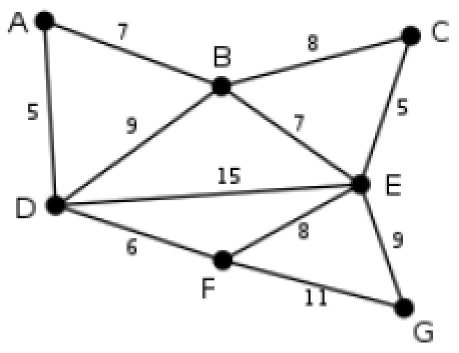
Prim's algorithm

It is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

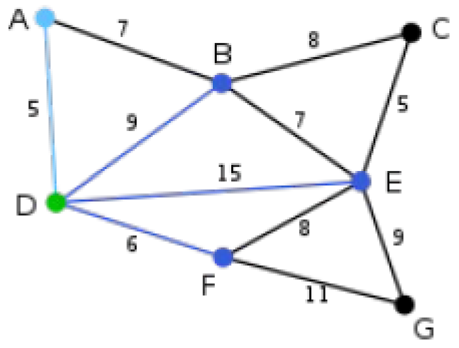
How it works

- This algorithm builds the MST one vertex at a time.
- It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex.
- Now, from either 'A' or 'B', it will find the next least costly vertex connection, Without creating a cycle (vertex C, for example).
- Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes.
- Eventually, all the vertices will be connected, without any cycles, and an MST
- will be the result.

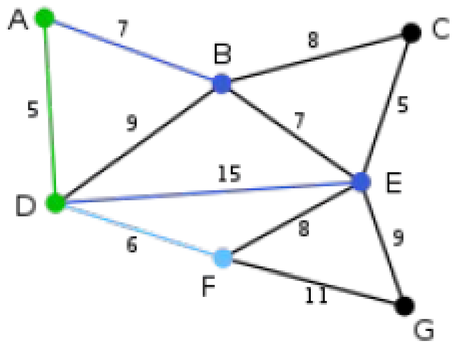
Example,



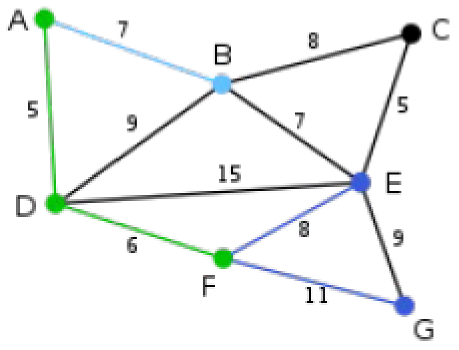
This is our original weighted graph. The numbers near the edges indicate their weight.



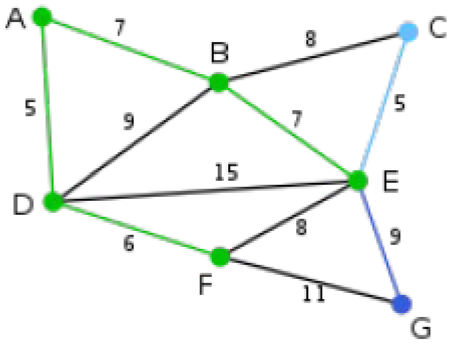
Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D** through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**.



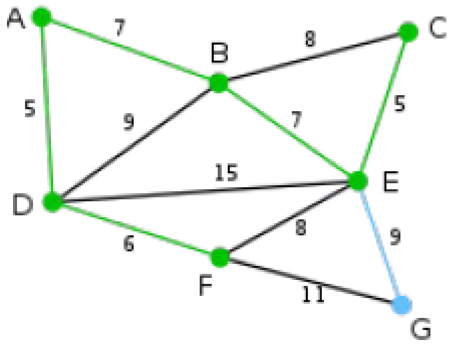
The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**, **E** is 15,and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.



