RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Discrete Structure**

Subject Code: **IT-302**

Semester: **3rd**

Subject Notes
Discrete Structures

UNIT-4

Graph

Definition − A graph (denoted as $G= (V, E)$) consists of a non-empty set of vertices or nodes V and a set of edges E.

Example − Let us consider, a Graph is $G=(V,E)$where $V=\{a, b, c, d\}$ and $E=\{\{a,b\},\{a,c\},\{b,c\},\{c,d\}\}$

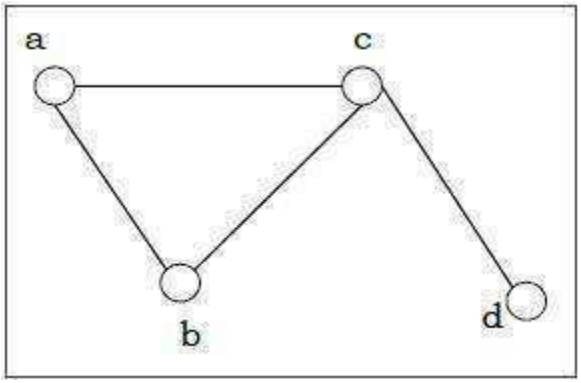


Figure 4.1 example of graph

Degree of a Vertex − the degree of a vertex V of a graph G (denoted by deg (V)) is the number of edges incident with the vertex V.

Vertex	Degree	Even / Odd
a	2	even
b	2	even
c	3	odd
d	1	odd

Even and Odd Vertex − If the degree of a vertex is even, the vertex is called an even vertex and, if the degree of a vertex is odd, the vertex is called an odd vertex

Degree of a Vertex − the degree of a graph is the largest vertex degree of that graph. For the above graph the degree of the graph is 3.

The Handshaking Lemma − in a graph, the sum of all the degrees of all the vertices is equal to twice the number of edges.

Types of Graphs

There are different types of graph

1. **Null Graph -** A null graph has no edges. The null graph of *n* vertices is denoted by *Nn*

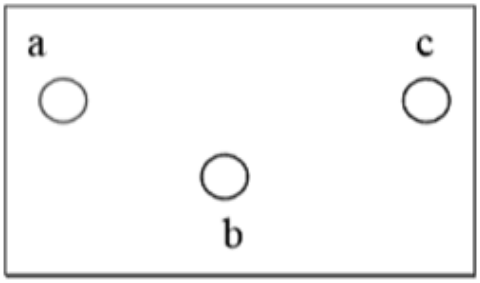


Figure 4.2 Null Graph

2. **Simple Graph** -A graph is called simple graph/strict graph if the graph is undirected and does not contain any loops or multiple edges.

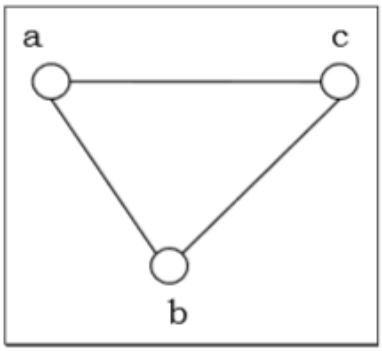


Figure 4.3 Simple Graph

3. **Directed and Undirected Graph -** A graph G=(V,E) is called a directed graph if the edge set is made of ordered vertex pair and a graph is called undirected if the edge set is made of unordered vertex pair.

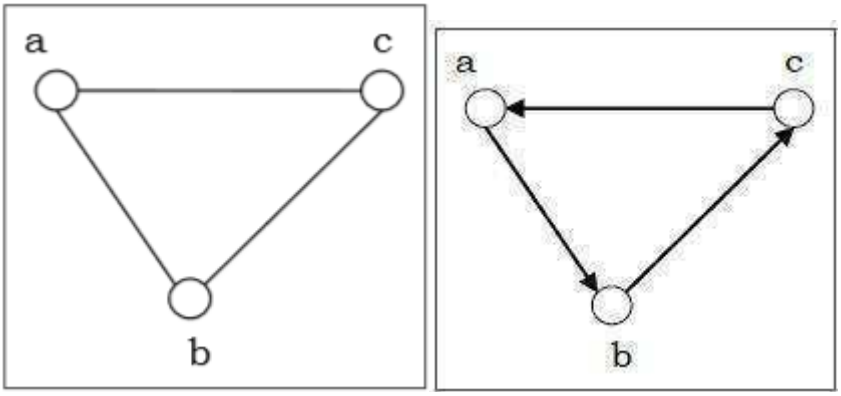


Figure 4.4 Directed and Undirected Graphs

4. **Connected and Disconnected Graph -** A graph is connected if any two vertices of the graph are connected by a path; while a graph is disconnected if at least two vertices of the graph are not connected by a path. If a graph G is disconnected, then every maximal connected subgraph of G is called a connected component of the graph G

.

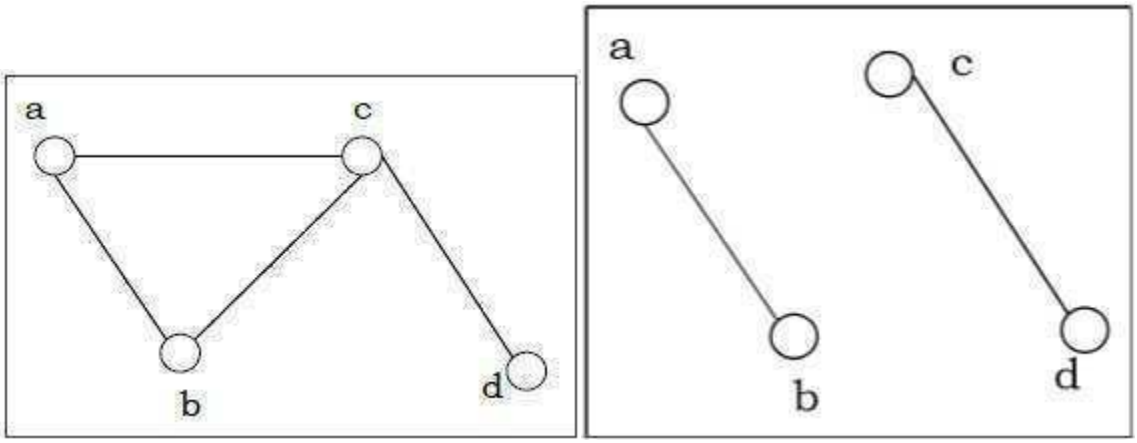


Figure 4.5 Connected and Disconnected Graph

5. **Regular Graph -** A graph is regular if all the vertices of the graph have the same degree. In a regular graph G of degree r, the degree of each vertex of G is r.

