## 8.7 Problems on Disjoint Sets

**Problem-1** Consider a list of cities $c_1, c_2,\ldots,c_n$. Assume that we have a relation $R$ such that, for any $i, j$, $R(c_i, c_j)$ is 1 if cities $c_i$ and $c_j$ are in the same state, and 0 otherwise. If $R$ is stored as a table, how much space does it require?

**Solution**: $R$ must have an entry for every pair of cities. There are $\Theta(n^2)$ of these.

**Problem-2** For the Problem-1, using a Disjoint sets ADT, give an algorithm that puts each city in a set such that $c_i$ and $c_j$ are in the same set if and only if they are in the same state.

**Solution:**

```
for (i = 1; i<= n; i++) {
        MAKESET(ci);
        for (j = 1; j <= i-1; j++) {
                if(R(cj, ci)) {
                        UNION(cj, ci);
                        break;
                }
        }
}
```

**Problem-3** For the Problem-1, when the cities are stored in the Disjoint sets ADT, if we are given two cities $c_i$ and $c_j$, how do we check if they are in the same state?

**Solution:** Cities $c_i$ and $c_j$ are in the same state if and only if FIND($c_i$) = FIND($c_j$).

**Problem-4** For the Problem-1, if we use linked-lists with UNION by size to implement the union-find ADT, how much space do we use to store the cities?

**Solution**: There is one node per city, so the space is $\Theta(n)$.

**Problem-5** For the Problem-1, if we use trees with UNION by rank, what is the worst-case running time of the algorithm from Problem-2?

**Solution**: Whenever we do a UNION in the algorithm from Problem-2, the second argument is a tree of size 1. Therefore, all trees have height 1, so each union takes time O(1). The worst-case running time is then $\Theta(n^2)$.

**Problem-6** If we use trees without union-by-rank, what is the worst-case running time of the algorithm from Problem-2. Are there more worst-case scenarios than Problem-5?

**Solution**: Because of the special case of the unions, union-by-rank does not make a difference for our algorithm. Hence, everything is the same as in Problem-5.





# GRAPH ALGORITHMS

# Chapter-9

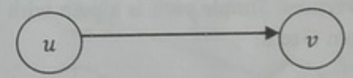
## 9.1 Introduction

In real world, many problems are represented in terms of objects and connections between them. For example, in an airline route map, we might be interested in questions like: "What's the fastest way to go from Hyderabad to New York?" *or* "What is the cheapest way to go from Hyderabad to New York?". To answer these questions we need information about interconnections (airline routes) between objects (towns). Graphs are the data structures are used for solving these kinds of problems.

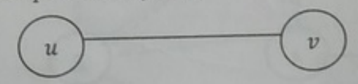
## 9.2 Glossary

**Graph:** A graph is a pair $(V, E)$, where $V$ is a set of nodes, called *vertices* and $E$ is a collection of pairs of vertices, called *edges*.

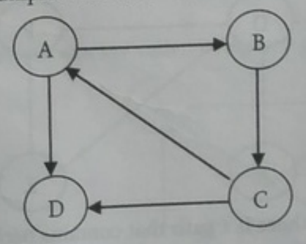
- *Vertices* and *edges* are positions and store elements
- Definitions that we use:
  - *Directed edge*:
    - ordered pair of vertices $(u, v)$
    - first vertex $u$ is the origin
    - second vertex $v$ is the destination
    - Example: One-way road traffic



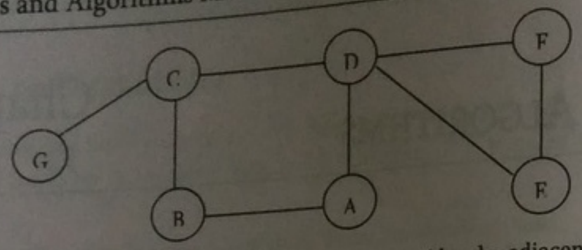

  - *Undirected edge*
    - unordered pair of vertices $(u, v)$
    - Example: Railway lines



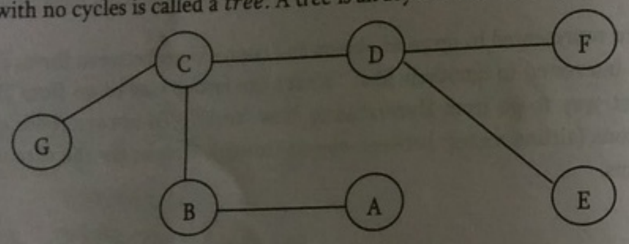

  - *Directed graph*
    - all the edges are directed
    - Example: route network



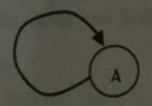

  - *Undirected graph*
    - all the edges are undirected
    - Example: flight network



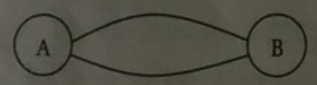

- When an edge connects two vertices, the vertices are said to be adjacent to each other and that the edge is incident on both vertices.
- A graph with no cycles is called a *tree*. A tree is an acyclic connected graph.



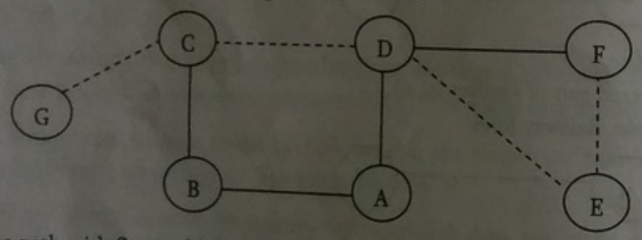

- A self loop is an edge that connects a vertex to itself.



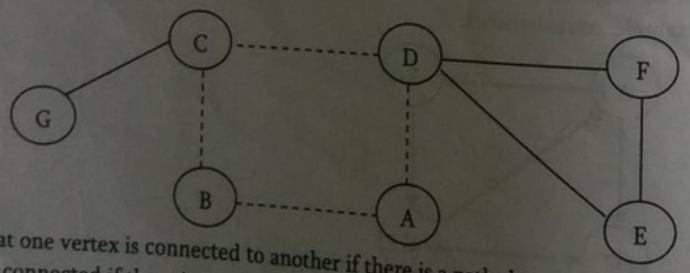

- Two edges are parallel if they connect the same pair of vertices.




- *D*egree of a vertex is the number of edges incident on it.
- A subgraph is a subset of a graphs edges (with associated vertices) that forms a graph.
- A path in a graph is a sequence of adjacent vertices. *S*imple path is a path with no repeated vertices. In the below graph, dotted lines represents a path from *G* to *E*.



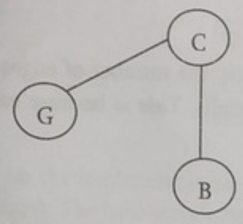

- A cycle is a path with first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).

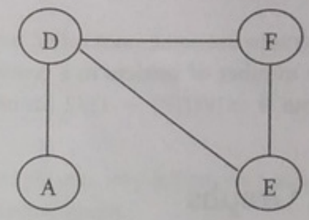



- We say that one vertex is connected to another if there is a path that contains both of them.
- A graph is connected if there is a path from *every* vertex to every other vertex.
- If a graph is not connected then it consists of a set of connected components.

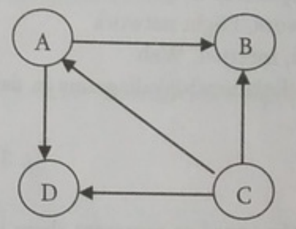




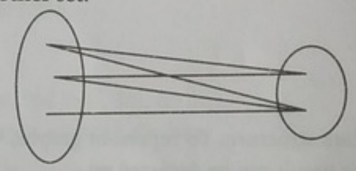


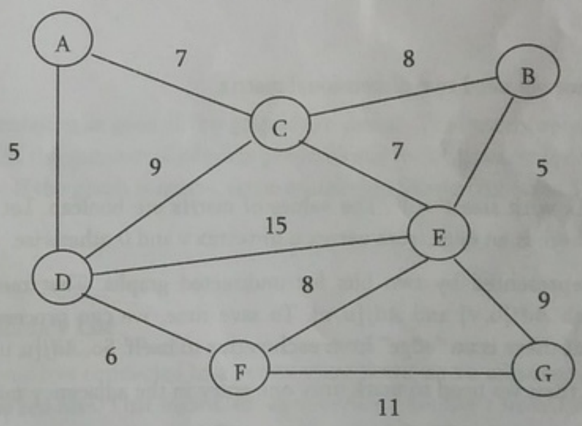

- A *directed* acyclic graph [DAG] is a directed graph with no cycles.



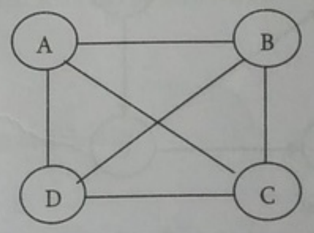

- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graphs vertices and is a single tree. A spanning forest of a graph is the union of spanning trees of its connected components.
- A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



- In *weighted graphs* integers *(weights)* are assigned to each edge to represent (distances or costs).




- Graphs with all edges present are called *complete* graphs.




- Graphs with relatively few edges (generally if it edges < $|V| \log |V|$) are called *sparse graphs*.
- Graphs with relatively few of the possible edges missing are called *dense*.

- Directed weighted graphs are sometimes called *network*.
- We will denote the number of vertices in a given graph by $|V|$, the number of edges by $|E|$. Note that $E$ can range anywhere from 0 to $|V|(|V| - 1)/2$ (in undirected graph). This is because each node can connect to every other node.

## 9.3 Applications of Graphs

- Representing relationships between components in electronic circuits
- Transportation networks: Highway network, Flight network
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

## 9.4 Graph Representation

As similar to other ADTs, to manipulate graphs we need to represent them in some useful form. Basically, there are three ways of doing this:

- Adjacency Matrix
- Adjacency List
- Adjacency Set

### Adjacency Matrix

#### Graph Declaration for Adjacency Matrix

First, let us see the components of the graph data structure. To represent graphs, we need number of vertices, number of edges and also their interconnections. So, the graph can be declared as:

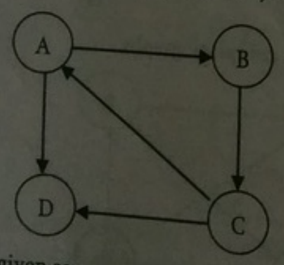
```
struct Graph {
        int V;
        int E;
        int **Adj; //Since we need two dimensional matrix
};
```

#### Description

In this method, we use a matrix with size $V \times V$. The values of matrix are boolean. Let us assume the matrix is $Adj$. The value $Adj[u, v]$ set to 1 if there is an edge from vertex u to vertex v and 0 otherwise.

In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from u to v is represented by 1 values in both $Adj[u, v]$ and $Adj[u, v]$. To save time, we can process only half of this symmetric matrix. Also, we can assume that there is an "edge" from each vertex to itself. So, $Adj[u, u]$ is set to 1 for all vertices.

If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the below directed graph.




Adjacency matrix for this graph can be given as:

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 |
| B | 0 | 0 | 1 | 0 |
| C | 1 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 0 |

Now, let us concentrate on the implementation. To read a graph, one way is first read the vertex names and then read pairs of vertex names (edges). The below code reads an undirected graph.

```
//This code creates a graph with adj matrix representation
struct Graph *adjMatrixOfGraph() {
        int i, u, v;
        struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
        if(!G) {
                printf("Memory Error");
                return;
        }
        scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
        G→Adj = malloc(sizeof(G→V * G→V));
        for(u = 0; u < G→V; u++)
                for(v = 0; v < G→V; v++)
                        G→Adj[v][v] = 0;
        for(i = 0; i < G→E; i++) {
                //Read an edge
                scanf("Reading Edge: %d %d", &u, &v);
                //For undirected graphs set both the bits
                G→ Adj[u][v] = 1;
                G→ Adj[v][u] = 1;
        }
        return G;
}
```

The adjacency matrix representation is good if the graphs are dense. The matrix requires $O(V^2)$ bits of storage and $O(V^2)$ time for initialization. If the number of edges is proportional to $V^2$, then there is no problem because $V^2$ steps are required to read the edges. If the graph is sparse, since initializing the matrix takes $V^2$ and it dominates the running time of algorithm.

