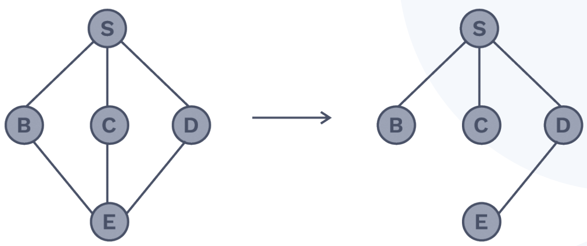
4 Graph-Based Algorithms

Breadth-First Search

- BFS is a simple algorithm used to traverse a graph, and it is a model used in many important graph algorithms.
- BFS can compute the distance(smallest number of edges) from s to other reachable vertices. It is the idea used behind Prim's minimum spanning tree and Dijkstra's single-source shortest path algorithms.
- BFS for a Graph G(V, E) explores the edges of G to "discover" each edge that is reachable from S, where V= set of vertices, E=Set of edges, S belongs to V, and S is the source vertex, and is distinguishable from other vertices.



- BFS uses white, grey, and black colours for the vertices to keep track the progress.
  White – not discovered and not explored
  Grey – discovered but not explored
  Black – discovered and explored
  Discovered – traversed the vertex
  Explored – the vertices that are reachable from a vertex V are discovered.
- The vertices that are adjacent to a black vertex will either be in black or grey.
- Grey vertices may have adjacent white vertices, and at a point, it acts as a border between white and black-coloured vertices
- BFS starts with source vertex S and then includes edges that connect S to its adjacent white vertices.
- For an edge (u,v) that is added to the BFS, u is the predecessor or parent to v in the Breadth-first tree.

Approach:

- The algorithm assumes that the graph G(V,E) is represented in an adjacency list, and the color of each vertex is stored in the u colour and the predecessors of u in u.π.(If there are no predecessors present for a vertex, S for example, then S.π =NIL).
- A queue Q is used to store the grey vertices.
- For a vertex u ∈ V, u.d stores the distance from the source to the vertex u.

```
BFS (G, s)
//initializing all vertices to white color,
infinite distance and connected to null
 1. For every vertex u ∈ G.V – {s}
 2. u.color = WHITE
 3. u.d = ∞
 4. u.π = NULL
 5. s.color = GREY
 6. s.d = 0
 7. s.π = NULL
 8. Q = ∅
 9. ENQUEUE(Q,s):
10. While Q is empty
11. u = DEQUEUE(Q)
12. For each v ∈ G.Adj[u]
13. If (v.color == WHITE)
14. v.color = GREY
15. v.d = u.d + 1
16. v.π = u
17. ENQUEUE(Q, v)
18. u.color = BLACK
```

