Applications of Graphs

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

1. Social network graphs: to tweet or not to tweet. Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who follows whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.
2. Transportation networks. In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as Google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.
3. Utility graphs. The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
4. Robot planning. Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles
5. Neural networks. Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.
6. Graphs in compilers. Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

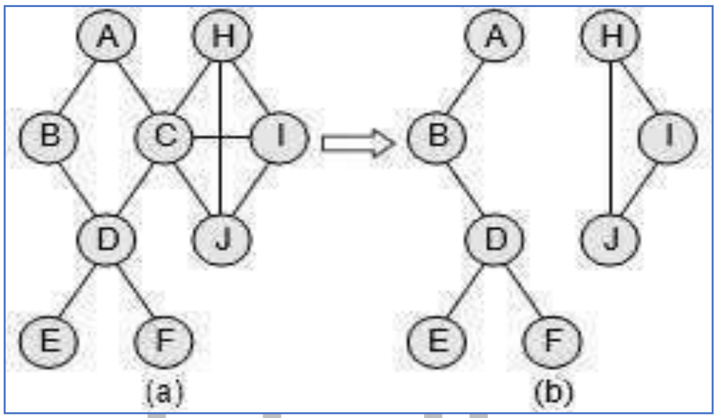
www.binils.com

BI- CONNECTIVITY

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

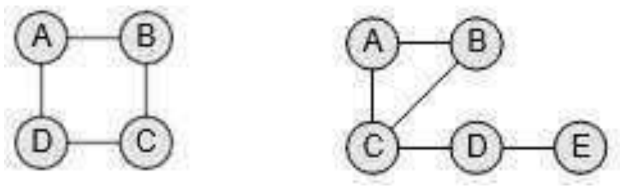
A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

- A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.
- In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

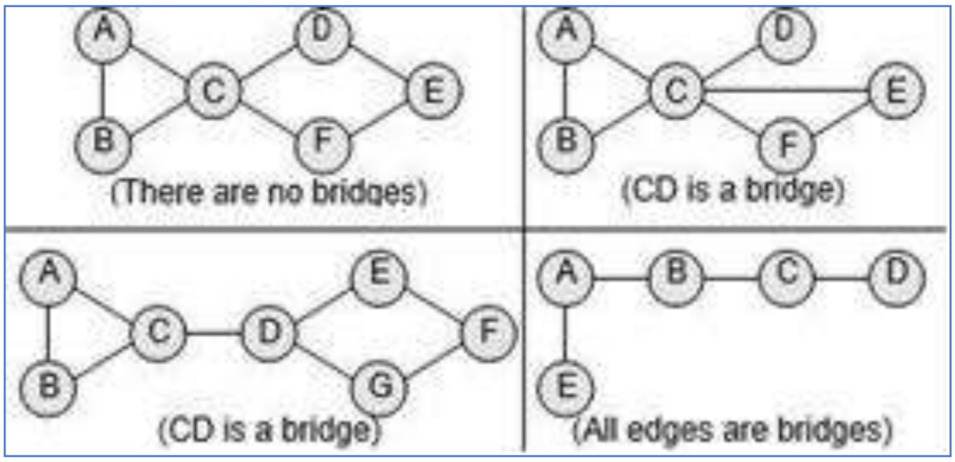


www.binils.com

Note that the graph is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph



Biconnected Graph



Graph with Bridges

As for vertices, there is a related concept for edges. An edge in a graph is called a bridge if removing that edge results in a disconnected graph. Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge.

CUT VERTEX

www.binils.com

Articulation point

The vertices whose removal would disconnect the graph are known as articulation points.

Steps to find Articulation point

1. Find DFS spanning tree
2. Number the vertex in the order in which they are visited. This number is referred as Num(v)
3. Compute the lowest numbered vertex for every vertex v in the DFS spanning tree which we call as Low(w)(i.e) reachable from v by taking 0 or more tree edges and then possible one back edge. By definition Low(v) is the
    a. Minimum of Num(v)
    b. The lowest Num(w) among all back edges
    c. The lowest Low(w) among all tree edges (i.e) min(Num(v),Num(w),Low(w))

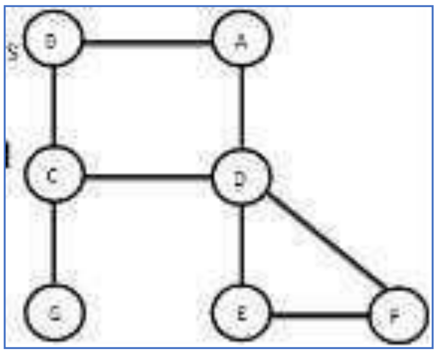
Use post order traversal to calculate Low(v). The root is articulation if and only if it has more than 2 children.

Any vertex V other than root is an articulation point if and only if V has some children such that

Low(w)≥Num(v).

Example

Checking whether finding the articulation point



Low(v)=min(Num(v),Num(w),Low(w))

Low(F)=min(Num(F),Num(D),Low(D)) =min(6,4) =4

Low(E)=min(Num(E),Num(F),Low(F)) =min(5,6,4) =4

Low(D)=min(Num(D),Num(E),Low(E),Num(A),Low(A)) =min(4,5,4,1) =1

Low(G)=min(Num(G)) =min(7) =7

Low(C)=min(Num(C),Num(D),Low(D),Num(G),Low(G)) =min(3,4,1,7,7) =1

Low(B)=min(Num(B),Num(C),Low(C)) =min(2,3,1) =1

Low(A)=min(Num(A),Num(B),Low(B))