### Adjacency List

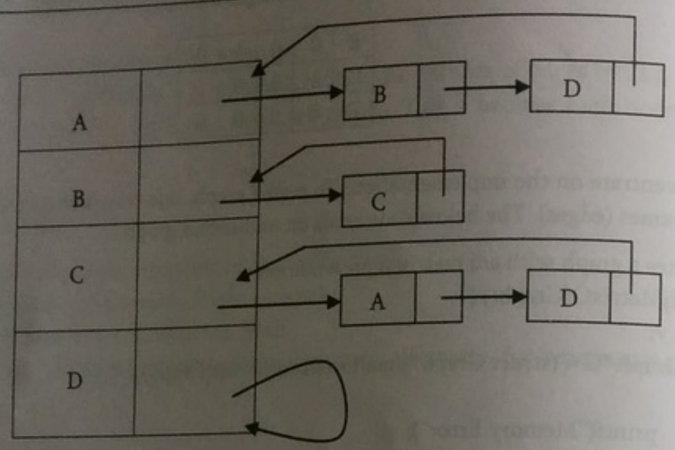
#### Graph Declaration for Adjacency List

In this representation all the vertices connected to a vertex $v$ are listed on an adjacency list for that vertex $v$. This can be easily implemented with linked lists. That means, for each vertex $v$ we use a linked list and list nodes represents the connections between $v$ and other vertices to which $v$ has an edge. The total number of linked lists is equal to the number of vertices in the graph. The graph ADT can be declared as:

```
struct Graph {
        int V;
        int E;
        int *Adj; //head pointers to linked list
};
```

#### Description

Considering the same example as that of adjacency matrix, the adjacency list representation can be given as:




Since vertex A is having an edge for B and D, we have added them in the adjacency list for A. Same is the case with other vertices as well.

```
//Nodes of the Linked List
struct ListNode {
        int vertexNumber;
        struct ListNode *next;
}

//This code creates a graph with adj list representation
struct Graph *adjListOfGraph() {
        int i, x, y;
        struct ListNode *temp;
        struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));
        if(!G) {
                printf("Memory Error");
                return;
        }
        scanf("Number of Vertices: %d, Number of Edges:%d", &G→V, &G→E);
        G→Adj = malloc(G→V * sizeof(struct ListNode));
        for(i = 0; i  < G→V; i++) {
                G→Adj[i] = (struct ListNode *) malloc(sizeof(struct ListNode));
                G→Adj[i]→vertexNumber = i;
                G→Adj[i]→next = G→ Adj[i];
        }
        for(i = 0;  i < E; i++) {
                //Read an edge
                scanf("Reading Edge: %d %d", &x, &y);
                temp = (struct ListNode *) malloc(struct ListNode);
                temp→vertexNumber = y;
                temp→next = G→Adj[x];
                G→ Adj[x]→next = temp;
                temp = (struct ListNode *) malloc(struct ListNode);
                temp→vertexNumber = y;
                temp→next = G→ Adj[y];
                G→Adj[y]→ next= temp;
        }
        retutn G;
```





}

For this representation, the order of edges in the input is *important*. This is because they determine the order of the vertices on the adjacency lists. Same graph can be represented in many different ways in an adjacency list. The order in which edges appear on the adjacency list affects the order in which edges are processed by algorithms.

### Disadvantages of Adjacency Lists

Using adjacency list representation we cannot perform some operations efficiently. As an example, consider the case of deleting a node. In adjacency list representation, if we delete a node from the adjacency list then that is enough. For each node on the adjacency list of that node specifies another vertex. We need to search other nodes linked list also for deleting it. This problem can be solved by linking the two list nodes which correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

## Adjacency Set

It is very much similar to adjacency list but instead of using Linked lists, Disjoint Sets [Union-Find] are used. For more details refer *Disjoint Sets ADT* chapter.

## Comparison of Graph Representations

Directed and undirected graphs are represented with same structures. For directed graphs, everything is the same, except that each edge is represented just once, an edge from $x$ to $y$ is represented by a 1 value in $Adj[x][y]$ in the adjacency matrix or by adding $y$ on $x's$ adjacency list. For weighted graphs, everything is same except that fill the adjacency matrix with weights instead of boolean values.

| Representation | Space | Checking edge between $v$ and $w$? | Iterate over edges incident to $v$? |
|---|---|---|---|
| List of edges | E | $E$ | $E$ |
| Adj Matrix | $V^2$ | 1 | $V$ |
| Adj List | $E+V$ | $Degree(v)$ | $Degree(v)$ |
| Adj Set | $E+V$ | $log(Degree(v))$ | $Degree(v)$ |

# 9.5 Graph Traversals

To solve problems on graphs, we need a mechanism for traversing the graphs. Graph traversal algorithms are also called as *graph search* algorithms. Like trees traversal algorithms (Inorder, Preorder, Postorder and Level-Order traversals), graph search algorithms can be thought of as starting at some source vertex in a graph, and "search" the graph by going through the edges and marking the vertices. Now, we will discuss two such algorithms for traversing the graphs.

- Depth First Search [DFS]
- Breadth First Search [BFS]

## Depth First Search [DFS]

DFS algorithm works similar to *preorder* traversal of the trees. Like *preorder* traversal, internally this algorithm also uses stack.

Let us consider the following example. Suppose a person is trapped inside a maze. To come out from that maze, the person visits each path and each intersection (in the worst case). Let us say the person uses two colors of paint, to mark the intersections already passed. When discovering a new intersection, it is marked grey, and he continues to go deeper. After reaching a "dead end" the person knows that there is no more unexplored path from the grey intersection, which now is completed and he marks it with black. This "dead end" is either an intersection which has already been marked grey or black, or simply a path that does not lead to an intersection.

Intersections of the maze are the vertices and the paths between the intersections are the edges of the graph. The process of returning from the "dead end" is called *backtracking*. We are trying to go away from starting vertex into the graph as deep as possible, until we have to backtrack to the preceding grey vertex. In DFS algorithm, we encounter the following types of edges.

| |
|---|
| *Tree edge*: encounter new vertex |
| *Back edge*: from descendent to ancestor |
| *Forward edge:* from ancestor to descendent |
| *Cross edge*: between a tree or subtrees |

For most algorithms boolean classification unvisited/visited is enough (for three color implementation refer problems section). That means, for some problems we need to use three colors, but for our discussion two colors are enough.

false ⟶ Vertex is unvisited

true ⟶ Vertex is visited

Initially all vertices are marked unvisited (false). DFS algorithm starts at a vertex $u$ in graph. By starting at vertex $u$ it considers the edges from $u$ to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex $u$. If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex. Follow this process until we reach the dead-end. At this point start *backtracking*. The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is give below: assume Visited[] is a global array.

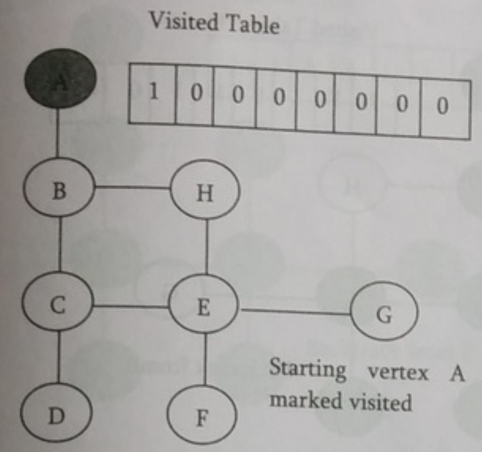
```
int Visited[G→V];
void DFS(struct Graph *G, int u) {
        Visited[u] = 1;
        for( int v = 0; v < G→V; v++ ) {
                /* For example, if the adjacency matrix is used for representing the
                   graph, then the condition to be used for finding unvisited adjacent
                   vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        DFS(v);
                }
        }
}
void DFSTraversal(struct Graph *G) {
        for (int i = 0; i< G→V;i++)
                Visited[i]=0;
        //This loop is required if the graph has more than one component
        for (int i = 0; i< G→V;i++)
                if(!Visited[i])
                        DFS(G, i);
}
```

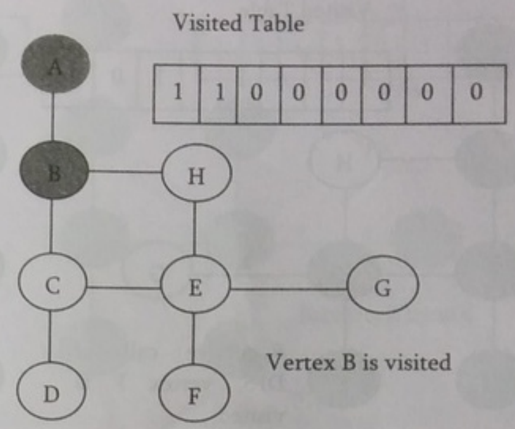
As an example consider the following graph. We could see that sometimes an edge leads to an already discovered vertex. These edges are called *back edges*, and the other edges are called *tree edges* because deleting the back edges from the graph generates a tree.

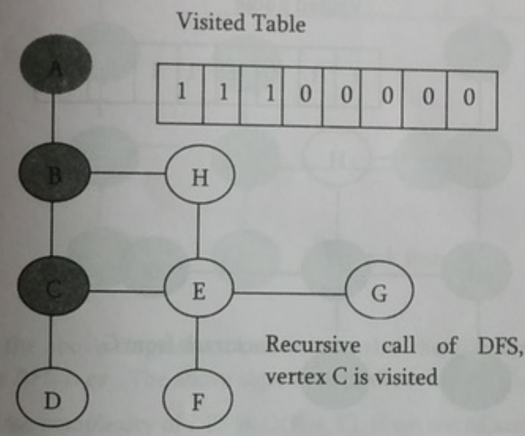
The final generated tree is called *DFS tree* and the order in which the vertices processed are called *DFS numbers* of the vertices. In the below graph gray color indicates that the vertex is visited (there is no other significance). We need to see when *Visited* table is being updated.