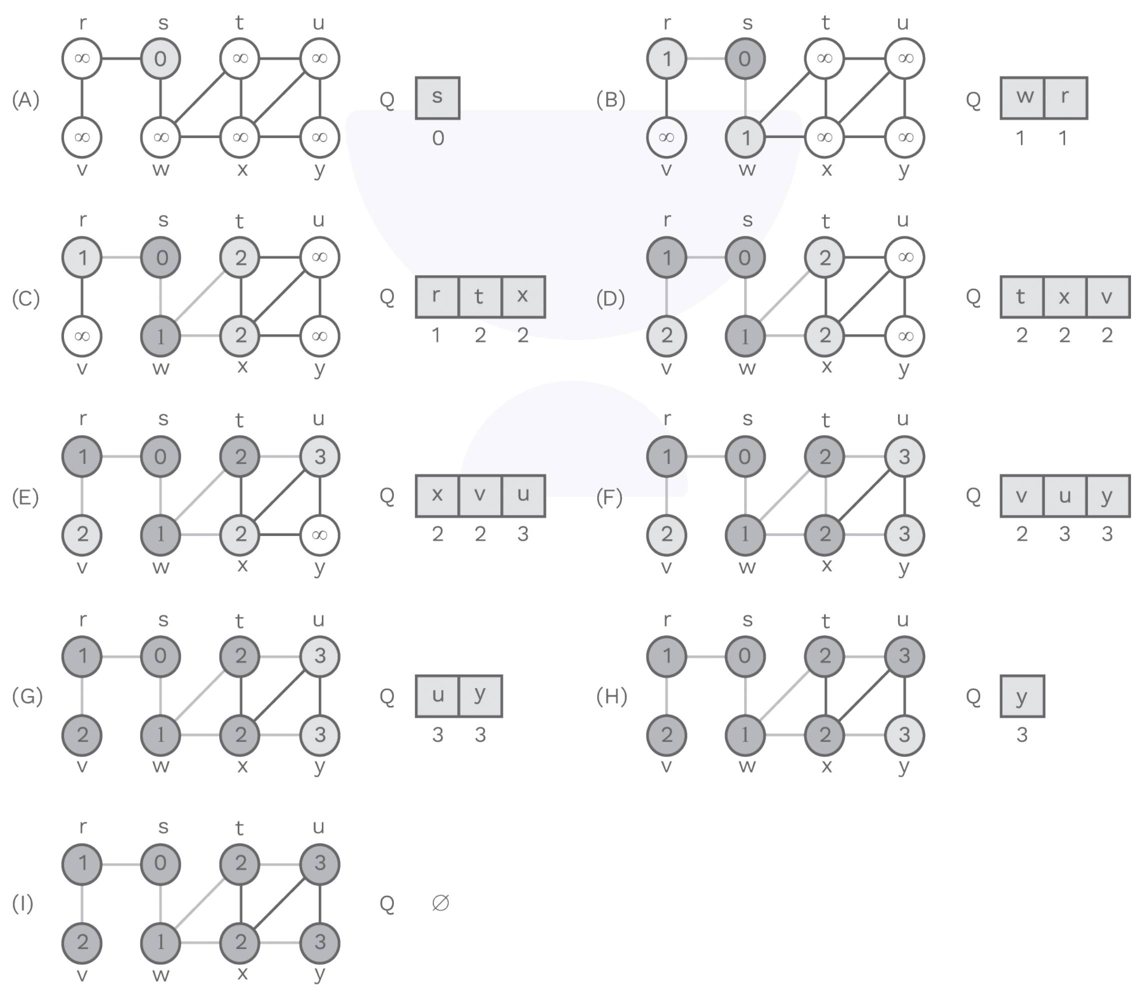
- Line 1-4 except the source vertex all the other vertex's colours are initialised to white, u.d vector to infinity, and u.π to null.
- Line 5 colours the source vertex to grey, line 6 initialises s.d to zero, and in line 7 s.π is initialised to null.
- In line 8, the queue Q is initially empty.
- Line 9 adds source vertex to the queue.
- Line 11 takes the first element in the queue into u, and for every vertex v adjacent to u (by means of for loop) it marks it into grey (line 14) if it is white (line 13) and intialises all the vertices v.d vector to u.d+1 (line 15), and predecessor of v, i.e., v.π as u(line 16).

- Line 17 adds each vertex discovered into queue and line 18 marks the explored vertex into black, i.e., u.colour = black.
- This procedure (from line 10 to line 18) is done until the queue gets empty.
- The order of vertices resulting from BFS depends on the order of adjacent vertices visited in line 12: the BFS may be different, but the distances computed by the algorithm do not.

Analysis

- All the vertices in the graph are white at first except the source S, and the vertices are not changed into white further in the algorithm, and vertices are enqueued and dequeued only once, which is tested in line 13.
- The enqueuing process and dequeuing processes of a vertex take O(1) time which in turn takes O(V) time.
- The algorithm scans the adjacency list of each vertex which takes O(E) time.
- Since initialisation takes O(V) time, the total time taken is O(V+E).
- Hence, BFS runs in time equal to the size of the adjacency-list representation of G, i.e., O(V+E).

Example:



Traversal vs. Search

- Traversal of a graph or tree involves examining every node in the graph or tree.
- Search may or may not involve all the nodes in visiting the graph in systematic manner.
- Traversal is a search, which involves all the nodes in the graph.
- By using Breadth-First Search (BFS), we can even find out whether a graph is connected or not.

- We can initialise the visited array of all the vertex to be zero (i.e., visited [i to n] = 0) and then run the breadth-first search starting from one of the vertexes, and after the search finishes, we should examine the visited array.
- In case if visited [ ] array indicates that some of the vertices are not visited, then the reason is definitely that the graph is not connected.
- Therefore, by using breadth-first search, we can say that whether the graph is connected or not.