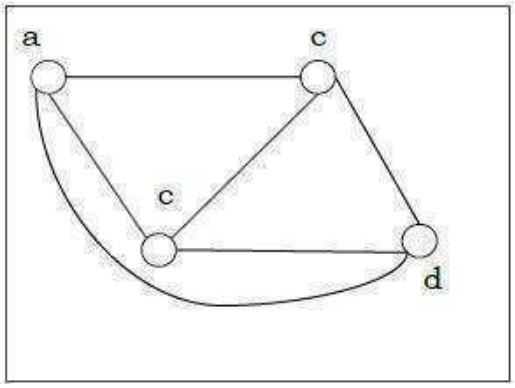


Figure 4.6 Regular Graph

6. **Complete Graph-** A graph is called complete graph if every two vertices pair are joined by exactly one edge. The complete graph with n vertices is denoted by Kn

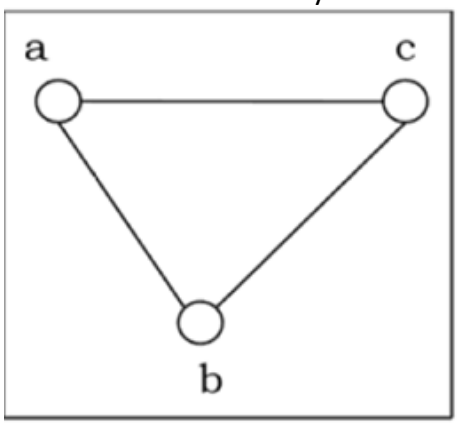


Figure 4.7 Complete Graph

7. **Cycle Graph-** If a graph consists of a single cycle, it is called cycle graph. The cycle graph with n vertices is denoted by Cn

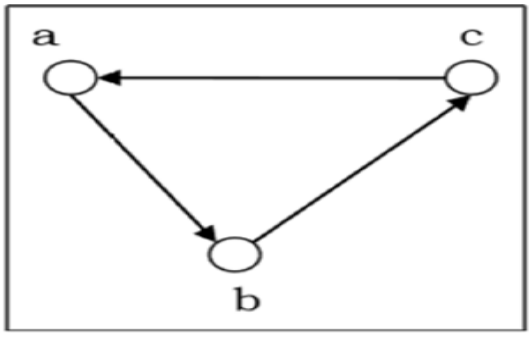


Figure 4.8 Cycle Graph

8. **Bipartite Graph -** If the vertex-set of a graph G can be split into two disjoint sets, V1 and V2, in such a way that each edge in the graph joins a vertex in V1 to a vertex in V2, and there are no edges in G that connect two vertices in V1 or two vertices in V2, then the graph G is called a bipartite graph.

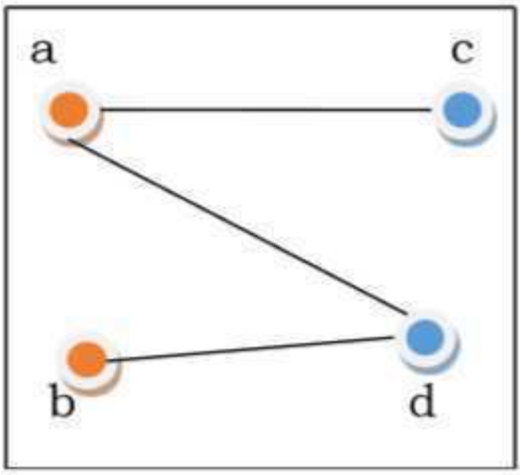


Figure 4.9 Bipartite Graph

9. **Complete Bipartite Graph-** A complete bipartite graph is a bipartite graph in which each vertex in the first set is joined to every single vertex in the second set. The complete bipartite graph is denoted by Kx,ywhere the graph G contains x vertices in the first set and y vertices in the second set.

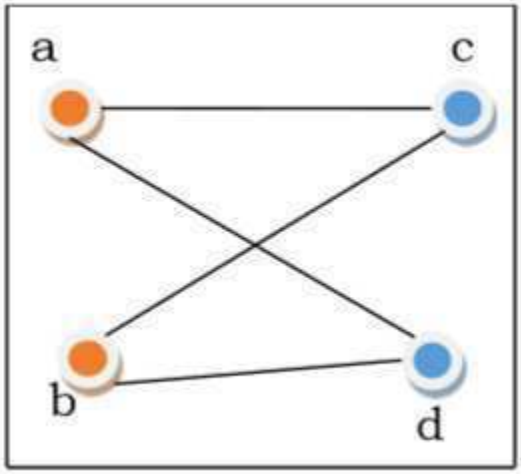


Figure 4.10 Complete Bipartite Graphs

10. **Planar vs. Non-planar graph**

Planar graph – A graph G is called a **planar** graph if it can be drawn in a plane without any edges crossed. If we draw graph in the plane without edge crossing, it is called embedding the graph in the plane.

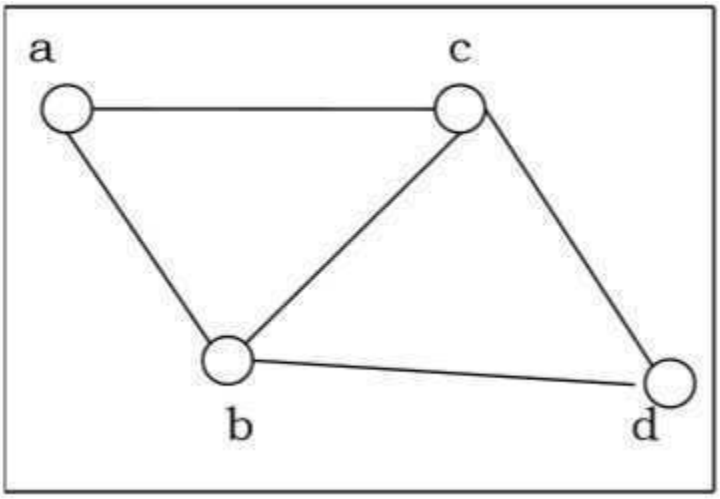


Figure 4.11 Planar graph

Non-planar graph – A graph is non-planar if it cannot be drawn in a plane without graph edges crossing.