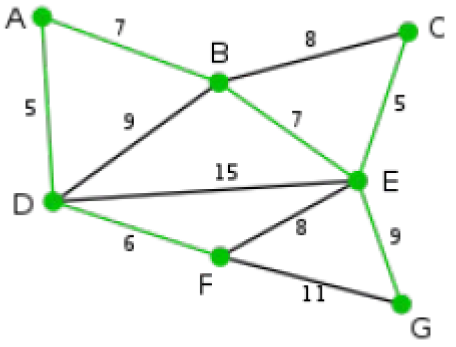
The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.



In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **BE**.



Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.



Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.

Algorithm:

```
PrimMST(G)
{
    T = Φ; // T is a set of edges of MST
    S = {s}; //s is randomly chosen vertex and S is set of vertices
    while(S != V)
    {
        e = (u,v) an edge of minimum weight incident to vertices in T and not forming a
        simple circuit in T if added to T i.e. u ε S and v ε V-S
        T = T ∪ {(u,v)};
        S = S ∪ {v};
}
}
```

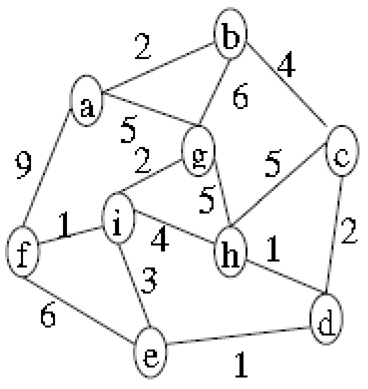
Analysis

In the above algorithm while loop execute for O(V) time. The edge of minimum weight incident on a vertex can be found in O(E), so the total time is O(EV). We can improve the performance of the above algorithm by choosing better data structures. If we use heap data structure edge of minimum weight can be selected in O(logE) time and the running time of prim's algorithm becomes O(ElogE) which is equivalent to O(ElogV).

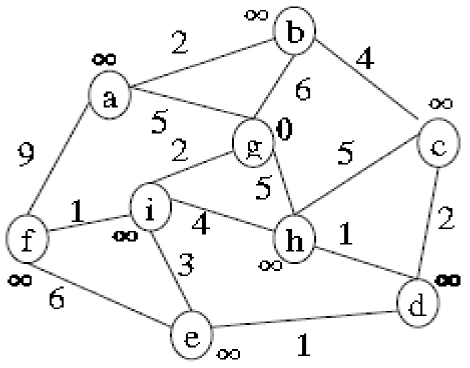
Shortest Path Algorithms

Dijkstra Algorithm

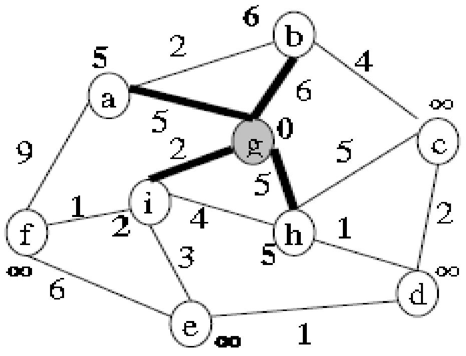
This is an approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as below:



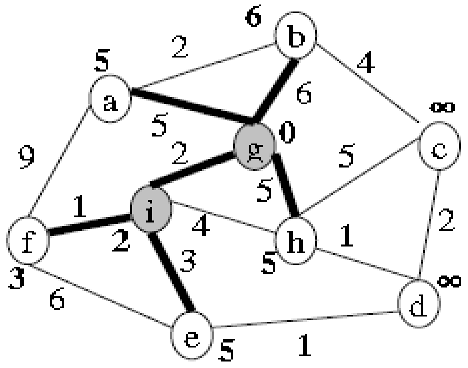
Initially, shortest path to all vertices from some given source is infinity and, suppose g is the is the source vertex.