Breadth-First Traversal

BFT can be executed using BFS.

Algorithm:

```
BFT (G, n) /*G is the graph, and 'n' is the number of vertices */
{
    for i = 1 to n do
            visited [i] = 0; /* visited[] is a
            global array of vertices. '0' value
            indicate it is not visited and '1'
            indicate it is visited.*/
    for i = 1 to n do
        if(visited [i] == 0) then
            BFS(i);
}
```

- For the time complexity of breadth-first traversal (BFT), we have to do an aggregate analysis.
- Aggregate analysis considers overall work done.
- In case if we are going to use adjacency list representation, then every node is going to have an adjacency list.
- Whenever we call BFS, then some part of the adjacency list will be completely visited, exactly one.
- Next time, when we call BFS on the remaining nodes, then the remaining nodes which are present on this list will be also visited exactly one.
- Therefore, overall, on average, all the nodes will be visited exactly one.
- In case of the undirected graph, the adjacency list contains 2E nodes.
- For initialisation of visited [ ] array, it takes O(V) time.
- Therefore, the total time complexity
  = O(V + 2E)
  = O(V + E)
- The time complexity is the same for both the directed graph as well as for undirected graph.
- Space complexity is also going to be the same as Breadth-First Search.

Conclusion

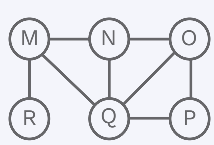
- The time and space complexity of breadth-first traversal is the same as breadth- first search.
- For a given graph,BFT calls BFS on every node.When BFS is called on a single node ,that means we are working on smaller part of the graph and then continue with the remaining part.
- So, it is as good as running the Breadth-First search on the entire graph exactly once.

Applications of Breadth First Traversal

- Shortest path and minimum spanning tree for weighted graph
- Path finding
- To test if a graph is bipartite
- Finding all nodes within one connected component
- Cycle detection in an undirected graph
- Social networking websites
- Crawlers in search engines

Previous Years' Question

The Breadth-First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



(A) MNOPQR (B) NQMPOR

(C) QMNPRO (D) QMNPOR

Solution: (D) **[2017 (Set-2)]**

Depth First Search (DFS)

- Depth-first search (DFS) is an algorithm for searching the vertices of a graph (traversing a graph to be specific), that works by exploring as far into the graph as possible before backtracking.
- **Input:**
  Graph G(V,E), where V is the set of vertices and E is the set of edges.
- To keep track of progress, a depth-first search colours each vertex
  - **White:** Vertices are white before they are discovered
  - **Gray:** Vertices are grey when they are discovered, but their outgoing edges are still in the process of being explored.
  - **Black:** Vertices are black when the entire subtree starting at the vertex has been explored
- The moment DFS discovers a vertex v in an adjacency list of already visited u,the algorithm marks the predecessor of attribute v. π=u.
- Depending upon the source vertex, there will be many subtrees possible for a DFS.
- The predecessor subgraph of a DFS is defined as

$$G_\pi = (V, E_\pi), \text{ where}$$
$$E_\pi = \{(V.\pi, V) : v \in V \text{ and } v.\pi \neq \text{NULL}\}$$

- The predecessor subgraph of DFS forms a forest with several depth-first trees.The edges in E.π are tree edges.
- DFS timestamps each vertex apart from creating a depth-first tree.The two timestamps given to a vertex:v.d is used to record when the vertex is discovered(changes v to grey colour) and v.f is used to assign appropriate finishing time after examining v's adjacency list (changes v to black).
- Since the adjacency matrix has at most $|V|^2$ entries,the timestamps range between 1 and $|V|^2$. The timestamp of discovering the vertex and finishing the vertex are v.d and v.f such that v.d<v.f, for every vertex v ∈ V.
- Vertex u is WHITE before u.d, GREY between u.d and u.f, and BLACK after u.f.
- The code below is DFS with Graph G(directed or undirected) as input. Time is a global variable.

```
DFS (G):
1.  For each vertex u ∈ G.V
2.  u.color ← WHITE
3.  u.π ← NULL
4.  Time ← 0
5.  For each vertex u ∈ G.V
6.  If u.color is WHITE
7.  DFS-VISIT(G,u)
end
DFS-VISIT(G,u)
1.  Time ← time + 1 // white vertex u has just been discovered
2.  u.d ← time
3.  u.color ← GREY
4.  For each v ∈ G.Adj[u] // explore edge (u,v)
5.  if v.color is WHITE
6   v.π ← u
7.  DFS-VISIT(G,v)
8.  u.color ← BLACK // blacken u; it is finished
9.  Time ← time + 1
10. u.f ← time
end
```