=min(1,2,1) =1 At vertex F Low(W) ≥Num(V) 1 ≥

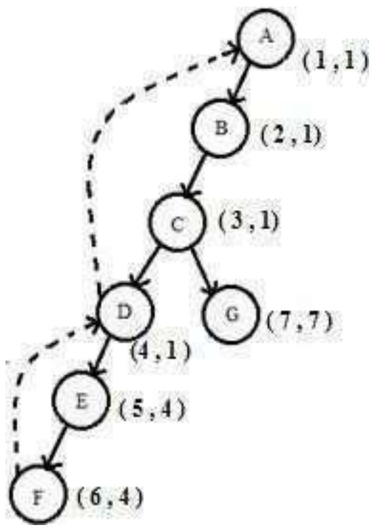
6

At vertex D Low(A) ≥ Num(D) 1 ≥ 4

Low(E) ≥ Num(D) 4 ≥ 4(articulation
point) At vertex E Low(F) ≥ Num(E) 4
≥5
At vertex C Low(D) ≥ Num(C) 1 ≥3

www.binils.com

Low(G) ≥Num(C) 6 ≥ 3(articulation point)



DFS Spanning Tree

www.binils.com

DEPTH-FIRST SEARCH ALGORITHM

- The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
- In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
- The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead-end, we backtrack to find another path P¢. The algorithm terminates when backtracking leads back to the starting node

  A. In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges.

www.binils.com

STEPS

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2

(waiting state) Step 3: Repeat Steps 4 and 5 until STACK is empty

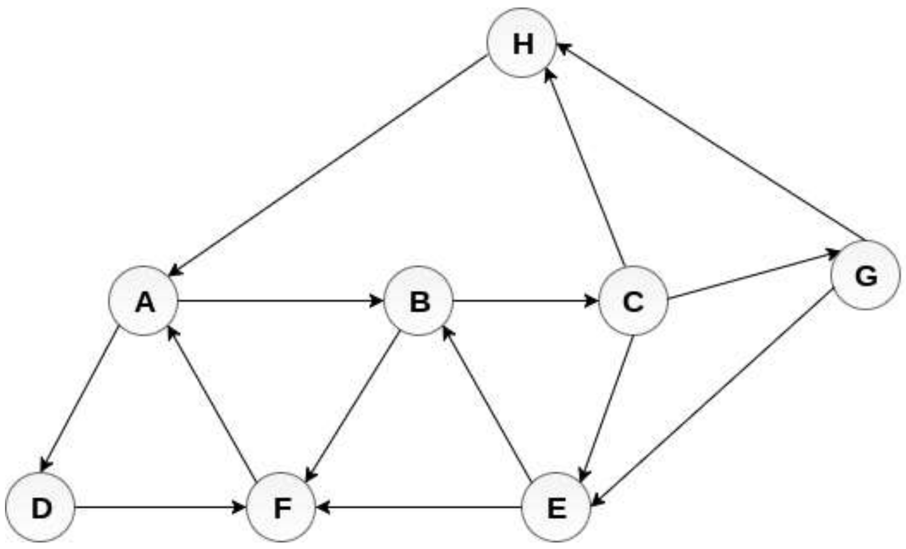
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]

Step 6: EXIT

EXAMPLE

Consider the graph G along with its adjacency list, given in the figure below. Calculate the order to print all the nodes of the graph starting from node H, by using depth first search **(DFS) algorithm.**



Adjacency Lists

A	B, D
B	C, F
C	E, G, H
G	E, H
E	B, F
F	A
D	F
H	A

Solution :

Push H onto the stack

STACK : H

www.binils.com

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

Print H
STACK :
A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

Print A
Stack : B,
D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

Print D
Stack : B,
F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

Print F

Stack :

B

Pop the top of the stack i.e. B and push all the

neighbours Print B

Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

Print C

Stack : E,

G

www.binils.com

Pop the top of the stack i.e. G and push all its neighbours.

Print G

Stack :

E

Pop the top of the stack i.e. E and push all its neighbours.

Print

E

Stack

:

Hence, the stack now becomes empty and all the nodes of the graph have been

traversed. The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

www.binils.com for Anna University | Polytechnic and Schools

Applications of Depth-First Search

Algorithm Depth-first search is useful

for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph

www.binils.com

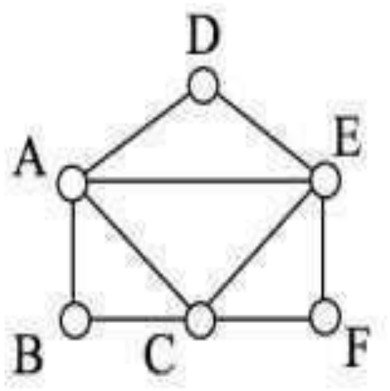
Download Binils Android App in Playstore Download Photoplex App

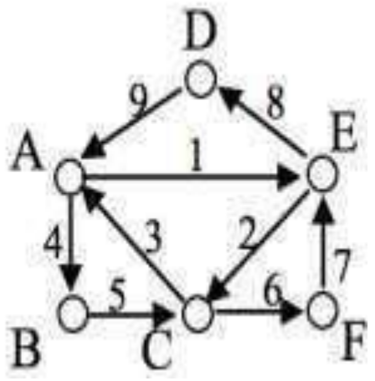
EULER CIRCUIT

An Euler circuit is a circuit that uses every edge in a graph with no repeats. Being a circuit, it must start and end at the same vertex.

EXAMPLE

The graph below has several possible Euler circuits. Here's a couple, starting and ending at vertex A: ADEACEFCBA and AECABCFEDA. The second is shown in arrows.





