

Data Structures and Algorithms

Lecture 9-10



WHAT ARE GRAPHS?



Why Graphs?

- - Represent pairwise relationships flexibly.
- - Useful in modeling real-world networks:
 - - Social relationships.
 - - Transportation systems.
 - - Communication networks.
- - Enable powerful algorithms (e.g., Dijkstra, A*).

Applications of Graphs

- - Social networks (e.g., Facebook, LinkedIn).
- - GPS routing and internet protocols.
- - Search engines (e.g., Google's PageRank).
- - Electrical circuits representation.
- - Artificial Intelligence (state-space problems).



GRAPH EXAMPLES

- To an algorithms person, a graph means a representation of the relationships between pairs of Objects.
- Road networks. When your smartphone's software computes driving directions, it searches through a graph that represents the road network, with vertices corresponding to intersections and edges corresponding to individual road segments.





GRAPH EXAMPLES

- **The World Wide Web.** The Web can be **modeled as a directed graph**, with the **vertices** corresponding to **individual Web pages**, and the **edges** corresponding to **hyperlinks**, directed from the page containing the hyperlink to the destination page.
- **Precedence constraints.** Graphs are also **useful in problems that lack an obvious network structure**. For example, imagine that you a first-year university student, **planning which courses to take and in which order**.





GRAPH EXAMPLES

- **Social networks.** A social network can be **represented as a graph** whose **vertices** correspond to **individuals** and **edges** to **some type of relationship**. For example, an **edge could indicate a friendship between its endpoints**, or that **one of its endpoints is a follower of the other**. Among the currently popular social networks, which ones are most naturally modeled as an **undirected** graph, and which ones as a **directed** graph?





WHAT ARE GRAPHS USED FOR?

- There are a lot of diverse problems that can be represented as graphs, and we want to answer questions about them
- For example:
 - How do we most efficiently route packets across the internet?
 - Are there natural “clusters” or “communities” in a graph?
 - Which character(s) are least related with _____?
 - How should I sign up for classes without violating pre-req constraints?

But first off, some terminology!



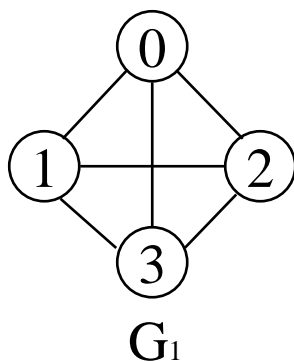
The Graph ADT (2/13)

- Definitions
 - A **graph** G consists of two sets
 - a finite, nonempty set of vertices $V(G)$
 - a finite, possible empty set of edges $E(G)$
 - $G(V,E)$ represents a graph
 - An undirected graph is one in which the pair of vertices in an edge is unordered
 - A directed graph is one in which each edge is a directed pair of vertices

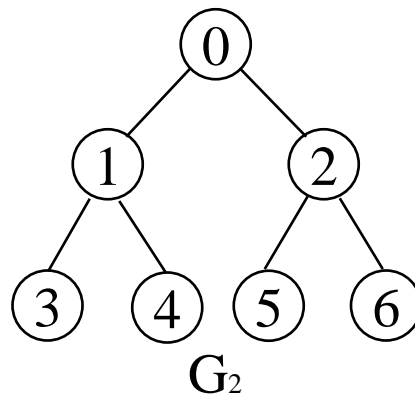
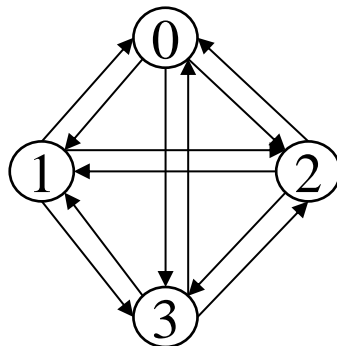
tail \longrightarrow **head**

The Graph ADT (3/13)

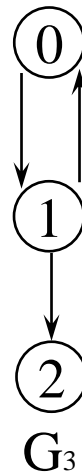
- Examples for Graph
 - complete undirected graph
 - complete directed graph



complete graph



incomplete graph



$V(G_1) = \{0, 1, 2, 3\}$

$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$

$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$

$V(G_3) = \{0, 1, 2\}$

$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$

Graphs Algorithms

- Elementary Graph Operations
 - Depth First and Breadth First Search
 - Spanning Tree
- Minimum Cost Spanning Trees
 - Kruskal's, Prim's and Sollin's Algorithm
- Shortest Paths
 - Transitive Closure
- Topological Sorts
 - Activity Networks
 - Critical Paths



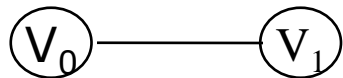
Differences: Trees vs. Graphs

Comparison	Tree	Graph
Relationship of node	Only one root node. Parent-Child relationship exists.	No root node. No Parent-Child relationship exists.
Path	Only one path between two nodes	One or more paths exist between two nodes
Edge	$N - 1$ (N = Number of nodes)	Can not defined
Loop	Loop is not allowed	Loop is allowed
Traversal	Preorder, Inorder, Postorder	BFS, DFS
Model type	Hierarchical	Network

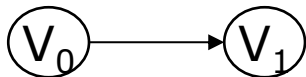


The Graph ADT (5/13)

- Adjacent and Incident
- If (v_0, v_1) is an edge in an undirected graph,
 - v_0 and v_1 are **adjacent**
 - The edge (v_0, v_1) is **incident** on vertices v_0 and v_1

