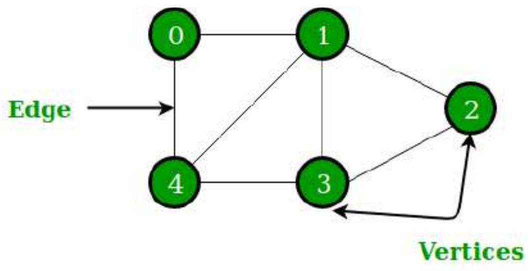
Chapter 9 Graphs

1. Graph

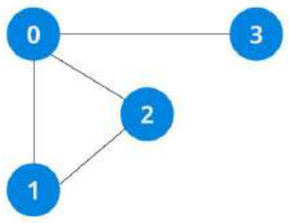
A Graph is a non-linear data structure consisting of a finite set of nodes and edges. The nodes are referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

A Graph is a finite set (V, E) of vertices (or nodes) and set of Edges (or arcs) which connect a pair of nodes.



The above figure shows a graph, where the set of vertices, V = {0, 1, 2, 3, 4} and the set of edges E = {(0, 1), (1, 2), (2, 3), (3, 4), (0, 4), (1, 4), (1, 3)}.

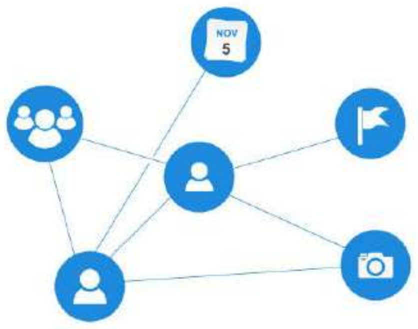
Let us consider another example:



In the graph, V = {0, 1, 2, 3}, E = {(0, 1), (0, 2), (0, 3), (1, 2)} and graph, G = {V, E}.

Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include city network, airline networks, telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.). For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes.

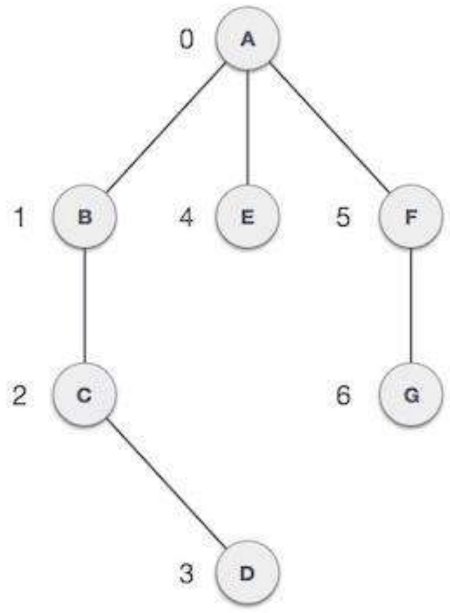
On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node. Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page etc., a new edge is created for that relationship.



All of facebook is then, a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

2. Graph Terminologies

Vertex − each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices.



Edge − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges.

Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices C and D are adjacent because there is an edge between them. Vertices B and E are not adjacent because there is no edge between them.

Path − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Incident - An edge is **incident** on a vertex if the vertex is an endpoint of the edge.

Outgoing edges – outgoing edges of a vertex are directed edges that the vertex is the origin. **Incoming edges - Incoming edges** of a vertex are directed edges that the vertex is the destination.

Degree - Degree of a vertex, *v*, denoted *deg*(*v*) is the number of incident edges.

Out-degree, *outdeg*(*v*), is the number of outgoing edges.

In-degree, *indeg*(*v*), is the number of incoming edges.

Parallel edges or multiple edges are edges of the same type and end-vertices.

Self-loop is an edge with the end vertices the same vertex.

Simple graphs have **no** parallel edges or self-loops.

Cycle is a path that starts and ends at the same vertex.

Simple path is a path with distinct vertices.

A directed graph is called **weekly connected** if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. The vertices in a weakly connected graph have their out-degree or in-degree of at least 1.

A bridge is an edge whose removal would disconnect the graph.

Forest is a graph without cycles.

Tree is a connected graph with no cycles. If we remove all the cycles from directed acyclic graph (DAG) it becomes tree and if we remove any edge in a tree it becomes forest.

Spanning tree of an undirected graph is a subgraph that is a tree which includes all of the vertices of the graph

The set of all neighbors of a vertex v of G = (V,E), denoted by N(v), is called the neighborhood of v. If A is a subset of V, we denote by N(A) the set of all vertices in G that are adjacent to at least one vertex in A. So, $N(A) = \bigcup_{v \in A} N(v)$.

The degree of a vertex in an undirected graph is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by deg(v).

In a graph with directed edges the in-degree of a vertex v, denoted by $deg^-(v)$,is the number of edges with v as their terminal vertex. The out-degree of v, denoted by $deg^+(v)$, is the number of edges with v as their initial vertex. (Note that a loop at a vertex contributes 1 to both the in-degree and the out-degree of this vertex.)

Weighted graph: A positive value assigned to each edge indicating its length (distance between the vertices connected by an edge) is called weight. The graph containing weighted edges is called a weighted graph. The weight of an edge e is denoted by w(e) and it indicates the cost of traversing an edge.

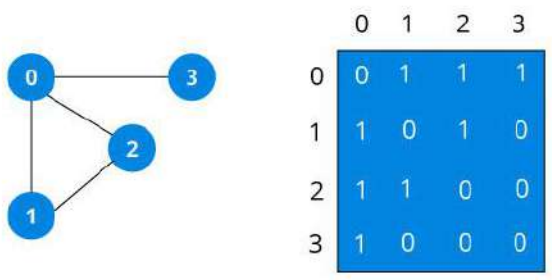
3. Graph Representation

Graphs are commonly represented in two ways:

- Adjacency Matrix
- Incidence Matrix
- Adjacency List

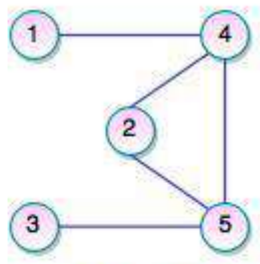
Adjacency Matrix

An adjacency matrix is 2D array of V x V vertices. Each row and column represent a vertex. If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j. For example, the matrix shows the adjacency matrix for the given graph:

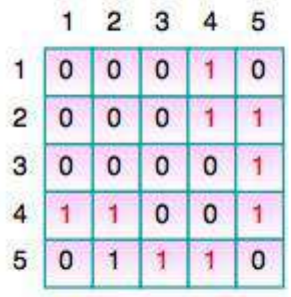


$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 \end{array}$$

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.



Undirected Graph

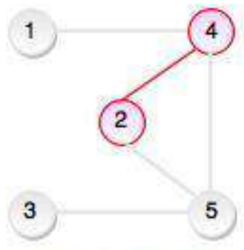


	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

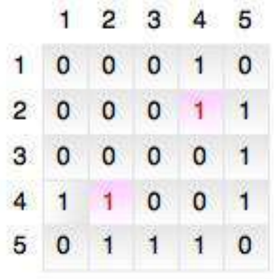
Adjacency Matrix

Fig. Adjacency Matrix Representation of Undirected Graph

The above graph represents undirected graph with the adjacency matrix representation. It shows adjacency matrix of undirected graph is symmetric. If there is an edge (2, 4), there is also an edge (4, 2).



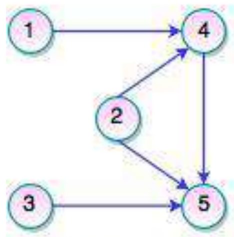
Undirected Graph
Edge (2, 4)



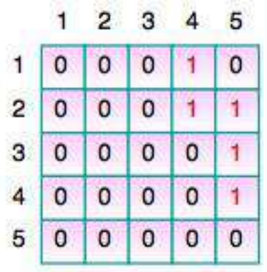
	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicated a directed edge from vertex i to vertex j.



Directed Graph



	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0

Adjacency Matrix

Fig. Adjacency Matrix Representation of Directed Graph

The above graph represents directed graph with the adjacency matrix representation. It shows adjacency matrix of directed graph which is never symmetric. If there is an edge (2, 4), there is not an edge (4, 2). It indicates direct edge from vertex i to vertex j.

Adjacency matrix is a way to represent a graph. It shows which nodes are adjacent to one another. Graph is represented using a square matrix. **A graph can be a sparse graph** if it contains less number of edges. It is called a **dense graph** if it contains more number of edges as compared to sparse graph. Adjacency matrix is best for dense graph. Adjacency matrix of an undirected graph is always a symmetric matrix which means an edge (i, j) implies the edge (j, i).

Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices (V x V), so it requires more space.

Advantages of Adjacency Matrix

- Adjacency matrix representation of graph is very simple to implement.
- Adding or removing time of an edge can be done in O(1) time. Same time is required to check, if there is an edge between two vertices.
- It is very convenient and simple to program.

Disadvantages of Adjacency Matrix

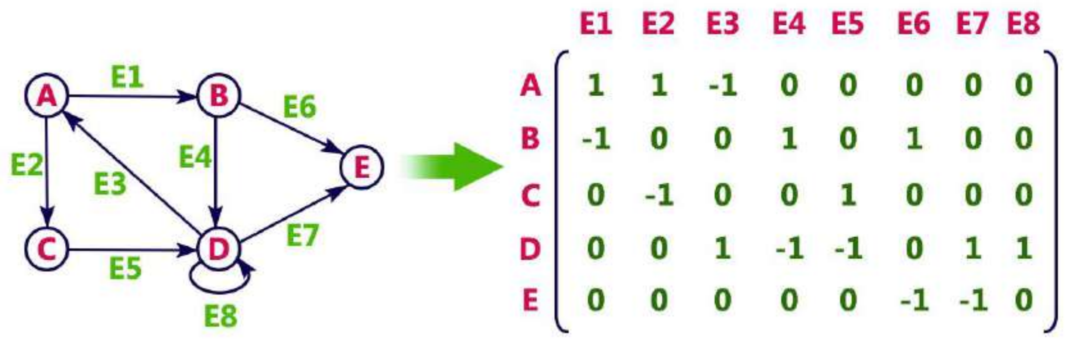
- It consumes huge amount of memory for storing big graphs.
- It requires huge efforts for adding or removing a vertex. If you are constructing a graph in dynamic structure, adjacency matrix is quite slow for big graphs.

Incidence Matrix

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1.

- 0 represents that the row edge is not connected to column vertex,
- 1 represents that the row edge is connected as the outgoing edge to column vertex and
- -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation:



$$\begin{array}{c|cccccccc} & E1 & E2 & E3 & E4 & E5 & E6 & E7 & E8 \\ \hline A & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ B & -1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ C & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & -1 & -1 & 0 & 1 & 1 \\ E & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \end{array}$$

Adjacency List

An adjacency list represents a graph as an array of linked list. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex. The adjacency list for the graph is as follows:

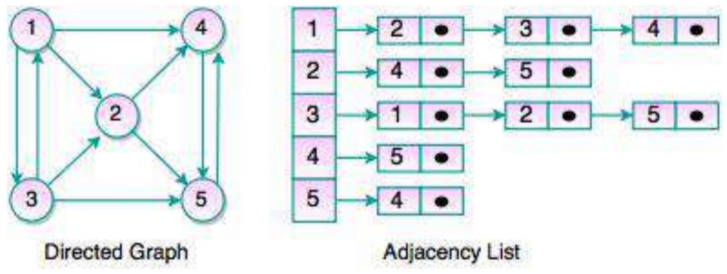
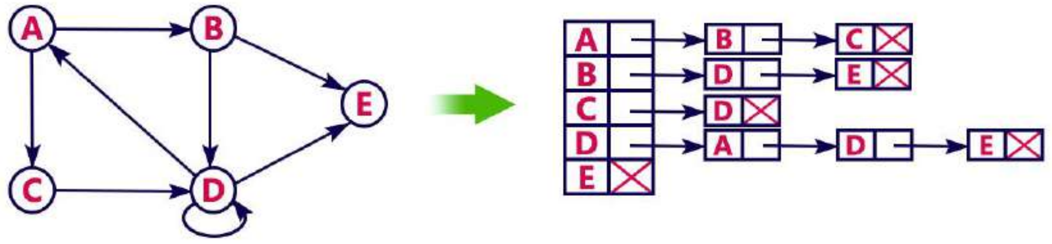
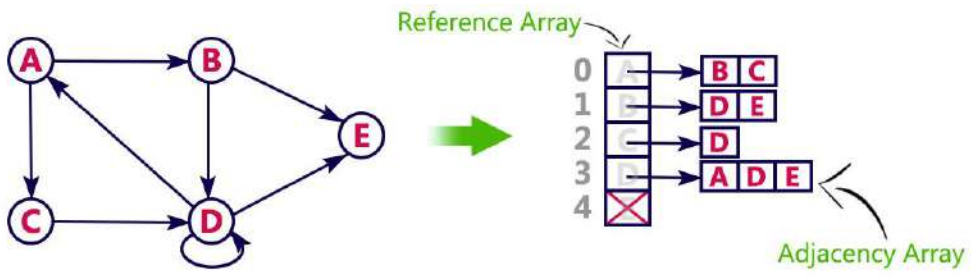


Fig. Adjacency List Representation of Directed Graph

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows.



An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

In summary,

- Adjacency list is another representation of graphs.
- It is a collection of unordered list, used to represent a finite graphs.
- Each list describes the set of neighbors of a vertex in the graph.
- Adjacency list requires less amount of memory.
- For every vertex, adjacency list stores a list of vertices, which are adjacent to the current one.
- In adjacency list, an array of linked list is used. Size of the array is equal to the number of vertices.
- In adjacency list, an entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex.
- Adjacency list allows to store the graph in more compact form than adjacency matrix.
- It allows to get the list of adjacent vertices in O(1) time.

Disadvantages of Adjacency List

- It is not easy for adding or removing an edge to/from adjacent list.
- It does not allow to make an efficient implementation, if dynamically change of vertices number is required.