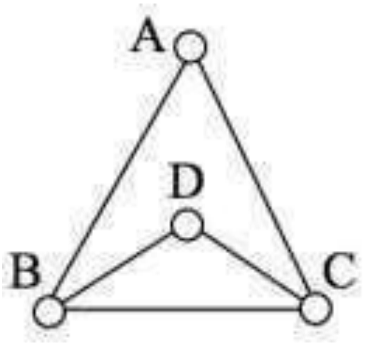
www.binils.com

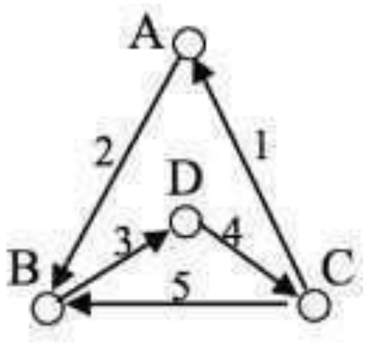
EULER PATH

An Euler path is a path that uses every edge in a graph with no repeats. Being a path, it does not have to return to the starting vertex.

EXAMPLE

In the graph shown below, there are several Euler paths. One such path is CABDCB. The path is shown in arrows to the right, with the order of edges numbered.



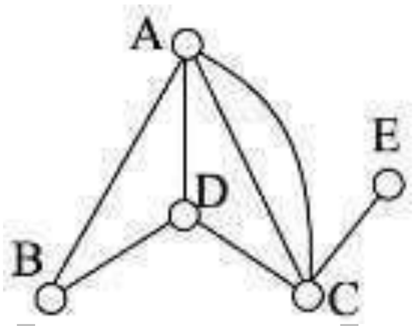


EULER'S PATH AND CIRCUIT THEOREMS

- A graph will contain an Euler path if it contains at most two vertices of odd degree.
- A graph will contain an Euler circuit if all vertices have even degree

EXAMPLE

In the graph below, vertices A and C have degree 4, since there are 4 edges leading into each vertex. B is degree 2, D is degree 3, and E is degree 1. This graph contains two vertices with odd degree (D and E) and three vertices with even degree (A, B, and C), so Euler's theorems tell us this graph has an Euler path, but not an Euler circuit.



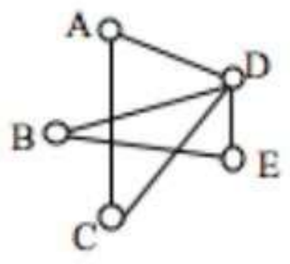
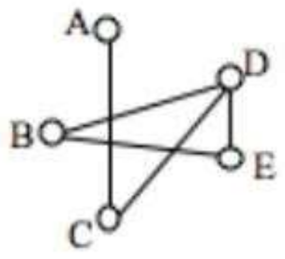
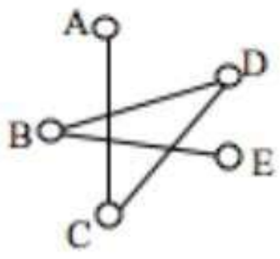
www.binils.com

FLEURY'S ALGORITHM

1. Start at any vertex if finding an Euler circuit. If finding an Euler path, start at one of the two vertices with odd degree.
2. Choose any edge leaving your current vertex, provided deleting that edge will not separate the graph into two disconnected sets of edges.
3. Add that edge to your circuit, and delete it from the graph.
4. Continue until you're done.

EXAMPLE

Find an Euler Circuit on this graph using Fleury's algorithm, starting at vertex A.

Original Graph. Choosing edge AD.	AD deleted. D is current. Can't choose DC since that would disconnect graph. Choosing DE	E is current. From here, there is only one option, so the rest of the circuit is determined.
		
Circuit so far: AD	Circuit so far: ADE	Circuit: ADEBDCA

HAMILTONIAN CIRCUITS AND PATHS

A Hamiltonian circuit is a circuit that visits every vertex once with no repeats. Being a circuit, it must start and end at the same vertex. A Hamiltonian path also visits every vertex once with no repeats but does not have to start and end at the same vertex.

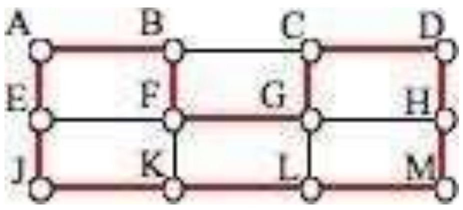
www.binils.com

Hamiltonian circuits are named for William Rowan Hamilton who studied them in the 1800's.

EXAMPLE

One Hamiltonian circuit is shown on the graph below. There are several other Hamiltonian circuits possible on this graph. Notice that the circuit only has to visit every vertex once; it does not need to use every edge.

This circuit could be notated by the sequence of vertices visited, starting and ending at the same vertex: ABFGCDHMLKJEA. Notice that the same circuit could be written in reverse order or starting and ending at a different vertex.



www.binils.com for Anna University | Polytechnic and Schools

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

- Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n - 1)! number of possibilities.
- Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.
- Let us consider a graph G = (V, E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.
- Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.
- For a subset of cities S Є {1, 2, 3, ... , n} that includes 1, and j Є S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.
- When |S| > 1, we define C(S, 1) = ∝ since the path cannot start and end at 1.

www.binils.com

Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that

$$\boxed{\begin{aligned} C(S,j) &= min\, C(S-\{j\},i)+d(i,j)\ where\ i \in S\ and\ i \neq jc(S,j) \\ &= minC(s-\{j\},i)+d(i,j)\ where\ i \in S\ and\ i \neq j \end{aligned}}$$

Download Binils Android App in Playstore Download Photoplex App

Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and
    containing 1 C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

www.binils.com

GRAPH TRAVERSAL ALGORITHMS

There are two standard methods of graph traversal, they are

1. Breadth-first search
2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth- first search scheme uses a stack. But both these algorithms make use of a variable STATUS. During the execution of the algorithm, every node in the graph will have the variable STATUS set to 1 or 2, depending on its current state.