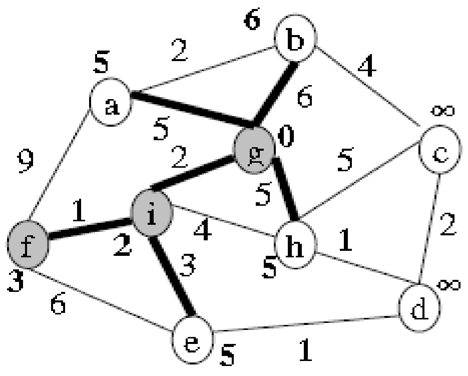
Take a vertex with shortest path (i.e vertex g), and relax all neighboring vertex. Vertex a, b, i and h gets relaxed.



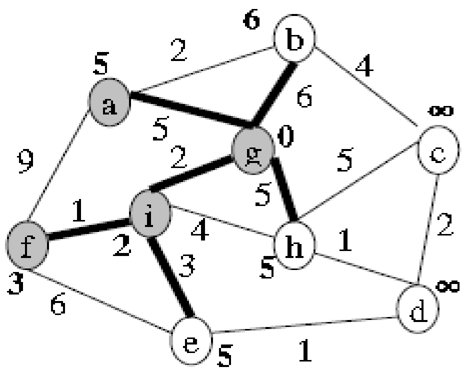
Again, Take a vertex with shortest path (i.e vertex i), and relax all neighboring vertex. Vertices f and e gets relaxed.



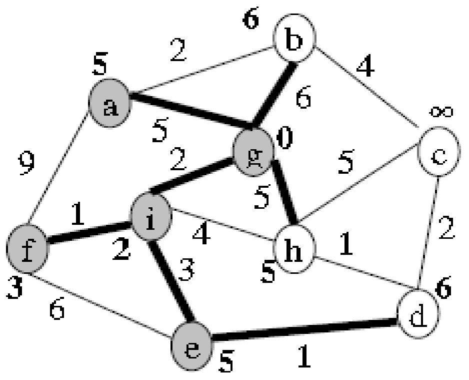
Take a vertex with shortest path (i.e vertex f), and relax all neighboring vertex. None of the vertices gets relaxed.



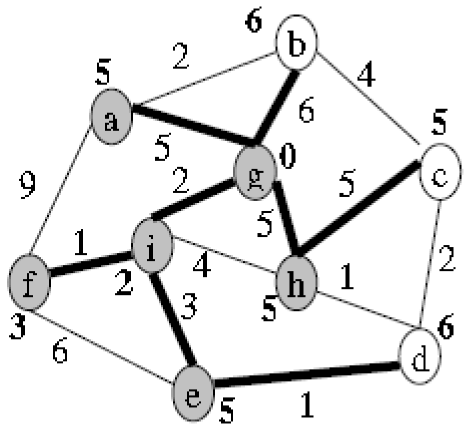
Take a vertex with shortest path (i.e vertex a or vertex e or vertex h), and relax all neighboring vertex. None of the vertices gets relaxed.



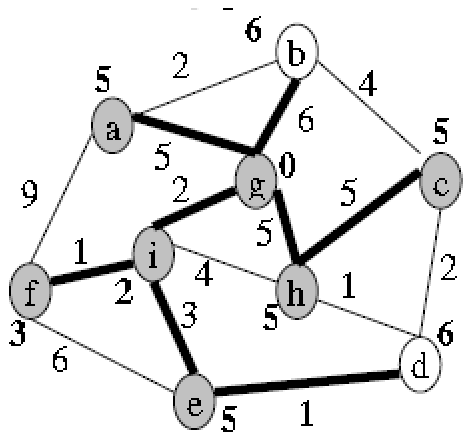
Take a vertex with shortest path (i.e vertex e or vertex h), and relax all neighboring vertex. Vertex d gets relaxed.



Take a vertex with shortest path (i.e vertex h), and relax all neighboring vertex. Vertex c gets relaxed



Take a vertex with shortest path (i.e vertex c), and relax all neighboring vertex. None of the vertices gets relaxed.



There will be no change for vertices b and d. continues above steps for b and d to complete. The tree is shown as dark connection.