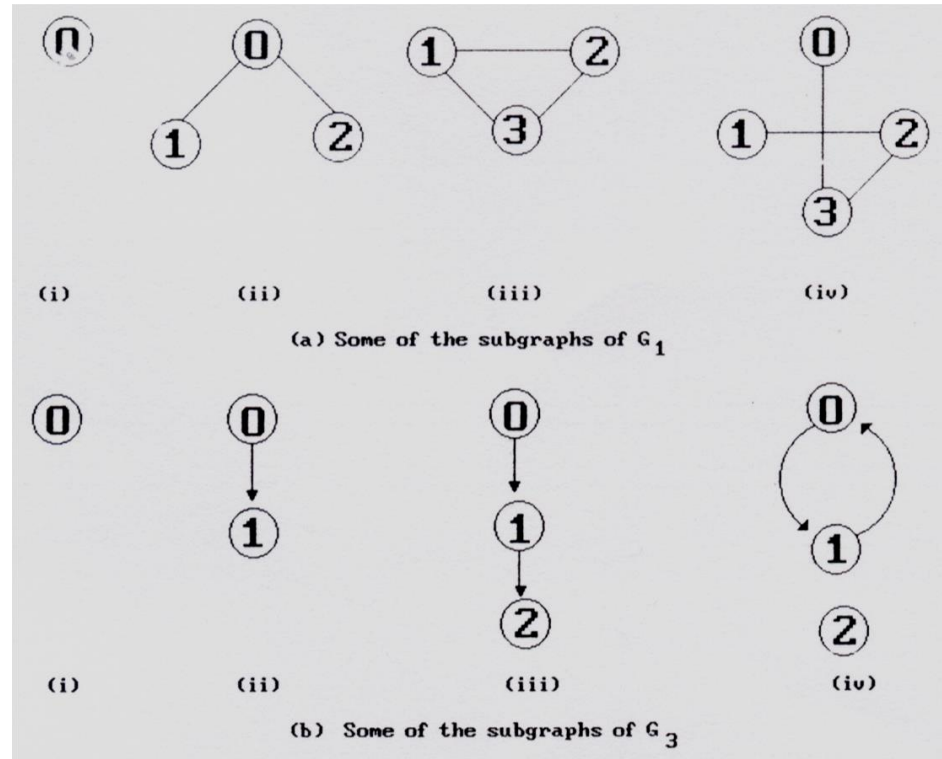
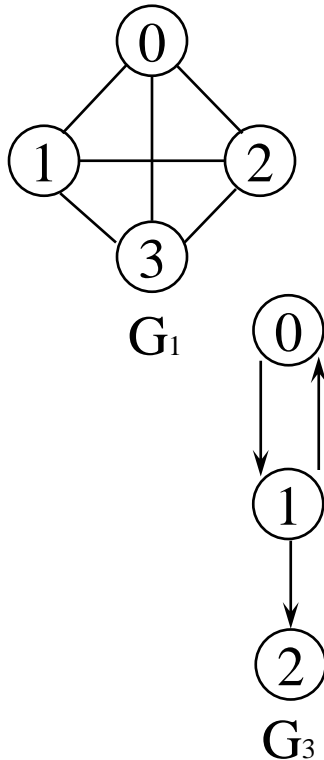


- If $\langle v_0, v_1 \rangle$ is an edge in a directed graph
 - v_0 is **adjacent to** v_1 , and v_1 is **adjacent from** v_0
 - The edge $\langle v_0, v_1 \rangle$ is incident on v_0 and v_1



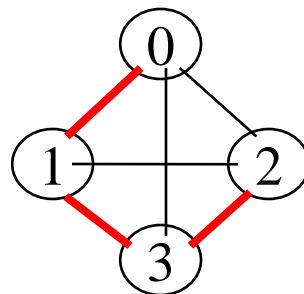
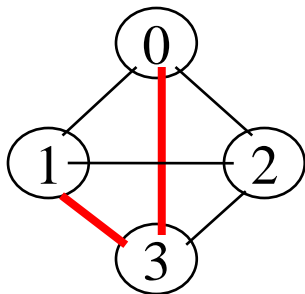
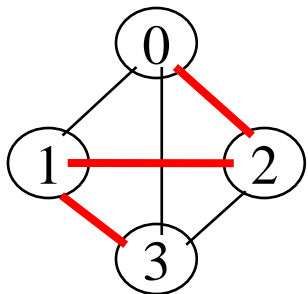
The Graph ADT (6/13)

- A subgraph of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.



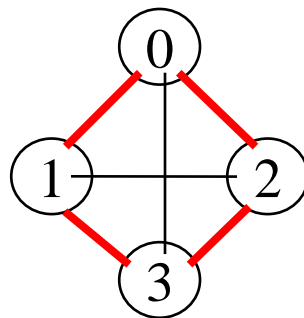
The Graph ADT (7/13)

- Path
 - A **path** from vertex v_p to vertex v_q in a graph G , is a sequence of vertices, $v_p, v_{i_1}, v_{i_2}, \dots, v_{i_n}, v_q$, such that $(v_p, v_{i_1}), (v_{i_1}, v_{i_2}), \dots, (v_{i_n}, v_q)$ are edges in an undirected graph.
 - A path such as $(0, 2), (2, 1), (1, 3)$ is also written as $0, 2, 1, 3$
 - The **length of a path** is the number of edges on it



The Graph ADT (8/13)

- Simple path and cycle
 - **simple path (simple directed path)**: a path in which all vertices, except possibly the first and the last, are distinct.
 - A **cycle** is a simple path in which the first and the last vertices are the same.



The Graph ADT (9/13)

- Connected graph
 - In an undirected graph G , two vertices, v_0 and v_1 , are **connected** if there is a path in G from v_0 to v_1 .
 - An undirected graph is **connected** if, for every pair of distinct vertices v_i, v_j , there is a path from v_i to v_j .
- Connected component
 - A **connected component** of an undirected graph is a maximal connected subgraph.
 - A **tree** is a graph that is connected and **acyclic** (i.e., has no cycle).