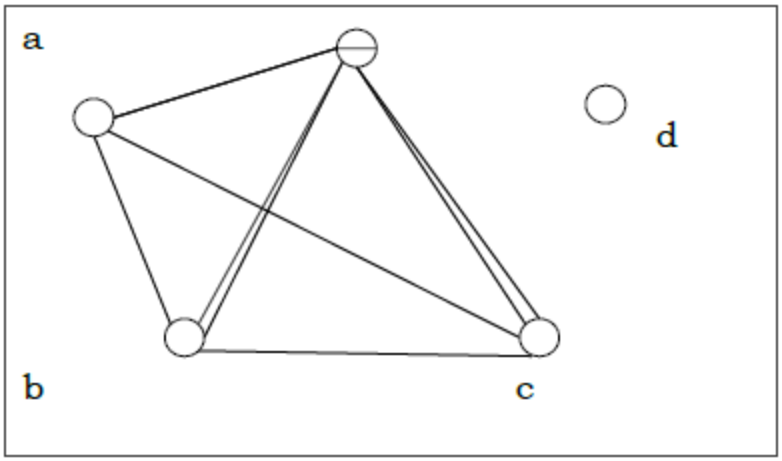


Figure 4.12 Non-planar graph

11. **Multi-Graph**- If in a graph multiple edges between the same set of vertices are allowed, it is called Multigraph. In other words, it is a graph having at least one loop or multiple edges.

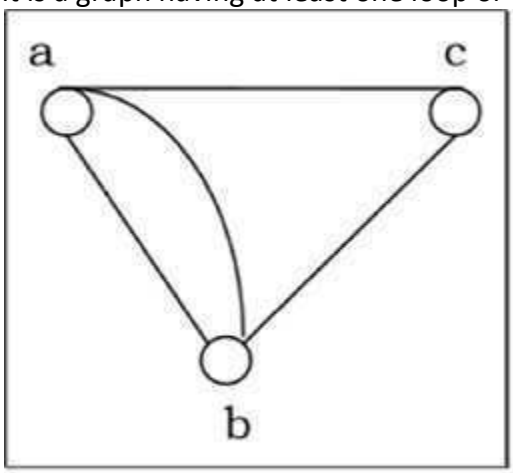


Figure 4.13 Multi-Graph

12. **Weighted Graph:** weighted graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).

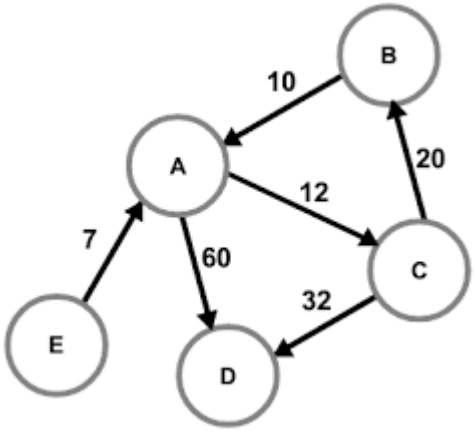


Figure 4.14 Weighted Graphs

13. **Isomorphic graph** - If two graphs G and H contain the same number of vertices connected in the same way, they are called isomorphic graphs (denoted by G≅H).
It is easier to check non-isomorphism than isomorphism. If any of these following conditions occurs, then two graphs are non-isomorphic –
    1) The number of connected components are different

2) Vertex-set cardinalities are different
3) Edge-set cardinalities are different
4) Degree sequences are different

Example

The following graphs are isomorphic –

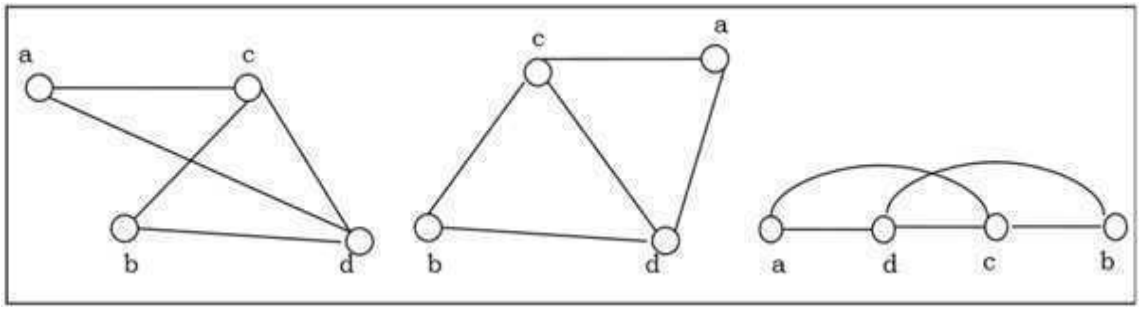


Figure 4.15 Isomorphic graph

Representation of Graphs

There are mainly two ways to represent a graph –

1. Adjacency Matrix
2. Adjacency List

1. **Adjacency Matrix -** AnAdjacency Matrix *A*[*V*][*V*] is a 2D array of size *V*×*V* where *V* is the number of vertices in a undirected graph. If there is an edge between *Vx* to *Vy* then the value of *A*[*Vx*][*Vy*]=1 and *A*[*Vy*][*Vx*]=1, otherwise the value will be zero. And for a directed graph, if there is an edge between *Vx* to *Vy*, then the value of *A*[*Vx*][*Vy*]=1 , otherwise the value will be zero.

Adjacency Matrix of an Undirected Graph

Let us consider the following undirected graph and construct the adjacency matrix –

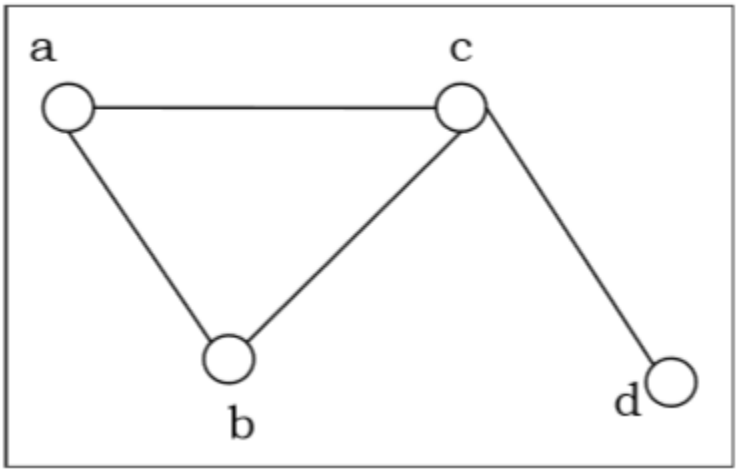


Figure 4.16 Adjacency Matrix of an Undirected Graph

Adjacency matrix of the above undirected graph will be –

	a	**b**	**c**	**d**
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency Matrix of a Directed Graph

Let us consider the following directed graph and construct its adjacency matrix –