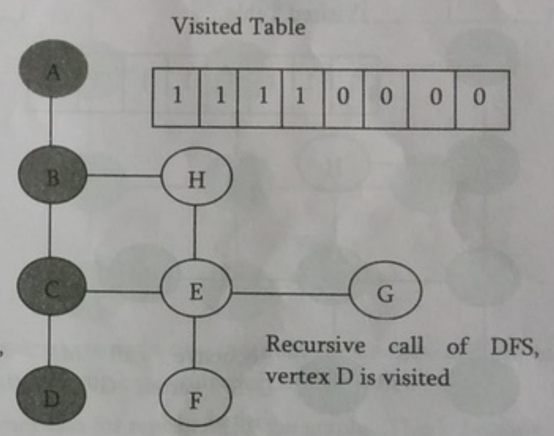


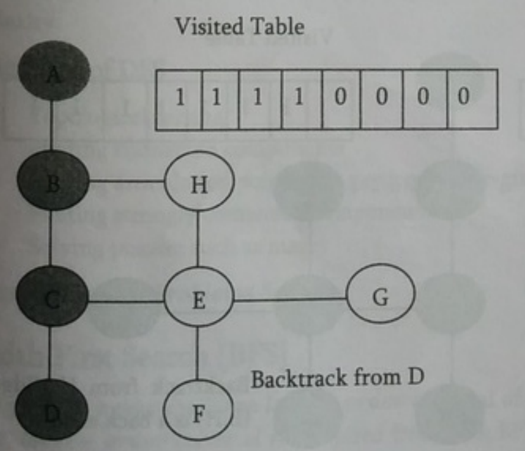


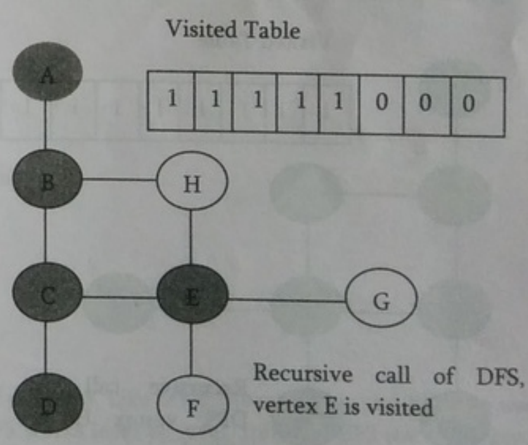


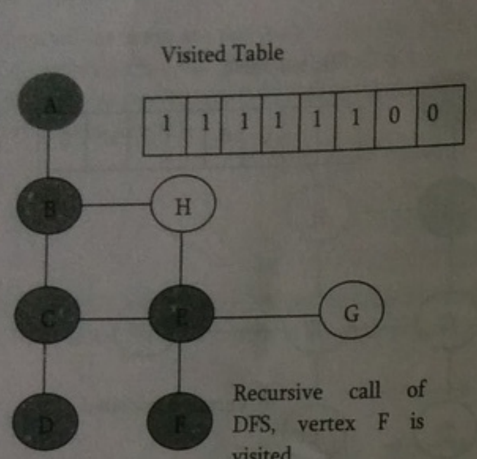



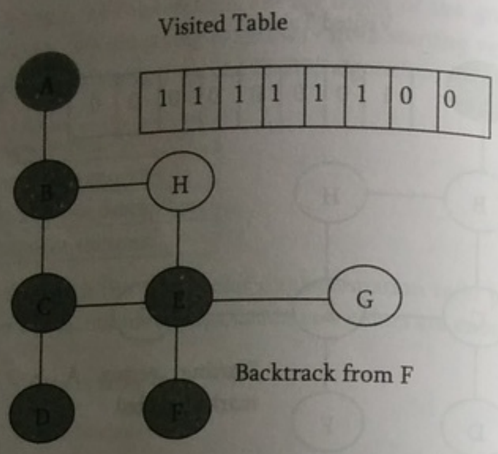



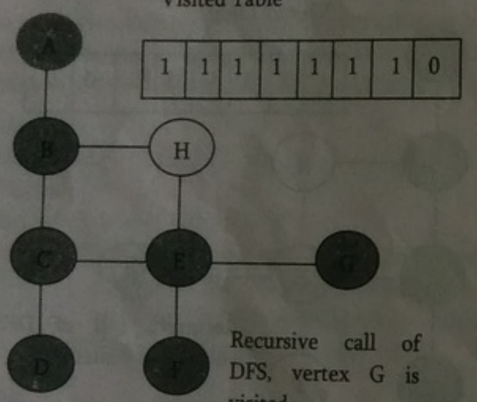





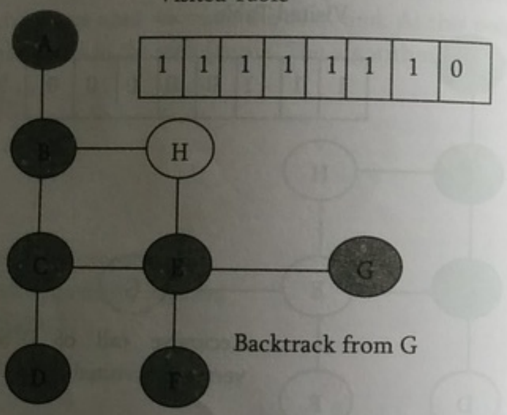

Recursive call of DFS, vertex F is visited



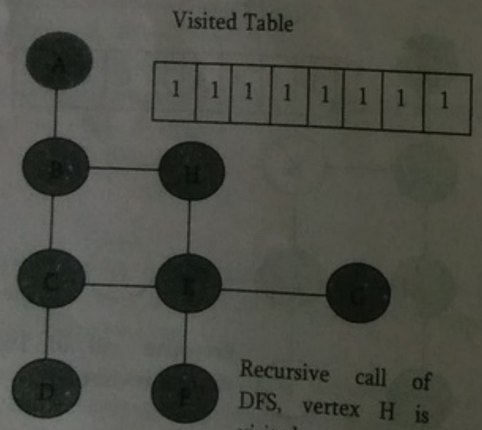

Backtrack from F



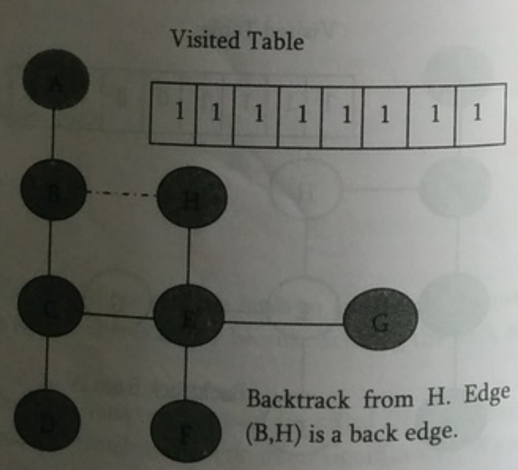

Recursive call of DFS, vertex G is visited




Backtrack from G



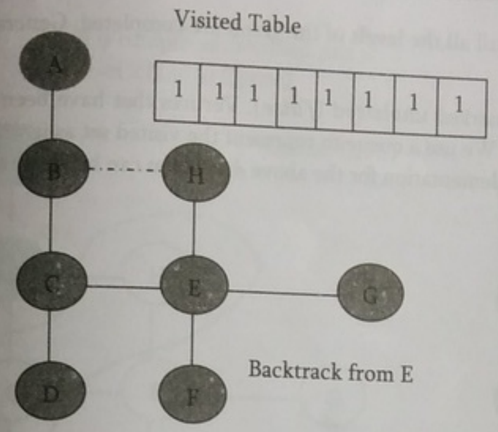

Recursive call of DFS, vertex H is visited



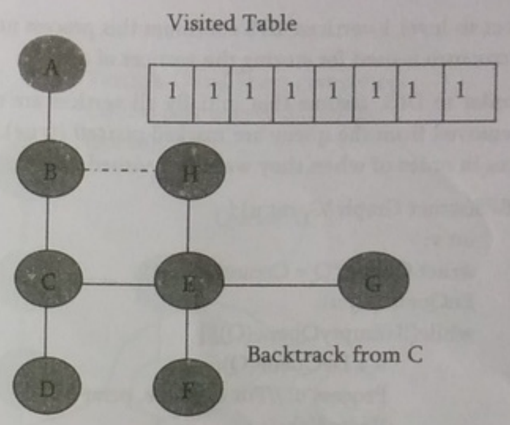

Backtrack from H. Edge (B,H) is a back edge.



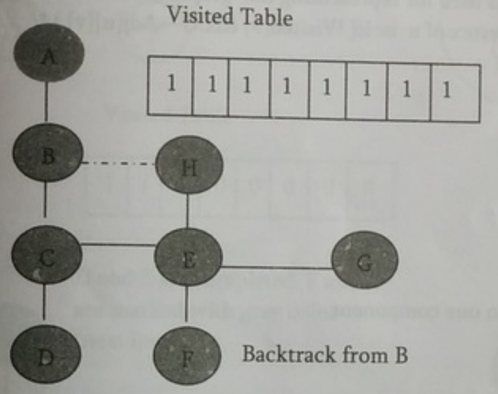




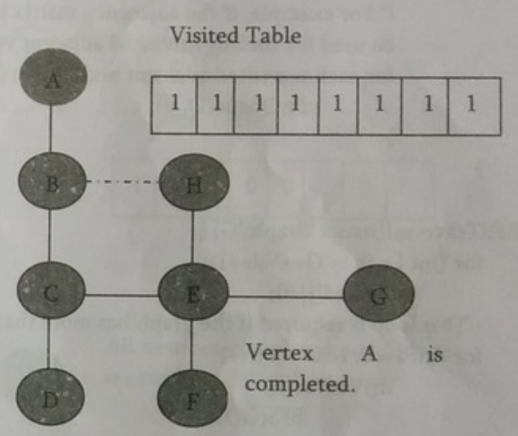

Backtrack from E




Backtrack from C




Backtrack from B




Vertex A is completed.

From the above diagrams, it can be seen that the DFS traversal creates a tree (without back edges) and we call such tree as *DFS tree*. The above algorithm works even if the given graph has connected components.

The time complexity of DFS is $O(V + E)$, if we use adjacency lists for representing the graphs. This is because, we are starting at a vertex and processing the adjacent nodes only if they are not visited. Similarly, if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, this gives $O(V^2)$ complexity.

### Applications of DFS

- Topological sorting
- Finding connected components
- Finding articulation points (cut vertices) of the graph
- Finding strongly connected components
- Solving puzzles such as mazes

For algorithms refer *Problems Section*.

## Breadth First Search [BFS]

BFS algorithm works similar to $level - order$ traversal of the trees. Like $level - order$ traversal BFS also uses queues. In fact, $level - order$ traversal got inspired from BFS. BFS works level by level. Initially, BFS starts at a given vertex, which is at level 0. In the first stage it visits all vertices at level 1 (that means, vertices whose distance is 1 from start vertex of the graph). In the second stage, it visits all vertices at second level. These new vertices are the one which are

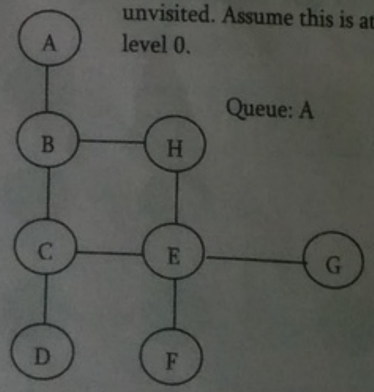
adjacent to level 1 vertices. BFS continues this process until all the levels of the graph are completed. Generally *queue* data structure is used for storing the vertices of a level.

As similar to DFS, assume that initially all vertices are marked *unvisited* (*false*). Vertices that have been processed and removed from the queue are marked *visited* (*true*). We use a queue to represent the visited set as it will keep the vertices in order of when they were first visited. The implementation for the above discussion can be given as:

```
void BFS(struct Graph *G, int u) {
        int v;
        struct Queue *Q = CreateQueue();
        EnQueue(Q, u);
        while(!IsEmptyQueue(Q)) {
                u = DeQueue(Q);
                Process u; //For example, print
                Visited[s]=1;
                /* For example, if the adjacency matrix is used for representing the  graph, then the condition
                be used for finding unvisited adjacent vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        EnQueue(Q, v);
                }
        }
}
void BFSTraversal(struct Graph *G) {
        for (int i = 0; i< G→V;i++)
                Visited[i]=0;
        //This loop is required if the graph has more than one component
        for (int i = 0; i< G→V;i++)
                if(!Visited[i])
                        BFS(G, i);
}
```

As an example, let us consider the same graph as that of DFS example. The BFS traversal can be shown as:

Starting vertex *A* is marked unvisited. Assume this is at level 0.

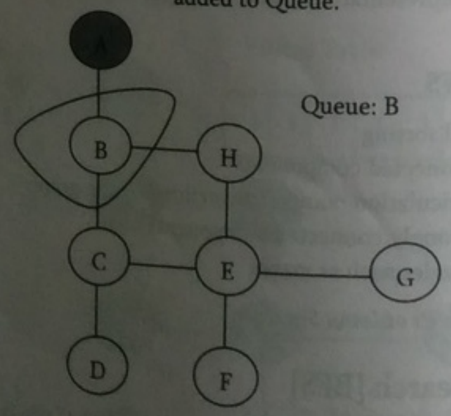



Queue: A

Visited Table

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Vertex *A* is completed. Circled part is level 1 and added to Queue.




Queue: B

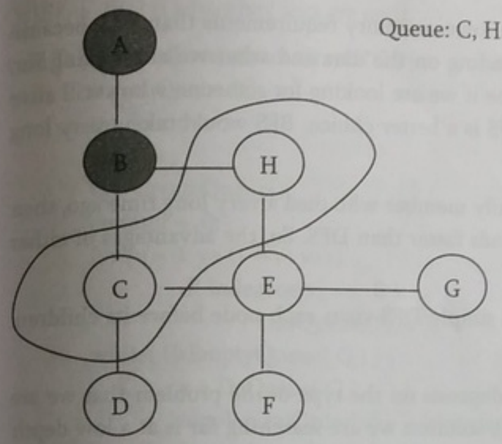
Visited Table

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|





B is completed. Selected part is level 2 (add to Queue).

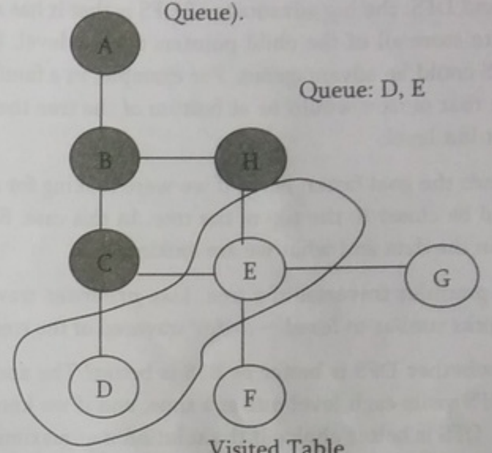



Queue: C, H

Visited Table

| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Vertices C and H are completed. Circled part is level 3 (add to Queue).

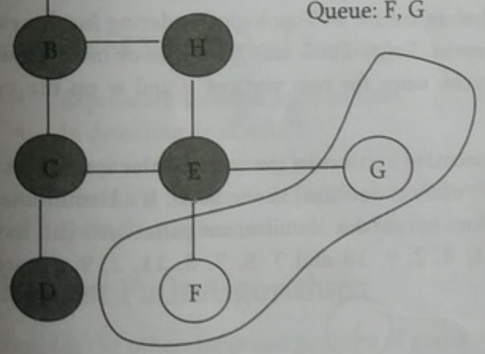



Queue: D, E

Visited Table

| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

D and E are completed. F and G are marked with gray color (next level).

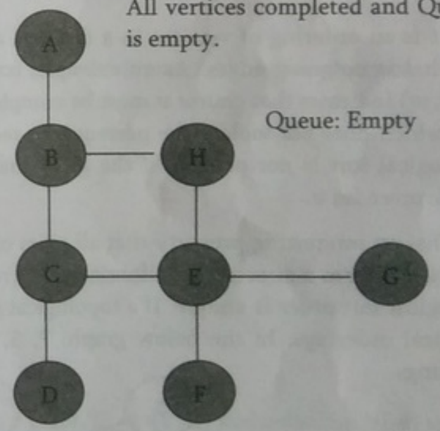



Queue: F, G

Visited Table

| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
|---|---|---|---|---|---|---|---|

All vertices completed and Queue is empty.