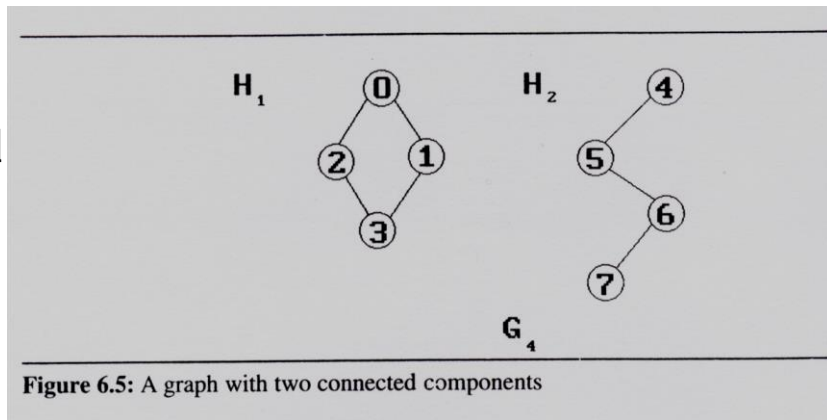
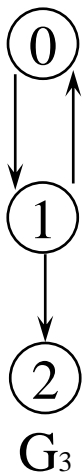


Figure 6.5: A graph with two connected components

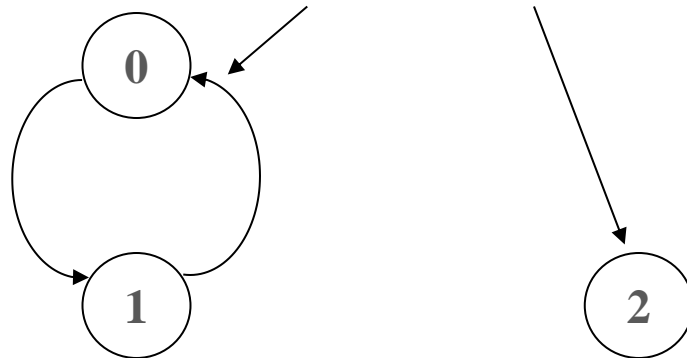
The Graph ADT (10/13)

- Strongly Connected Component
 - A directed graph is **strongly connected** if there is a directed path from v_i to v_j and also from v_j to v_i
 - A **strongly connected component** is a maximal subgraph that is strongly connected



not strongly connected

strongly connected component
(maximal strongly connected subgraph)

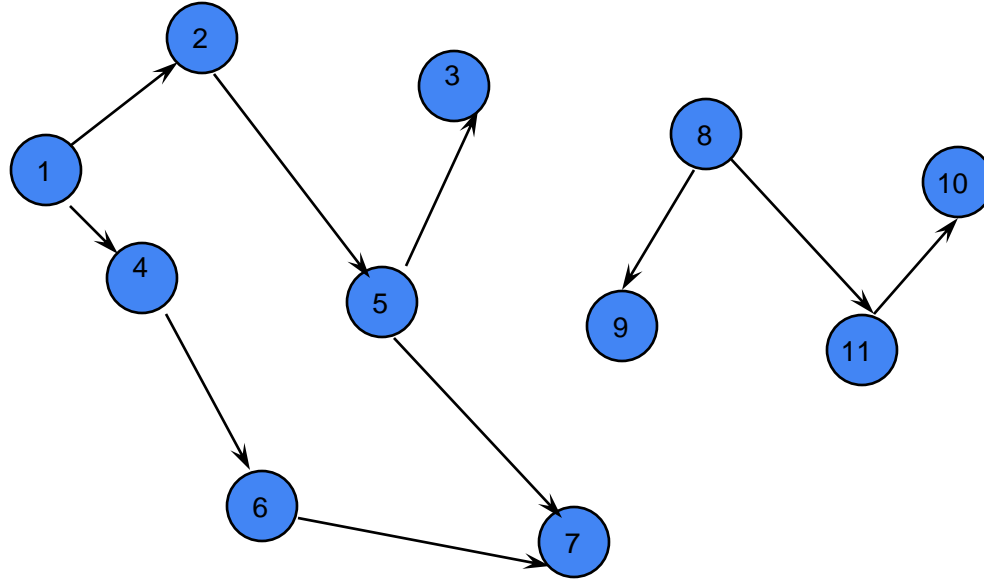


The Graph ADT (11/13)

- Degree
 - The **degree** of a vertex is the number of edges incident to that vertex.
- For directed graph
 - **in-degree** (v) : the number of edges that have v as the head
 - **out-degree** (v) : the number of edges that have v as the tail
- If d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

$$e = (\sum_{i=0}^{n-1} d_i) / 2$$

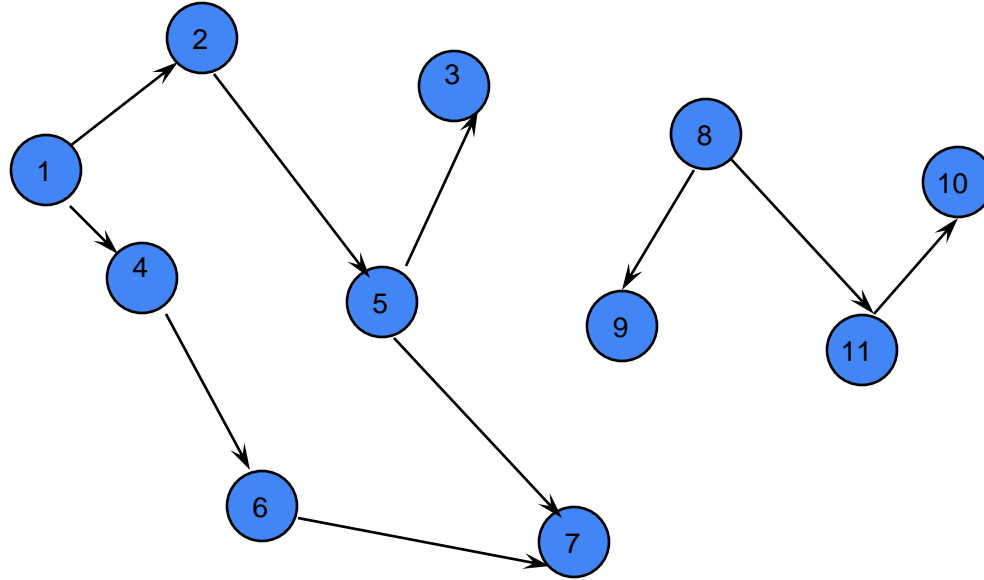
In-Degree Of A Vertex



in-degree is number of incoming edges

$\text{indegree}(2) = 1$, $\text{indegree}(8) = 0$

Out-Degree Of A Vertex



out-degree is number of outbound edges

$\text{outdegree}(2) = 1$, $\text{outdegree}(8) = 2$

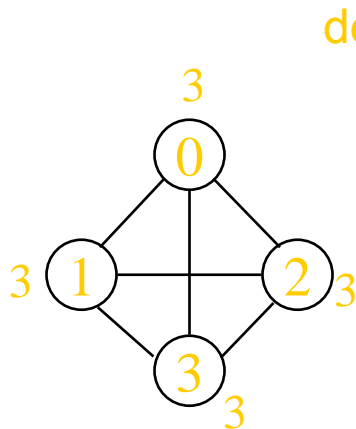
The Graph ADT (12/13)