4. Graph applications

Since they are powerful abstractions, graphs can be very important in modeling data. In fact, many problems can be reduced to known graph problems. Here we outline just some of the many applications of graphs.

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.

3. Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. Document link graphs. The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
5. Protein-protein interactions graphs. Vertices represent proteins and edges represent interactionsbetweenthemthatcarryoutsomebiologicalfunctioninthecell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humanshaveover120Kproteinswithmillionsofinteractionsamong them.
6. Network packet traffic graphs. Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.
7. Scene graphs. In graphics and computer games scene graphs represent the logical or special relationships between objects in a scene. Such graphs are very important in the computer games industry.
8. Finite element meshes. In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.
9. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.
10. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 1011 neurons and close to 1015 synapses.
11. Graphs in quantum field theory. Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).
12. Semantic networks. Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.
13. Graphs in epidemiology. Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

14. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
15. Constraint graphs. Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.
16. Dependence graphs. Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.
17. In **Computer science** graphs are used to represent the flow of computation.
18. **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
19. In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
20. In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
21. In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
22. **Computer Science:** In computer science, graph is used to represent networks of communication, data organization, computational devices etc.
23. **Physics and Chemistry:** Graph theory is also used to study molecules in chemistry and physics.
24. **Social Science:** Graph theory is also widely used in sociology.
25. **Mathematics:** In this, graphs are useful in geometry and certain parts of topology such as knot theory.
26. **Biology:** Graph theory is useful in biology and conservation efforts.

5. Graphs as an ADT

Description

A graph data structure consists of a finite (and possibly mutable) set of *vertices* (also called *nodes* or *points*), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges* (also called *links* or *lines*), and for a directed graph are also known as *arrows*.

Operations

The basic operations provided by a graph data structure *G* usually include:

a) `Graph()` creates a new, empty graph.
b) `adjacent`(*G*, *x*, *y*): tests whether there is an edge from the vertex *x* to the vertex *y*;
c) `neighbors`(*G*, *x*): lists all vertices *y* such that there is an edge from the vertex *x* to the vertex *y*;
d) `add_vertex`(*G*, *x*): adds the vertex *x*, if it is not there;
e) `remove_vertex`(*G*, *x*): removes the vertex *x*, if it is there;
f) `add_edge`(*G*, *x*, *y*): adds the edge from the vertex *x* to the vertex *y*, if it is not there;
g) `remove_edge`(*G*, *x*, *y*): removes the edge from the vertex *x* to the vertex *y*, if it is there;
h) `get_vertex_value`(*G*, *x*): returns the value associated with the vertex *x*;
i) `set_vertex_value`(*G*, *x*, *v*): sets the value associated with the vertex *x* to *v*.

Directed graph implementation

```
#include <iostream>
#include <vector>
using namespace std;

// data structure to store graph edges
struct Edge {
     int src, dest;
};

// class to represent a graph object
class Graph
{
public:
     // construct a vector of vectors to represent an adjacency
list
     vector<vector<int>> adjList;

     // Graph Constructor
     Graph(vector<Edge> const &edges, int N)
     {
          // resize the vector to N elements of type vector<int>
          adjList.resize(N);

          // add edges to the directed graph
```

```
            for (auto &edge: edges)
            {
                 // insert at the end
                 adjList[edge.src].push_back(edge.dest);

                 // Uncomment below line for undirected graph
                 // adjList[edge.dest].push_back(edge.src);
            }
       }
};

// print adjacency list representation of graph
void printGraph(Graph const& graph, int N)
{
       for (int i = 0; i < N; i++)
       {
            // print current vertex number
            cout << i << " --> ";

            // print all neighboring vertices of vertex i
            for (int v : graph.adjList[i])
                 cout << v << " ";
            cout << endl;
       }
}

// Graph Implementation using STL
int main()
{
       // vector of graph edges as per above diagram.
       // Please note that initialization vector in below format
will
       // work fine in C++11, C++14, C++17 but will fail in C++98.
       vector<Edge> edges =
       {
            { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },
            { 3, 2 }, { 4, 5 }, { 5, 4 }
       };

       // Number of nodes in the graph
       int N = 6;

       // construct graph
       Graph graph(edges, N);

       // print adjacency list representation of graph
       printGraph(graph, N);
```

```
     return 0;
}
```

C++ Graph Implementation Using Adjacency List

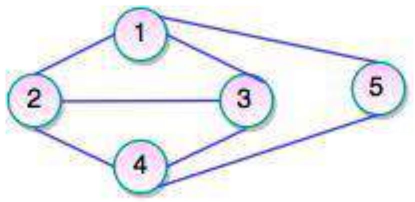
```
#include <iostream>
using namespace std;
// stores adjacency list items
struct adjNode {
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge {
    int start_ver, end_ver, weight;
};
class DiaGraph{
    // insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head)   {
        adjNode* newNode = new adjNode;
        newNode->val = value;
        newNode->cost = weight;

        newNode->next = head; // point new node to current head
        return newNode;
    }
    int N;  // number of nodes in the graph
public:
    adjNode **head;   //adjacency list as array of pointers
    // Constructor
    DiaGraph(graphEdge edges[], int n, int N)  {
        // allocate new node
        head = new adjNode*[N]();
        this->N = N;
        // initialize head pointer for all vertices
        for (int i = 0; i < N; ++i)
            head[i] = nullptr;
        // construct directed graph by adding edges to it
        for (unsigned i = 0; i < n; i++)  {
            int start_ver = edges[i].start_ver;
            int end_ver = edges[i].end_ver;
            int weight = edges[i].weight;
            // insert in the beginning
            adjNode* newNode = getAdjListNode(end_ver, weight,
head[start_ver]);
            // point head pointer to new node
            head[start_ver] = newNode;
             }
    }
```

```
       // Destructor
      ~DiaGraph() {
     for (int i = 0; i < N; i++)
         delete[] head[i];
         delete[] head;
      }
};
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != nullptr) {
        cout << "(" << i << ", " << ptr->val
            << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}
int main()
{
    // graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
    };
    int N = 6;      // Number of vertices in the graph
    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);
    // construct graph
    DiaGraph diagraph(edges, n, N);
    // print adjacency list representation of graph
    cout<<"Graph adjacency list "<<endl<<"(start_vertex, end_vertex,
weight):"<<endl;
    for (int i = 0; i < N; i++)
    {
        // display adjacent vertices of vertex i
        display_AdjList(diagraph.head[i], i);
    }
    return 0;
}
```

6. Graphs types

Undirected Graph

- If a graph contains unordered pair of vertices, is said to be an Undirected Graph.
- In this graph, pair of vertices represents the same edge.

- 
  Fig. Undirected Graph

  Set of Vertices V = {1, 2, 3, 4, 5}
  Set of Edges E = {(1, 2), (1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}
- In an undirected graph, the nodes are connected by undirected arcs.
- It is an edge that has no arrow. Both the ends of an undirected arc are equivalent, there is no head or tail.

Directed Graph

- If a graph contains ordered pair of vertices, is said to be a Directed Graph.
- If an edge is represented using a pair of vertices (V_1, V_2), the edge is said to be directed from V_1 to V_2.
- The first element of the pair V_1 is called the start vertex and the second element of the pair V_2 is called the end vertex.

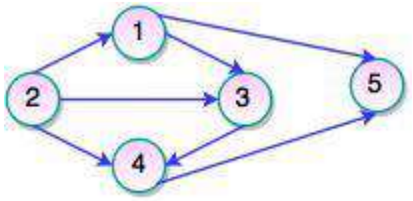


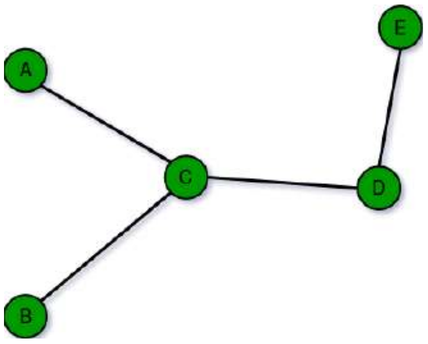
Fig. Directed Graph

Set of Vertices V = {1, 2, 3, 4, 5, 5}

Set of Edges W = {(1, 3), (1, 5), (2, 1), (2, 3), (2, 4), (3, 4), (4, 5)}

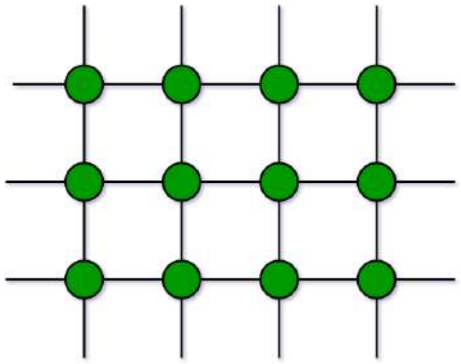
Finite Graphs

A graph is said to be finite if it has finite number of vertices and finite number of edges.



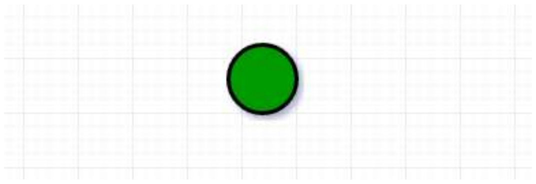
Infinite Graph

A graph is said to be infinite if it has infinite number of vertices as well as infinite number of edges.



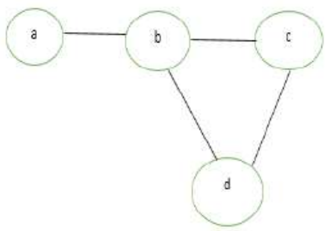
Trivial Graph

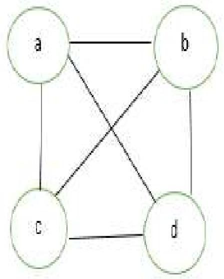
A graph is said to be trivial if a finite graph contains only one vertex and no edge.



Simple Graph

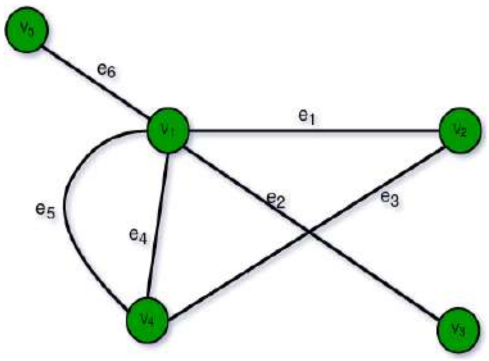
A simple graph is a graph which does not contains more than one edge between the pair of vertices. A simple railway tracks connecting different cities is an example of simple graph.





Multi Graph

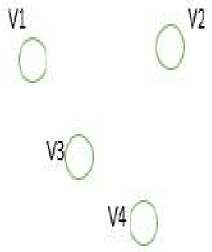
Any graph which contain some parallel edges but doesn't contain any self-loop is called multi graph. For example A Road Map.



- **Parallel Edges:** If two vertices are connected with more than one edge than such edges are called parallel edges that is many roots but one destination.
- **Loop:** An edge of a graph which join a vertex to itself is called loop or a self-loop.

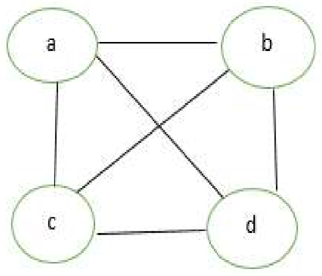
Null Graph

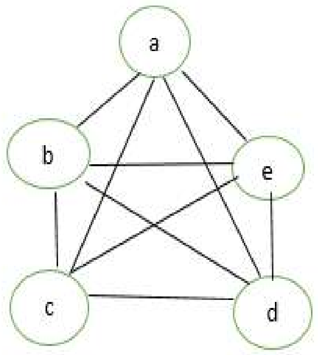
A graph of order n and size zero that is a graph which contain n number of vertices but do not contain any edge.



Complete Graph

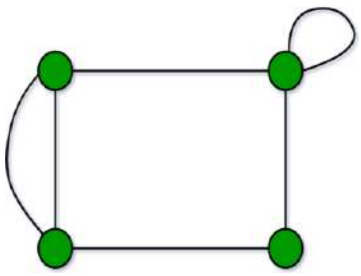
A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.





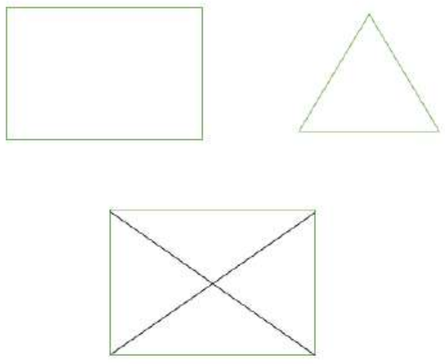
Pseudo Graph

A graph G with a self-loop and some multiple edges is called pseudo graph.



Regular Graph

A simple graph is said to be regular if all vertices of a graph G are of equal degree. All complete graphs are regular but vice versa is not possible.



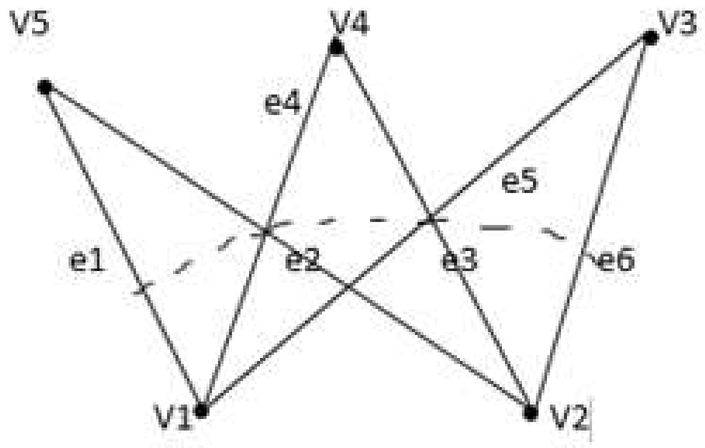
Bipartite Graph

A graph G = (V, E) is said to be bipartite graph if its vertex set V(G) can be partitioned into two non-empty disjoint subsets. V1(G) and V2(G) in such a way that each edge e of E(G) has its one end in V1(G) and other end in V2(G).