Algorithm:

```
Dijkstra(G,w,s)
{
        for each vertex vÎ V
            do d[v] = ∞
        p[v] = Nil
        d[s] = 0
        S = Φ
        Q = V
        While(Q!= Φ)
        {
            u = Take minimum from Q and delete.
            S = S ∪ {u}
            for each vertex v adjacent to u
                if d[v] > d[u] + w(u,v)
                then d[v] = d[u] + w(u,v)
        }
    }
```

Analysis

In the above algorithm, the first for loop block takes O(V) time. Initialization of priority queue Q takes O(V) time. The while loop executes for O(V), where for each execution the block inside the loop takes O(V) times . Hence the total running time is O(V^2).

Flyod's Warshall Algorithm

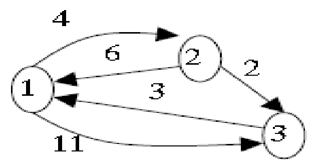
The algorithm being discussed uses dynamic programming approach. The algorithm being presented here works even if some of the edges have negative weights. Consider a weighted graph G = (V,E) and denote the weight of edge connecting vertices i and j by wij. Let W be the adjacency matrix for the given graph G. Let D^k denote an n x n matrix such that $D^k(i,j)$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices from 1,2,…,k as intermediate vertices in the path. If we consider shortest path with intermediate vertices as above then computing the path contains two cases. $D^k(i,j)$ does not contain k as intermediate vertex and $D^k(i,j)$ contains k as intermediate vertex. Then we have the following relations $D^k(i,j) = D^{k-1}(i,j)$, when k is not an intermediate vertex, and $D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)$, when k is an intermediate vertex. So from the above relations we obtain:

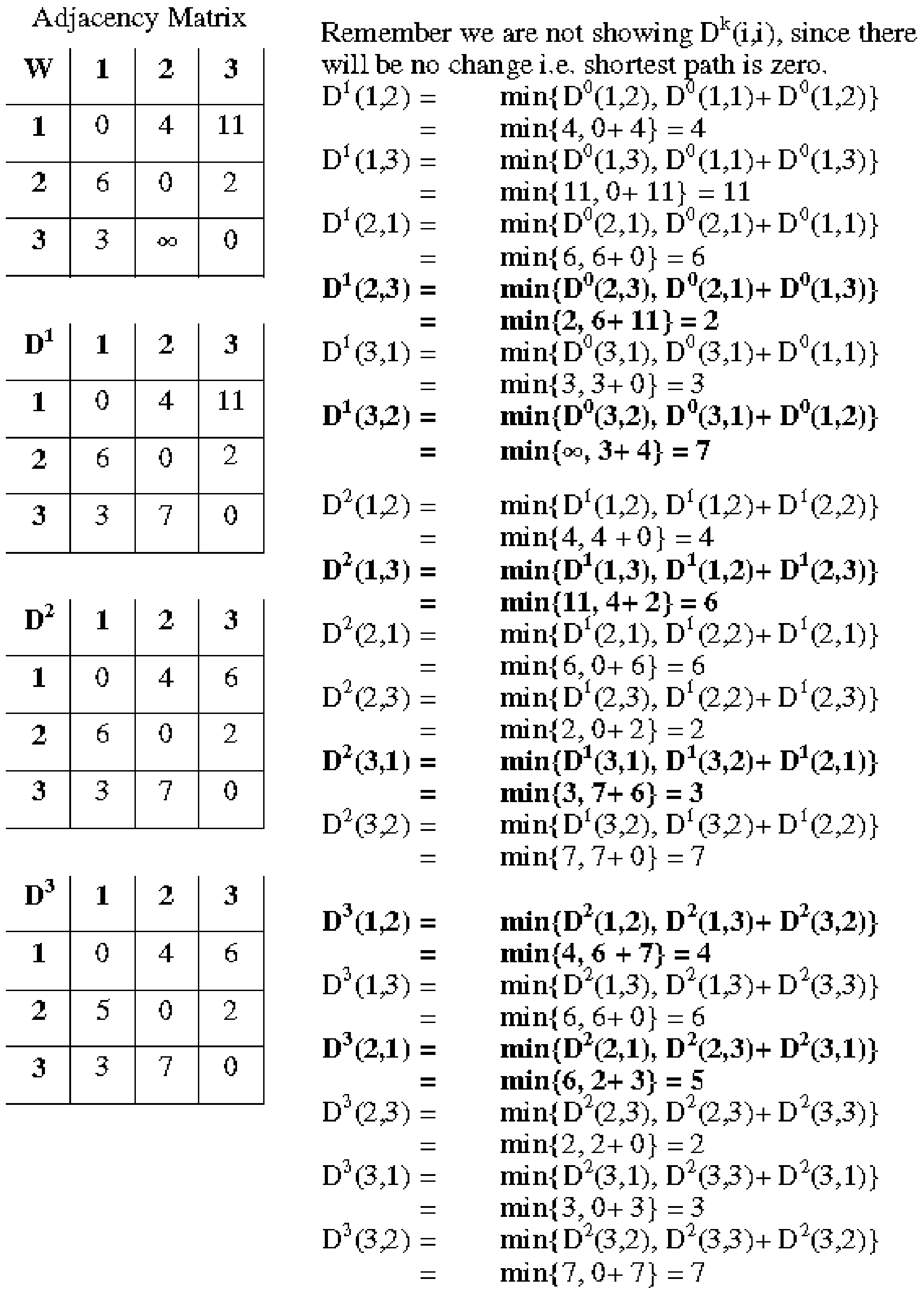
$$D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\}.$$