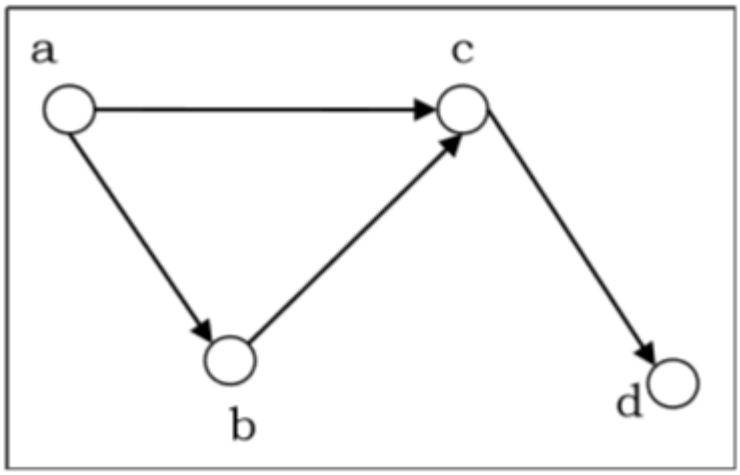


Figure 4.17 Adjacency Matrix of a Directed Graph

Adjacency matrix of the above directed graph will be –

	a	**b**	**c**	**d**
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

2. **Adjacency List - In** adjacency list, an array (*A*[*V*]) of linked lists is used to represent the graph G with *V* number of vertices. An entry *A*[*Vx*] represents the linked list of vertices adjacent to the *Vx*–*th*vertex. The adjacency list of the undirected graph is as shown in the figure below –

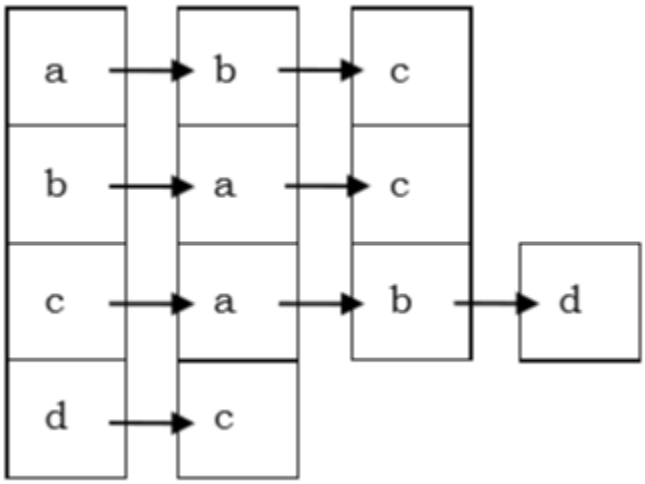


Figure 4.18 Adjacency List

Cycles and connectivity

Definitions:

- **Walk**: finite sequence of edges in which any two consecutive edges are adjacent or identical. (Initial vertex, Final vertex, length)
- **Trail**: walk with all distinct edges
- **Path**: A path is a sequence of vertices with the property that each vertex in the sequence is adjacent to the vertex next to it. A path that does not repeat vertices is called a simple path.
- **Circuit**: A circuit is path that begins and ends at the same vertex.
- **Cycle**: A circuit that doesn't repeat vertices is called a cycle.

Shortest path in weighted graph: Given a graph where edges are labeled with weights (or distances) and a source vertex, what is the shortest path between the source and some other vertex? Problems requiring us to answer such queries are broadly known as shortest-paths problems. Shortest-paths problem come in several flavors. For example, the single-source shortest path problem requires finding the shortest paths between a given source and all other vertices; the single-pair shortest path problem requires finding the shortest path between given a source and a given destination vertex; the all-pairs shortest path problem requires finding the shortest paths between all pairs of vertices.

Shortest Paths: Problem Statement

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v
- Length (or distance) of a path is the sum of the weights of its edges Length (or distance) of a path is the sum of the weights of its edges

Applications

- Internet packet routing
- Flight reservations
- Driving directions

Assumptions

1) Graph is simple
    a. No parallel edges and no self-loops
2) Graph is connected
    a. If not, run the algorithm for each connected component
3) Graph is undirected Graph is undirected
    a. It is simple to extend to directed case
4) No negative weight edges
    a. There is an algorithm to compute shortest paths in a graph with negative edges
    b. It has higher time complexity
    c. Does not work if there is a negative cost cycle Does not work if there is a negative cost cycle
    d. Makes no sense to compute shortest paths in the presence of negative cycles
        i. in a graph with a negative cycle, shortest path has cost negative infinity

Dijkstra's algorithm

Steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While sptSet doesn't include all vertices
    a) Pick a vertex u which is not there in sptSetand has minimum distance value.
    b) Include u to sptSet.
    c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:

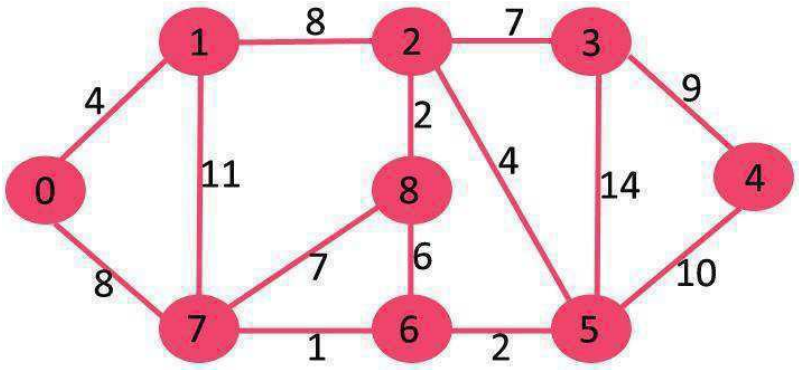


Figure 4.19 example of shortest Path

- The set sptSetis initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.
- Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

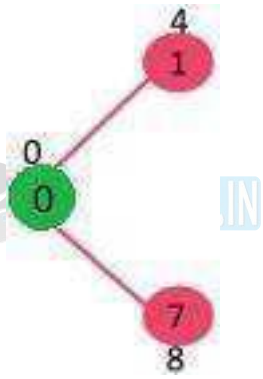


Figure 4.19 (a)

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
- The distance value of vertex 2 becomes 12.