The partition V1 U V2 = V is called Bipartite of G. Here in the figure:

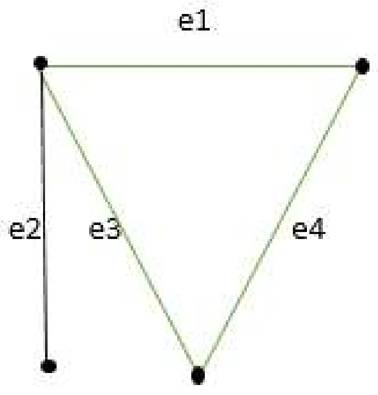
V1(G)={V5, V4, V3}

V2(G)={V1, V2}



Labelled Graph

If the vertices and edges of a graph are labelled with name, data or weight then it is called labelled graph. It is also called *Weighted Graph*.



Digraph Graph

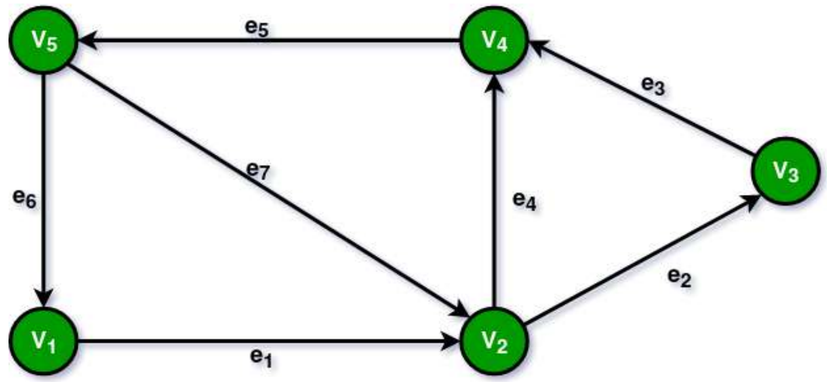
A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.

Here in the figure:

e1 = (V1, V2)

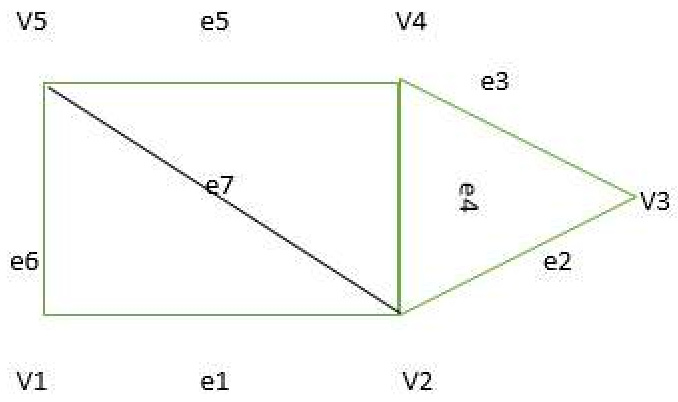
e2 = (V2, V3)

e4 = (V2, V4)



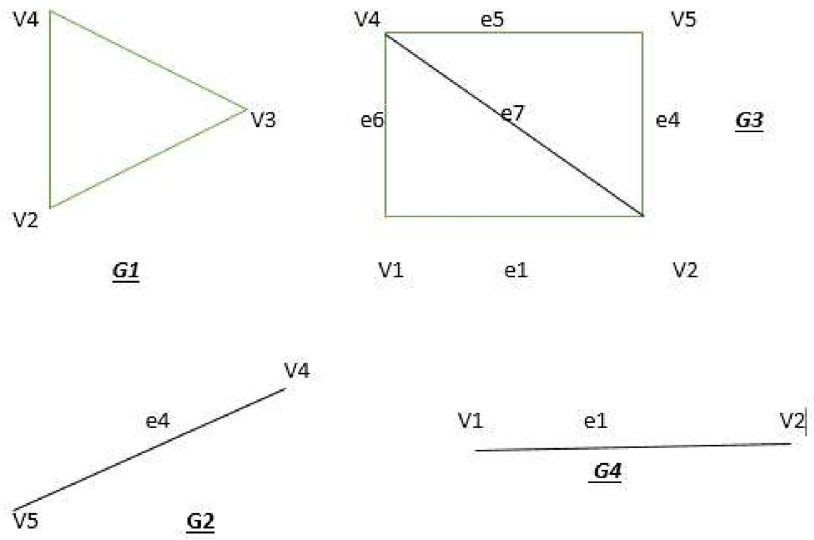
Subgraph

A graph G = (V1, E1) is called subgraph of a graph G(V, E) if V1(G) is a subset of V(G) and E1(G) is a subset of E(G) such that each edge of G1 has same end vertices as in G.



Types of Subgraph:

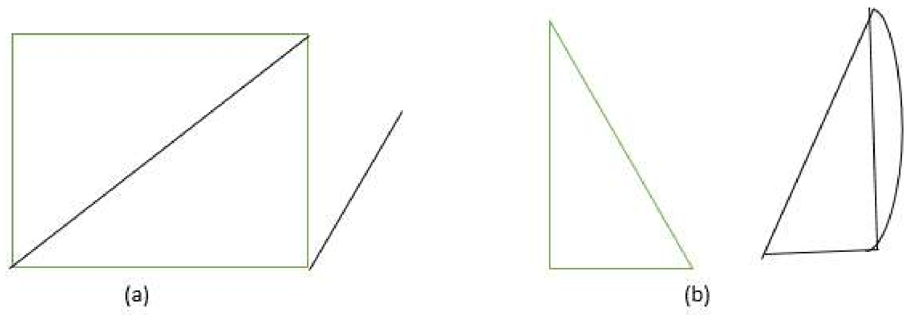
- **Vertex disjoint subgraph:** Any two graph G1 = (V1, E1) and G2 = (V2, E2) are said to be vertex disjoint of a graph G = (V, E) if V1(G1) intersection V2(G2) = null. In figure there is no common vertex between G1 and G2.
- **Edge disjoint subgraph:** A subgraph is said to be edge disjoint if E1(G1) intersection E2(G2) = null. In figure there is no common edge between G1 and G2.



Note: Edge disjoint subgraph may have vertices in common but vertex disjoint graph cannot have common edge, so vertex disjoint subgraph will always be an edge disjoint subgraph.

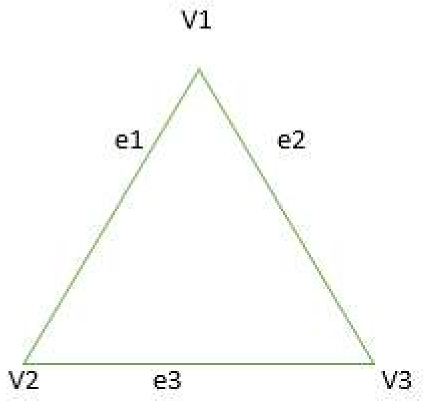
Connected or Disconnected Graph

A graph G is said to be connected if for any pair of vertices (Vi, Vj) of a graph G are reachable from one another. Or a graph is said to be connected if there exist at least one path between each and every pair of vertices in graph G, otherwise it is disconnected. A null graph with n vertices is disconnected graph consisting of n components. Each component consist of one vertex and no edge.



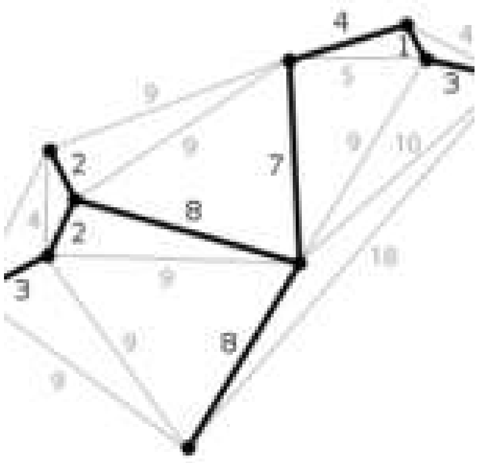
Cyclic Graph

A graph G consisting of n vertices and n> = 3 that is V1, V2, V3- – – – – – – – Vn and edges (V1, V2), (V2, V3), (V3, V4)- – – – – – – – — -(Vn, V1) are called cyclic graph.



Acyclic Graph

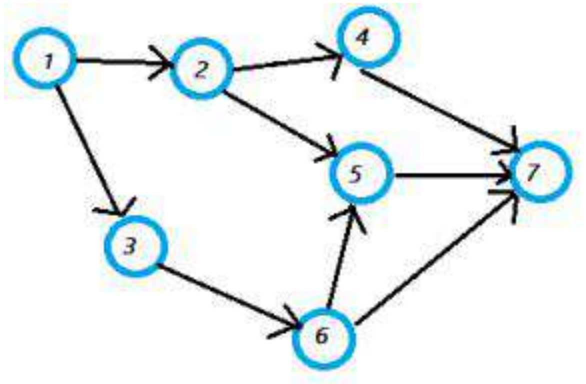
An acyclic graph is a graph without cycles (a cycle is a complete circuit). When following the graph from node to node, you will never visit the same node twice.

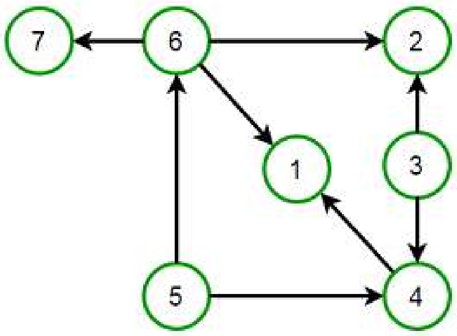


This graph (the thick black line) is acyclic, as it has no cycles (complete circuits). A connected acyclic graph, like the one above, is called a **tree**. If one or more of the tree “branches” is disconnected, the acyclic graph is a called a **forest**.

Directed Acyclic Graph

A directed acyclic graph is an acyclic graph that has a direction as well as a lack of cycles.





The parts of the above graph are:

- Vertices set = {1, 2, 3, 4, 5, 6, 7}.
- Edge set = {(1,2), (1,3), (2,4), (2,5), (3,6), (4,7), (5,7), (6,7)}.

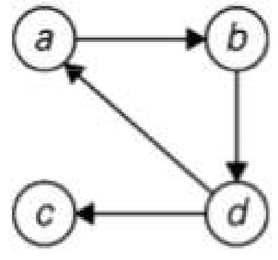
A directed acyclic graph has a topological ordering. This means that the nodes are ordered so that the starting node has a lower value than the ending node. A DAG has a unique topological ordering if it has a directed path containing all the nodes; in this case the ordering is the same as the order in which the nodes appear in the path.

In computer science, DAGs are also called *wait-for-graphs*. When a DAG is used to detect a deadlock, it illustrates that a resources has to *wait for* another process to continue. Directed acyclic graphs (DAGs) are used to model probabilities, connectivity, and causality. A "graph" in this sense means a structure made from nodes and edges.

7. Transitive closure and Warshall's algorithm

Transitive closure

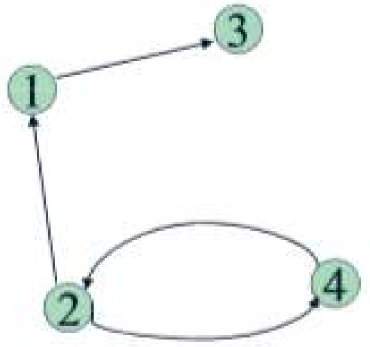
The **transitive closure** of a directed graph with *n* vertices can be defined as the *n*-by-*n* boolean matrix *T*, in which the element in the *i*th row and *j*th column is 1 if there exist a directed path from the *i*th vertex to the *j*th vertex, otherwise it is zero.



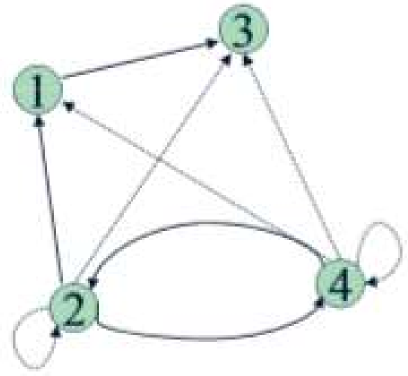
$$A = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array} \qquad T = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 1 & 1 \\ b & 1 & 1 & 1 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

(a) **(b)** **(c)**

Fig: (a) Digraph G (b) Adjacency matrix of G (c) Transitive closure



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

Figure: Another example of finding Transitive closure

Warshall's Algorithm

Warshall's algorithm uses the adjacency matrix to find the transitive closure of a directed graph.

Warshall's algorithm calculates the transitive closure by generating a sequence of n matrices, where n is the number of vertices.

$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}$

Recall that a path in a simple graph can be defined by a sequence of vertices.

The definition of the element at the ith row and jth column in the kth matrix ($R^{(k)}$), $r_{ij}^{(k)}$ is one if and only if there exist a path from v_i to v_j such that all the intermediate vertex, w_q is among the first k vertices, ie. $1 \le q \le k$.

The $R^{(0)}$ matrix represent paths without any intermediate vertices, so it is the adjacency matrix.

The $R^{(n)}$ matrix has ones if there is a path between the vertices with intermediate vertices from any of the n vertices of the graph, so it is the transitive closure.

Consider the case $r_{ij}^{(k)}$ is one and $r_{ij}^{(k\text{-}1)} = 0$. This can occur only if that there is an intermediate path through v_k from from v_i to v_j. More specifically the list of vertices has the form

v_i, w_q (where $1 \leq q < k$), v_k. w_q (where $1 \leq q < k$), v_j

This can happen only if $r_{ik}^{(k\text{-}1)} = r_{kj}^{(k\text{-}1)} = 1$. Note the k subscript.