- Degree (cont'd)
- We shall refer to a directed graph as a *digraph*. When we use the term *graph*, we assume that it is an undirected graph

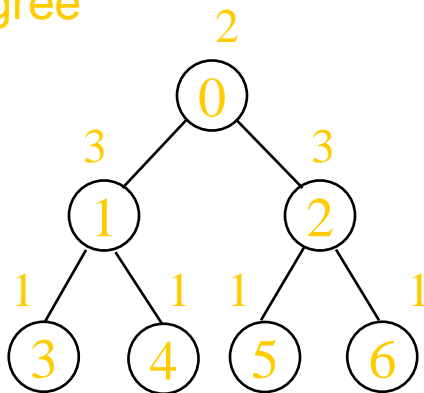
directed graph

undirected graph

in-degree & out-degree



G_1



G_2



in:1, out: 1

in: 1, out: 2

in: 1, out: 0

G_3

KINDS OF GRAPHS



TWO KINDS OF GRAPHS:

1. Undirected Graph
2. Directed Graph



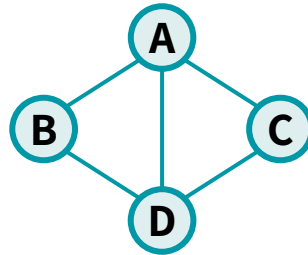
Two KINDS OF GRAPHS:

- In an **undirected graph**, each edge corresponds to an **unordered pair** $\{v, w\}$ of vertices, which are called the endpoints of the edge. In an undirected graph, **there is no difference between an edge (v, w) and an edge (w, v) .**

UNDIRECTED GRAPHS

An undirected graph has
a set of vertices (V) & a set of edges (E)

Formally,
 $G = (V, E)$



$V = \{A, B, C, D\}$

$E = \{ \{A, B\}, \{A, C\}, \{A, D\}, \{B, D\}, \{C, D\} \}$





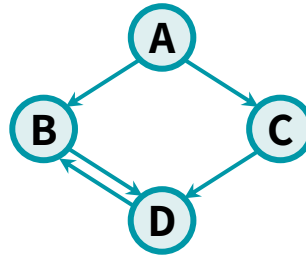
Two KINDS OF GRAPHS:

- In a **directed graph**, each edge (v,w) is an **ordered** pair, with the edge traveling from the first vertex v (called the **tail**) to the second w (the **head**);

DIRECTED GRAPHS

Formally,
 $G = (V, E)$

A directed graph has
a set of vertices (V) & a set of **DIRECTED** edges (E)



The **in-degree** of vertex D is 2. The **out-degree** of vertex D is 1.
Vertex D's **incoming neighbors** are A, B, & C
Vertex D's **outgoing neighbor** is B



GRAPH REPRESENTATIONS OPTIONS



GRAPH REPRESENTATIONS OPTIONS:

OPTION 1: ADJACENCY MATRIX

OPTION 2: ADJACENCY LIST



GRAPH REPRESENTATIONS

OPTION 1: ADJACENCY MATRIX

- The adjacency matrix representation of G is a square $n \times n$ matrix A equivalently, a two-dimensional array—with only zeroes and ones as entries. Each entry A_{ij} is defined as:

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ belongs to } E \\ 0 & \text{otherwise.} \end{cases}$$





GRAPH REPRESENTATIONS

OPTION 2: ADJACENCY LIST

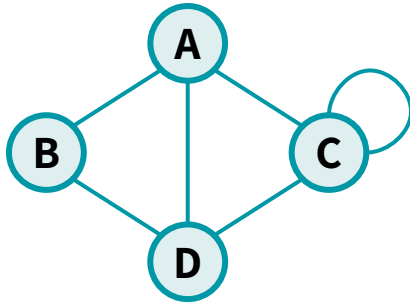
- Ingredients for Adjacency Lists
 1. An array containing the graph's vertices.
 2. An array containing the graph's edges.
 3. For each edge, a pointer to each of its two endpoints.
 4. For each vertex, a pointer to each of the incident edges.





GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(An undirected graph)

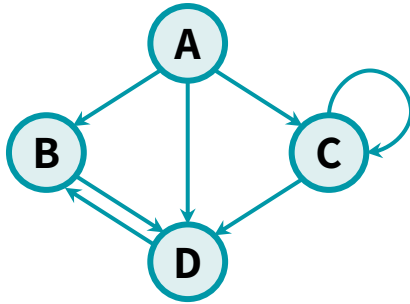
		(destination)			
		A	B	C	D
(source)	A	0	1	1	1
	B	1	0	0	1
	C	1	0	1	1
	D	1	1	1	0





GRAPH REPRESENTATIONS

OPTION 1: **ADJACENCY MATRIX**



(A directed graph)

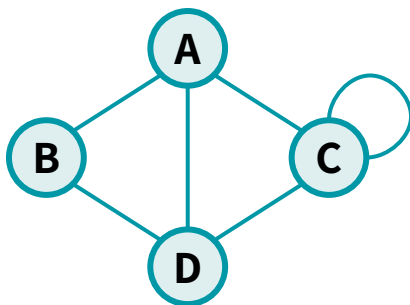
		(destination)			
		A	B	C	D
(source)	A	0	1	1	1
	B	0	0	0	1
	C	0	0	1	1
	D	0	1	0	0