Analyzing DFS(G):

- All the vertices are coloured white and their π attributes are initialised to null in lines 1-3 of DFS (G).
- The time variable is reset in line 4.
- From line 5 to line 7, For any vertex u ∈ V is applied DFS-VISIT(G,u) if it is white.
- For each time DFS_VISIT(G,u) is called on a vertex u, then u becomes the new root.
- This DFS-VISIT(G,u) returns the vertex with u.d and u.f initialised.

Analyzing DFS-VISIT(G,u):

- The global variable time is incremented in line, 1, and the new value of discovery time u.d is updated in line 2. The vertex is coloured grey in line 3.
- From the 4th to 6th line, the vertices that are adjacent to input vertex u are checked. If they are white, their predecessor, i.e., v.π, is initialised to u.
- Since every vertex adjacent to u is considered in line 4, DFS explores the edge (u,v) for v ∈ Adj[u].

- After all the vertices adjacent to u are explored,8th line to 10th line in algorithm colours the vertex to black,increments time, and u.f is noted.

> **Note:**
> The order of vertices returned by DFS depends on the order of vertices discovered in line 5 of DFS algorithm, and line 4 of DFS-VISIT algorithm.

- Apart from the time to execute calls of DFS_VISIT,the loops in lines 1-3, and 5-7 in DFS gives Θ (V) time complexity.
- The algorithm DFS_VISIT discovers every vertex exactly once, and the vertex on which DFS_VISIT is called should be a white vertex, and the DFS_VISIT will colour it to grey at the very first step.
- The loop lines 4-7 execute |Adj[v]| times in the execution of DFS-VISIT algorithm.
  $\sum_{v \in V} |\, Adj[v] \,| = \Theta(E)$, the total time for executing lines 4–7 of DFS-VISIT is Θ (E).
  The total time taken by DFS is therefore Θ (V +E).

Depth-First Traversal

- Depth-first traversal is also exactly the same as Breadth-first traversal.
- Here, instead of calling BFS inside that traversal function, we will call DFS.
- The time and space complexity of depth-first Traversal and depth-first Search is same.

```
DFT (G, n)/* G is the graph & n is the number of vertices */
{
for i = 1 to n do
visited[i] = 0; // visited [] is an array
for i = 1 to n do
if(visited[i] == 0) then
DFS(i);
}
```

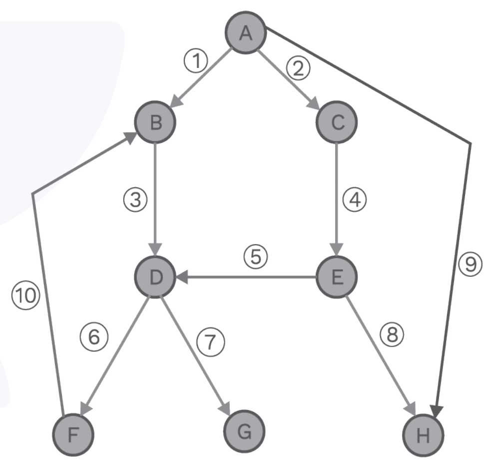
Conclusion

- Time complexity in case of adjacency list
  BFS = BFT = DFS = DFT = O(V + E)
- Time complexity in case of adjacency matrix
  BFS = BFT = DFS = DFT = $O(V^2)$
- Space complexity of
  BFS = BFT = DFS = DFT = O(V).