If $r_{ij}^{(k\text{-}1)} = 1$ then $r_{ij}^{(k)}$ should be one.

In sumarry

$r_{ij}^{(k)} = r_{ij}^{(k\text{-}1)}$ **or** ($r_{ik}^{(k\text{-}1)}$ **and** $r_{kj}^{(k\text{-}1)}$)

That is-

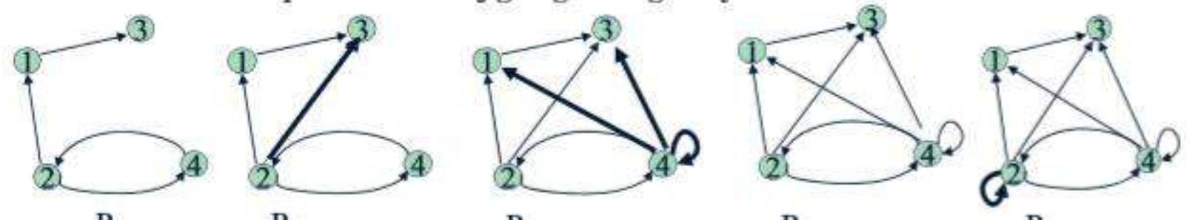
a. $r_{ij}^{(k)} = r_{ij}^{(k\text{-}1)}$:- means if an element in row i and j is 1 in $r_{ij}^{(k\text{-}1)}$, it remains 1 in $r_{ij}^{(k)}$. and
b. $r_{ij}^{(k)} = (r_{ik}^{(k\text{-}1)}$ **and** $r_{kj}^{(k\text{-}1)})$:- if an element in row row i and j is 0 in $r_{ij}^{(k\text{-}1)}$, it has to be changed to 1 in $r_{ij}^{(k)}$, if and only if the element in its row i and column k and the element in its row k and column j are both 1s in $r_{ij}^{(k\text{-}1)}$.

```
Algorithm Warshall(A[1...n, 1...n]) // A is the adjacency matrix
    //Input: adjacency matrix A of a digraph with n vertices
    //Output: transitive closure of the digraph
    $R^{(0)} \leftarrow A$
    for $k \leftarrow 1$ to $n$ do
        for $i \leftarrow 1$ to $n$ do
            for $j \leftarrow$ to $n$ do
                $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ or $(R^{(k-1)}[i, k]$ and $R^{(k-1)}[k, j])$
    return $R^{(n)}$
```

The worst case cost is $\Theta(n^3)$, so it is not better than the brute force algorithm.

In fact, for a sparse graph the brute force algorithm is faster.

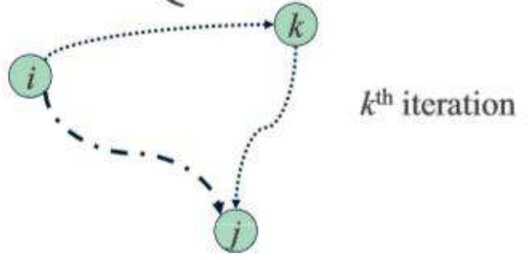
- **Main idea: a path exists between two vertices i, j, iff**
  - **there is an edge from i to j; or**
  - **there is a path from i to j going through vertex 1; or**
  - **there is a path from i to j going through vertex 1 and/or 2; or**
  - **there is a path from i to j going through vertex 1, 2, and/or 3; or**
  - **...**
  - **there is a path from i to j going through any of the other vertices**



$$R_0 = \begin{matrix} 0&0&1&0\\ 1&0&0&1\\ 0&0&0&0\\ 0&1&0&0 \end{matrix} \qquad R_1 = \begin{matrix} 0&0&1&0\\ 1&0&\mathbf{1}&1\\ 0&0&0&0\\ 0&1&0&0 \end{matrix} \qquad R_2 = \begin{matrix} 0&0&1&0\\ 1&0&1&1\\ 0&0&0&0\\ \mathbf{1}&1&\mathbf{1}&\mathbf{1} \end{matrix} \qquad R_3 = \begin{matrix} 0&0&1&0\\ 1&0&1&1\\ 0&0&0&0\\ 1&1&1&1 \end{matrix} \qquad R_4 = \begin{matrix} 0&0&1&0\\ 1&\mathbf{1}&1&1\\ 0&0&0&0\\ 1&1&1&1 \end{matrix}$$

- **On the *k*th iteration, the algorithm determine if a path exists between two vertices *i, j* using just vertices among 1,…,*k* allowed as intermediate**

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-}1) \\ \quad\text{or} & \\ (R^{(k-1)}[i,k] \;\text{ and }\; R^{(k-1)}[k,j]) & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } i \text{ using just } 1,\ldots,k\text{-}1) \end{cases}$$



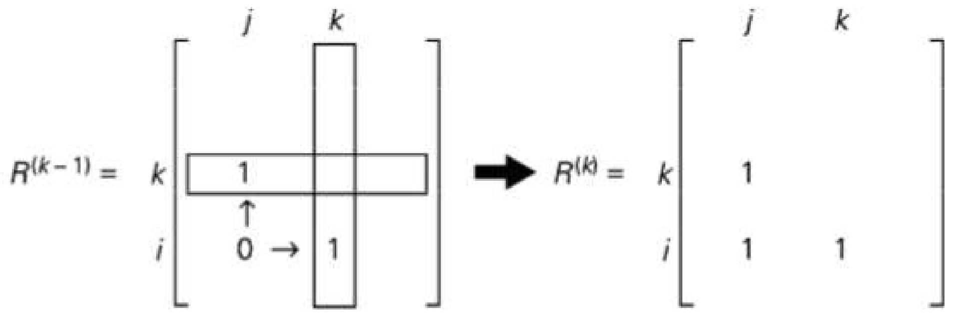
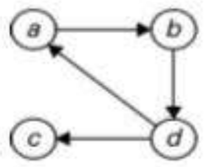


Figure: Rule for changing zeros to ones in Warshall's algorithm



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.4 Application of Warshall's algorithm to the digraph shown. New ones are in bold.

8. Graphs traversal

Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a graph ***traversal*** and is similar in concept to a ***tree traversal***. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain might consist of a large collection of states, with connections between various pairs of states. Solving this sort of problem requires getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it might not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph might contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

Graph traversal algorithms can solve both of these problems by flagging vertices as `VISITED` when appropriate. At the beginning of the algorithm, no vertex is flagged as `VISITED`. The flag for a vertex is set when the vertex is first visited during the traversal. If a flagged vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking whether they have the `VISITED` flag set. If not all vertices are flagged, we can continue the traversal from another unvisited vertex. Note that this process works regardless of whether the graph is directed or undirected.

Using graph traversal we visit all the vertices of the graph without getting into looping path. There are two graph traversal techniques and they are as follows...

a) **BFS (Breadth First Search)**
b) **DFS (Depth First Search)**

BFS (Breadth-First Search) Traversal

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom.

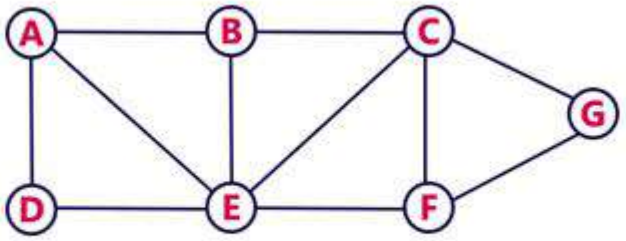
We use the following steps to implement BFS traversal...

- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

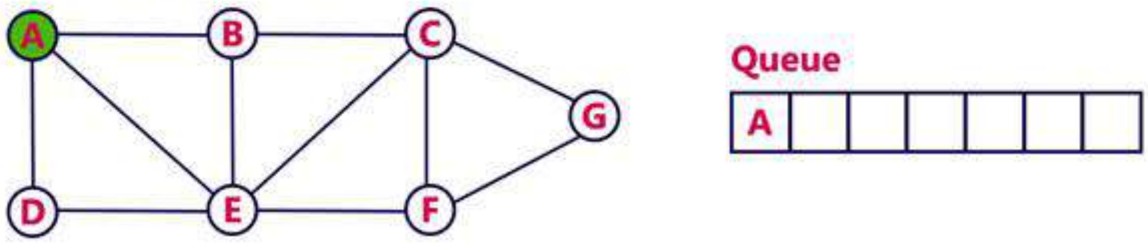
Example:

Consider the following example graph to perform BFS traversal



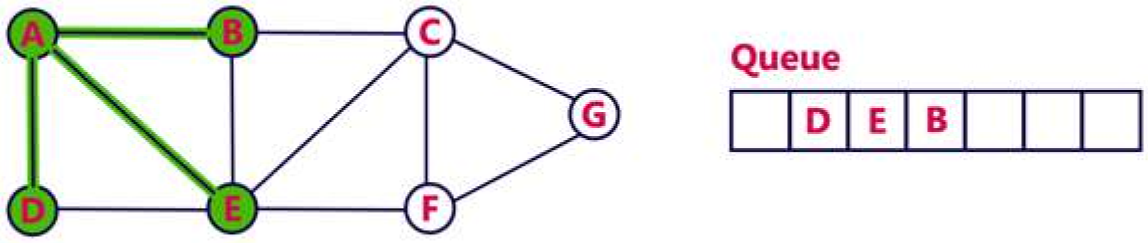
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



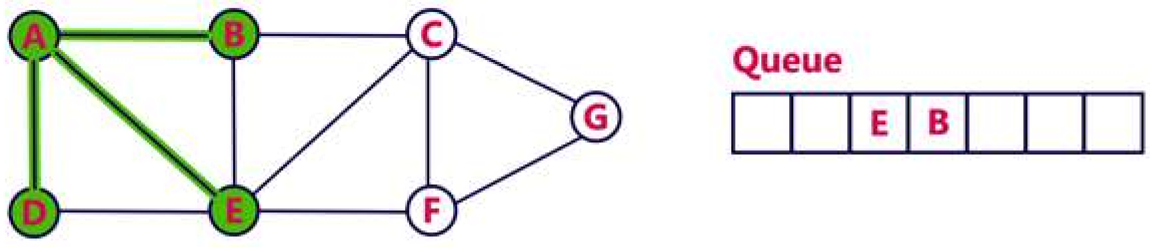
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



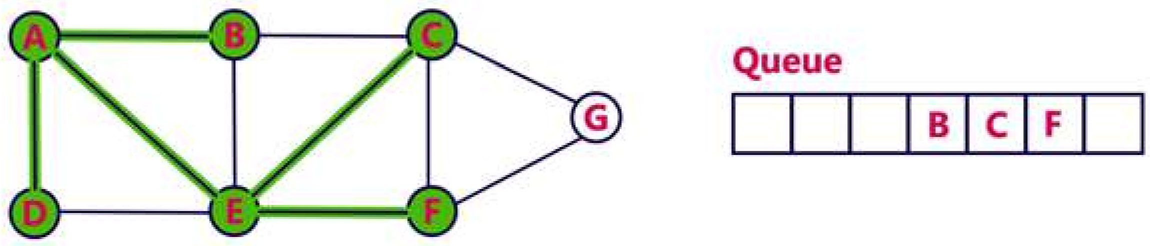
Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



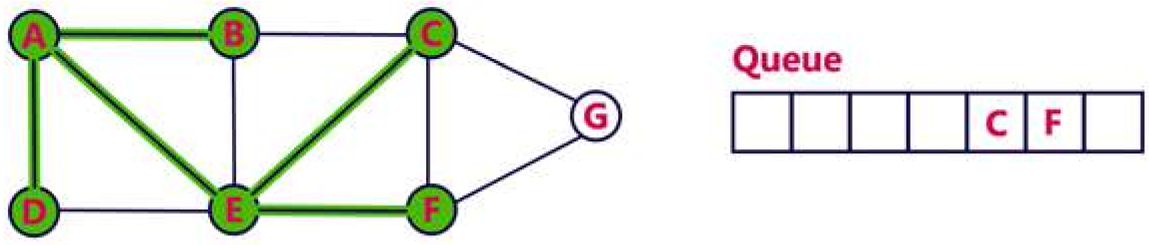
Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



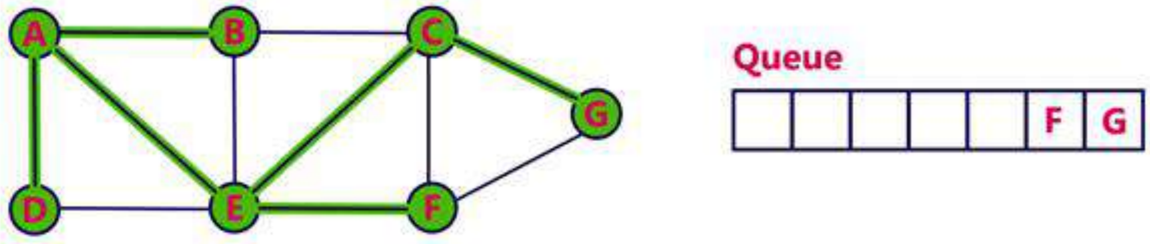
Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



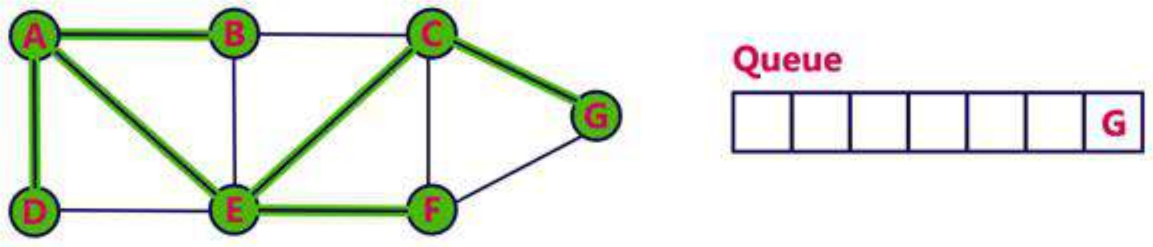
Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



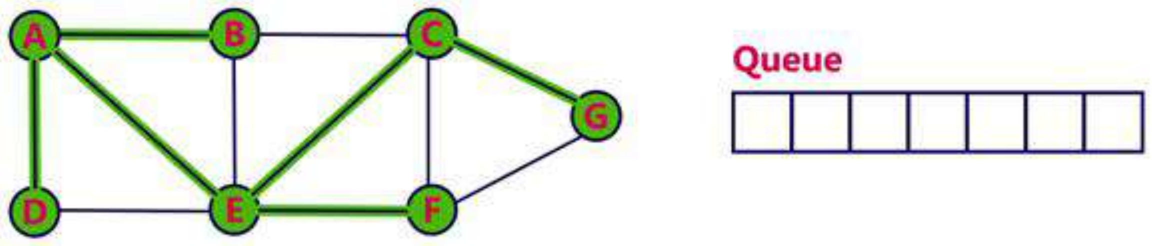
Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

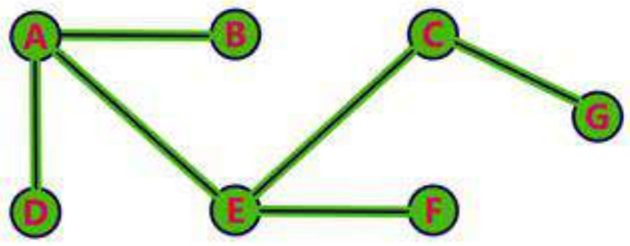


Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Program to print BFS traversal from a given

```
// source vertex. BFS(int s) traverses vertices
// reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using
// adjacency list representation
class Graph
```

```
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;
public:
    Graph(int V);  // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);
};

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

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent
    // vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued
        // vertex s. If a adjacent has not been visited,
        // then mark it visited and enqueue it
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);
    return 0;
}
```

DFS (Depth-First Search) Traversal

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops.