BREADTH-FIRST SEARCH ALGORITHM

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

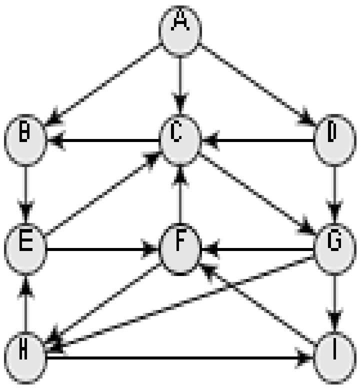
Algorithm to breadth-first Search

```
Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2

(waiting state) Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set
their STATUS

= 2 (waiting state) [END OF LOOP]
```

Step 6: EXIT

Example

Consider the graph G given. The adjacency list of G is also given. Assume that G represents the daily flights between different cities and we want to fly from city A to I with minimum stops. That is, find the minimum path P from A to I given that every edge has a length of 1.



Adjacency List
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays:

QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be

processed, ORIG is used to keep track of the origin of each edge.

Initially, FRONT = REAR = –1.

The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

FRONT = 0	QUEUE = A
REAR = 0	ORIG \0

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueue the neighbours of A. Also, add A as the ORIG of its neighbours.

```
FRONT = 1      QUEUE = A    B    C    D

REAR = 3       ORIG   \0    A    A    A
```

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its

neighbours.

```
FRONT =    QUEUE =    B    C    D    E
2                A

REAR = 4    ORIG  \0    A    A    A    B
```

d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

```
FRONT = 3 QUEUE = A   B    C    D    E    G

REAR = 5   ORIG  \0    A    A    A    B

           C
```

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also, add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

```
FRONT = 4    QUEUE = A     B    C    D    E

           G REAR = 5   ORIG \0    A    A

           A          B         C
```

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

```
FRONT = 5 QUEUE = A    B    C    D    E    G
           FREAR = 6   ORIG \0    A    A    A    B
           C    E
```

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT=6	QUEUE =	A	B	C	D	E	G	F	H	I
REAR = 9	ORIG	\0	A	A	A	B	C	E	G	G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE. Now, backtrack from I using ORIG to find the minimum path P. Thus, we have P as A -> C -> G -> I

Features of Breadth-First Search

Algorithm Space complexity

- In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is therefore proportional to the number of nodes at the deepest level of the graph. Given a graph with branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level O(bd).
- If the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as O (| E | + | V |), where | E | is the total number of edges in G and | V | is the number of nodes or vertices.

Time complexity

In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity of this algorithm asymptotically approaches O(bd). However, the time complexity can also be expressed as O(| E | +
| V |), since every vertex and every edge will be explored in the worst case.

Completeness

Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality

Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

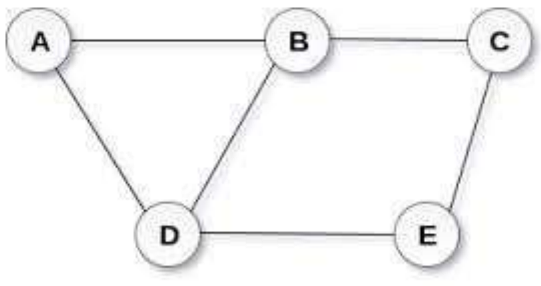
Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph

GRAPHS

DEFINITION

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.



Undirected Graph

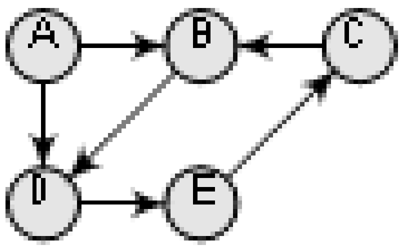
In the above graph, V(G) = {A, B, C, D and E} and E(G) = {(A, B), (B, C), (A, D), (B, D), (D, E), (C,E)}.

There are five vertices or nodes and six edges in the graph.

www.binils.com

DIRECTED AND UNDIRECTED GRAPHS

- In an undirected graph, edges do not have any direction associated with them. That is, if an edge is drawn between nodes A and B, then the nodes can be traversed from A to B as well as from B to A.
- In a directed graph, edges form an ordered pair. If there is an edge from A to B, then there is a path from A to B but not from B to A. The edge (A, B) is said to initiate from node A (also known as initial node) and terminate at node B (terminal node).



Directed Graphs

GRAPH TERMINOLOGY

Adjacent nodes or neighbours

For every edge, e = (u, v) that connects nodes u and v, the nodes u and v are the end- points and are said to be the adjacent nodes or neighbours.

Degree of a node

Degree of a node u, deg(u), is the total number of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an isolated node.

Closed path

A path P is known as a closed path if the edge has the same end-points. That is, if vo=vn.

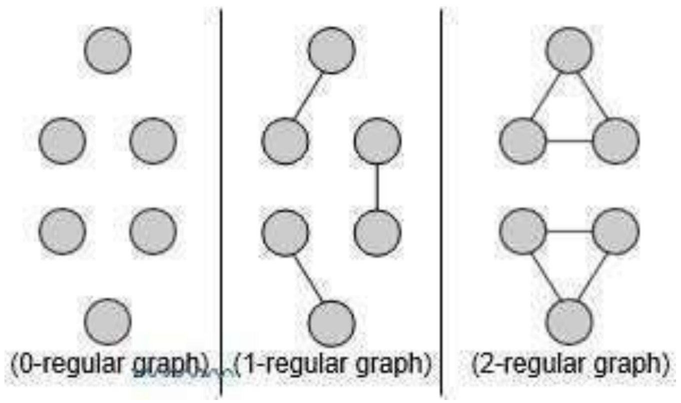
Simple path

A path P is known as a simple path if all the nodes in the path are distinct with an exception that v0 may be equal to vn. If v0= vn then the path is called a closed simple path.

www.binils.com

Cycle

A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices (except the first and last vertices).



Regular Graph

Types of Graphs

Connected graph

A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say that there are no isolated nodes in a connected graph. A connected graph that does not have any cycle is called a tree.