Queue: Empty

Visited Table

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Time complexity of BFS is $O(V + E)$, if we use adjacency lists for representing the graphs and $O(V^2)$ for adjacency matrix representation.

### Applications of BFS

- Finding all connected components in a graph
- Finding all nodes within one connected component
- Finding the shortest path between two nodes

- Testing a graph for bipartiteness

### Comparing DFS and BFS

Comparing BFS and DFS, the big advantage of DFS is that it has much lower memory requirements than BFS, because it's not required to store all of the child pointers at each level. Depending on the data and what we are looking for, either DFS or BFS could be advantageous. For example, in a family tree if we are looking for someone who's still alive and if we assume that person would be at bottom of the tree then DFS is a better choice. BFS would take a very long time to reach that last level.

DFS algorithm finds the goal faster. Now, if we were looking for a family member who died a very long time ago, then that person would be closer to the top of the tree. In this case, BFS finds faster than DFS. So, the advantages of either vary depending on the data and what we are looking for.

DFS is related to preorder traversal of a tree. Like *preorder* traversal simply DFS visits each node before its children. BFS algorithm works similar to *level – order* traversal of the trees.

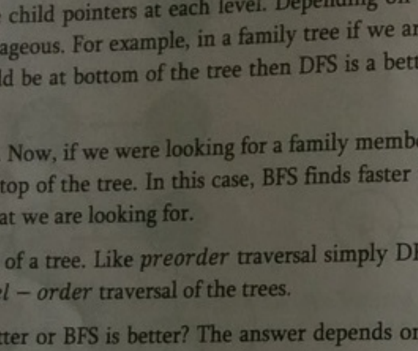
If someone asks whether DFS is better or BFS is better? The answer depends on the type of the problem that we are trying to solve. BFS visits each level one at a time, and if we know the solution we are searching for is at a low depth then BFS is good. DFS is better choice if the solution is at maximum depth. Below table shows the differences between DFS and BFS in terms of their applications.

| Applications | DFS | BFS |
|---|---|---|
| Spanning forest, connected components, paths, cycles | Yes | Yes |
| Shortest paths | | Yes |
| Minimal use of memory space | Yes | |

## 9.6 Topological Sort

*Topological sort* is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure at universities. A directed *edge* $(v, w)$ indicates that course $v$ must be completed before course $w$. Topological ordering for this example is the sequence which does not violate the prerequisite requirement. Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since for two vertices $v$ and $w$ on the cycle, $v$ precedes $w$ and $w$ precedes $v$.

Topological sort has an interesting property that all pairs of consecutive vertices in the sorted order are connected by edges then these edges form a directed Hamiltonian path [refer *Problems Section*] in the DAG. If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings. In the below graph: 7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.




Initially, *indegree* is computed for all vertices and start with the vertices which are having indegree 0. That means consider the vertices which do not have any prerequisite. To keep track of vertices with indegree zero we can use a queue.





All vertices of indegree 0 are placed on queue. While the queue is not empty, a vertex $v$ is removed, and all edges adjacent to $v$ have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering then is the order in which the vertices DeQueue. The time complexity of this algorithm is $O(|E| + |V|)$ if adjacency lists are used.

```
void TopologicalSort( struct Graph *G ) {
        struct Queue *Q;
        int counter;
        int v, w;
        Q = CreateQueue();
        counter = 0;
        for (v = 0; v< G→V; v++)
                if( indegree[v] == 0 )
                        EnQueue( Q, v );
        while( !IsEmptyQueue( Q ) ) {
                v = DeQueue( Q );
                topologicalOrder[v] = ++counter;
                for each w adjacent to v
                        if( --indegree[w] == 0 )
                                EnQueue ( Q, w );
        }
        if( counter != G→V)
                printf("Graph has cycle");
        DeleteQueue( Q );
}
```

Total running time of topological sort is $O(V + E)$.

**Note:** Topological sorting problem can be solved with DFS and refer *Problems Section* for algorithm.

### Applications of Topological Sorting

- Representing course prerequisites
- In detecting deadlocks
- Pipeline of computing jobs
- Checking for symbolic link loop
- Evaluating formulae in spreadsheet

## 9.7 Shortest Path Algorithms

Let us consider the other important problem of graph. Given a graph $G = (V, E)$ and a distinguished vertex s, we need to find the shortest path from $s$ to every other vertex in $G$. There are variations in the shortest path algorithms which depend on the type of the input graph and are given below.

**Variations of Shortest Path Algorithms**

| Shortest path in unweighted graph |
|---|
| Shortest path in weighted graph |
| Shortest path in weighted graph with negative edges |

**Applications of Shortest Path Algorithms**

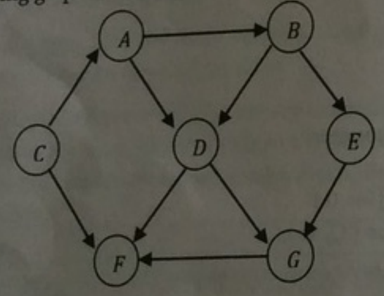
- Finding fastest way to go from one place to another
- Finding cheapest way to fly/send data from one city to another

## Shortest Path in Unweighted Graph

Let $s$ be the input vertex from which we want to find shortest path to all other vertices. Unweighted graph is a special case of the weighted shortest-path problem, with all edges a weight of 1. The algorithm is similar to BFS and we need to use the following data structures:

- A distance table with three columns (each row corresponds to a vertex):
  - Distance from source vertex.
  - Path - contains the name of the vertex through which we got the shortest distance.
- A queue is used to implement breadth-first search. It contains vertices whose distance from the source node has been computed and their adjacent vertices are to be examined.

As an example, consider the following graph and its adjacency list representation.




The adjacency list for this graph is:

$A: B \rightarrow D$
$B: D \rightarrow E$
$C: A \rightarrow F$
$D: F \rightarrow G$
$E: G$
$F: -$
$G: F$

Let $s = C$. The distance from $C$ to $C$ is 0. Initially, distances to all other nodes are not computed, and we initialize the second column in the distance table for all vertices (except $C$) with -1 as below.

| Vertex | Distance[v] | Previous vertex which gave Distance[v] |
|---|---|---|
| $A$ | -1 | - |
| $B$ | -1 | - |
| $C$ | 0 | - |
| $D$ | -1 | - |
| $E$ | -1 | - |
| $F$ | -1 | - |
| $G$ | -1 | - |

**Algorithm**

```
void UnweightedShortestPath(struct Graph *G, int s) {
        struct Queue *Q = CreateQueue();
        int v, w;
        EnQueue(Q, s);
        for (int i = 0; i< G→V;i++)
                Distance[i]=-1;
        Distance[s]= 0;
        while (!IsEmptyQueue(Q)) {
                v = DeQueue(Q);                    Each vertex examined at most once
                for each w adjacent to v
```





```
                if(Distance[w] == -1)      {
                        Distance[w] = Distance[v] + 1;
                        Path[w] = v;
                        EnQueue(Q, w);      ←  Each vertex EnQueue'd at most once
                }
        }
        DeleteQueue(Q);
}
```

Running time: $O(|E| + |V|)$, if adjacency lists are used. In for loop, we are checking the outgoing edges for a given vertex and the sum of all examined edges in the while loop is equal to the number of edges which gives $O(|E|)$.

If we use matrix representation the complexity is $O(|V|^2)$, because we need to read an entire row in the matrix of length $|V|$ in order to find the adjacent vertices for a given vertex.

## Shortest path in Weighted Graph [Dijkstra's]

A famous solution for shortest path problem was given by *Dijkstra*. *Dijkstra's* algorithm is a generalization of BFS algorithm. The regular BFS algorithm cannot solve the shortest path problem as it cannot guarantee that the vertex at the front of the queue is the vertex closest to source $s$.

Before going to code let us understand how the algorithm works. As similar to unweighted shortest path algorithm, we use the distance table. The algorithm works by keeping the shortest distance of vertex $v$ from the source in *Distance* table. The value $Distance[v]$ holds the distance from s to v. The shortest distance of the source to itself is zero. *Distance* table for all other vertices is set to $-1$ to indicate that those vertices are not already processed.

| Vertex | Distance[v] | Previous vertex which gave Distance[v] |
|---|---|---|
| $A$ | -1 | - |
| $B$ | -1 | - |
| $C$ | 0 | - |
| $D$ | -1 | - |
| $E$ | -1 | - |
| $F$ | -1 | - |
| $G$ | -1 | - |

After the algorithm finishes *Distance* table will have the shortest distance from source $s$ to each other vertex $v$. To simplify the understanding of *Dijkstra's* algorithm, let us assume that the given vertices are maintained in two sets. Initially the first set contains only the source element and the second set contains all the remaining elements. After the $k^{th}$ iteration, the first set contains $k$ vertices which are closest to the source. These $k$ vertices are the one for which we have already computed shortest distances from source.

**Notes on Dijkstra's Algorithm**

- It uses greedy method: Always pick the next closest vertex to the source.
- Uses priority queue to store unvisited vertices by distance from $s$.
- Does not work with negative weights.

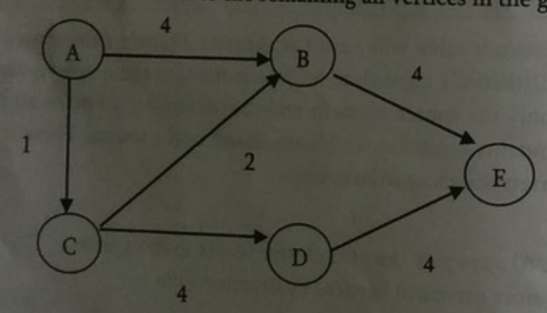
**Difference between Unweighted Shortest Path and Dijkstra's Algorithm**

1) To represent weights in adjacency list, each vertex contains the weights of the edges (in addition to their identifier).
2) Instead of ordinary queue we use priority queue [distances are the priorities] and the vertex with the smallest distance is selected for processing.
3) The distance to a vertex is calculated by the sum of the weights of the edges on the path from the source to that vertex.

4) We update the distances in case the newly computed distance is smaller than old distance which we have already computed.

```
void Dijkstra(struct Graph *G, int s) {
        struct PriorityQueue *PQ = CreatePriorityQueue();
        int v, w;
        EnQueue(PQ, s);
        for (int i = 0; i< G→V;i++)
                Distance[i]=-1;
        Distance[s] = 0;
        while ((!IsEmptyQueue(PQ)) {
                v = DeleteMin(PQ);
                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];
                        if(Distance[w] == -1) {
                                Distance[w] = new distance d;
                                Insert w in the priority queue with priority d
                                Path[w] = v;
                        }
                        if(Distance[w]  > new distance d) {
                                Distance[w] = new distance d;
                                Update priority of vertex w to be d;
                                Path[w] = v;
                        }
                }
        }
}
```

The above algorithm can be better understood using an example. The example will explain each step that is taken and how *Distance* is calculated. The below weighted graph has 5 vertices from $A - E$. The value between the two vertices is known as the edge cost between two vertices. For example the edge cost between $A$ and $C$ is 1. Dijkstra's algorithm can be used to find shortest path from the source $A$ to the remaining all vertices in the graph.




Initially the *Distance* table is:

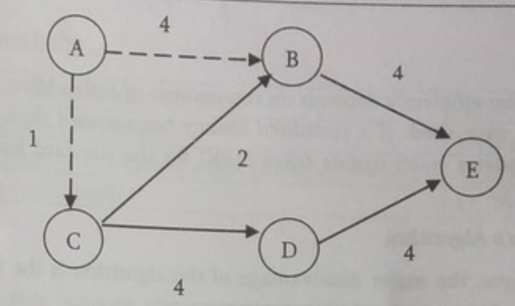
| Vertex | Distance[v] | Previous vertex which gave Distance[v] |
|---|---|---|
| *A* | 0 | - |
| *B* | -1 | - |
| *C* | -1 | - |
| *D* | -1 | - |
| *E* | -1 | - |

After the first step, from the vertex $A$, we can reach $B$ and $C$. So, in the *Distance* table we update the reachability of $B$ and $C$ with their costs and same is shown below.