Algorithm:

```
FloydWarshalAPSP(W,D,n) // W is adjacency matrix of graph G.
{
    for(i=1;i<=n;i++)
        for(j=1;j<=1;j++)
            D[i][j] = W[i][j]; // initially D[][] is D0.
    for(k=1;k<=n;k++)
        for(i=1;i<=n;i++)
            for(j=1;j<=1;j++)
                D[i][j] = min{D[i][j], D[i][k]+ D[k][j]};
}
```

Example:





Adjacency Matrix

W	**1**	**2**	**3**
1	0	4	11
2	6	0	2
3	3	∞	0

D^1	**1**	**2**	**3**
1	0	4	11
2	6	0	2
3	3	7	0

D^2	**1**	**2**	**3**
1	0	4	6
2	6	0	2
3	3	7	0

D^3	**1**	**2**	**3**
1	0	4	6
2	5	0	2
3	3	7	0

Remember we are not showing $D^k(i,i)$, since there will be no change i.e. shortest path is zero.

$D^1(1,2) = \min\{D^0(1,2), D^0(1,1)+ D^0(1,2)\}$
$= \min\{4, 0+ 4\} = 4$
$D^1(1,3) = \min\{D^0(1,3), D^0(1,1)+ D^0(1,3)\}$
$= \min\{11, 0+ 11\} = 11$
$D^1(2,1) = \min\{D^0(2,1), D^0(2,1)+ D^0(1,1)\}$
$= \min\{6, 6+ 0\} = 6$
$D^1(2,3) = \min\{D^0(2,3), D^0(2,1)+ D^0(1,3)\}$
$= \min\{2, 6+ 11\} = 2$
$D^1(3,1) = \min\{D^0(3,1), D^0(3,1)+ D^0(1,1)\}$
$= \min\{3, 3+ 0\} = 3$
$D^1(3,2) = \min\{D^0(3,2), D^0(3,1)+ D^0(1,2)\}$
$= \min\{\infty, 3+ 4\} = 7$