We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

Whenever a vertex *v* is visited during the search, DFS will recursively visit all of *v* 's unvisited neighbors.

Equivalently, DFS will add all edges leading out of *v* to a stack. The next vertex to be visited is determined by popping the stack and following that edge.

The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a ***depth-first search tree***.

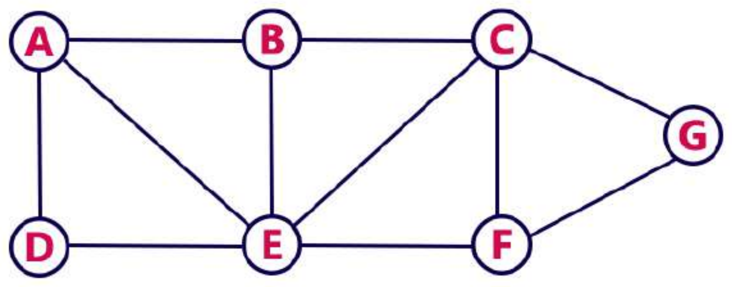
This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs.

We use the following steps to implement DFS traversal:

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

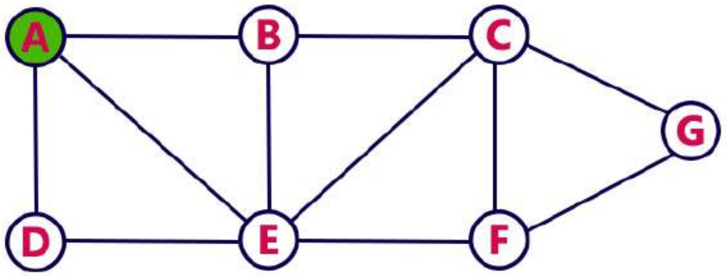
Example

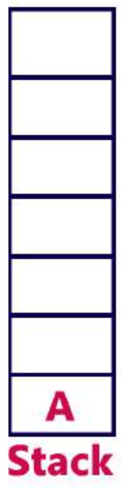
Consider the following example graph to perform DFS traversal



Step 1:

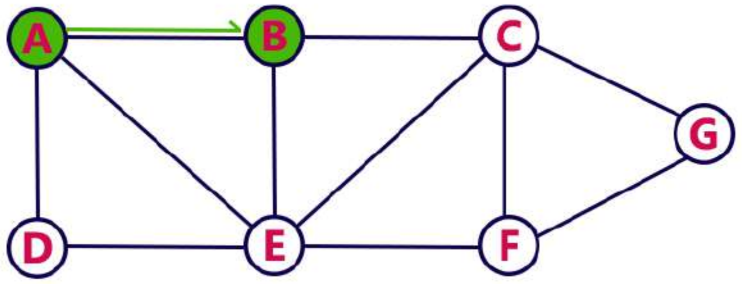
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

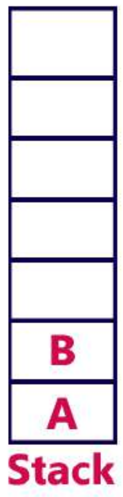




Step 2:

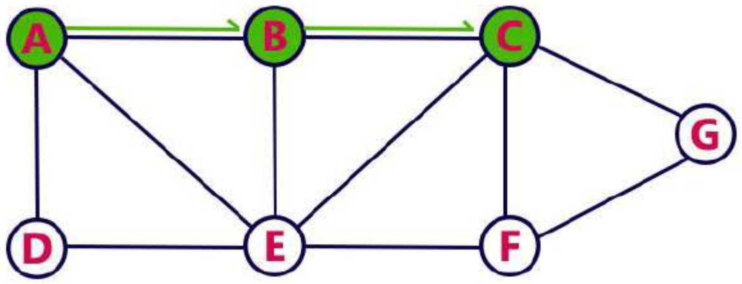
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

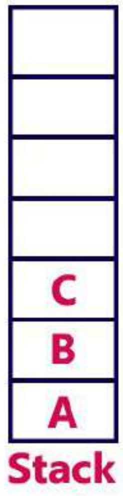




Step 3:

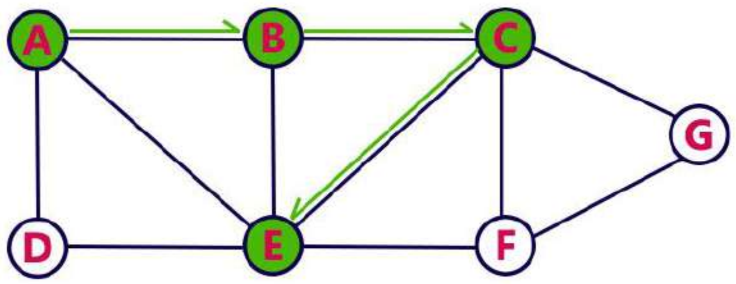
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

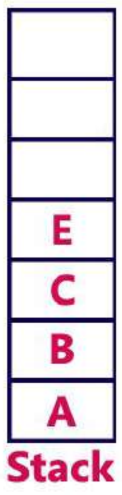




Step 4:

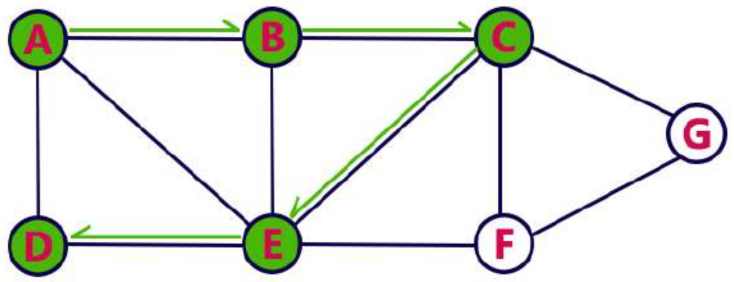
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

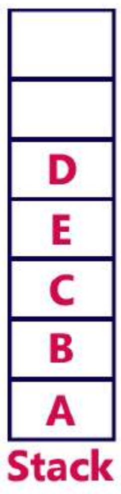




Step 5:

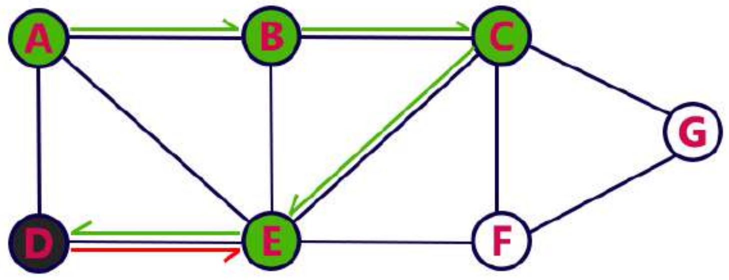
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

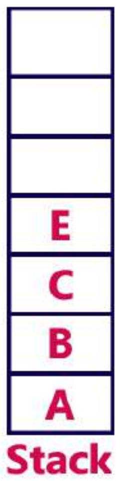




Step 6:

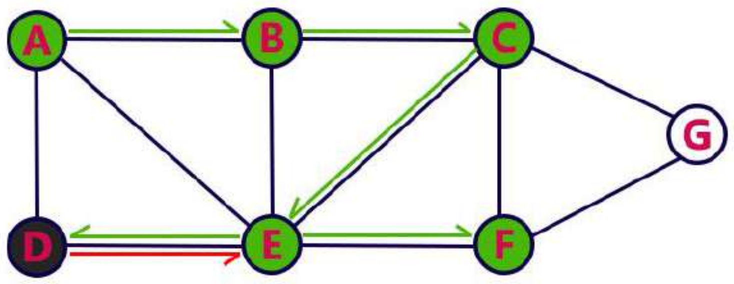
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

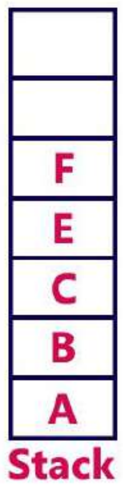




Step 7:

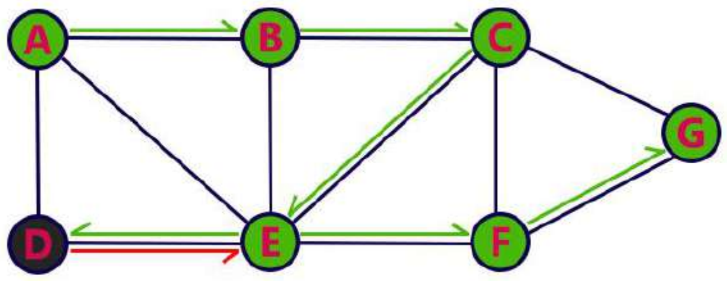
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

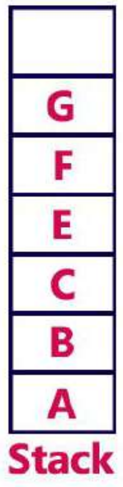




Step 8:

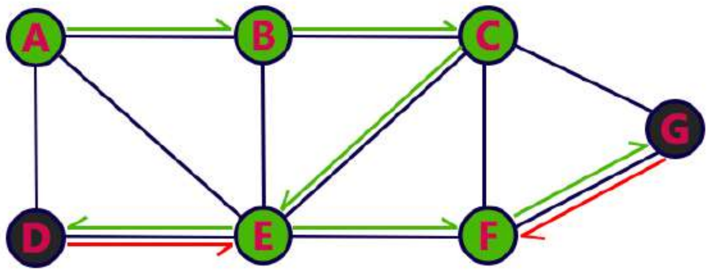
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

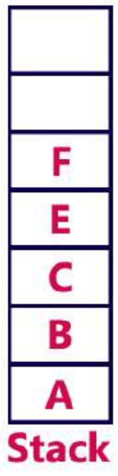




Step 9:

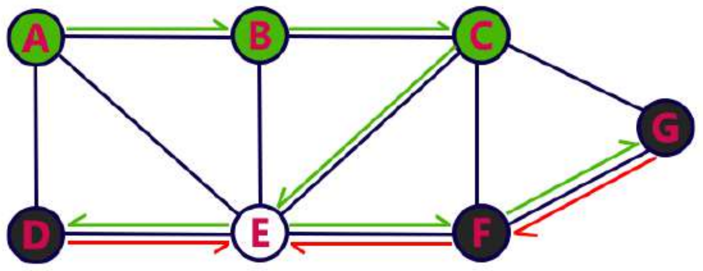
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

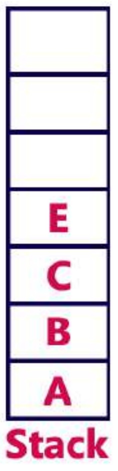




Step 10:

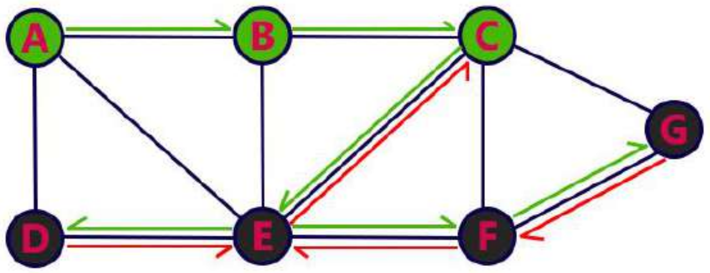
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

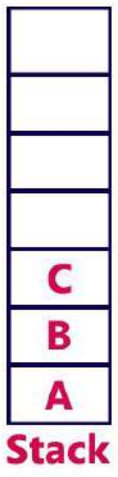




Step 11:

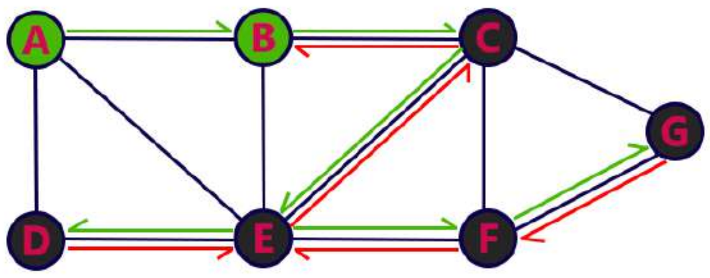
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