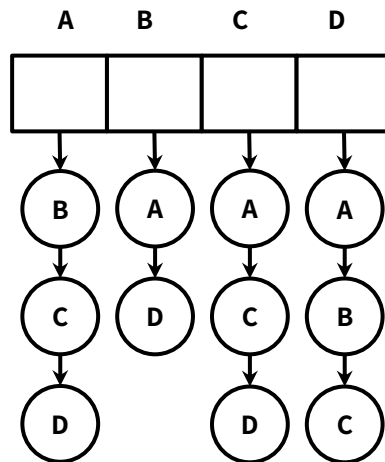


GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**



(An undirected graph)



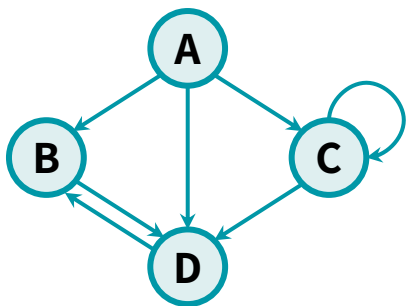
Each list stores a node's neighbors



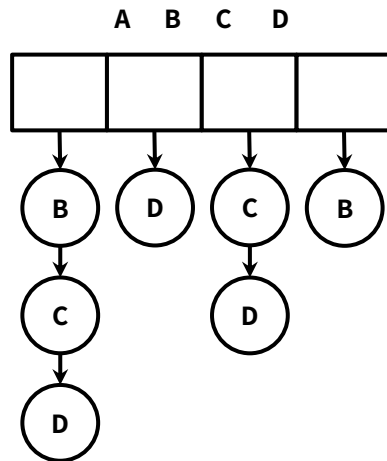


GRAPH REPRESENTATIONS

OPTION 2: **ADJACENCY LISTS**



(A directed graph)



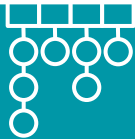
Tracks *outgoing* neighbors.

(You could also do the same for incoming neighbors as well)





GRAPH REPRESENTATIONS

GRAPH REPRESENTATIONS TECHNIQUES	ADJACENCY MATRIX	ADJACENCY LISTS
For a graph $G = (V, E)$ where $ V = n$, and $ E = m$	$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$	
SPACE REQUIREMENTS	$O(n^2)$	$O(n + m)$

Generally, better for
sparse graphs
(where $m \ll n^2$).

**We'll assume this
representation,
unless otherwise
stated.**





GRAPH REPRESENTATIONS

Generally Adjacency Lists representation, better than Adjacency Matrix representation for sparse graphs (where $m < n^2$).

We'll assume Adjacency Lists representation, unless otherwise stated.



Sparse Graphs vs. Dense Graphs





Sparse Graphs vs. Dense Graphs

- A graph is **sparse** if the number of edges is relatively close to linear in the **number of vertices**.
- A graph is **dense** if the number of edges is closer to quadratic in the **number of vertices**.





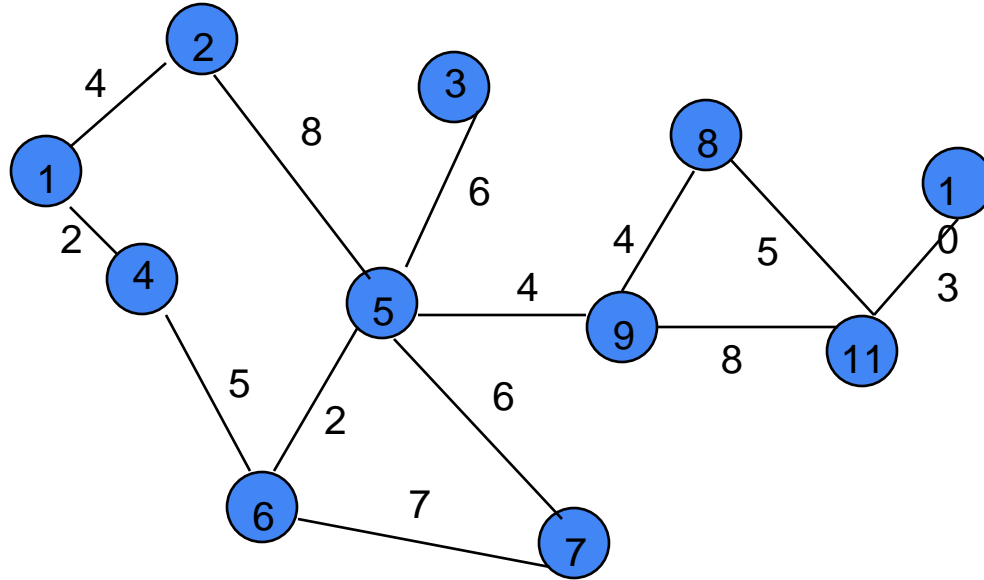
Spanning Tree



Spanning Tree

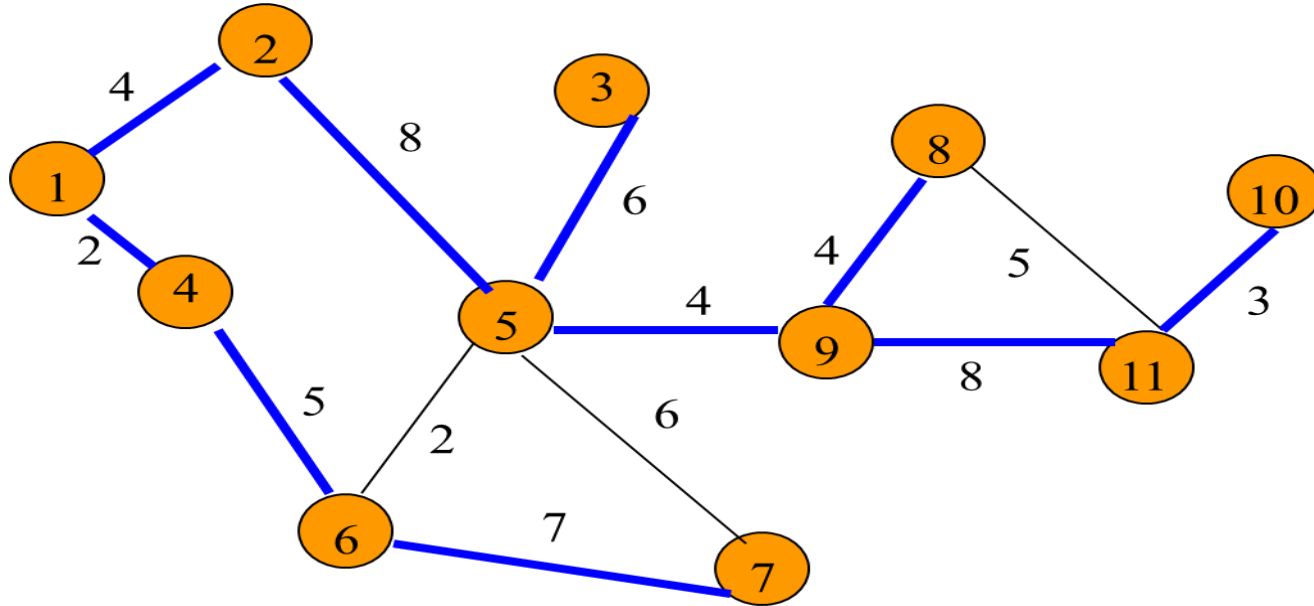
- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
 - If original graph has n vertices, the spanning tree has n vertices and $n-1$ edges.