Applications of Depth First Search

- Detecting cycle in a graph:
  If there is a back edge in DFS, it has a cycle. Hence DFS can detect a cycle in a graph.
- Path finding:
  The DFS algorithm can be tweaked to find a path between two given vertices, u and z.
- Used as logic behind topological sorting
- To test if a graph is bipartite
- Finding strongly connected components of a graph

Triee edge, Back edge and Cross edges in DFS of graph



- **Tree Edge [Red edges]**
  Formed after applying DFS on a graph.

- **Forward Edge [vertex A to vertex H]**
  Edge (u,v), in which v is a descendent of u.

- **Back Edge [vertex F to vertex B]**
  Edges (u,v) in which v is the ancestor of u.

- **Cross Edge [vertex E to vertex D]**
  Edges(u,v) in which v is neither ancestor nor descendent to u.

Undirected graph

- In an undirected graph, forward, and backward edges are the same.
- Cross edges are not possible in an undirected graph.

Difference between DFS and BFS

Depth-First Search	Breadth-First Search
1. Backtracking is possible from a dead end. 2. Stack is used to traverse the vertices in LIFO order. 3. Search is done in one particular direction.	1. Backtracking is not possible. 2. Queue is used to traverse the vertices in FIFO order. 3. The vertices at the same level are maintained in parallel.

Topological sort:

- DFS is the logic behind topological sort in a directed acyclic graph (DAG).
- A topological sort of a DAG G=(V, E) is a linear ordering of all its vertices, such that if G contains an edge (u, v), then u appears before v in the ordering.
- If a cycle exists in a graph, there will be no linear ordering possible.

Topological Sorting(G)

Step 1: Call DFS on the graph, so that it calculates the finishing time of each vertex.

Step 2: Based on the finishing time, insert them into a linked list and return the list of vertices.

Analysis

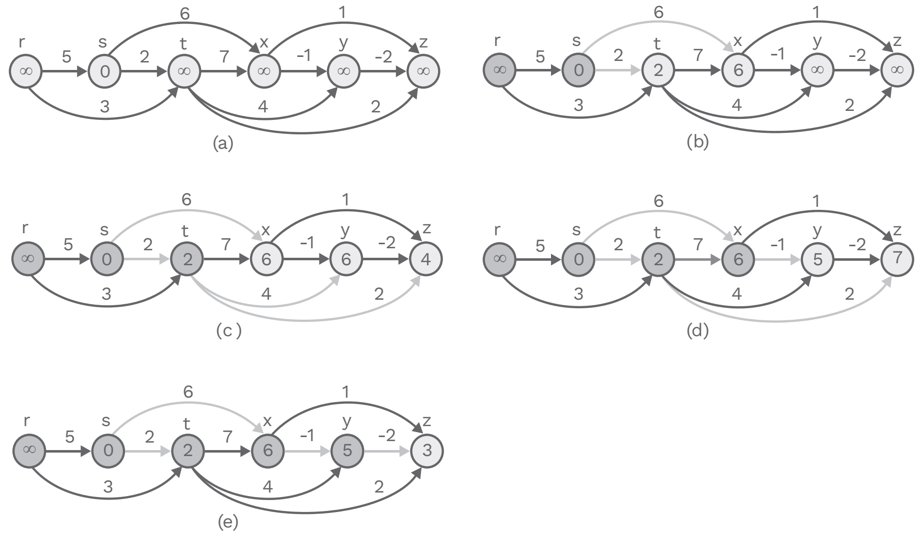
- DFS takes Θ (V + E) time, and O(V) to add all vertices one by one into the linked list.

1. Therefore, the topological sort takes O(V + E) time in total.
   DAG-SHORTEST-PATHS(G, w, s)
   a) sort the vertices of G topologically
   b) start with source
   c) for each vertex u, taken in topologically sorted order
   d) for each vertex v $\in$ G.Adj[u]
   e) RELAX(u, v, w)

Analysis

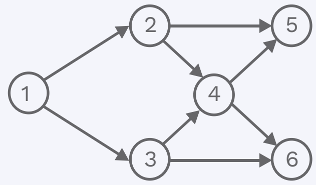
- The topological sort of line 1 takes Θ (V +E) time.
- The call of INITIALISE-SINGLE-SOURCE in line 2 takes Θ (V) time.
- The for loop of lines 3–5 makes one iteration per vertex.
- Altogether, the for loop of lines 4–5 relaxes each edge exactly once.
- Because each iteration of the inner for loop takes Θ (1) time.
- The total time is Θ (V +E).