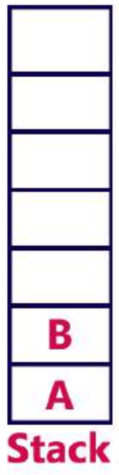




Step 12:

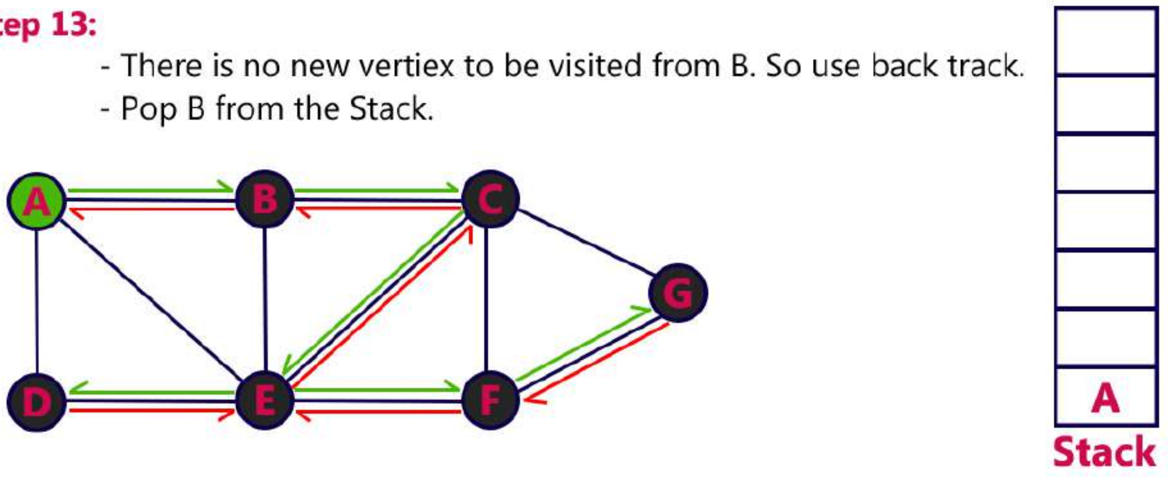
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.





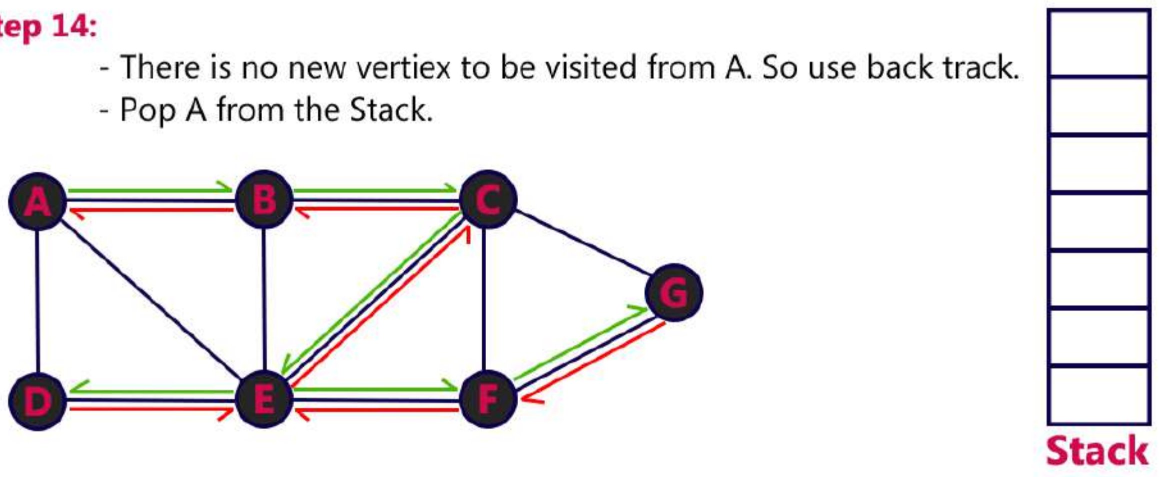
Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

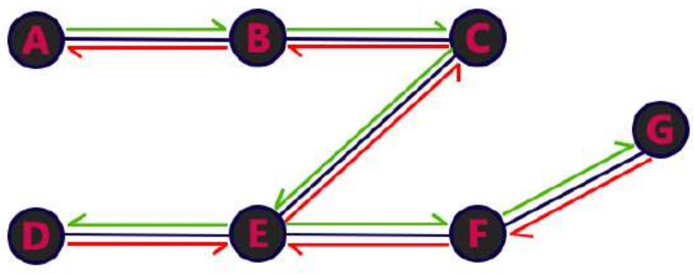


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



C++ program to print DFS traversal from

```
// a given vertex in a  given graph
#include<bits/stdc++.h>
using namespace std;
```

```
// Graph class represents a directed graph
// using adjacency list representation
class Graph
{
    int V;    // No. of vertices

    // Pointer to an array containing
    // adjacency lists
    list<int> *adj;

    // A recursive function used by DFS
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);   // Constructor

    // function to add an edge to graph
    void addEdge(int v, int w);

    // DFS traversal of the vertices
    // reachable from v
    void DFS(int v);
};

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

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and
    // print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices adjacent
    // to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
```

```
            DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function
    // to print DFS traversal
    DFSUtil(v, visited);
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"
            " (starting from vertex 2) \n";
    g.DFS(2);

    return 0;
}
```

9. Spanning Tree and Minimum Cost Spanning Tree

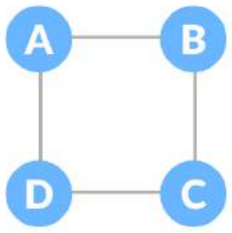
A spanning tree is a sub-graph of an undirected and a connected graph, which includes all the vertices of the graph having a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.

The edges may or may not have weights assigned to them.

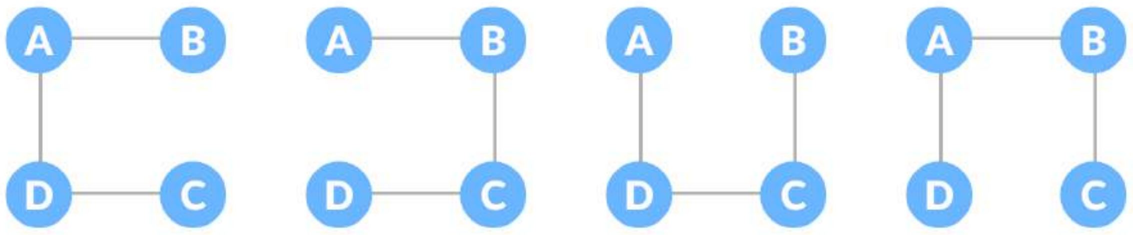
The total number of spanning trees with n vertices that can be created from **a complete graph** is equal to $n^{(n-2)}$.

Example of a Spanning Tree

Let's understand the spanning tree with examples below. Let a graph:



The spanning trees that can be created from the above graph are:



General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

1. A connected graph G can have more than one spanning tree.
2. All possible spanning trees of graph G, have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Mathematical Properties of Spanning Tree

1. Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
2. From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
3. A complete graph can have maximum $\mathbf{n}^{n-2}$ number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Spanning Tree Applications

1. Computer Network Routing Protocol
2. Cluster Analysis
3. Civil Network Planning

Minimum Spanning Tree (Minimum Cost Spanning Tree)

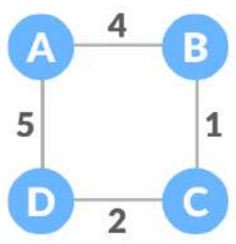
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

The ***minimal-cost spanning tree*** (MCST) problem takes as input a connected, undirected graph **G**G, where each edge has a distance or weight measure attached. The MCST is the graph containing the vertices of **G** along with the subset of **G** 's edges that (1) has minimum total cost as measured by summing the values for all of the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

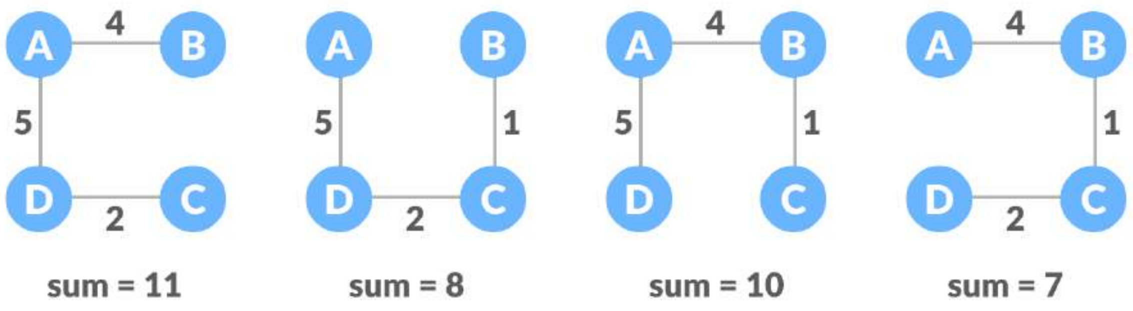
The MCST contains no cycles. If a proposed MCST did have a cycle, a cheaper MCST could be had by removing any one of the edges in the cycle. Thus, the MCST is a free tree with $|\mathbf{V}|-1$ edges. The name "minimum-cost spanning tree" comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost.

Example of a Minimum Cost Spanning Tree

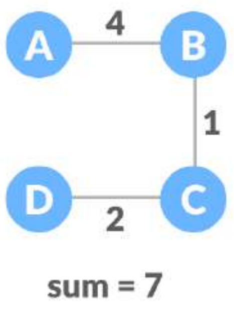
Let a graph as:



The possible spanning trees from the above graph are:



The minimum spanning tree from the above spanning trees is:



Minimum Spanning tree Applications

1. To find paths in the map
2. To design networks like telecommunication networks, water supply network and electrical grids.

The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

Prim's Algorithm

Prim's algorithm is used to find a minimum cost spanning tree. This algorithm takes a graph as input and finds the subset of the edges of that graph which

1. form a tree that includes every vertex
2. has the minimum sum of weights among all the trees that can be formed from the graph

Prim's algorithm to find minimum cost spanning tree uses the greedy approach which finds the local optimum in the hopes of finding a global optimum.

Prim's algorithm starts from one vertex and keep adding edges with the lowest weight until we we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

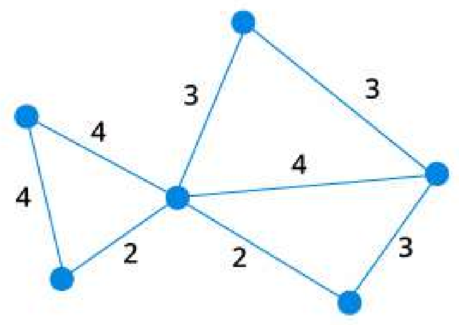
Prim's algorithm is very simple. Start with any Vertex *N*N in the graph, setting the MCST to be *N* initially. Pick the least-cost edge connected to *N*. This edge connects *N* to another vertex; call this *M*. Add Vertex *M* and Edge (*N*,*M*) to the MCST. Next, pick the least-cost edge coming from either *N* or *M* to any other vertex in the graph. Add this edge and the new vertex it reaches to the MCST. This process continues, at each step expanding the MCST by selecting the least-cost edge from a vertex currently in the MCST to a vertex not currently in the MCST.

Prim's algorithm is quite similar to Dijkstra's algorithm for finding the single-source shortest paths. The primary difference is that we are seeking not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MCST.

Example of Prim's algorithm

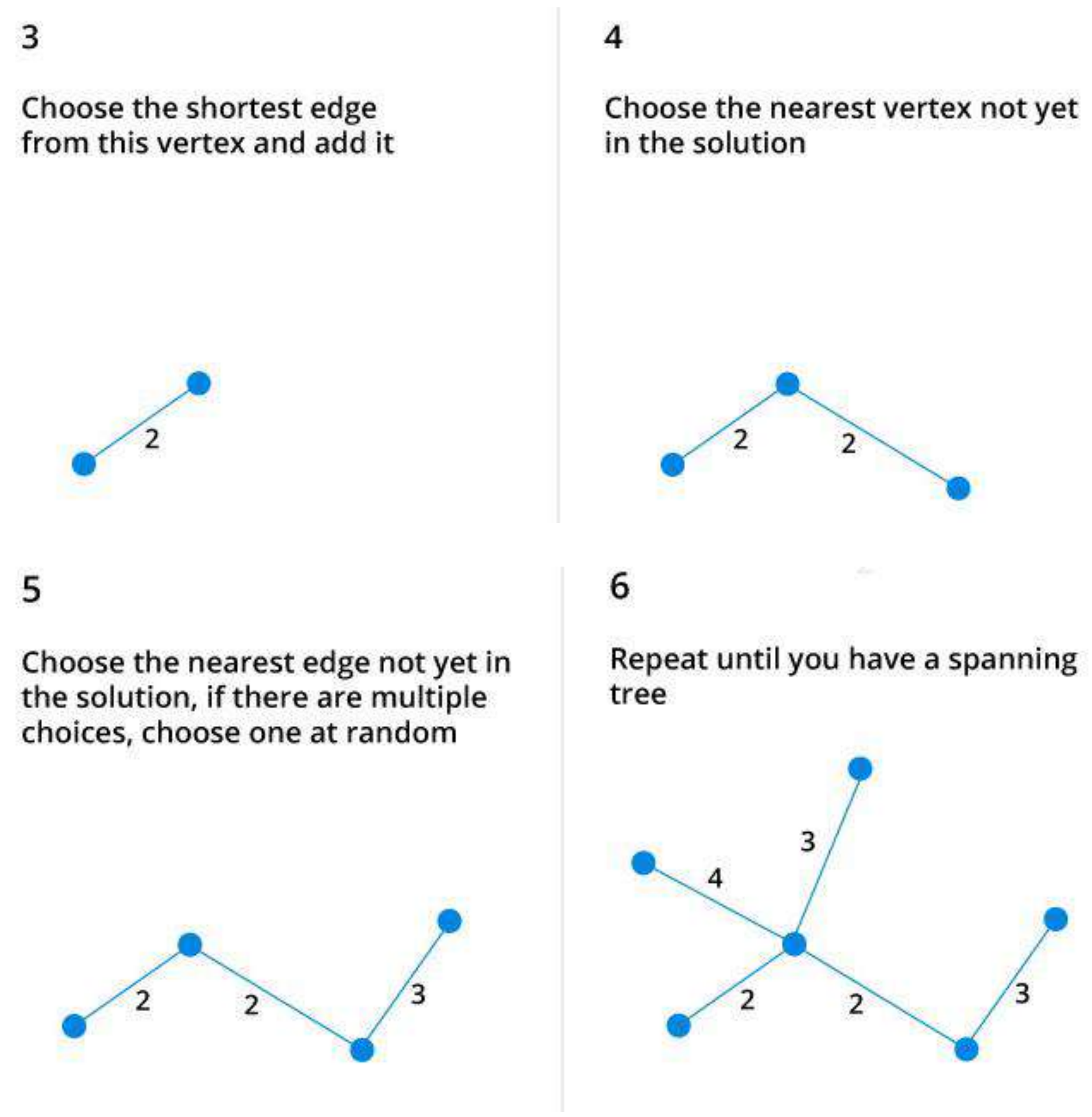
1

Start with a weighted graph



2

Choose a vertex


3
Choose the shortest edge from this vertex and add it
2
4
Choose the nearest vertex not yet in the solution
2
2
5
Choose the nearest edge not yet in the solution, if there are multiple choices, choose one at random
2
2
3
6
Repeat until you have a spanning tree
3
4
2
2
3