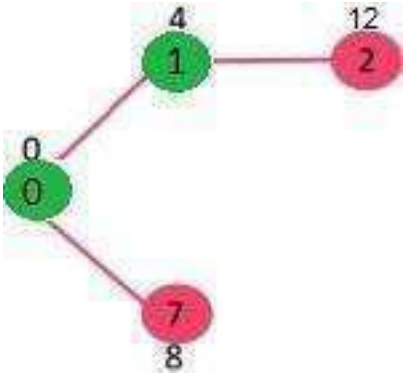


Figure 4.19 (b)

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

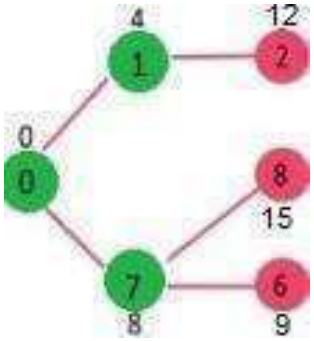


Figure 4.19 (c)

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

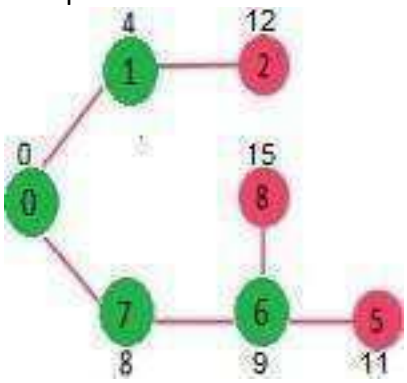


Figure 4.19 (d)

- We repeat the above steps until sptSet doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

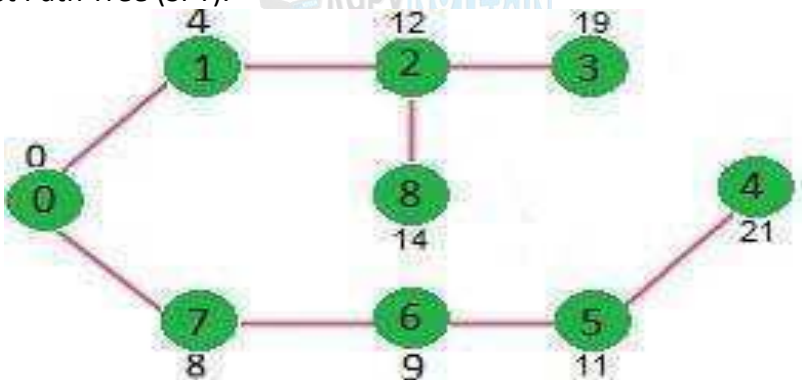


Figure 4.19 (e)

Introduction to Eulerian paths and Circuits

Euler Graphs - A connected graph *G* is called an Euler graph, if there is a closed trail which includes every edge of the graph *G* . An Euler path is a path that uses every edge of a graph exactly once. An Euler path starts and ends at different vertices. An Euler circuit is a circuit that uses every edge of a graph exactly once. An Euler circuit always starts and ends at the same vertex. A connected graph *G* is an Euler graph if and only if all vertices of *G* are of even degree, and a connected graph *G* is Eulerian if and only if its edge set can be decomposed into cycles.

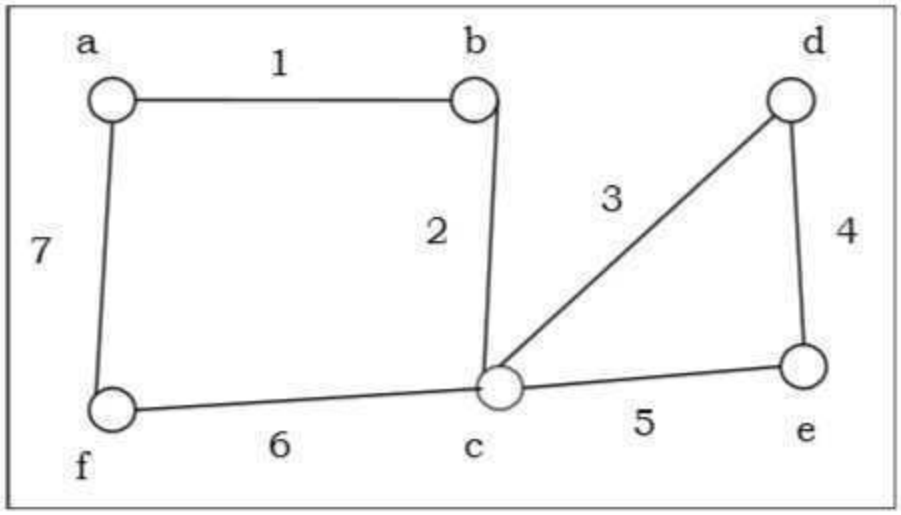


Figure 4.20 Euler Graph

The above graph is an Euler graph as "*a*1*b*2*c*3*d*4*e*5*c*6*f*7*g*"covers all the edges of the graph.

Eulerian path and circuit for undirected graph

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

How to find whether a given graph is Eulerian or not?

- The problem is same as following question. "Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once".
- A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path.
- The problem is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.
- Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

Eulerian Cycle an undirected graph has Eulerian cycle if following two conditions are true.

1) All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges).
2) All vertices have even degree.

Eulerian Path An undirected graph has Eulerian Path if following two conditions are true.

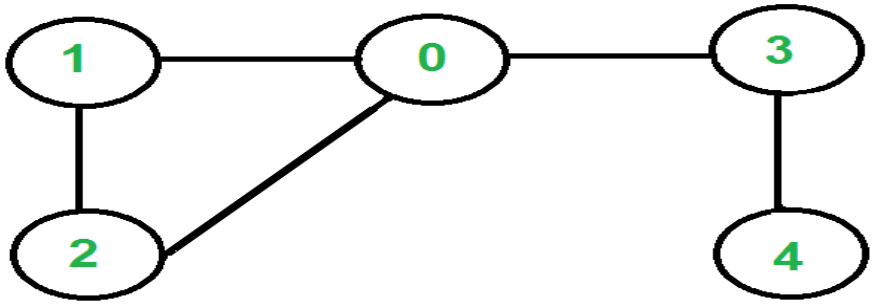
1) Same as condition
    a. For Eulerian Cycle
2) If zero or two vertices have odd degree and all other vertices have even degree.

Note: that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note: that a graph with no edges is considered Eulerian because there are no edges to traverse.

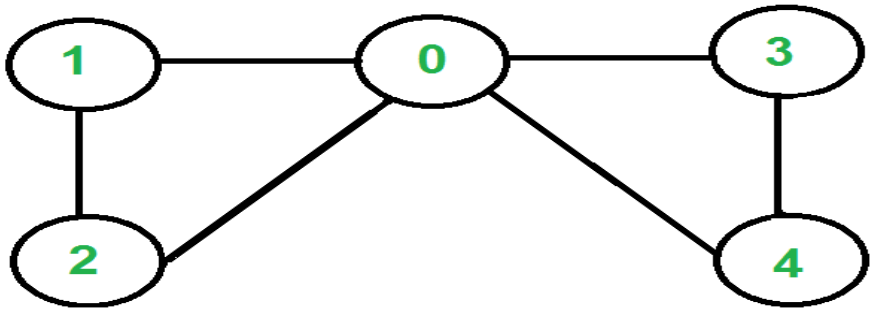
How does this work?