Examples of a directed acyclic graph (DAG):



Previous Years' Question

Consider the DAG with consider V={1, 2, 3, 4, 5, 6}, shown below. Which of the following is NOT a topological ordering? **[2007]**

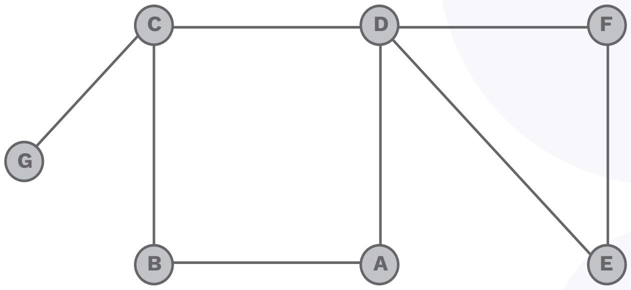


(A) 1 2 3 4 5 6
(B) 1 3 2 4 5 6
(C) 1 3 2 4 6 5
(D) 3 2 4 1 6 5

Solution: (D)

Solved Examples

1. Consider the following graph (G)



Number of cut vertex or articulation points is _______.

Solution: 2

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.

Removal of "D" vertex divides the graph into two connected components ({E, F} and {A, B, C, G}).

Similarly, removal of "C" vertex divides the graph into ({G} and {A, B, C, F}).

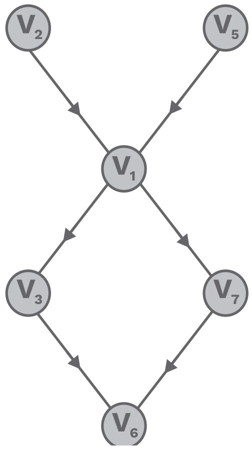
For this graph D and C are the cut vertices.

2. Topological sort can be applied to
(A) Undirected graph
(B) All types of graphs
(C) Directed acyclic graph
(D) None of the above

Solution: (C)

Topological sort can be applied to directed acyclic graphs.

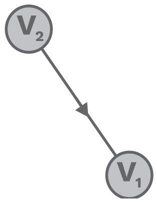
3. Consider the directed acyclic graph with $V=\{V_1, V_2, V_3, V_5, V_6, V_7\}$ shown below.



Which of the following is not a topological ordering?
(A) $V_2V_5V_1V_7V_3V_6$
(B) $V_5V_2V_1V_7V_3V_6$
(C) $V_1V_5V_2V_7V_3V_6$
(D) None of the above

Solution: (C)

Here, every edge has a dependency, like this



this edge means that V_2 comes before V_1 in topological ordering. Initially, V_2 and V_5 doesn't have any dependency. So any one of them can be done independently.

So, either start with V_2 or V_5.

So, the topological ordering is given below:

$V_2V_5V_1V_7V_3V_6$

Or

$V_5V_2V_1V_7V_3V_6$

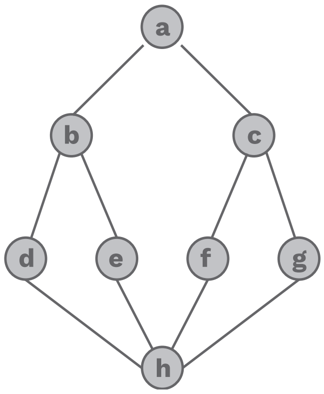
Another topological ordering is also possible, but $V_1V_5V_2V_7V_3V_6$ is not correct topological ordering because it starts with V_1 before V_2 or V_5.

Hence, (C) the correct option.

4. Consider the following sequence of nodes for the undirected graph given below.

1. a b d h e f c g 2. a c g h f e b d
3. a b d h f c g e 4. a b d h g c f e

If a Depth-First Search (DFS) is started at node a using stack data structure. The nodes are listed in the order they are first visited.



Which all of the above is/are not possible output(s)?