A C++ program for Prim's Minimum

```
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<"
\n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices not yet included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first vertex.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V - 1; count++)
    {
        // Pick the minimum key vertex from the
        // set of vertices not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.
        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices
of m
            // mstSet[v] is false for vertices not yet included in
MST
            // Update the key only if graph[u][v] is smaller than
key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v]
< key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

int main()
{
    /* Let us create the following graph
        2  3
    (0)--(1)--(2)
    |  / \  |
    6| 8/ \5 |7
    |  / \  |
    (3)-------(4)
            9     */
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    // Print the solution
    primMST(graph);

    return 0;
```

```
}
```

Prim's vs Kruskal's Algorithm

Kruskal's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

Kruskal's Algorithm

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

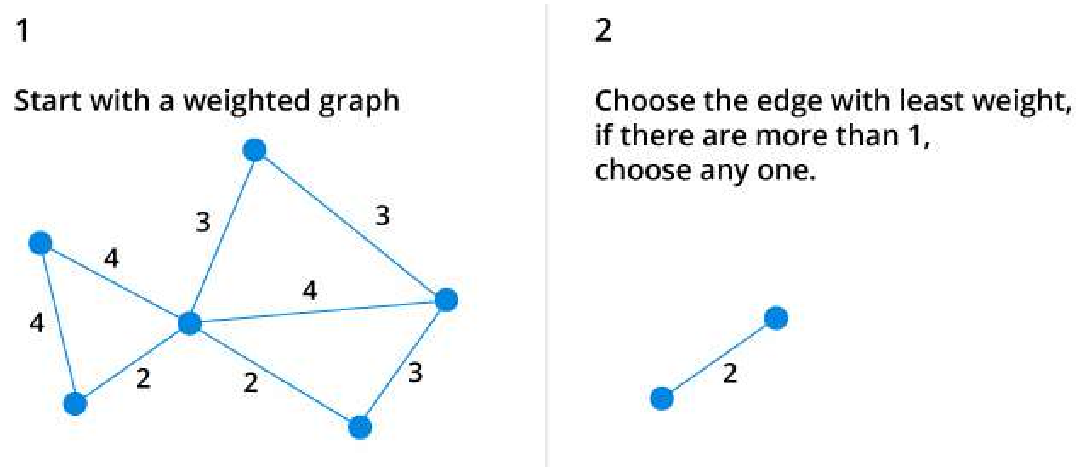
1. form a tree that includes every vertex
2. has the minimum sum of weights among all the trees that can be formed from the graph

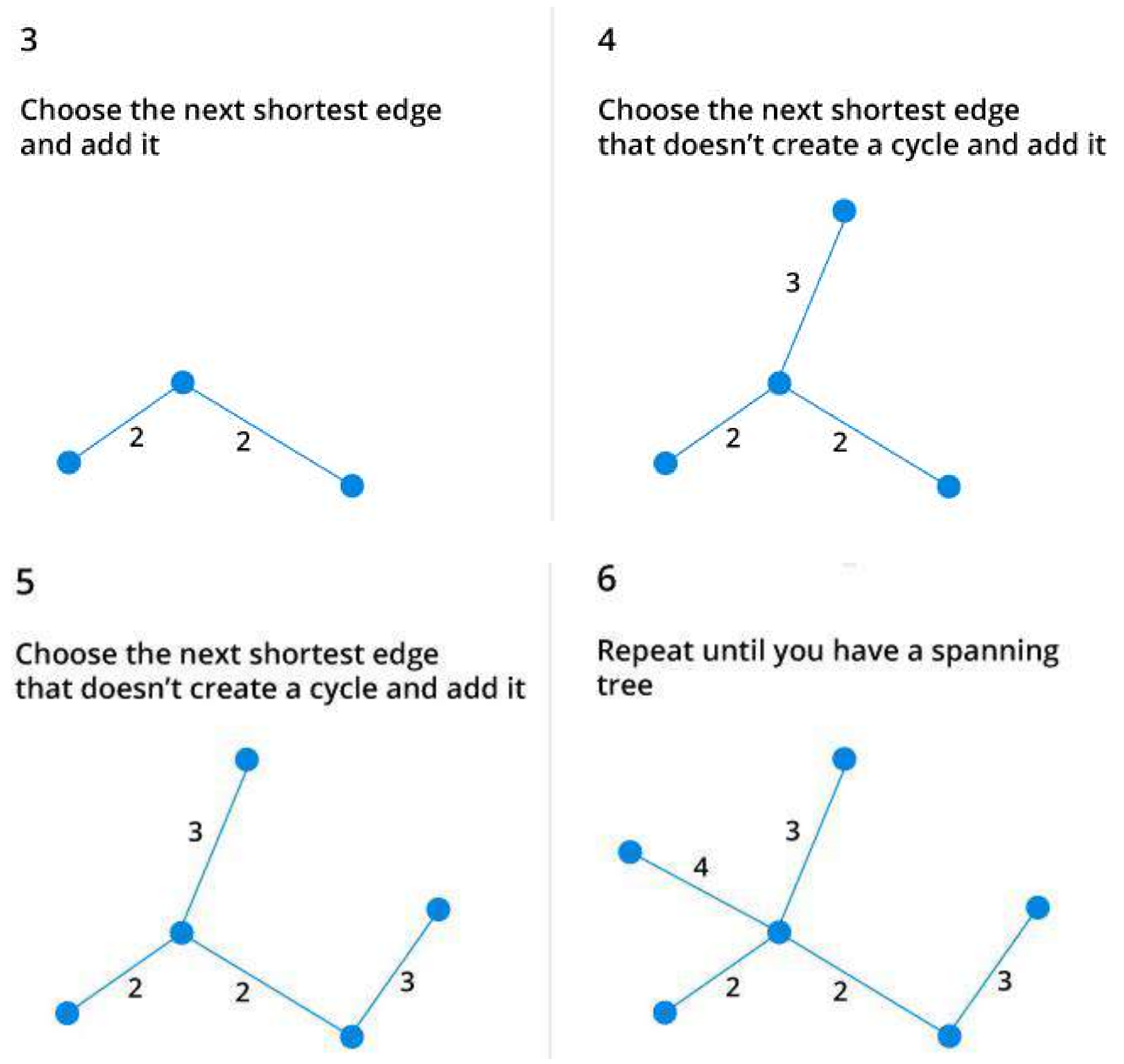
It is a greedy algorithms which find the local optimum in the hopes of finding a global optimum. This algorithm starts from an edge with the lowest weight and keep adding edges until a minimum spanning tree is built.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm


1
Start with a weighted graph
3
3
4
4
4
2
2
3
2
Choose the edge with least weight, if there are more than 1, choose any one.
2


3
Choose the next shortest edge and add it
2
2
4
Choose the next shortest edge that doesn't create a cycle and add it
3
2
2
5
Choose the next shortest edge that doesn't create a cycle and add it
3
2
2
3
6
Repeat until you have a spanning tree
4
3
2
2
3

Kruskal's algorithm is also a simple, greedy algorithm. First partition the set of vertices into |V| ***disjoint sets***, each consisting of one vertex. Then process the edges in order of weight. An edge is added to the MCST, and two disjoint sets combined, if the edge connects two vertices in different disjoint sets. This process is repeated until only one disjoint set remains.

The edges can be processed in order of weight by using a min-heap. This is generally faster than sorting the edges first, because in practice we need only visit a small fraction of the edges before completing the MCST.

C++ program to implement Kruskal's algorithm

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
#define edge pair<int,int>

class Graph {
private:
    vector<pair<int, edge>> G; // graph
    vector<pair<int, edge>> T; // mst
    int *parent;
    int V; // number of vertices/nodes in graph
public:
    Graph(int V);
    void AddWeightedEdge(int u, int v, int w);
    int find_set(int i);
    void union_set(int u, int v);
    void kruskal();
    void print();
};
Graph::Graph(int V) {
    parent = new int[V];

    //i 0 1 2 3 4 5
    //parent[i] 0 1 2 3 4 5
    for (int i = 0; i < V; i++)
        parent[i] = i;

    G.clear();
    T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
    G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
    // If i is the parent of itself
    if (i == parent[i])
        return i;
    else
        // Else if i is not the parent of itself
        // Then i is not the representative of his set,
        // so we recursively call Find on its parent
        return find_set(parent[i]);
}