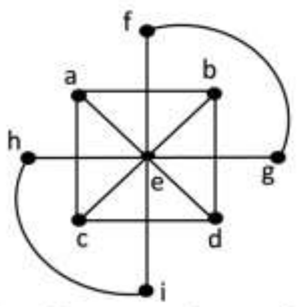


Fig: Connected graph

www.binils.com

Complete graph

A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n-1)/2 edges, where n is the number of nodes in G.

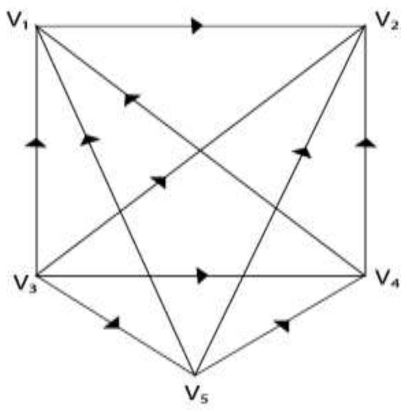


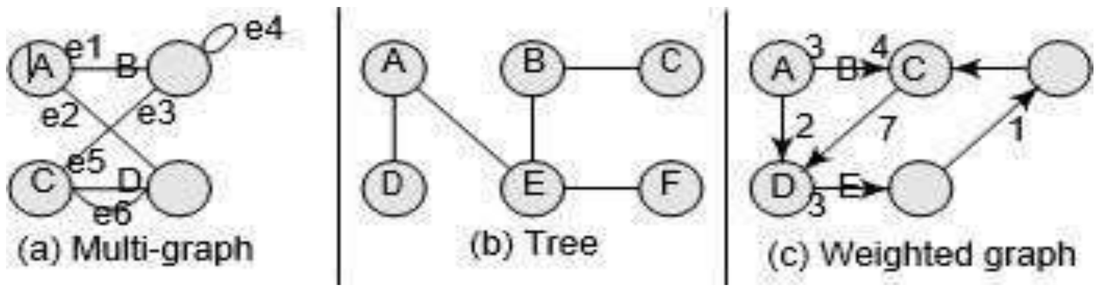
Fig: K_5

Clique

In a undirected graph G=(V, E), clique is a subset of the vertex, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled graph or weighted graph

A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge denoted by w(e) is a positive value which indicates the cost of traversing the edge.



Multiple edges

www.binils.com

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

Loop

An edge that has identical end-points is called a loop. That is, e = (u, u).

Multi-graph

A graph with multiple edges and/or loops is called a multi-graph.

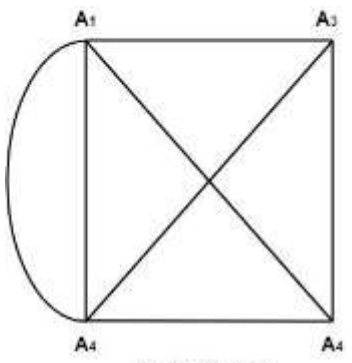


Fig:Multigraph

Size of a graph

The size of a graph is the total number of edges in it.

DIRECTED GRAPHS

A directed graph G, also known as a digraph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other.

Terminology of a Directed graph

Out-degree of a node - The out-degree of a node u, written as outdeg(u), is the number of edges that originate at u. In-degree of a node - The in-degree of a node u, written as indeg(u), is the number of edges that terminate at u.

www.binils.com

Degree of a node - The degree of a node, written as deg(u), is equal to the sum of in-degree and out-degree of that node. Therefore, deg(u) = indeg(u) + outdeg(u).

Isolated vertex - A vertex with degree zero. Such a vertex is not an end-point of

any edge. Pendant vertex (also known as leaf vertex) - A vertex with degree one.

Cut vertex - A vertex which when deleted would disconnect the remaining graph.

Source - A node u is known as a source if it has a positive out-degree but a zero

in-degree. Sink - A node u is known as a sink if it has a positive in-degree but a

zero out-degree.

Reachability -A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Strongly connected directed graph

A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is, if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally connected graph

A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path from u to v or a path from v to u, but not both.

Weakly connected digraph

A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. That is, in such a graph, it is possible to reach any node from any other node by traversing edges in any direction (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

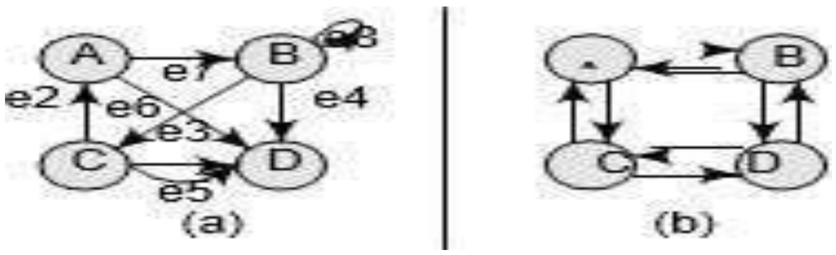
Parallel/Multiple edges

Distinct edges which connect the same end-points are called multiple edges. That is, e = (u, v) and e' = (u, v) are known as multiple edges of G.

www.binils.com

Simple directed graph

A directed graph G is said to be a simple directed graph if and only if it has no parallel edges



a) Directed acyclic graph and b) Strongly connected directed graph

REPRESENTATION OF GRAPH

There are three common ways of storing graphs in the computer’s memory. They are:

- Sequential representation by using an adjacency matrix.
- Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.
- Adjacency multi-list which is an extension of linked representation