In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex; therefore all vertices must have even degree.



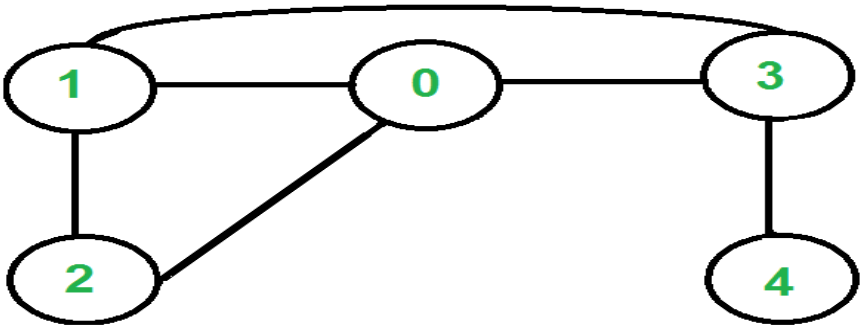
The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)

Figure 4.21 (a) example of Eulerian path



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2" Note that all vertices have even degree

Figure 4.21 (b) example of Eulerian cycle



The graph is not Eulerian. Note that there are four vertices with odd degree (0, 1, 3 and 4)

Figure 4.21 (c) examples with no Eulerian path

Hamiltonian Graphs - A connected graph *G* is called Hamiltonian graph if there is a cycle which includes every vertex of *G* and the cycle is called Hamiltonian cycle. Hamiltonian walk in graph *G* is a walk that passes through each vertex exactly once. If *G* is a simple graph with n vertices, where $n \geq 3$ If $deg(v) \geq n2$ for each vertex *v*, then the graph *G* is Hamiltonian graph. This is called Dirac's Theorem. If *G* is a simple graph with *n* vertices, where $n \geq 2$ if $deg(x)+deg(y) \geq n$ for each pair of non-adjacent vertices x and y, then the graph *G* is Hamiltonian graph. This is called Ore's theorem.

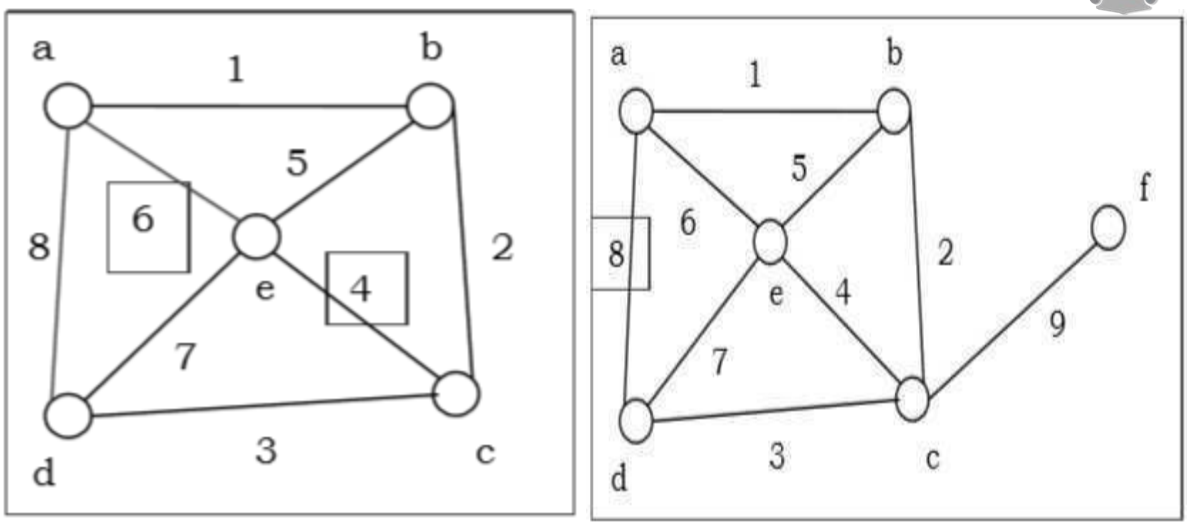


Figure 4.22 Hamiltonian Graphs

Hamiltonian Circuits A Hamiltonian circuit is a circuit that visits each of the vertices once and only once and ends on the same vertex as it began. For example, in this network the Hamiltonian circuit is marked in red. As it is a circuit, we cannot have repeated edges. We do not need to use all the edges, just visit each vertex once.

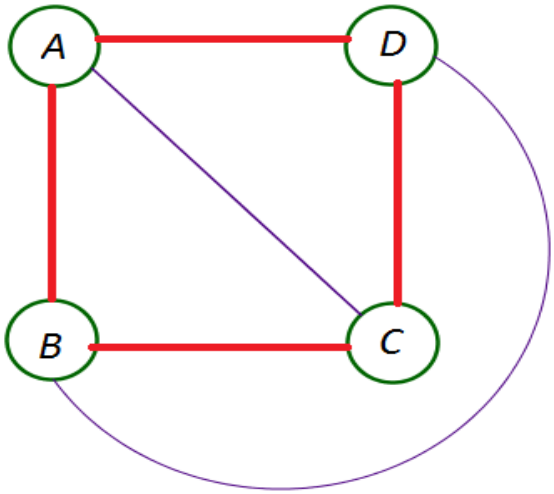


Figure 4.23 Hamiltonian Circuits

Other Hamiltonian circuits here include ABDCA, ABDCA, DBACD, DBACD and many others. If a graph has a Hamiltonian circuit then it automatically has a Hamiltonian path. (By just dropping off the last vertex in the circuit we create a path, for example, ABDC and DBAC from above)

Hamiltonian path: A Hamiltonian path is a path that visits each of the vertices once and only once but can begin and end on different vertices. For example, the Hamiltonian path in the network here could be ABCDE, ABCDE.

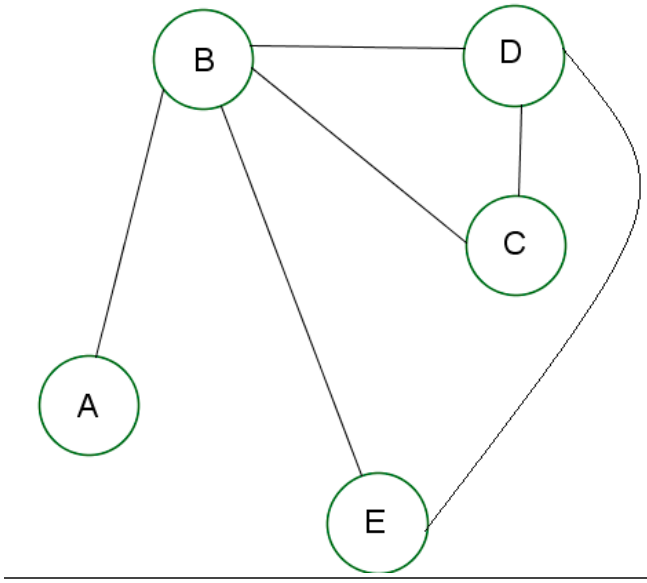


Figure 4.23 example of Hamiltonian path

Unlike with the Euler paths and Euler circuits, there is no single rule or theorem to help us identify if a Hamiltonian path or Hamiltonian circuit exists. The only thing we can do is look for them.
The following network has both Hamiltonian path and Hamiltonian circuit - can you find them both?