Minimum Cost Spanning Tree



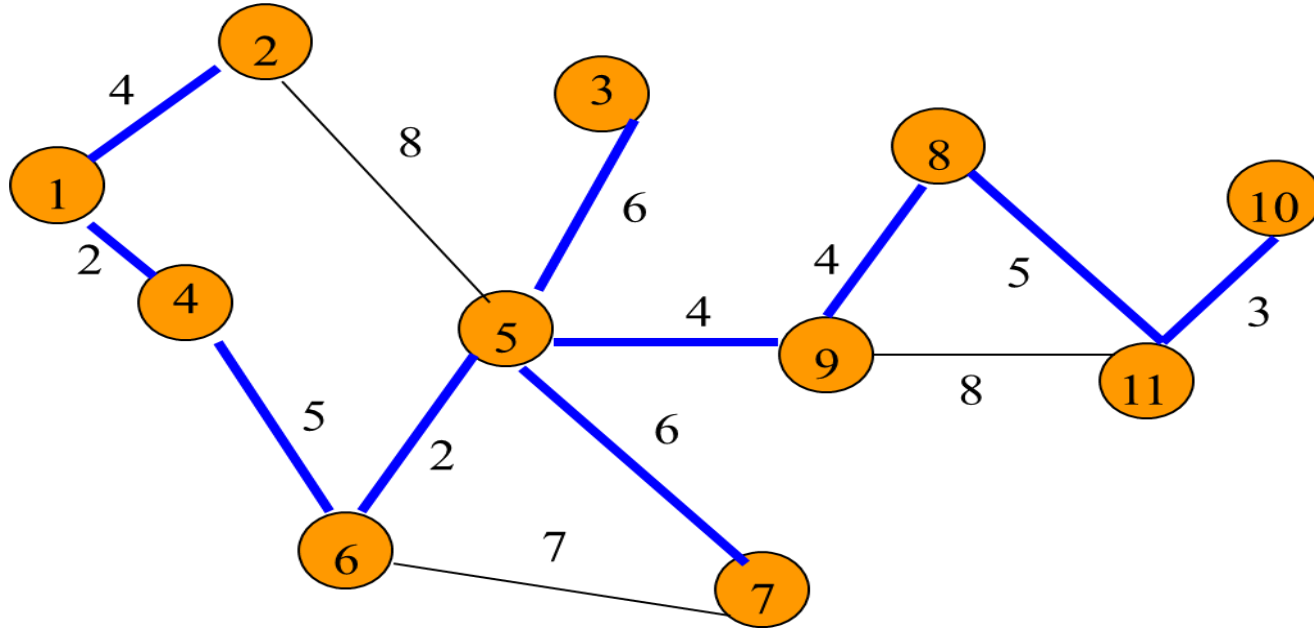
- Tree cost is sum of edge weights/costs.

A Spanning Tree



Spanning tree cost = 51.

Minimum Cost Spanning Tree



Spanning tree cost = 41.



BREADTH-FIRST SEARCH (BFS)

One way to explore a graph!



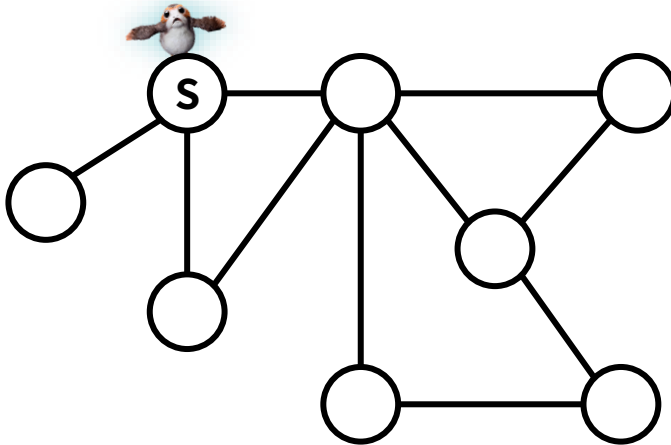


BREADTH-FIRST SEARCH

- Breadth-first search explores the vertices of a graph in layers, in order of increasing distance from the starting vertex.

An analogy:

A bird is exploring a maze from above (with a bird's eye view)