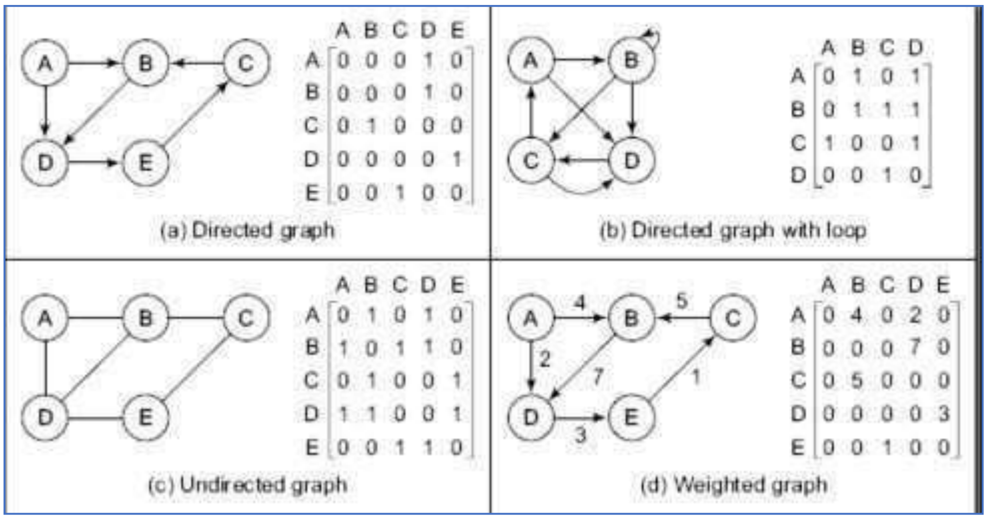
Adjacency matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n ¥ n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero

$$a_{ij} \begin{cases} 1 & \text{[if } v_i \text{ is adjacent to } v_j\text{, that is there is an edge } (v_i, v_j)\text{]A} \\ 0 & \text{[otherwise]} \end{cases}$$

Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. therefore, a change in the order of nodes will result in a different adjacency matrix.

www.binils.com



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 0 & 0 & 1 & 0 \\ B & 0 & 0 & 0 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(a) Directed graph

$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ C & 1 & 0 & 0 & 1 \\ D & 0 & 0 & 1 & 0 \end{array}$$

(b) Directed graph with loop

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

(c) Undirected graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 2 & 0 \\ B & 0 & 0 & 0 & 7 & 0 \\ C & 0 & 5 & 0 & 0 & 0 \\ D & 0 & 0 & 0 & 0 & 3 \\ E & 0 & 0 & 1 & 0 & 0 \end{array}$$

(d) Weighted graph

From the above examples, we can draw the following conclusions:

- For a simple graph (that has no loops), the adjacency matrix has 0s on the diagonal.

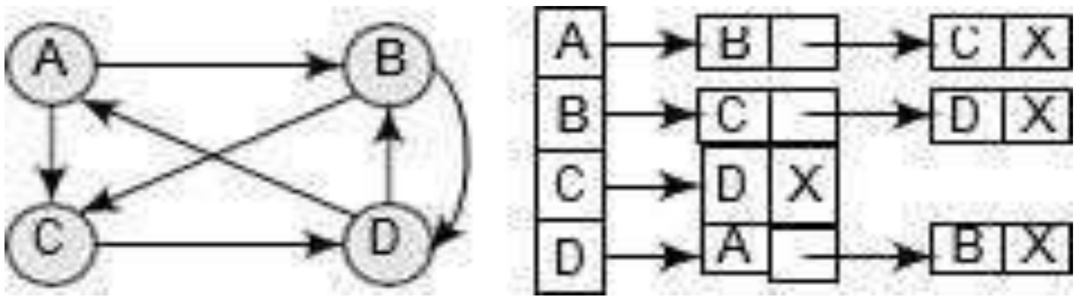
- The adjacency matrix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is O(n2), where n is the number of nodes in the graph.
- Number of 1s (or non-zero entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the weights of the edges connecting the nodes.

ADJACENCY LIST REPRESENTATION

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

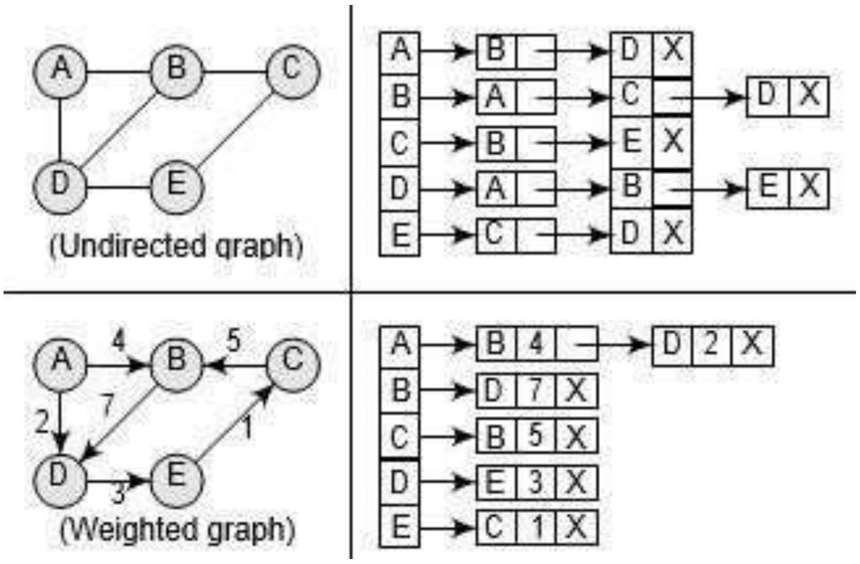
- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

www.binils.com



Graph G and its Adjacency List

Adjacency list for an undirected graph and a weighted graph



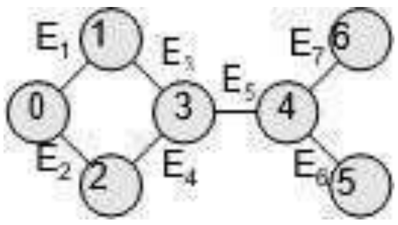
ADJACENCY MULTI-LIST REPRESENTATION

A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node i points to the adjacency list for node i.

www.binils.com

In a multi-list representation, the information about an edge (vi , vj) of an undirected graph can be stored using the following attributes:
M: A single bit field to indicate whether the edge has been examined or not. Vj : A vertex in the graph that is connected to vertex vi by an edge.
Vi : A vertex in the graph that is connected to vertex vi by an edge.