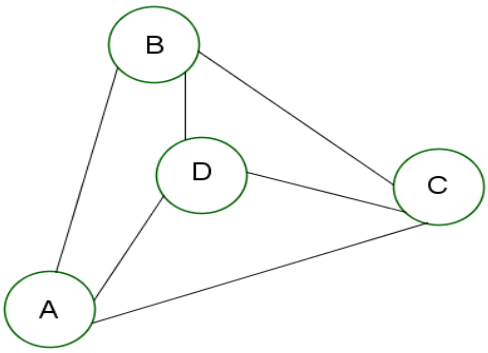


Figure 4.23 examples of both Hamiltonian path & Hamiltonian Circuits

Remember, the Hamiltonian path is a path where we visit each of the vertices only once. There are many Hamiltonian Paths here,

Graph Coloring- Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

Chromatic number: The chromatic number of a graph G is the smallest number of colors needed to color the vertices of G so that no two adjacent vertices share the same color i.e., the smallest value of k possible to obtain a k-coloring.

Method to Color a Graph

The steps required to color a graph G with n number of vertices are as follows –
Step 1 – Arrange the vertices of the graph in some order.
Step 2 – Choose the first vertex and color it with the first color.

Step 3 – Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Example

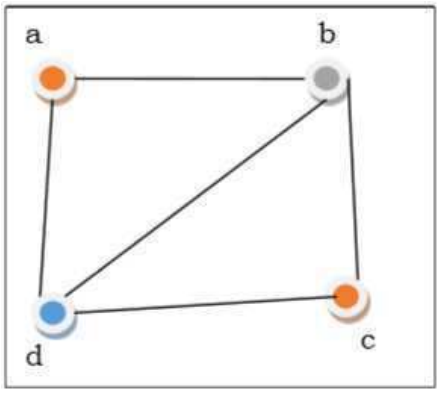


Figure 4.24 Graph coloring

In the above figure, at first vertex *a* is colored red. As the adjacent vertices of vertex a are again adjacent, vertex *b* and vertex *d* are colored with different color, green and blue respectively. Then vertex *c* is colored as red as no adjacent vertex of *c* is colored red. Hence, we could color the graph by 3 colors. Hence, the chromatic number of the graph is 3.

Applications of Graph Coloring- Some applications of graph coloring include

1. Register Allocation
2. Map Coloring
3. Bipartite Graph Checking
4. Mobile Radio Frequency Assignment
5. Making time table, etc.

Isomorphism of Graphs - If two graphs G and H contain the same number of vertices connected in the same way, they are called isomorphic graphs (denoted by $G \cong H$).

It is easier to check non-isomorphism than isomorphism. If any of these following conditions occurs, then two graphs are non-isomorphic –

1. The number of connected components are different
2. Vertex-set cardinalities are different
3. Edge-set cardinalities are different
4. Degree sequences are different

Example

The following graphs are isomorphic –

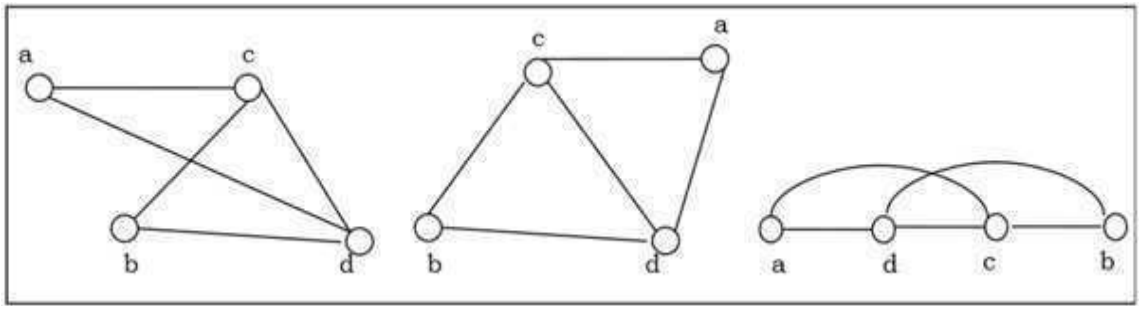


Figure 4.25 Isomorphism of Graph

Homomorphism - A homomorphism from a graph *G* to a graph *H* is a mapping (May not be a bijective mapping)$h:G\rightarrow H$ such that – $(x,y)\in E(G)\rightarrow(h(x),h(y))\in E(H)$. It maps adjacent vertices of graph *G* to the adjacent vertices of the graph *H.*

Properties of Homomorphisms

1. A homomorphism is an isomorphism if it is a bijective mapping.
2. Homomorphism always preserves edges and connectedness of a graph.
3. The compositions of homomorphisms are also homomorphisms.
4. To find out if there exists any homomorphic graph of another graph is aNPcomplete problem.

Graph Traversal

Graph traversal is the problem of visiting all the vertices of a graph in some systematic order. There are mainly two ways to traverse a graph.

1. **Breadth First Search**
2. **Depth First Search**

1. **Breadth First Search** - Breadth First Search (BFS) starts at starting level-0 vertex X of the graph G. Then we visit all the vertices that are the neighbors of X . After visiting, we mark the vertices as "visited," and place them into level-1. Then we start from the level-1 vertices and apply the same method on every level-1 vertex and so on. The BFS traversal terminates when every vertex of the graph has been visited.

BFS Algorithm

The concept is to visit all the neighbor vertices before visiting other neighbor vertices of neighbor vertices.

1. Initialize status of all nodes as “Ready”.
2. Put source vertex in a queue and change its status to “Waiting”.
3. Repeat the following two steps until queue is empty –
4. Remove the first vertex from the queue and mark it as “Visited”.
5. Add to the rear of queue all neighbors of the removed vertex whose status is “Ready”. Mark their status as “Waiting”.

Problem

Let us take a graph (Source vertex is ‘a’) and apply the BFS algorithm to find out the traversal order.