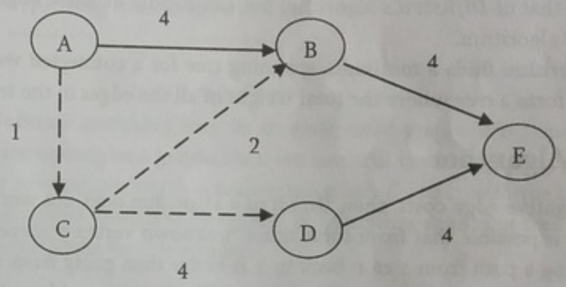
| | | |
|---|---|---|
| *A* | 0 | - |
| *B* | 4 | A |
| *C* | 1 | A |
| *D* | -1 | - |
| *E* | -1 | - |




*Shortest path from $B, C$ from $A$*

Now, let us select the minimum distance among all. The minimum distance vertex is $C$. That means, we have to reach other vertices from these two vertices ($A$ and $C$). For example $B$ can be reached from $A$ and also from $C$. In this case we have to select the one which gives low cost. Since reaching $B$ through $C$ is giving minimum cost $(1 + 2)$, we update the *Distance* table for vertex $B$ with cost 3 and the vertex from which we got this cost as $C$.

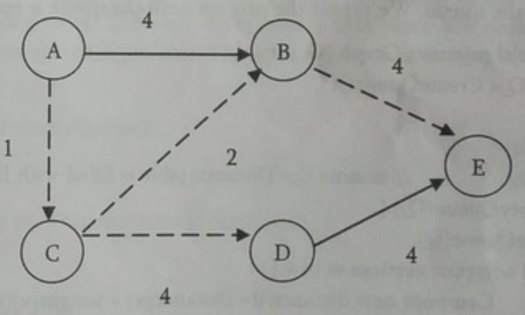
| | | |
|---|---|---|
| *A* | 0 | - |
| *B* | 3 | C |
| *C* | 1 | A |
| *D* | 5 | C |
| *E* | -1 | - |



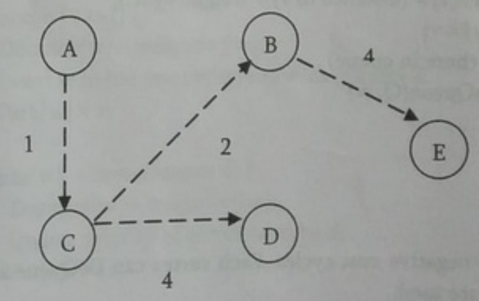

*Shortest path to $B, D$ using $C$ as intermediate vertex*

The only vertex remaining is $E$. To reach $E$, we have to see all the paths through which we can reach $E$ and select the one which gives minimum cost. We can see that if we use $B$ as intermediate vertex through $C$ then we get the minimum cost.

| | | |
|---|---|---|
| *A* | 0 | - |
| *B* | 3 | C |
| *C* | 1 | A |
| *D* | 5 | C |
| *E* | 7 | B |




The final minimum cost tree which Dijkstra's algorithm generates is:



**Performance**

In Dijkstra's algorithm, the efficiency depends on the number of DeleteMins ($V$ DeleteMins) and updates for priority queues ($E$ updates) that were used. If a *standard binary heap* is used then the complexity is $O(ElogV)$. The term $ElogV$ comes from $E$ updates (each update takes $logV$) for the standard heap. If the set used is an array then the complexity is $O(E + V^2)$.

**Disadvantages of Dijkstra's Algorithm**

- As discussed above, the major disadvantage of the algorithm is the fact that it does a blind search there by consuming a lot of time waste of necessary resources.
- Another disadvantage is that it cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the right shortest path.

**Relatives of Dijkstra's Algorithm**

- The *Bellman– Ford* algorithm computes single-source shortest paths in a weighted digraph. It uses the same concept as that of *Dijkstra's* algorithm but can handle negative edges as well. It has more running time than *Dijkstra's* algorithm.
- Prim's algorithm finds a minimum spanning tree for a connected weighted graph. It implies that a subset of edges that form a tree where the total weight of all the edges in the tree is minimized.

## Bellman-Ford Algorithm

If the graph has negative edge costs, then *Dijkstra's* algorithm does not work. The problem is that once a vertex $u$ is declared known, it is possible that from some other, unknown vertex $v$ there is a path back to $u$ that is very negative. In such a case, taking a path from $s$ to $v$ back to $u$ is better than going from $s$ to $u$ without using $v$. A combination of the Dijkstra's algorithm and unweighted algorithms will solve the problem. Initialize the queue with $s$. Then, at each stage, we *DeQueue* a vertex $v$. We find all vertices $w$ adjacent to $v$ such that,

$$distance\ to\ v\ +\ weight(v,w) <\ \text{old distance to w}$$

We update w old distance and path, and place $w$ on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue. We repeat the process until the queue is empty.

```
void BellmanFordAlgorithm(struct Graph *G, int s) {
        struct Queue *Q = CreateQueue();
        int v, w;
        EnQueue(Q, s);
        Distance[s] = 0;              // assume the Distance table is filled with INT_MAX
        while ((!IsEmptyQueue(Q)) {
                v = DeQueue(Q);
                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];
                        if(old distance to w > new distance d ) {
                            Distance[v] = (distance to v) + weight[v][w]);
                            Path[w] = v;
                            if(w is there in queue)
                                EnQueue(Q, w)
                        }
                }
        }
}
```

This algorithm works if there are no negative-cost cycles. Each vertex can DeQueue at most $|V|$ times, so the running time is $O(|E| . |V|)$ if adjacency lists are used.



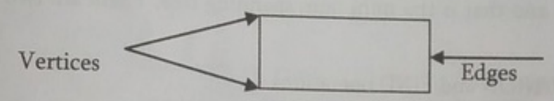


## Overview of Shortest Path Algorithms

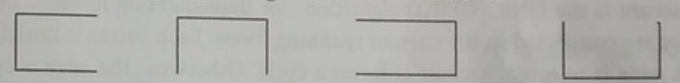
| | |
|---|---|
| Shortest path in unweighted graph [*Modified BFS*] | $O(\|E\| + \|V\|)$ |
| Shortest path in weighted graph [*Dijkstra's*] | $O(\|E\| log \|V\|)$ |
| Shortest path in weighted graph with negative edges [*Bellman – Ford*] | $O(\|E\|.\|V\|)$ |
| Shortest path in weighted acyclic graph | $O(\|E\| + \|V\|)$ |

# 9.8 Minimal Spanning Tree

*Spanning tree* of a graph is a subgraph that contains all the vertices and is also a tree. A graph may have many spanning trees. As an example, consider a graph with 4 vertices as shown below. Let us assume that the corners of the graph are vertices.




For this simple graph, we can have multiple spanning trees as shown below.



The algorithm we will discuss now is *minimum spanning tree* in an undirected graph. We assume that the given graphs are weighted graph. If the graphs are unweighted graphs then we can still the weighted graph algorithms by treating all weights are equal. A *minimum spanning tree* of an undirected graph $G$ is a tree formed from graph edges that connects all the vertices of $G$ with minimum total cost (weights). A minimum spanning tree exists only if the graph is connected. There are two famous algorithms for this problem:

- *Prim's* Algorithm
- *Kruskal's* Algorithm

## Prim's Algorithm

Prim's algorithm is almost same as Dijkstra's algorithm. Similar to Dijkstra's algorithm, in Prim's algorithm also we keep values *distance* and *paths* in distance table. The only exception is that since the definition of *distance* is different and as result the updating statement also changes little. The update statement is simpler than before.

```
void Prims(struct Graph *G, int s) {
        struct PriorityQueue *PQ = CreatePriorityQueue();
        int v, w;
        EnQueue(PQ, s);
        Distance[s] = 0;              // assume the Distance table is filled with -1
        while ((!IsEmptyQueue(PQ)) {
                v = DeleteMin(PQ);
                for all adjacent vertices w of v {
                        Compute new distance d= Distance[v] + weight[v][w];
                        if(Distance[w] == -1) {
                                Distance[w] = weight[v][w];
                                Insert w in the priority queue with priority d
                                Path[w] = v;
                        }
                        if(Distance[w]  > new distance d) {
                                Distance[w] = weight[v][w];
                                Update priority of vertex w to be d;
                                Path[w] = v;
                        }
```

```
        }
    }
}
```

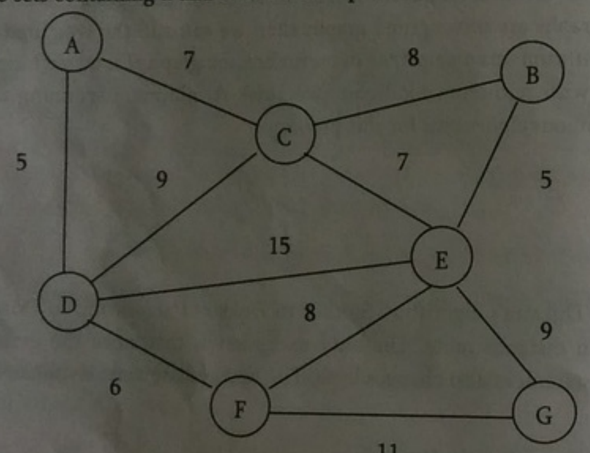
The entire implementation of this algorithm is identical to that of Dijkstra's algorithm. The running time is $O(|V|^2)$ without heaps [good for dense graphs], and $O(ElogV)$ using binary heaps [good for sparse graphs].

## Kruskal's Algorithm

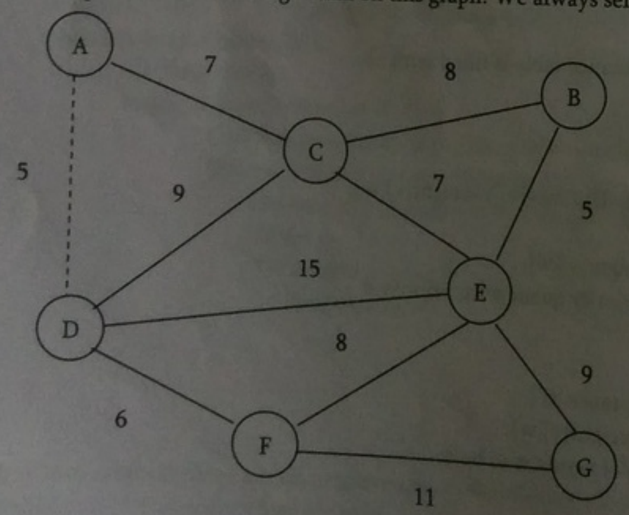
The algorithm starts with $V$ different trees ($V$ is the vertices in graph). While constructing the minimum spanning tree, every time it selects the edge which has minimum weight and adds that edge if it doesn't creates a cycle. So, initially, there are $|V|$ single-node trees in the forest. Adding an edge merges two trees into one. When the algorithm completes, there will be only one tree, and that is the minimum spanning tree. There are two ways of implementing Kruskal's algorithm:

- Bye using Disjoint Sets: Using UNION and FIND operations
- By using Priority Queues: Maintains weights in priority queue

The appropriate data structure is the UNION/FIND algorithm [for implementing forests]. Two vertices belong to the same set if and only if they are connected in the current spanning forest. Each vertex is initially in its own set. If $u$ and $v$ are in the same set, the edge is rejected, because it forms a cycle. Otherwise, the edge is accepted, and a UNION is performed on the two sets containing $u$ and $v$. As an example, consider the following graph (edges shows the weights).



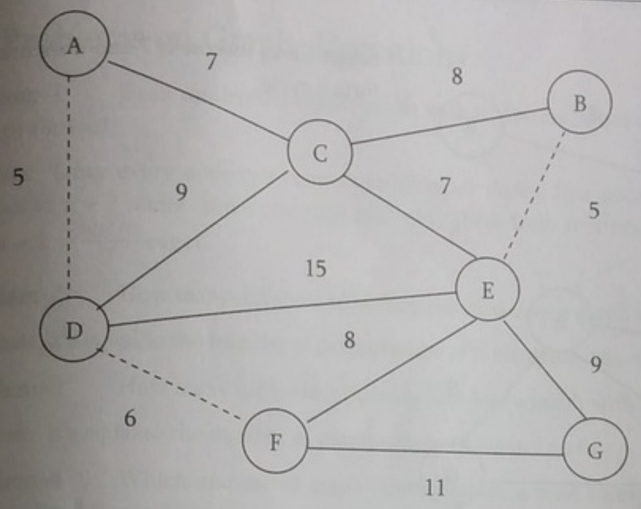

Now let us perform Kruskal's algorithm on this graph. We always select the edge which is having minimum weight.



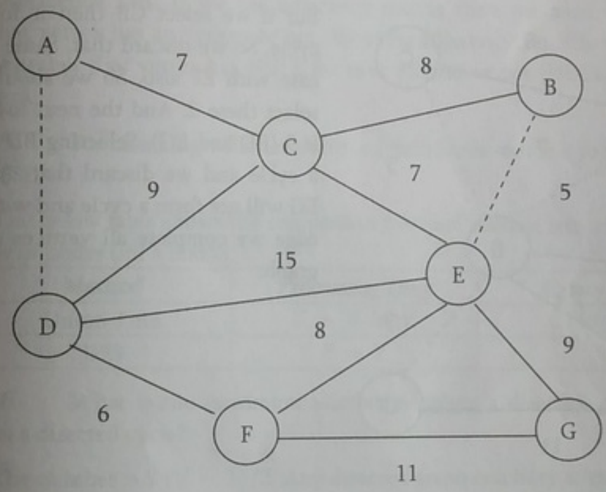

From the above graph, the edges which are having minimum weight (cost) are: AD and BE. Among these two we can select one of them and let us assume that we have selected AD (dotted line).