(A) 1,3, and 4 only

(B) 1 and 3 only

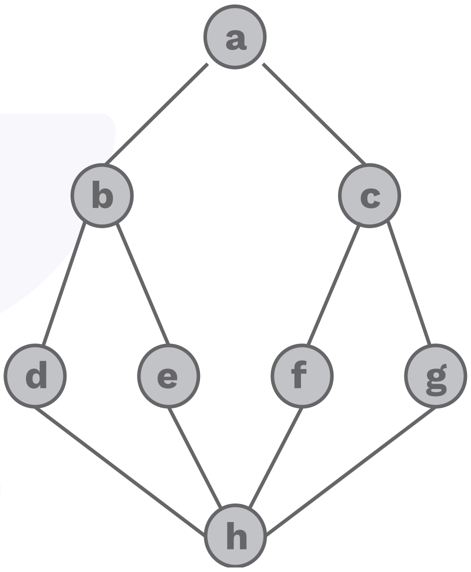
(C) 1, 2, 3, and 4 only

(D) None of the above

Solution: (D)

All the sequences of nodes are the possible output of Depts First Search (DFS).

5. The Breadth-First Search algorithm has been implemented using the queue data structure. The possible order of visiting the nodes of the following graph is (MSQ)



(A) abcdefgh

(B) acbfgdeh

(C) abcedgfh

(D) abchdefg

Solution: (A), (B), and (C)

The sequence of nodes given in options (a), (b), and (c) are the correct possible order of visiting the nodes by using breadth-first search, because breadth-first search visits the "breadth" first, i.e., if it is visiting a node, then after visiting that node, it will visit the neighbour nodes (children) first before moving on to the next level neighbours.

Chapter Summary

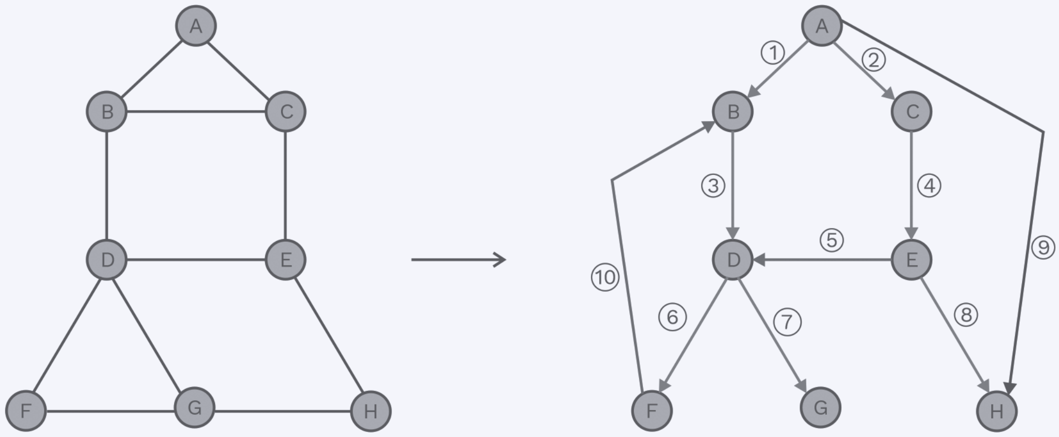
- **BFS** – Simple algorithm for traversing a graph (Breadth-wise).
- **Traversal Vs. Search** – Traversal goes through each vertex, but the search may or may not.
- **DFS** – Algorithm used for traversing a graph (depth-wise).

Difference between DFS and BFS

Depth First Search	Breadth-First Search
1. Backtracking is possible from a dead end. 2. Vertices from which exploration is incomplete are processed in a LIFO order. 3. Search is done in one particular direction.	1. Backtracking is not possible. 2. The vertices to be explored are organised as a FIFO queue. 3. The vertices at the same level are maintained in parallel.

- **Tree edge, Back edge, and Cross edge in DFS of a Graph:**

Tree edge: Formed after applying DFS on a graph.Red edges are tree edges.
Forward edge: Edge (u,v) in which v is a descendent of u.
eg: 9.
Back edge: Edges (u,v) in which v is the ancestor of u.
eg : 10.



Cross edge : Edges(u,v) in which v is neither ancestor nor descendent to u.
eg : 5

- **Topological Sort :** "A topological sort of a DEG G=(V, E) is a linear ordering of all its vertices, such that if G contains an edge (u, v), then u appears before v in the ordering."