$D^2(1,2) = \min\{D^1(1,2), D^1(1,2)+ D^1(2,2)\}$
$= \min\{4, 4 + 0\} = 4$
$D^2(1,3) = \min\{D^1(1,3), D^1(1,2)+ D^1(2,3)\}$
$= \min\{11, 4+ 2\} = 6$
$D^2(2,1) = \min\{D^1(2,1), D^1(2,2)+ D^1(2,1)\}$
$= \min\{6, 0+ 6\} = 6$
$D^2(2,3) = \min\{D^1(2,3), D^1(2,2)+ D^1(2,3)\}$
$= \min\{2, 0+ 2\} = 2$
$D^2(3,1) = \min\{D^1(3,1), D^1(3,2)+ D^1(2,1)\}$
$= \min\{3, 7+ 6\} = 3$
$D^2(3,2) = \min\{D^1(3,2), D^1(3,2)+ D^1(2,2)\}$
$= \min\{7, 7+ 0\} = 7$

$D^3(1,2) = \min\{D^2(1,2), D^2(1,3)+ D^2(3,2)\}$
$= \min\{4, 6 + 7\} = 4$
$D^3(1,3) = \min\{D^2(1,3), D^2(1,3)+ D^2(3,3)\}$
$= \min\{6, 6+ 0\} = 6$
$D^3(2,1) = \min\{D^2(2,1), D^2(2,3)+ D^2(3,1)\}$
$= \min\{6, 2+ 3\} = 5$
$D^3(2,3) = \min\{D^2(2,3), D^2(2,3)+ D^2(3,3)\}$
$= \min\{2, 2+ 0\} = 2$
$D^3(3,1) = \min\{D^2(3,1), D^2(3,3)+ D^2(3,1)\}$
$= \min\{3, 0+ 3\} = 3$
$D^3(3,2) = \min\{D^2(3,2), D^2(3,3)+ D^2(3,2)\}$
$= \min\{7, 0+ 7\} = 7$

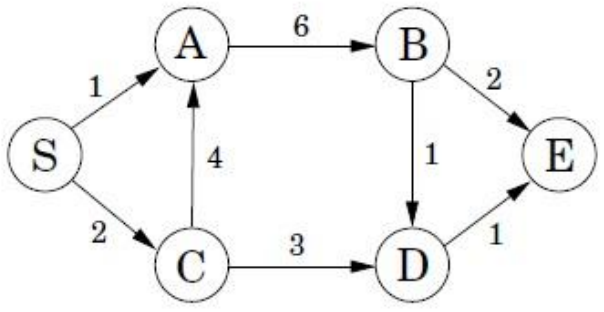
Analysis:

Clearly the above algorithm's running time is $O(n^3)$, where n is cardinality of set V of vertices.

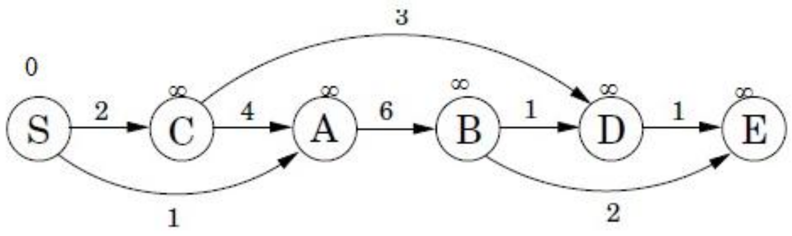
Directed Acyclic Graph (DAG)

DAG, here directed means that each edge has an arrow denoting that the edge can be traversed in only that particular direction. Acyclic means that the graph has no cycles, i.e., starting at one node, you can never end up at the same node. DAG can be used to find shortest path from a given source node to all other nodes. To find shortest path by using DAG, first of all sort the vertices of graph topologically and then relax the vertices in topological order.