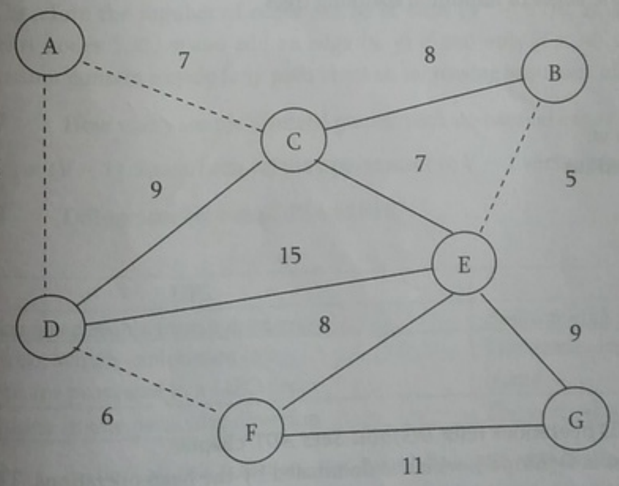




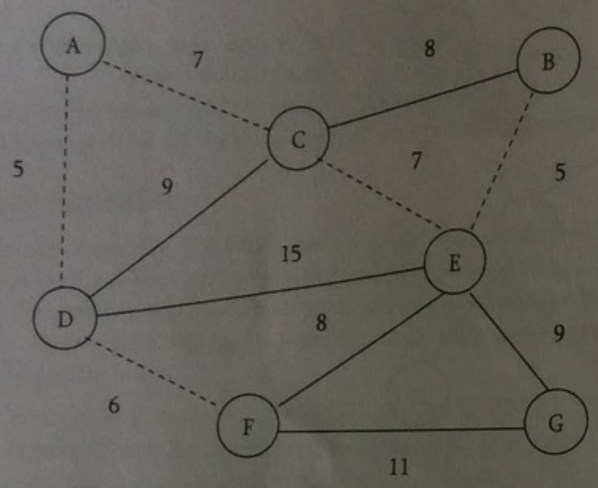

DF is the next edge which having the low cost (6).



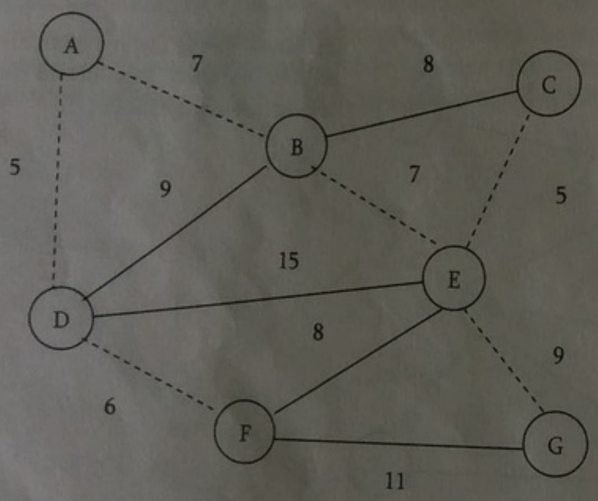

BE is now having the low cost among all and we select that (dotted lines indicates selected edges).




Next, AC and CE are having the low cost of 7 and let us assume that we have selected AC.




Select CE as its cost is 7 and does not form a cycle.




Next low cost edges are CB and EF. But if we select CB then it forms a cycle. So we discard that. Same is the case with EF also. So we should not select these 2. And the next low cost is 9 (BD and EG). Selecting BD forms a cycle and we discard that. Adding EG will not form a cycle and with this edge we complete all vertices of the graph.

```
void Kruskal(struct Graph *G) {
        S = ϕ;  // At the end S will contains the edges of minimum spanning trees
        for (int v = 0; v< G→V; v++)
           MakeSet (v);

        Sort edges of E by increasing weights w;
        for each edge (u, v) in E { //from sorted list
                if(FIND (u) ≠ FIND (v)) {
                        S = S ∪ {(u, v)};
                        UNION (u, v);
                }
        }
        return S;
}
```

**Note:** For implementation of UNION and FIND operations refer *Disjoint Sets ADT* chapter. The worst-case running time of this algorithm is $O(ElogE)$, which is dominated by the heap operations. That means, since we are constructing the heap with E edges we need $O(ElogE)$ time for doing that.





## 9.9 Problems on Graph Algorithms

**Problem-1** In an undirected simple graph with $n$ vertices, what is the maximum number of edges? Self loops are not allowed.

**Solution:** Since every node can connect to all other nodes, first node can connect o $n-1$ nodes. Second node can connect to $n-2$ nodes [since one edge is already there from first node]. The total number of edges is: $1+2+3+\ldots+n-1=\frac{n(n-1)}{2}$ edges.

**Problem-2** How many different adjacency matrices does a graph with $n$ vertices and $E$ edges have?

**Solution:** It's equal to the number of permutations of $n$ elements. i.e., $n!$.

**Problem-3** How many different adjacency lists does a graph with $n$ vertices have?

**Solution:** It's equal to the number of permutations of edges. i.e., $E!$.

**Problem-4** Which undirected graph representation is most appropriate for determining whether or not a vertex is isolated (is connected to no other vertex)?

**Solution: Adjacency List**. If we use adjacency matrix then we need to check the complete row for determining whether that vertex has any edges or not. By using adjacency list it is very easy to check and it can be done just by checking whether that vertex has NULL for next pointer or not [NULL indicates that vertex is not connected to any other vertex].

**Problem-5** For checking whether there is a path from source $s$ to target $t$, which one is best among disjoint sets and DFS?

**Solution:** The below table shows the comparison between disjoint sets and DFS. The entries in table represent the case for any pair of nodes (for $s$ and $t$).

| Method | Processing Time | Query Time | Space |
|---|---|---|---|
| Union-Find | $V + E\,logV$ | $logV$ | $V$ |
| DFS | $E + V$ | 1 | $E + V$ |

**Problem-6** What is the maximum number of edges a directed graph with $n$ vertices can have and still not contain a directed cycle?

**Solution:** The number is $V(V-1)/2$. Any directed graph can have at most $n^2$ edges. However, since the graph has no cycles it cannot contain a self loop and for any pair $x, y$ of vertices at most one edge from $(x, y)$ and $(y, x)$ can be included. Therefore the number of edges can be at most $(V^2-V)/2$ as desired. It is possible to achieve $V(V-1)/2$ edges. Label $n$ nodes $1, 2\ldots n$ and add an edge (x, y) if and only if $x < y$. This graph has the appropriate number of edges and cannot contain a cycle (any path visits an increasing sequence of nodes).

**Problem-7** How many simple directed graphs with no parallel edges and self loops are possible in terms of $V$?

**Solution:** $(V) \times (V-1)$. Since, each vertex can connect to $V-1$ vertices without self loops.

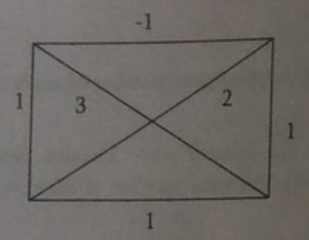
**Problem-8** Differences between DFS and BFS?

**Solution:**

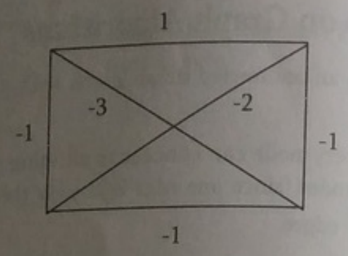
| DFS | BFS |
|---|---|
| Backtracking is possible from a dead end | Backtracking is not possible |
| Vertices from which exploration is incomplete are processed in a LIFO order | The vertices to be explored are organized as a FIFO queue |
| Search is done in one particular direction | The vertices at the same level are maintained in parallel |

**Problem-9** Earlier of this chapter, we have discussed minimum spanning tree algorithms. Now, give an algorithm for finding the maximum-weight spanning tree in a graph?

**Solution:**




Given graph




Transformed graph with negative edge weights

Using the given graph, construct a new graph with same nodes and edges. But instead of using same weights take the negative of their weights. That means, weight of an edge = negative of weight of the corresponding edge in the given graph. Now, we can use existing *minimum spanning tree* algorithms on this new graph. As a result, we will get the maximum weight spanning tree in the original one.

**Problem-10** Give an algorithm for checking whether a given graph $G$ has simple path from source $s$ to destination $d$. Assume the graph $G$ is represented using adjacent matrix.

**Solution:** Let us assume that the structure for the graph is:

```
struct Graph {
        int V;                  //Number of vertices
        int E;                  //Number of edges
        int ** adjMatrix;  //Two dimensional array for storing the connections
};
```

For each vertex call *DFS* and check whether the current vertex is same as destination vertex or not. If they are same, then return 1. Otherwise, call the *DFS* on its unvisited neighbors. One important thing to note here is that, we are calling the DFS algorithm on vertices which are not visited already.

```
void HasSimplePath(struct Graph *G, int s, int d) {
        int t;
        Viisited[s] = 1;
        if(s == d) return 1;
        for(t = 0; t < G→V; t++) {
                if(G→adjMatrix[s][t] && !Viisited[t])
                        if(DFS(G, t, d))
                                return 1;
        }
        return 0;
}
```

Time Complexity:O($E$). In the above algorithm, for each node since we are not calling *DFS* on all of its neighbors (discarding through *if* condition), Space Complexity: O($V$).

**Problem-11** Count simple paths for a given graph $G$ has simple path from source s to destination d? Assume the graph is represented using adjacent matrix.

**Solution:** As similar to the discussion of Problem-10, start at one node and call DFS on that node. As a result of this call, it visits all the nodes in the given graph which it can reach. That means it visits all the nodes of the connected component of that node. If there are any nodes left without visiting, then again start at one of those nodes and call DFS. Before the first call of DFS in each connected component, increment the connected components *count*. Continue this process until all of the graph nodes are visited. As a result, at the end we will get the total number of connected components. The implementation based on this logic is given below.

```
void CountSimplePaths(struct Graph * G, int s, int d) {
```





```
        int t;
        Viisited[s] = 1;
        if(s == d) {
                count++;
                Visited[s] = 0;
                return;
        }
        for(t = 0, t < G→V; t++) {
                if(G→adjMatrix[s][t] && !Viisited[t]) {
                        DFS(G, t, d);
                        Visited[t] = 0;
                }
        }
}
```

**Problem-12** **All pairs shortest path problem:** Find the shortest graph distances between every pair of vertices in a given graph. Let us assume that given graph doesn't have negative edges.

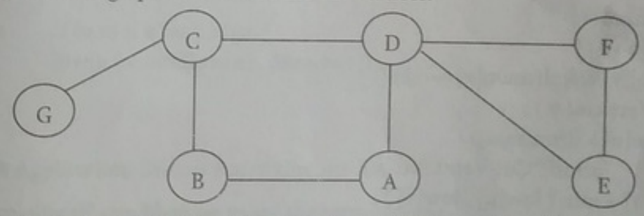
**Solution:** The problem can be solved using $n$ applications of *Dijkstra's* algorithm. That means we apply the *Dijkstra's* algorithm on each vertex of the given graph. This algorithm does not work if the graph has edges with negative weights.

**Problem-13** For the Problem-12, how do we solve the all pairs shortest path problem if the graph has edges with negative weights?

**Solution:** This can be solved by using $Floyd - Warshall\ algorithm$. This algorithm also works in the case of a weighted graph where the edges have negative weights. This algorithm is an example of Dynamic Programming and refer *Dynamic Programming* chapter.

**Problem-14** **DFS Application:** *Cut Vertex* or *Articulation Points*

**Solution:** In an undirected graph, a *cut vertex* (or articulation point) is a vertex and if we remove it then the graph splits into two disconnected components. As an example, consider the following figure. Removal of "$D$" vertex divides the graph in to two connected components ($\{E, F\}$ and $\{A, B, C, G\}$). Similarly, removal $C$ vertex divides the graph into ($\{G\}$ and $\{A, B, D, E, F\}$). For this graph $A$ and $C$ are the cut vertices.