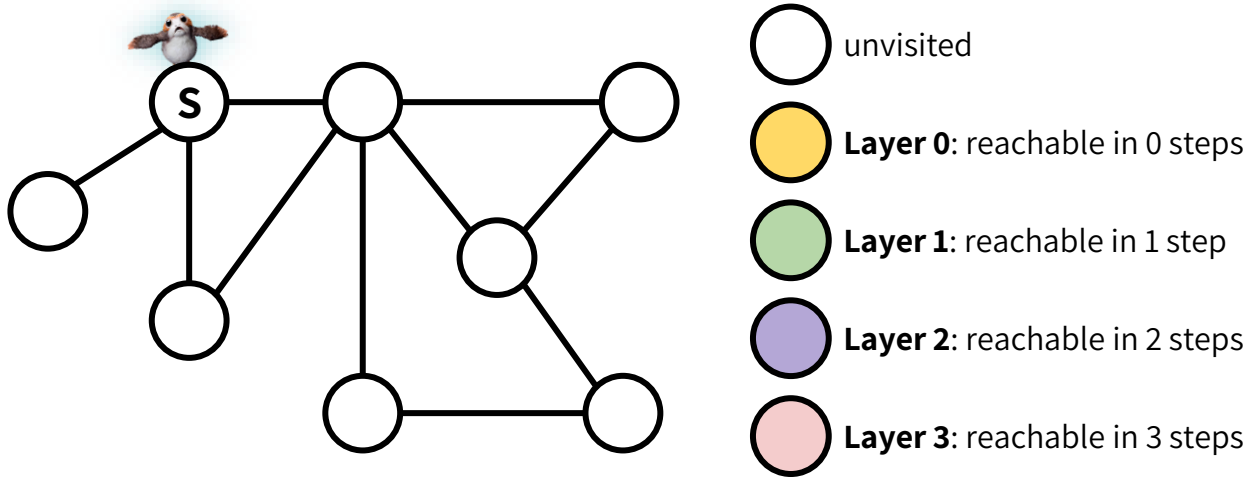




BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)

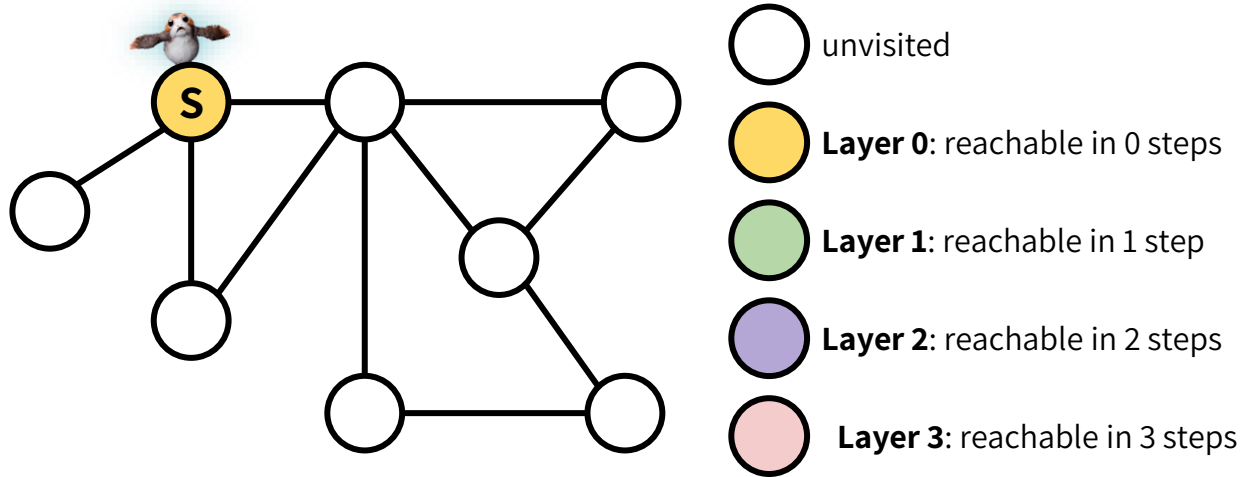




BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)

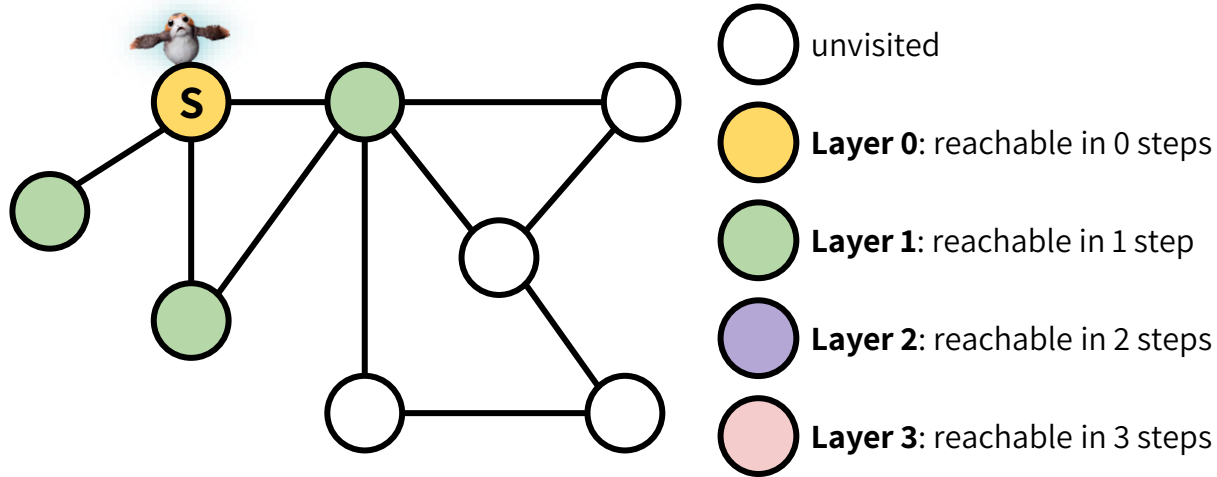




BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)