Link i for vj : A link that points to another node that has an edge incident on v . Link j for vi : A link that points to another node that has an edge incident on vi .



Undirected Graph

The adjacency multi-list for the graph can be given as:

Edge 1	0	1	Edge 2	Edge 3
Edge 2	0	2	NULL	Edge 4
Edge 3	1	3	NULL	Edge 4
Edge 4	2	3	NULL	Edge 5
Edge 5	3	4	NULL	Edge 6
Edge 6	4	5	Edge 7	NULL
Edge 7	4	6	NULL	NULL

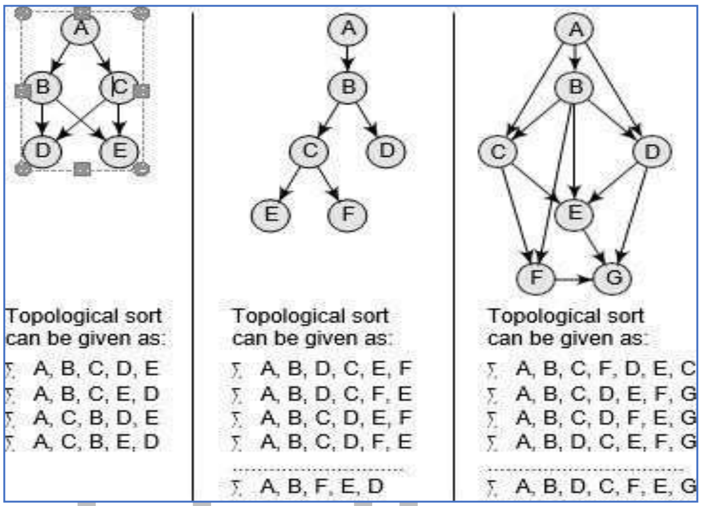
Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

VERTEX	LIST OF EDGES
0	Edge 1, Edge 2
1	Edge 1, Edge 3
2	Edge 2, Edge 4
3	Edge 3, Edge 4, Edge 5
4	Edge 5, Edge 6, Edge 7
5	Edge 6
6	Edge 7

www.binils.com

TOPOLOGICAL SORTING

- Topological sort of a directed acyclic graph (DAG) G is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.
- A topological sort of a DAG G is an ordering of the vertices of G such that if G contains an edge (u, v), then u appears before v in the ordering. Note that topological sort is possible only on directed acyclic graphs that do not have any cycles. For a DAG that contains cycles, no linear ordering of its vertices is possible.
- In simple words, a topological ordering of a DAG G is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.



www.binils.com

Topological Sorting

Algorithm

The two main steps involved in the topological sort algorithm include:

- Selecting a node with zero in-degree
- Deleting N from the graph along with its edges

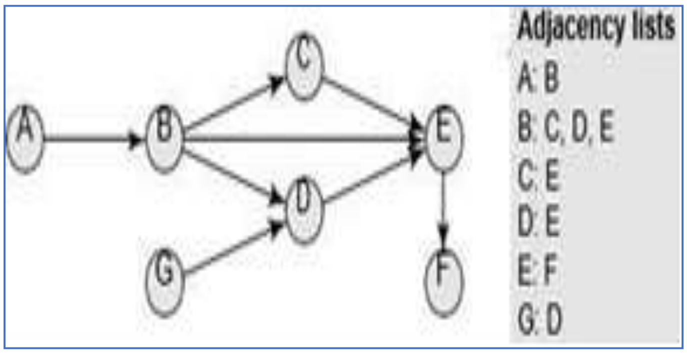
```
Step 1: Find the in-degree INDEG(N) of every node in the
graph Step 2: Enqueue all the nodes with a zero in-
degree
Step 3: Repeat Steps 4 and 5 until the QUEUE is empty

Step 4: Remove the front node N of the QUEUE by setting FRONT =
FRONT + 1 Step 5: Repeat for each neighbour M of node N:
    a) Delete the edge from N to M by setting INDEG(M) = INDEG(M) – 1

    b) IF INDEG(M) = , then Enqueue M, that is, add M to the rear of the
    queue [END OF INNER LOOP]
    [END OF LOOP]

Step 6: Exit
```

We will use a QUEUE to hold the nodes with zero in-degree. The order in which the nodes will be deleted from the graph will depend on the sequence in which the nodes are inserted in the QUEUE. Then, we will use a variable INDEG, where INDEG(N) will represent the in-degree of node N.

www.binils.com

Example

Consider a directed acyclic graph G given below. We use the algorithm given above to find a topological sort T of G. The steps are given as below



Step 1: Find the in-degree INDEG(N) of every node in the

graph INDEG(A) = 0 INDEG(B) = 1 INDEG(C) = 1

INDEG(D) = 2

INDEG(E) = 3 INDEG(F) = 1 INDEG(G) = 0

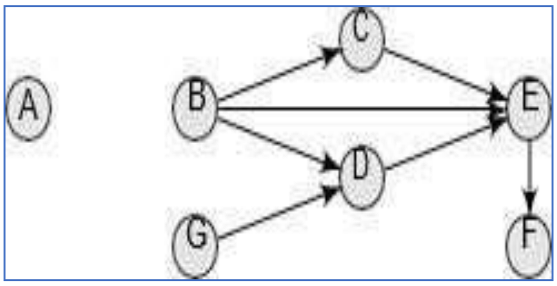
Step 2: Enqueue all the nodes with a zero in-degree FRONT = 1 REAR = 2
QUEUE = A, G