void Graph::union_set(int u, int v) {
    parent[u] = parent[v];
}
void Graph::kruskal() {
    int i, uRep, vRep;
    sort(G.begin(), G.end()); // increasing weight
    for (i = 0; i < G.size(); i++) {
        uRep = find_set(G[i].second.first);
        vRep = find_set(G[i].second.second);
        if (uRep != vRep) {
            T.push_back(G[i]); // add to tree
            union_set(uRep, vRep);
        }
    }
}
void Graph::print() {
    cout << "Edge :" << " Weight" << endl;
    for (int i = 0; i < T.size(); i++) {
        cout << T[i].second.first << " - " << T[i].second.second << ":"
                << T[i].first;
        cout << endl;
    }
}
int main() {
    Graph g(6);
    g.AddWeightedEdge(0, 1, 4);
    g.AddWeightedEdge(0, 2, 4);
    g.AddWeightedEdge(1, 2, 2);
    g.AddWeightedEdge(1, 0, 4);
    g.AddWeightedEdge(2, 0, 4);
    g.AddWeightedEdge(2, 1, 2);
    g.AddWeightedEdge(2, 3, 3);
    g.AddWeightedEdge(2, 5, 2);
    g.AddWeightedEdge(2, 4, 4);
    g.AddWeightedEdge(3, 2, 3);
    g.AddWeightedEdge(3, 4, 3);
    g.AddWeightedEdge(4, 2, 4);
    g.AddWeightedEdge(4, 3, 3);
    g.AddWeightedEdge(5, 2, 2);
    g.AddWeightedEdge(5, 4, 3);
    g.kruskal();
    g.print();
    return 0;
}
```

Kruskal's vs Prim's Algorithm

Prim's algorithm is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

Difference between Prim’s and Kruskal’s algorithm for MST

Both **Prim’s and Kruskal’s algorithm** finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few **major differences between them**.

Prim’s Algorithm	**Kruskal’s Algorithm**

It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm has a time complexity of O(V^2), V being the number of vertices.	Kruskal's algorithm's time complexity is O(logV), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

10. Shortest-path algorithm

Shortest-Paths Problems

On a road map, a road connecting two towns is typically labeled with its distance. We can model a road network as a directed graph whose edges are labeled with real numbers. These numbers represent the distance (or other cost metric, such as travel time) between two vertices. These labels may be called ***weights***, ***costs***, or ***distances***, depending on the application. Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices. This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices.

For example, in Figure 14.5.1, the cost of the path from A to B to D is 15. The cost of the edge directly from A to D is 20. The cost of the path from A to C to B to D is 10. Thus, the shortest path from A to D is 10 (rather than along the edge connecting A to D). We use the notation d(A,D)=10 to indicate that the shortest distance from A to D is 10. In Figure 14.5.1, there is no path from E to B, so we set d(E,B)=∞. We define w(A,D)=20 to be the weight of edge (A,D), that is, the weight of the direct connection from A to D. Because there is no edge from E to B ,w(E,B)=∞. Note that w(D,A)=∞ because the graph of Figure 14.5.1 is directed. We assume that all weights are positive.

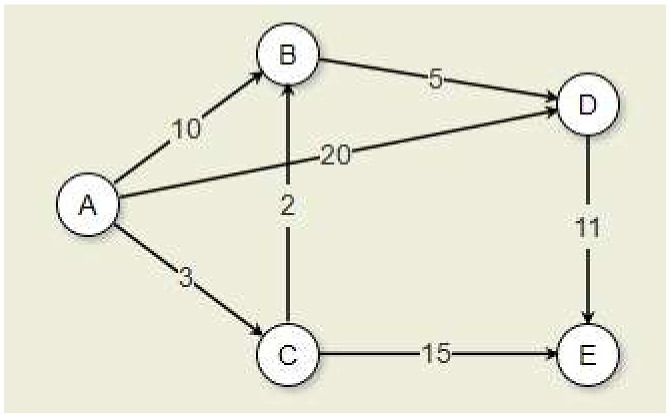


Figure 14.5.1: Example graph for shortest-path definitions.

Single-Source Shortest Paths

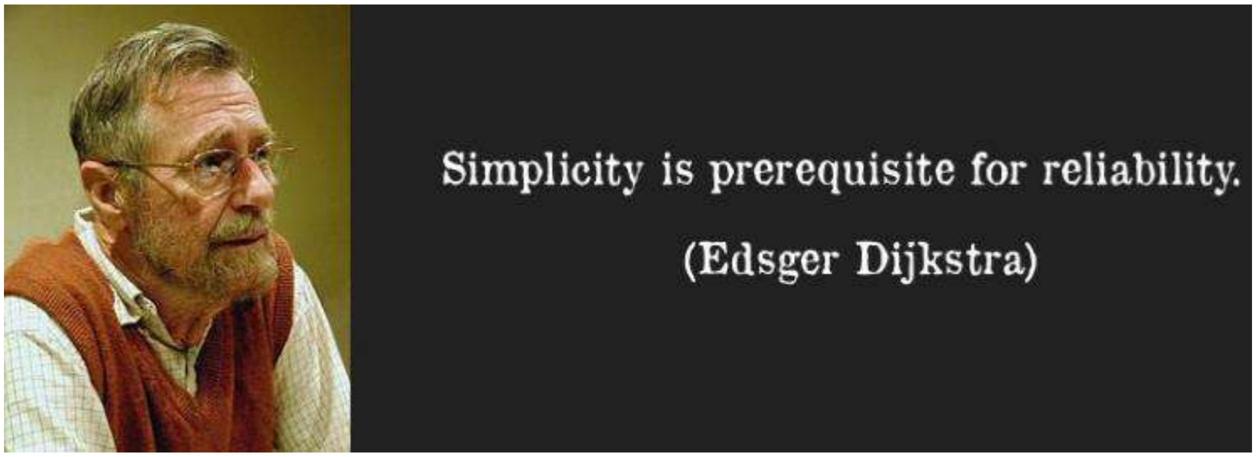
We will now present an algorithm to solve the ***single-source shortest paths problem***. Given Vertex S in Graph **G**, find a shortest path from S to every other vertex in **G**. We might want only the shortest path between two vertices, SS and T. However in the worst case, finding the shortest path from S to T requires us to find the shortest paths from S to every other vertex as well. So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices. The algorithm described here will only compute the distance to every such vertex, rather than recording the actual path.

Computer networks provide an application for the single-source shortest-paths problem. The goal is to find the cheapest way for one computer to broadcast a message to all other computers on the network. The network can be modeled by a graph with edge weights indicating time or cost to send a message to a neighboring computer.

Dijkstra's algorithm

Dijkstra's algorithm finds the shortest path between any two vertices of a graph. It is a solution to the single-source shortest path problem in graph theory

- Both directed and undirected graphs
- All edges must have nonnegative weights
- Graph must be connected



(*Edsger Wybe Dijkstra- May 11, 1930 – August 6, 2002 Dutch computer scientist from Netherlands Received the 1972 A. M. Turing Award, widely considered the most prestigious award in computer science Known for his many essays on programming*)

Dijkstra’s algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

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 neighbours to find the shortest subpath to those neighbours.

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.

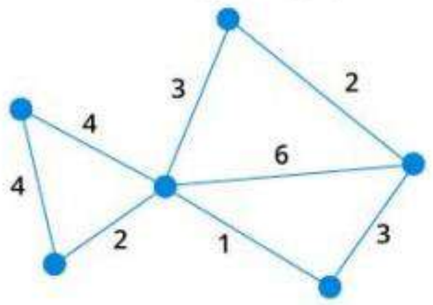
Here's a description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node `C`.
3. For each neighbour `N` of your current node `C`: add the current distance of `C` with the weight of the edge connecting `C`->`N`. If it's smaller than the current distance of `N`, set it as the new current distance of `N`.
4. Mark the current node `C` as visited.
5. If there are non-visited nodes, go to step 2.

Example of Dijkstra's algorithm

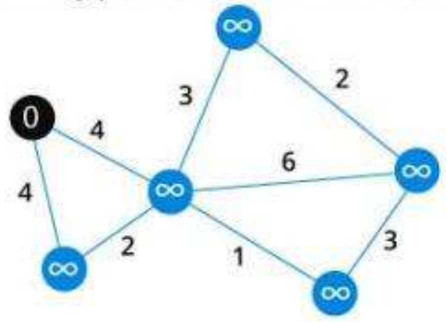
1

Start with a weighted graph



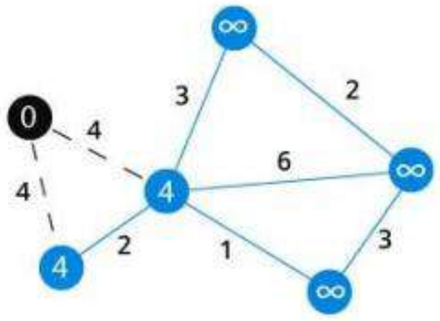
2

Choose a starting vertex and assign infinity path values to all other vertices



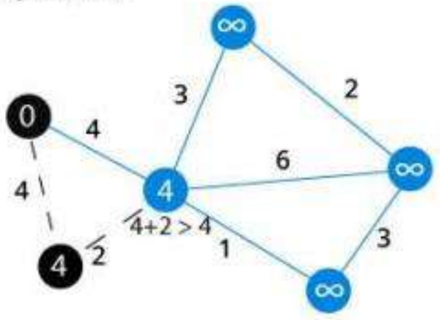
3

Go to each vertex adjacent to this vertex and update its path length



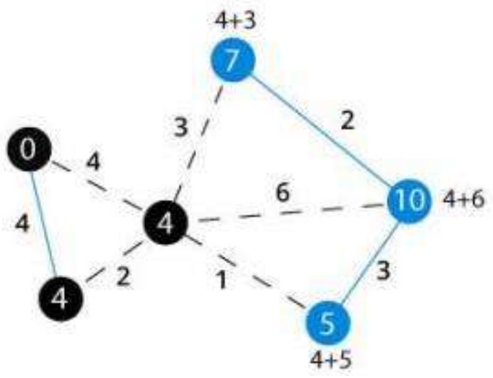
4

If the path length of adjacent vertex is lesser than new path length, don't update it.



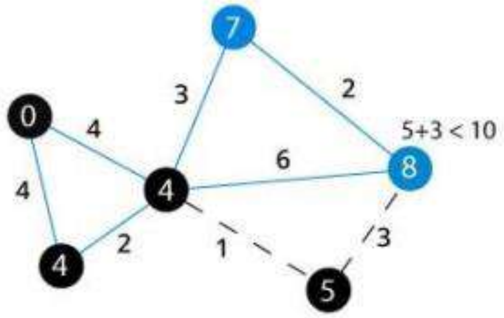
5

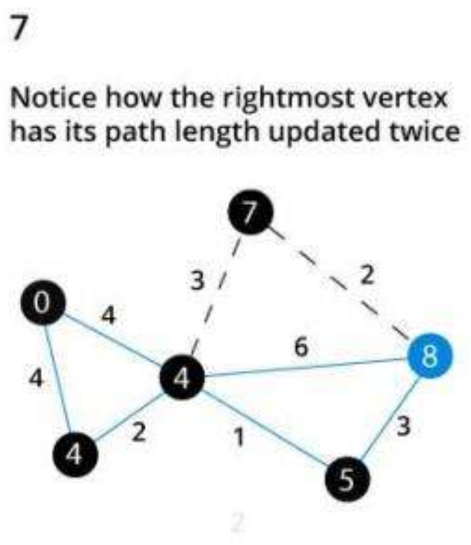
Avoid updating path lengths of already visited vertices

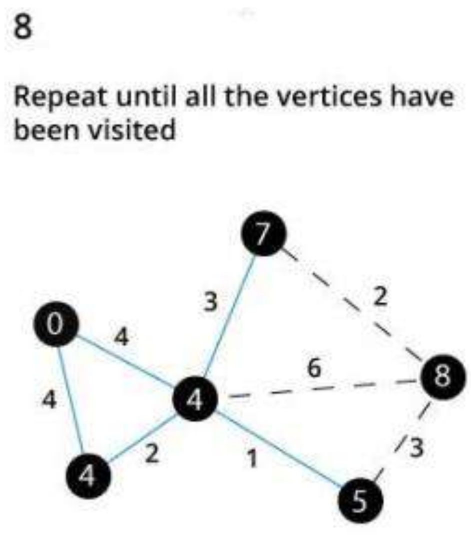


6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7




7
Notice how the rightmost vertex has its path length updated twice
0
4
4
4
2
4
7
3
2
8
6
1
3
5



Time Complexity of Dijkstra's Algorithm is O (V^2) but with min-priority queue it drops down to O (V + E log V).

Algorithm Steps using min-priority queue:

1. Set all vertices distances = infinity except for the source vertex, set the source distance = 0.
2. Push the source vertex in a min-priority queue in the form (distance , vertex), as the comparison in the min-priority queue will be according to vertices distances.
3. Pop the vertex with the minimum distance from the priority queue (at first the popped vertex = source).
4. Update the distances of the connected vertices to the popped vertex in case of "current vertex distance + edge weight < next vertex distance", then push the vertex with the new distance to the priority queue.
5. If the popped vertex is visited before, just continue without using it.
6. Apply the same algorithm again until the priority queue is empty.

Implementing Algorithm

It works by maintaining a distance estimate **D**(*X*) for all vertices *X* in **V**. The elements of **D** are initialized to the value `INFINITE`. Vertices are processed in order of distance from *S*. Whenever a vertex *v* is processed, **D**(*X*) is updated for every neighbor *X* of *V*. Here is an implementation for Dijkstra's algorithm. At the end, array `D` will contain the shortest distance values.

```
// Compute shortest path distances from s, store them in D
static void Dijkstra(Graph G, int s, int[] D) {
  for (int i=0; i<G.nodeCount(); i++)    // Initialize
```

```
    D[i] = INFINITY;
  D[s] = 0;
  for (int i=0; i<G.nodeCount(); i++) {  // Process the vertices
    int v = minVertex(G, D);     // Find next-closest vertex
    G.setValue(v, VISITED);
    if (D[v] == INFINITY) return; // Unreachable
    int[] nList = G.neighbors(v);
    for (int j=0; j<nList.length; j++) {
      int w = nList[j];
      if (D[w] > (D[v] + G.weight(v, w)))
        D[w] = D[v] + G.weight(v, w);
    }
  }
}
```

All-Pairs Shortest Paths

We next consider the problem of finding the shortest distance between all pairs of vertices in the graph, called the ***all-pairs shortest paths problem***.

Floyd–Warshall's Algorithm

Floyd–Warshall's Algorithm is used to find the shortest paths between all pairs of vertices in a graph, where each edge in the graph has a weight which is positive or negative. The biggest advantage of using this algorithm is that all the shortest distances between any 2 vertices could be calculated in O (V^3), where V is the number of vertices in a graph.

The Algorithm Steps:

For a graph with N vertices:

1. Initialize the shortest paths between any 2 vertices with Infinity.
2. Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on until using all N vertices as intermediate nodes.
3. Minimize the shortest paths between any 2 pairs in the previous operation.
4. For any 2 vertices (i, j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]).

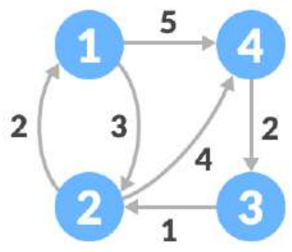
The dist[i][k] represents the shortest path that only uses the first K vertices, dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

```
for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
        }
    }
}
```

Time Complexity of Floyd–Warshall's Algorithm is O(V3), where V is the number of vertices in a graph.

Example

Consider a graph as



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^1 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.
   Each cell A[i][j] is filled with the distance from the ith vertex to the jth vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

$$A^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & \infty & 4 \\ 3 & \infty & 1 & 0 & \infty \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

2. Now, create a matrix A^1 using matrix A^0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way. Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).That is, if the

direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with A[i][k] + A[k][j].

$$A^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & & \\ 3 & \infty & & 0 & \\ 4 & \infty & & & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & \infty & 1 & 0 & 8 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

For example: For A1[2, 4], the direct distance from vertex 2 to 4 is 4 and the sum of the distance from vertex 2 to 4 through vertex (ie. from vertex 2 to 1 and from vertex 1 to 4) is 7. Since 4 < 7, A0[2, 4] is filled with 4.

3. In a similar way, A^2 is created using A^3. The elements in the second column and the second row are left as they are. In this step, k is the second vertex. The remaining steps are the same as in step 2.

$$A^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & & \\ 2 & 2 & 0 & 9 & 4 \\ 3 & & 1 & 0 & \\ 4 & & \infty & & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & \infty & \infty & 2 & 0 \end{array}$$

4. Similarly, A3 and A4 is also created.

$$A^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & \infty & \\ 2 & & 0 & 9 & \\ 3 & \infty & 1 & 0 & 8 \\ 4 & & & 2 & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 9 & 5 \\ 2 & 2 & 0 & 9 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

$$A^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & & & 5 \\ 2 & & 0 & & 4 \\ 3 & & & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 7 & 5 \\ 2 & 2 & 0 & 6 & 4 \\ 3 & 3 & 1 & 0 & 5 \\ 4 & 5 & 3 & 2 & 0 \end{array}$$

5. A^4 gives the shortest path between each pair of vertices.

Floyd-Warshall's Algorithm

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            $A^k[i, j] = \min (A^{k-1}[i, j], A^{k-1}[i, k] + A^{k-1}[k, j])$
return A
```