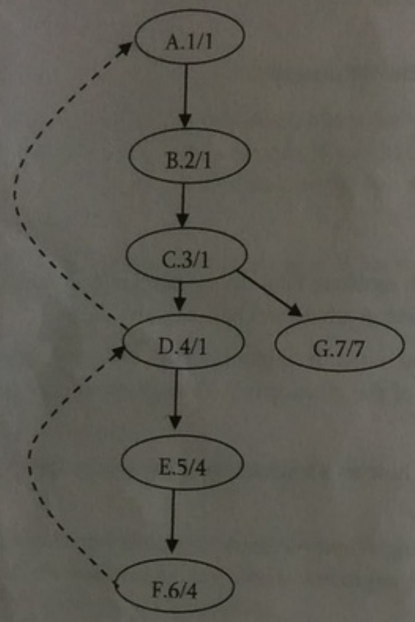



**Note:** A connected, undirected graph is called $bi-connected$ if the graph is still connected after removing any vertex.

*DFS* provides a linear-time algorithm (O($n$)) to find all cut vertices in a connected graph. Starting at any vertex, call a *DFS* and number the nodes as they are visited. For each vertex $v$, we call this DFS number $dfsnum$(v). The tree generated with DFS traversal is called *DFS spanning tree*. Then, for every vertex $v$ in the *DFS* spanning tree, we compute the lowest-numbered vertex, which we call $low(v)$, that is reachable from $v$ by taking zero or more tree edges and then possibly one back edge (in that order).

Based on the above discussion we need the following information for this algorithm. The $dfsnum$ of each vertex in the *DFS* tree (once it gets visited), and for each vertex $v$, the lowest depth of neighbors of all descendants of $v$ in the *DFS* tree, called the *low*. The $dfsnum$ can be computed during DFS. The low of $v$ can be computed after visiting all descendants of $v$ (i.e., just before $v$ gets popped off the *DFS* stack) as the minimum of the $dfsnum$ of all neighbors of $v$ (other than the parent of $v$ in the *DFS* tree) and the *low* of all children of $v$ in the *DFS* tree.

The root vertex is a cut vertex if and only if it has at least two children. A non-root vertex u is a cut vertex if and only if there is a son $v$ of $u$ such that $low(v) \geq dfsnum(u)$. This property can be tested once the *DFS* returned from every child of $u$ (that means, just before u gets popped off the DFS stack), and if true, $u$ separates the graph into different bi-connected components. This can be represented by computing one bi-connected component out of every such $v$ (a component which contains $v$ will contain the sub-tree of $v$, plus $u$), and then erasing the sub-tree of $v$ from the tree.




For the above graph, the *DFS* tree with $dfsnum/low$ can be given as shown in above figure. The implementation for the above discussion is:

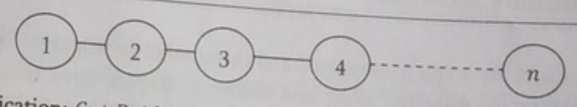
```
int adjMatrix [256] [256] ;
int dfsnum [256], num = 0, low [256];
void CutVertices( int u ) {
        low[u] = dfsnum[u] = num++;
        for (int v = 0 ; v < 256; ++v ) {
                if(adjMatrix[u][v] && dfsnum[v] == -1) {
                        CutVertices( v ) ;
                        if(low[v] > dfsnum[u])
                                printf("Cut Vetex:%d",u);
                        low[u] = min ( low[u] , low[v] ) ;
                }
                else    // (u,v) is a back edge
                        low[u ] = min(low[u] , dfsnum[v]) ;
        }
}
```

**Problem-15** Let $G$ be a connected graph of order $n$. What is the maximum number of cut-vertices that $G$ can contain?

**Solution:** $n - 2$. As an example consider the following graph. In the below graph except the vertices 1 and $n$ all the remaining vertices are cut vertices. This is because removing 1 and $n$ vertices do not split the graph into two. This is case where we can get maximum number of cut vertices.





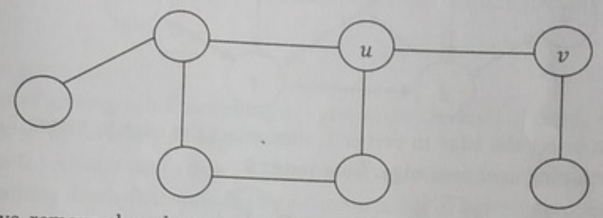



**Problem-16** **DFS Application:** *Cut Bridges* or *Cut Edges*

**Solution:**

**Definition:** Let $G$ be a connected graph. An edge $uv$ in $G$ is called a *bridge* of $G$ if $G - uv$ is disconnected.

As an example, consider the following graph.




In the above graph, if we remove the edge $uv$ then the graph splits into two components. For this graph, $uv$ is a bridge. The discussion we had for cut vertices holds good for bridges also. The only change is instead of printing the vertex we give the edge. The main observation is that an edge $(u, v)$ cannot be a bridge if it is part of a cycle. If$(u, v)$ is not part of a cycle then it is a bridge.

We can detect cycles in *DFS* by the presence of back edges. $(u, v)$ is a bridge if and only if none of v or $v's$ children has a back edge to $u$ or any of $u's$ ancestor. To detect whether any of $v's$ children has a back edge to $u's$ parent, we can use a similar idea above to see what is the smallest $dfsnum$ reachable from the subtree rooted at $v$.

```
int dfsnum[256], num = 0, low [256];
void Bridges( struct Graph *G, int u ) {
        low[u] = dfsnum[u] = num++;
        for (int v = 0 ; G→V; ++v ) {
                if(G→adjMatrix[u][v] && dfsnum[v] == -1) {
                        cutVertices( v ) ;
                        if(low[v] > dfsnum[u])
                                print (u,v)  as a bridge
                        low[u] = min ( low[u] , low[v] ) ;
                }
                else    // (u,v) is a back edge
                        low[u ] = min(low[u] , dfsnum[v]) ;
        }
}
```

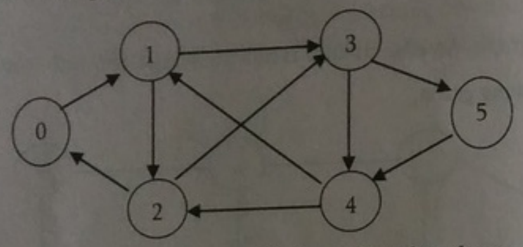
**Problem-17** **DFS Application:** Discuss *Euler* Circuits

**Solution:** Before discussing this problem let us see the terminology:

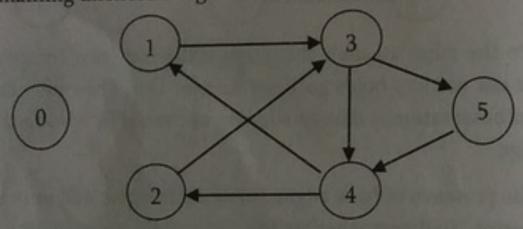
- *Eulerian tour* – a path that contains all edges without repetition.
- *Eulerian circuit* – a path that contains all edges without repetition starts and ends in the same vertex.
- *Eulerian graph* – a graph that contains an Eulerian circuit.
- *Even vertex*: a vertex that has an even number of incident edges.
- *Odd vertex*: a vertex that has an odd number of incident edges.

*Euler* circuit: For a given graph we have to reconstruct them using a pen, drawing each line exactly once. We should not lift the pen from the paper while the drawing. That means, we must find a path in the graph that visits every edge exactly once and this problem is called as an *Euler path* (also called as *Euler tour*) or *Euler circuit problem*. This puzzle has a simple solution based on DFS.

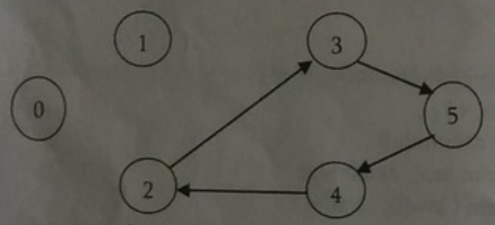
An *Euler* circuit exists if and only if the graph is connected and the number of neighbors of each vertex is even. Start with any node, select any untraversed outgoing edge, and follow it. Repeat until there are no more remaining unselected outgoing edges. For example, consider the following graph: A legal Euler Circuit of this graph is 0 1 3 4 1 2 3 5 4 2 0.




If we start at vertex 0, we can select the edge to vertex 1, then select the edge to vertex 2, then select the edge to vertex 0. There are now no remaining unchosen edges from vertex 0:




We now have a circuit 0,1,2,0 that does not traverse every edge. So, we pick some other vertex that is on that circuit, say vertex 1. We then do another depth first search of the remaining edges. Say we choose the edge to node 3, then 4, then 1. Again we are stuck there are no more unchosen edges from node 1. We now splice this path 1,3,4,1 into the old path 0,1,2,0 to get: 0,1,3,4,1,2,0. The unchosen edges now look like this:




We can pick yet another vertex to start another DFS. If we pick vertex 2, and splice the path 2,3,5,4,2, then we get the final circuit 0,1,3,4,1,2,3,5,4,2,0.

A very similar problem is to find a simple cycle, in an undirected graph that visits every vertex. This is known as the *Hamiltonian cycle problem*. Although it seems almost identical to the *Euler* circuit problem, no efficient algorithm for it is known.

**Notes:**

- A connected undirected graph is *Eulerian* if and only if every graph vertex has an even degree, or exactly two vertices with odd degree.
- A directed graph is *Eulerian* if it is strongly connected and every vertex has equal *in* and *out* degree.

**Application:** A postman has to visit a set of streets in order to deliver mails and packages. It is needed to find a path that starts and ends at the post-office, and that passes through each street (edge) exactly once. This way the postman will deliver mails and packages to all streets he has to, and in the same time will spend minimum efforts/time for the road.

**Problem-18** **DFS Application:** Finding Strongly Connected Components.

**Solution:** This is another application of DFS. In a directed graph, two vertices $u$ and $v$ are strongly connected if and only if there exists a path from $u$ to $v$ and there exists a path from $v$ to $u$. The strongly connectedness is an equivalence relation.





- A vertex is strongly connected with itself
- If a vertex $u$ is strongly connected to a vertex $v$, then $v$ is strongly connected to $u$
- If a vertex $u$ is strongly connected to a vertex $v$ and $v$ is strongly connected to a vertex $x$, then $u$ is strongly connected to $x$

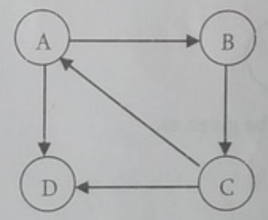
What this says is, for a given directed graph we can divide it into strongly connected components. This problem can be solved by performing two depth-first searches. With two DFS searches we can test whether a given directed graph is strongly connected or not. We can also produce the subsets of vertices that are strongly connected.

**Algorithm**

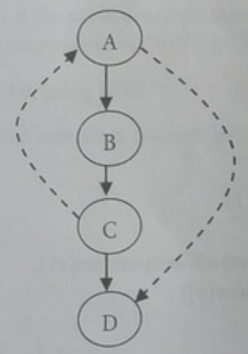
- Perform DFS on given graph $G$
- Number vertices of given graph $G$ according to a post-order traversal of depth-first spanning forest.
- Construct graph $G_r$ by reversing all edges in $G$
- Perform DFS on $G_r$: Always start a new DFS (initial call to Visit) at the highest-numbered vertex
- Each tree in resulting depth-first spanning forest corresponds to a strongly-connected component.

**Why this algorithm works?**

Let us consider two vertices, $v$ and $w$. If they are in same strongly connected component, then there are paths from $v$ to $w$ and from $w$ to $v$ in the original graph $G$, and hence also in $G_r$. If two vertices $v$ and $w$ are not in the same depth-first spanning tree of $G_r$, clearly they cannot be in the same strongly connected component. As an example, consider the graph shown below. Let us assume this graph is $G$.



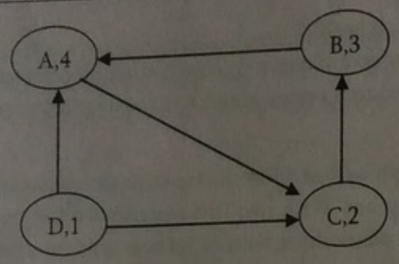

Now, as per the algorithm, performing *DFS* on this graph $G$ produces below diagram. The dotted line from $C$ to $A$ indicates a back edge.



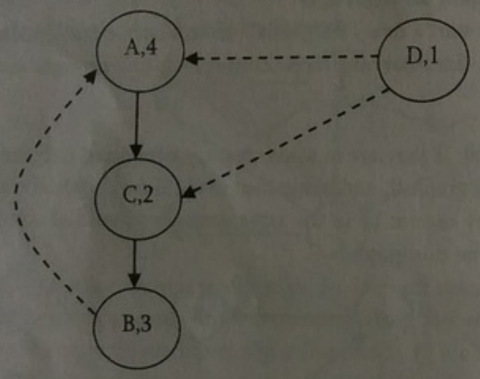

Now, performing post order traversal on this tree gives: $D, C, B$ and $A$.

| Vertex | Post Order Number |
|---|---|
| A | 4 |
| B | 3 |
| C | 2 |
| D | 1 |

Now reverse the given graph, $G$ and call it as $G_r$ and at the same time assign postorder numbers to the vertices. The reversed graph $G_r$ will look like:




Last step is, performing DFS on this revered graph $G_r$. While doing *DFS* we need to consider the vertex which has largest DFS number. So, first we start at $A$ and with *DFS* we go to $C$ and then $B$. At B, we cannot move further. This says that $\{A, B, C\}$ is strongly connected component. Now the only remaining element is $D$ and we end our second *DFS* at $D$ itself. So the connected components are: $\{A, B, C\}$ and $\{D\}$.