Step 3: Remove the front element A from the queue by setting FRONT = FRONT + 1, so FRONT = 2 REAR = 2 QUEUE = A, G

Step 4: Set INDEG(B) = INDEG(B) – 1, since B is the neighbour of A. Note that INDEG(B) is 0, so add it on the queue. The queue now becomes

FRONT = 2 REAR = 3 QUEUE = A, G, B

Delete the edge from A to B. The graph now becomes as shown in the figure below



www.binils.com

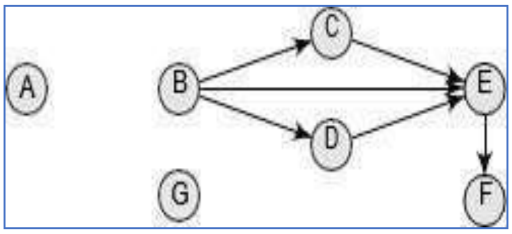
Step 5: Remove the front element B from the queue by setting FRONT =

FRONT + 1, so FRONT = 2 REAR = 3 QUEUE = A, G, B

Step 6: Set INDEG(D) = INDEG(D) – 1, since D is the neighbour of G. Now,

INDEG(C) = 1 INDEG(D) = 1 INDEG(E) = 3 INDEG(F) = 1

Delete the edge from G to D. The graph now becomes as shown in the figure below



Step 7: Remove the front element B from the queue by setting FRONT =

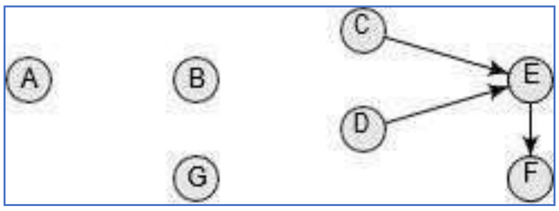
FRONT + 1, so FRONT = 4 REAR = 3 QUEUE = A, G, B

Step 8: Set INDEG(C) = INDEG(C) – 1, INDEG(D) = INDEG(D) – 1, INDEG(E) = INDEG(E) – 1, since C, D, and E are the

neighbours of B. Now, INDEG(C) = 0, INDEG(D) = 1 and INDEG(E) = 2

Step 9: Since the in-degree of node c and D is zero, add C and D at the rear of the queue. The queue can be given as below:

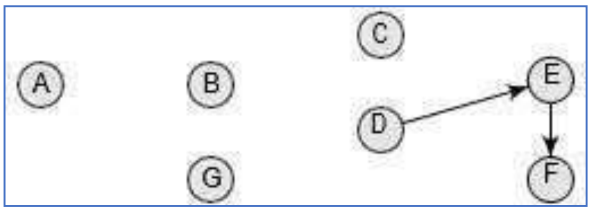
FRONT = 4 REAR = 5 QUEUE = A, G, B, C, D The graph now becomes as shown in the figure below



Step 10: Remove the front element C from the queue by setting FRONT = FRONT + 1, so FRONT = 5 REAR = 5 QUEUE = A, G, B, C, D

www.binils.com

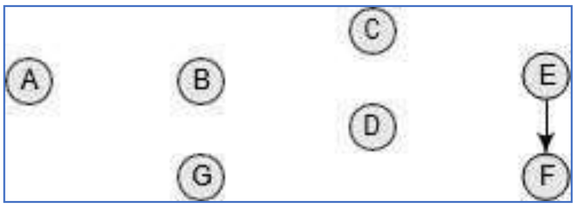
Step 11: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of C. Now, INDEG(E) = 1 The graph now becomes as shown in the figure below



Step 12: Remove the front element D from the queue by setting FRONT = FRONT + 1, so FRONT = 6 REAR = 5 QUEUE = A, B, G, C, D

Step 13: Set INDEG(E) = INDEG(E) – 1, since E is the neighbour of D. Now, INDEG(E) = 0, so add E to the queue. The queue now becomes. FRONT = 6 REAR = 6 QUEUE = A, G, B, C, D, E

Step 14: Delete the edge between D an E. The graph now becomes as shown in the figure below

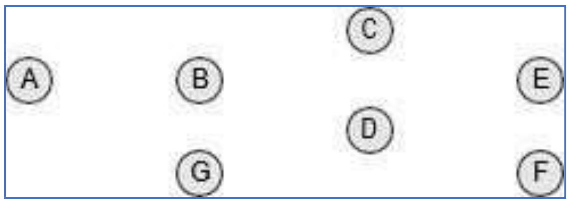


Step 15: Remove the front element D from the queue by setting FRONT = FRONT + 1, so FRONT = 7 REAR = 6 QUEUE = A, G, B, C, D, E

Step 16: Set INDEG(F) = INDEG(F) – 1, since F is the neighbour of E. Now INDEG(F) = 0, so add F to the queue. The queue now becomes,

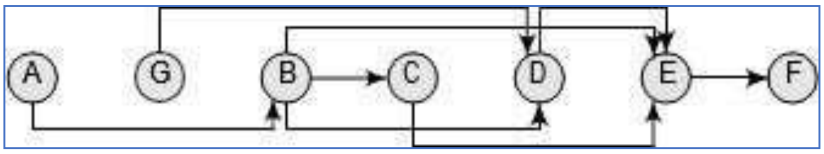
FRONT = 7 REAR = 7 QUEUE = A, G, B, C, D, E, F

Step 17: Delete the edge between E and F. The graph now becomes as shown in the figure below



www.binils.com

There are no more edges in the graph and all the nodes have been added to the queue, so the topological sort T of G can be given as: A, G, B, C, D, E, F. When we arrange these nodes in a sequence, we find that if there is an edge from u to v, then u appears before v.



Topological Sort of G