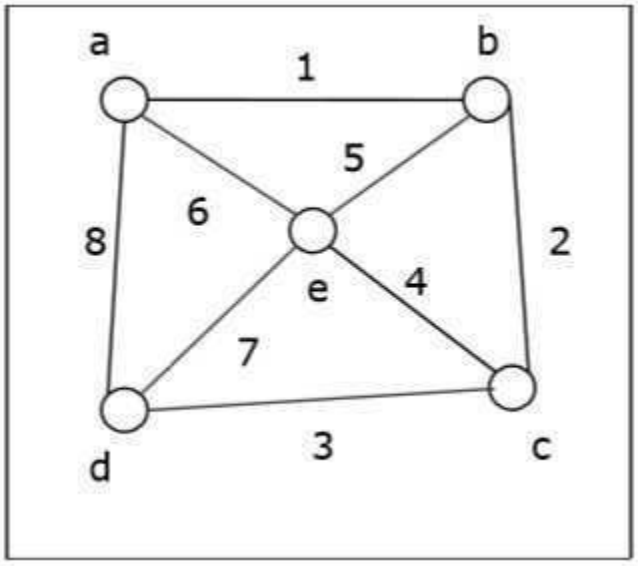


Figure 4.26 examples of BFS

Solution –

- Initialize status of all vertices to “Ready”.
- Put a in queue and change its status to “Waiting”.
- Remove a from queue, mark it as “Visited”.
- Add a’s neighbors in “Ready” state b, d and e to end of queue and mark them as “Waiting”.

- Remove b from queue, mark it as “Visited”, put its “Ready” neighbor c at end of queue and mark c as “Waiting”.
- Remove d from queue and mark it as “Visited”. It has no neighbor in “Ready” state.
- Remove e from queue and mark it as “Visited”. It has no neighbor in “Ready” state.
- Remove c from queue and mark it as “Visited”. It has no neighbor in “Ready” state.
- Queue is empty so stop.

So the traversal order is −

$a \rightarrow b \rightarrow d \rightarrow e \rightarrow c$

The alternate orders of traversal are −

$a \rightarrow b \rightarrow e \rightarrow d \rightarrow c$

Or, $a \rightarrow d \rightarrow b \rightarrow e \rightarrow c$

Or, $a \rightarrow e \rightarrow b \rightarrow d \rightarrow c$

Or, $a \rightarrow b \rightarrow e \rightarrow d \rightarrow c$

Or, $a \rightarrow d \rightarrow e \rightarrow b \rightarrow c$

Application of BFS

1. Finding the shortest path
2. Minimum spanning tree for un-weighted graph
3. GPS navigation system
4. Detecting cycles in an undirected graph
6. Finding all nodes within one connected component

Complexity Analysis - Let $G(V, E)$ be a graph with $|V|$ number of vertices and $|E|$ number of edges. If breadth first search algorithm visits every vertex in the graph and checks every edge, then its time complexity would be −

$$O(|V|+|E|).O(|E|)$$

It may vary between $O(1)$ and $O(|V2|)$

1. **Depth First Search -** Depth First Search (DFS) algorithm starts from a vertex *v* , then it traverses to its adjacent vertex (say x) that has not been visited before and mark as "visited" and goes on with the adjacent vertex of *x* and so on. If at any vertex, it encounters that all the adjacent vertices are visited, then it backtracks until it finds the first vertex having an adjacent vertex that has not been traversed before. Then, it traverses that vertex, continues with its adjacent vertices until it traverses all visited vertices and has to backtrack again. In this way, it will traverse all the vertices reachable from the initial vertex *v.*

DFS Algorithm

The concept is to visit all the neighbor vertices of a neighbor vertex before visiting the other neighbor vertices.

- Initialize status of all nodes as “Ready”
- Put source vertex in a stack and change its status to “Waiting”
- Repeat the following two steps until stack is empty –
- Pop the top vertex from the stack and mark it as “Visited”
- Push onto the top of the stack all neighbors of the removed vertex whose status is “Ready”. Mark their status as “Waiting”.

Problem

Let us take a graph (Source vertex is ‘a’) and apply the DFS algorithm to find out the traversal order.

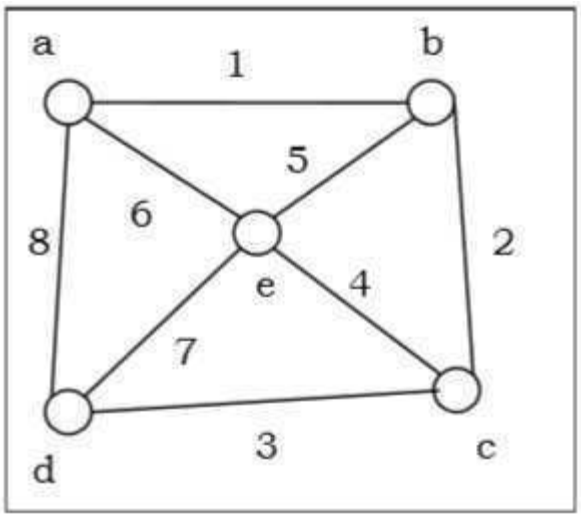


Figure 4.27 examples of DFS

Solution

- Initialize status of all vertices to “Ready”.
- Push a in stack and change its status to “Waiting”.
- Pop a and mark it as “Visited”.
- Push a’s neighbors in “Ready” state e, d and b to top of stack and mark them as “Waiting”.
- Pop b from stack, mark it as “Visited”, push its “Ready” neighbor c onto stack.
- Pop c from stack and mark it as “Visited”. It has no “Ready” neighbor.
- Pop d from stack and mark it as “Visited”. It has no “Ready” neighbor.
- Pop e from stack and mark it as “Visited”. It has no “Ready” neighbor.
- Stack is empty. So stop.

So the traversal order is −

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

The alternate orders of traversal are −

$a \rightarrow e \rightarrow b \rightarrow c \rightarrow d$

Or, $a \rightarrow b \rightarrow e \rightarrow c \rightarrow d$

Or, $a \rightarrow d \rightarrow e \rightarrow b \rightarrow c$

Or, $a \rightarrow d \rightarrow c \rightarrow e \rightarrow b$

Or, $a \rightarrow d \rightarrow c \rightarrow b \rightarrow e$

Complexity Analysis - Let $G(V,E)$ be a graph with $|V|$ number of vertices and $|E|$ number of edges. If DFS algorithm visits every vertex in the graph and checks every edge, then the time complexity is −

$$\ominus (|V|+|E|)$$

Applications

1. Detecting cycle in a graph
2. To find topological sorting
3. To test if a graph is bipartite
4. Finding connected components
5. Finding the bridges of a graph
6. Finding biconnectivity in graphs
7. Solving the Knight’s Tour problem
8. Solving puzzles with only one solution

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in