The implementation based on this discussion can be given as:

```
//Graph represented in adj matrix.
int adjMatrix [256][256], table[256];
vector <int> st ;
int counter = 0 ;
//This table contains the DFS Search number
int dfsnum [256], num = 0, low[256] ;
void StronglyConnectedComponents( int u ) {
        low[u] = dfsnum[ u ] = num++;
        Push(st, u) ;
        for( int v = 0 ; v < 256; ++v ) {
                if(graph[u][v] && table[v] == -1) {
                        if( dfsnum[v] == -1)
                                StronglyConnectedComponents(v) ;
                        low[u] = min(low[u] , low[v]) ;
                }
        }
        if(low[u] == dfsnum[u]) {
                while( table[u] != counter) {
                        table[st.back()] = counter;
                        Push(st) ;
                }
                ++ counter;
        }
}
```

**Problem-19** Count the number of connected components of Graph $G$ which is represented in adjacent matrix.





**Solution:** This problem can be solved with one extra counter in *DFS*.

```
//Visited[] is a global array.
int Visited[G→V];
void DFS(struct Graph *G, int u) {
        Visited[u] = 1;
        for( int v = 0; v < G→V; v++ ) {
                /* For example, if the adjacency matrix is used for representing the
                   graph, then the condition to be used for finding unvisited adjacent
                   vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        DFS(v);
                }
        }
}
void DFSTraversal(struct Graph *G) {
        int count = 0;
        for (int i = 0; i< G→V;i++)
                Visited[i]=0;
        //This loop is required if the graph has more than one component
        for (int i = 0; i< G→V;i++)
                if(!Visited[i]) {
                        DFS(G, i);
                        count++;
                }

        return count;
}
```

Time Complexity: Same as that of DFS and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

**Problem-20** Can we solve the Problem-19, using BFS?

**Solution: Yes.** This problem can be solved with one extra counter in BFS.

```
void BFS(struct Graph *G, int u) {
        int v,
        Queue Q = CreateQueue();
        EnQueue(Q, u);
        while(!IsEmptyQueue(Q)) {
                u = DeQueue(Q);
                Process u; //For example, print
                Visited[s]=1;
                /* For example, if the adjacency matrix is used for representing the
                   graph, then the condition be used for finding unvisited adjacent
                   vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
                for each unvisited adjacent node v of u {
                        EnQueue(Q, v);
                }
        }
}
```

```
void BFSTraversal(struct Graph *G) {
        for (int i = 0; i< G→V;i++)
                Visited[i]=0;
        //This loop is required if the graph has more than one component
        for (int i = 0; i< G→V; i++)
                if(!Visited[i])
                        BFS(G, i);
}
```

Time Complexity: Same as that of *BFS* and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

**Problem-21** Let us assume that $G(V, E)$ is an undirected graph. Give an algorithm for finding a spanning tree which takes $O(|E|)$ time complexity (not necessarily a minimum spanning tree).

**Solution:** The test for a cycle can be done in constant time, by marking vertices that have been added to the set $S$. An edge will introduce a cycle, if both its vertices have already been marked.

**Algorithm:**

```
S = {}; //Assume S is a set
for each edge e ∈ E {
        if(adding e to S doesn't form a cycle) {
                add e to S;
                mark e;
        }
}
```

**Problem-22** Is there any other way of solving the Problem-20?

**Solution:** Yes. We can run *BFS* and find the *BFS* tree for the graph (level order tree of the graph). Then start at the root element and keep moving to next levels and at the same time we have to consider the nodes in the next level only once. That means, if we have a node with multiple input edges then we should consider only of them, otherwise they will form a cycle.

**Problem-23** Detecting a cycle in undirected graph

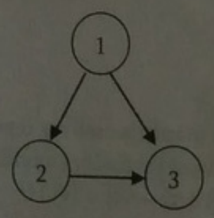
**Solution:** An undirected graph is acyclic if and only if a *DFS* yields no back edges, edges $(u, v)$ where $v$ has already been discovered and is $u$ ancestor.

- Execute *DFS* on the graph.
- If there is a back edge - the graph has a cycle.

If the graph does not contain a cycle then $|E| < |V|$ and *DFS* cost $O(|V|)$. If the graph contains a cycle, then a back edge is discovered after $2|V|$ steps at most.

**Problem-24** Detecting a cycle in DAG

**Solution:**




Cycle detection on a graph is a different than on a tree. This is because in graph a node can have multiple parents. In a tree, the algorithm for detecting a cycle is to do a depth first search, marking nodes as they are encountered. If a previously marked node is seen again, then a cycle exists. This won't work on a graph. Let us consider the graph shown in below figure. If we use tree cycle detection algorithm, then it will report wrong result. That means, it says





that this graph has cycle in it. But the given graph does not have any cycle in it. This is because node 3 will be seen twice in a *DFS* starting at node 1.

The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a *DFS* of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. But, if a node is seen a second time before all of its descendants has been visited, then there must be a cycle.

Can you see why this is? Suppose there is a cycle containing node A. This means that A must be reachable from one of its descendants. So when the *DFS* is visiting that descendant, it will see *A* again, before it has finished visiting all of *A's* descendants. So there is a cycle.

In order to detect cycles, we can modify the depth first search.

```
int DetectCycle(struct Graph *G) {
        for (int i = 0; i< G→V; i++) {
                Visited[s]=0;
                Predecessor[i] = 0;
        }
        for (int i = 0; i < G→V;i++) {
                if(!Visited[i] && HasCycle(G, i))
                        return 1;
        }
        return false;;
}
int HasCycle(struct Graph *G, int u) {
        Visited[u]=1;
        for (int i = 0; i< G→V; i++) {
                if(G→Adj[s][i]) {
                        if(Predecessor[i] != u && Visited[i])
                                return 1;
                        else {
                                Predecessor[i] = u;
                                return  HasCycle(G, i);
                        }
                }
        return 0;
}
```

Time Complexity: $O(V + E)$.

**Problem-25** Given a directed acyclic graph, give an algorithm for finding the depth of it.

**Solution:** We can solve this problem by following the similar approach which we used for finding the depth in trees. In trees, we have solved this problem using level order traversal (with one extra special symbol to indicate the end of the level).

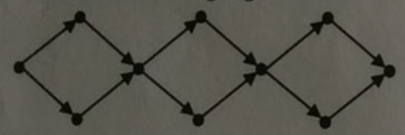
```
//Assuming the given  graph is a DAG
int DepthInDAG( struct Graph *G ) {
        struct Queue *Q;
        int counter;
        int v, w;
        Q = CreateQueue();
        counter =  0;
        for (v = 0; v< G→V; v++)
                if( indegree[v] ==  0 )
```

```
                EnQueue( Q, v );
        EnQueue( Q, '$' );
        while( !IsEmptyQueue( Q )) {
                v = DeQueue( Q );
                if(v == '$') {
                        counter++;
                        if(!IsEmptyQueue( Q ))
                                EnQueue( Q, '$' );

                }
                for each w adjacent to v
                        if( --indegree[w] == 0 )
                                EnQueue ( Q, w );
        }
        DeleteQueue( Q );
        return counter;
}
```

Total running time is $O(V + E)$.

**Problem-26** How many topological sorts of the following dag are there?



**Solution:** If we observer the above graph there are three stages with 2 vertices. In the early discussion of this chapter, we have seen that topological sort picks the elements with zero indegree at any point of time. At each of the two vertices stages, we can either first process the top vertex of bottom vertex. As a result at each of these stages we have two possibilities. So the total number of possibilities is the multiplication of possibilities at each stage and that is, $2 \times 2 \times 2 = 8$.

**Problem-27** **Unique topological ordering:** Design an algorithm to determine whether a directed graph has a unique topological ordering.

**Solution:** A directed graph has a unique topological ordering if and only if there is a directed edge between each pair of consecutive vertices in the topological order. This can also be defined as: a directed graph has a unique topological ordering if and only if it has a Hamiltonian path. If the digraph has multiple topological orderings, then a second topological order can be obtained by swapping a pair of consecutive vertices.

**Problem-28** Suppose let us consider the courses prerequisites at *IIT Bombay*. Suppose that all prerequisites must be obeyed, every course is offered every semester, and there is no limit to the number of courses we can take in one semester. We would like to know the minimum number of semesters required to complete the major. Describe the data structure we would use to represent this problem, and outline a linear time algorithm for solving it.

**Solution:** Use a directed acyclic graph (DAG). The vertices represent courses and the edges represent the prerequisite relation between courses at *IIT Bombay*. It is a DAG, because the prerequisite relation has no cycles.

The number of semesters required to complete the major is one more than the longest path in the dag. This can be calculated on the DFS tree recursively in linear time. The longest path out of a vertex $x$ is 0 if $x$ has outdegree 0, otherwise it is $1 + max\{longest\ path\ out\ of\ y \mid (x,y)\ is\ an\ edge\ of\ G\}$.

**Problem-29** At one of the universities (say, *IIT Bombay*), there are a list of courses along with their prerequisites. That means, two lists are given:





$A$ - Courses list

$B$ – Prerequisites: B contains couples $(x, y)$ where $x, y \in A$ indicating that course $x$ can't be taken before course $y$.

Let us consider a student who wants to take only one course in a semester. Design a schedule for this student.

**Example:** A = {C-Lang, Data Structures, OS, CO, Algorithms, Design Patterns, Programming }. B = { (C-Lang, CO), (OS, CO), (Data Structures, Algorithms), (Design Patterns, Programming) }. *One possible schedule could be*:

Semester 1: Data Structures
Semester 2: Algorithms
Semester 3: C-Lang
Semester 4: OS
Semester 5: CO
Semester 6: Design Patterns
Semester 7: Programming

**Solution:** The solution to this problem is exactly same as that of topological sort. Assume that the courses names are integers in the range $[1..n]$, $n$ is known ($n$ is not constant). The relations between the courses will be represented by a directed graph $G = (V, E)$, where $V$ are the set of courses and if course $i$ is prerequisite of course $j$, $E$ will contain the edge $(i, j)$. Let us assume that the graph will be represented as an Adjacency list.

First, let's observe another algorithm to topologically sort a DAG in $O(|V| + |E|)$.

- Find in-degree of all the vertices - $O(|V| + |E|)$
- Repeat:
  Find a vertex v with in-degree=0 - $O(|V|)$
  Output v and remove it from G, along with its edges - $O(|V|)$
  Reduce the in-degree of each node u such as $(v, u)$ was an edge in G and keep a list of vertices with in-degree=0 - $O(degree(v))$
  Repeat the process until all the vertices were removed

Time complexity of this algorithm is also same as that of topological sort and it is $O(|V| + |E|)$.

**Problem-30** In Problem-29, a student who wants to take all the courses in $A$, in the minimal number of semesters. That means, student is ready to take any number of courses in a semester. Design a schedule for this scenario. *One possible schedule is*:
*Semester* 1: C-Lang, OS, Design Patterns
*Semester* 2: Data Structures, CO, Programming
*Semester* 3: Algorithms

**Solution:** A variation of the above topological sort algorithm with a slight change: In each semester, instead of taking one subject, take all the subjects with zero indegree. That means, execute the algorithm on all the nodes with degree 0 (instead of dealing with one source in each stage, all the sources will be dealt and printed).

Time Complexity: $O(|V| + |E|)$.

**Problem-31** **LCA of a DAG:** Given a DAG and two vertices $v$ and $w$, find the *lowest common ancestor* (LCA) of $v$ and $w$. The LCA of $v$ and $w$ is an ancestor of $v$ and $w$ that has no descendants which are also ancestors of $v$ and $w$.

**Hint:** Define the height of a vertex $v$ in a DAG to be the length of the longest path from *root* to $v$. Among the vertices that are ancestors of both $v$ and $w$, the one with the greatest height is an LCA of $v$ and $w$.

**Problem-32** **Shortest ancestral path:** Given a DAG and two vertices $v$ and $w$, find the *shortest ancestral path* between $v$ and $w$. An ancestral path between $v$ and $w$ is a common ancestor $x$ along with a shortest path from $v$ to $x$ and a shortest path from $w$ to $x$. The shortest ancestral path is the ancestral path whose total length is minimized.