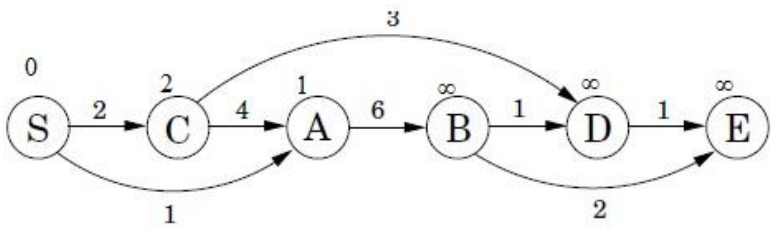
Example:



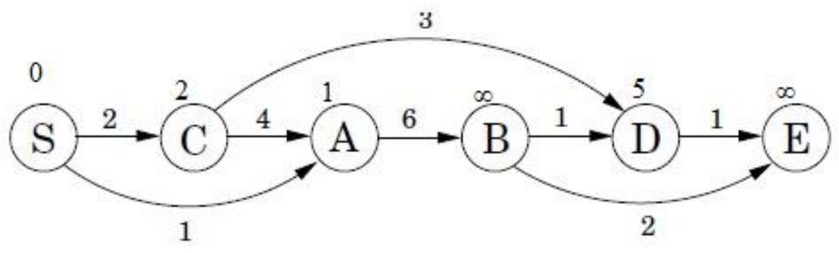
Step1: Sort the vertices of graph topologically



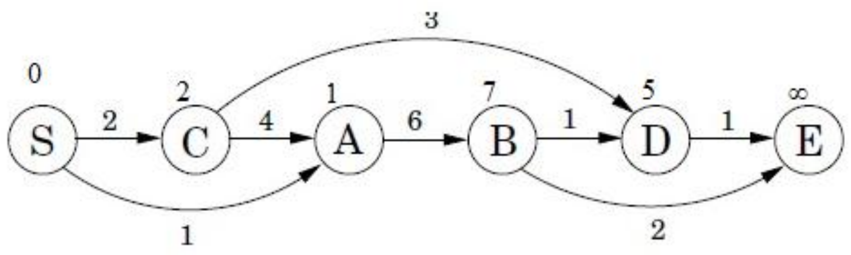
Step2: Relax from S ∞



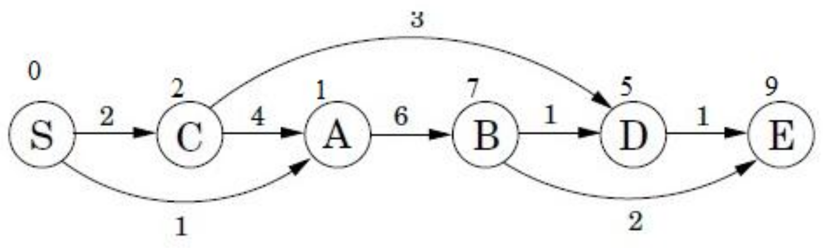
Step3: Relax from C



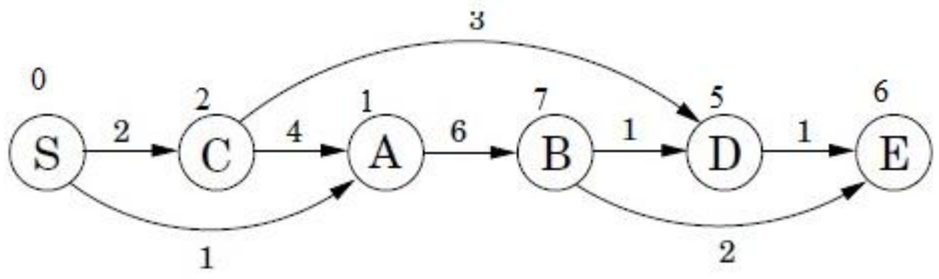
Step4: Relax from A



Step5: Relax from B



Step6: Relax from D



Algorithm

```
DagSP(G,w,s)
{
        Topologically Sort the vertices of G
        for each vertex vε V
                d[v] = ∞
```

```
    d[s] = 0
    for each vertex u, taken in topologically sorted order
        for each vertex v adjacent to u
                if d[v] > d[u] + w(u,v)
                        d[v] = d[u] + w(u,v)
}
```

Analysis:

In the above algorithm, the topological sort can be done in O(V+E) time (Since this is similar to DFS! see book.).The first for loop block takes O(V) time. In case of second for loop it executes in O(V2) Time so the total running time is O(V2). Aggregate analysis gives us the running time O(E+V).