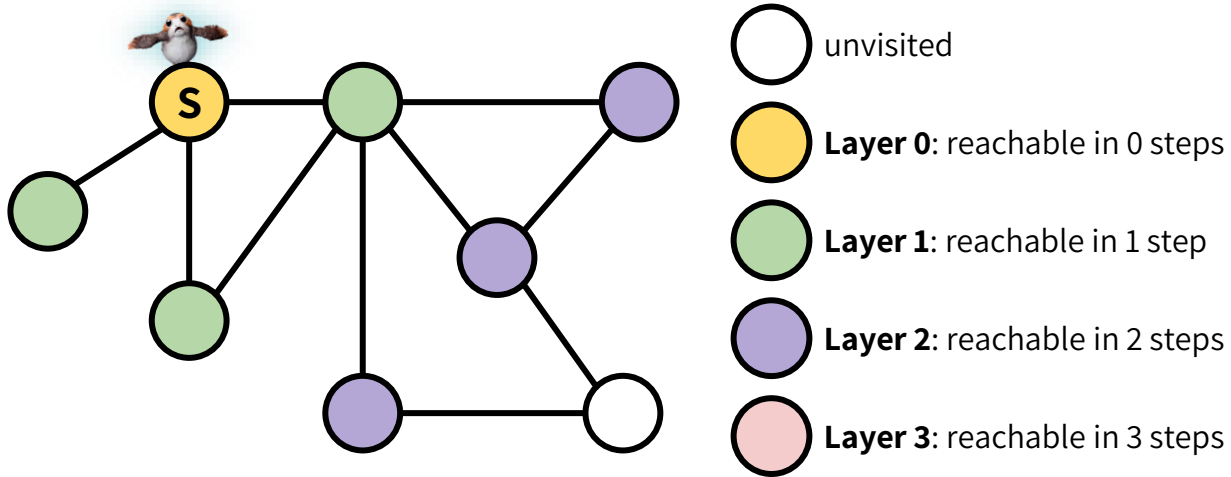




BREADTH-FIRST SEARCH

An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)

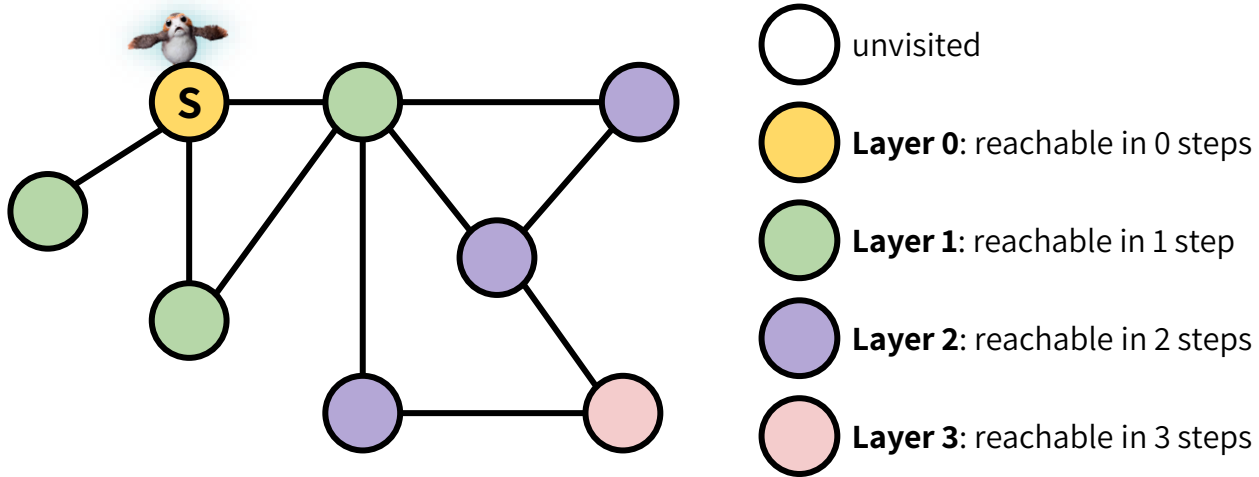




BREADTH-FIRST SEARCH

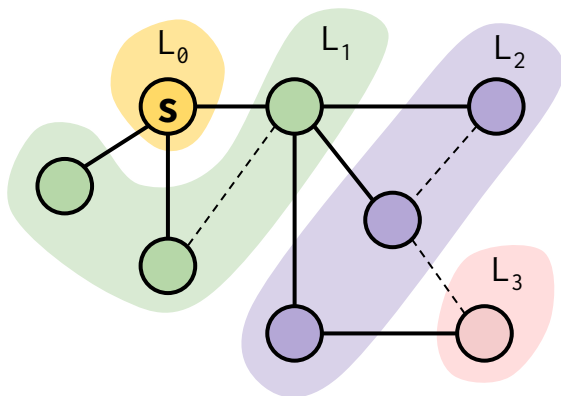
An analogy:

A bird is exploring a labyrinth from above (with a bird's eye view)





BREADTH-FIRST SEARCH



L_i = The set of nodes we can reach in i steps from s

BFS(s):

Set $L_i = []$ for $i = 0, \dots, n-1$

$L_0 = s$

for $i = 0, \dots, n-1$:

 for u in L_i :

 for v in u .neighbors:

 if v not yet visited:

 mark v as visited

 add v to L_{i+1}



Applications of Breadth First Search





1. Shortest Path for unweighted graph:

In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges.

2. Peer-to-Peer Networks:

In Peer-to-Peer Networks like [BitTorrent](#), Breadth First Search is used to find all neighbor nodes.

[BitTorrent](#) is a hyper distribution communications protocol for peer-to-peer file sharing ("P2P") which is used to distribute data and electronic files over the Internet.





3. Crawlers in Search Engines:

Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.

4. Social Networking Websites:

In social networks, we can find people within a given distance 'k' from a person using Breadth First Search till 'k' levels.





5. GPS Navigation systems:

Breadth First Search is used to find all neighboring locations.

6. Broadcasting in Network:

In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7. To test if a graph is Bipartite:

We can either use Breadth First or Depth First Traversal.





8. Path Finding:

We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

9. Cycle detection in undirected graph:

In undirected graphs, either Breadth First Search or Depth First Search can be used to detect a cycle. We can use BFS to detect cycle in a directed graph also.



Advantages and Disadvantages of Breadth First Search



Advantages of Breadth First Search:

- BFS will never get trapped exploring the useful path forever.
- If there is a solution, BFS will definitely find it.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Advantages of Breadth First Search:

- Low storage requirement – linear with depth.
- Easily programmable.

Disadvantages of Breadth First Search:

- The main drawback of BFS is its memory requirement.
- Since each level of the graph must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is $O(bd)$, where b is the branching factor (the number of children at each node, the outdegree) and d is the depth.
- As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.



DEPTH-FIRST SEARCH

One way to explore a graph!





DFS (Iterative Version)

Input: graph $G = (V, E)$ in adjacency-list representation, and a vertex $s \in V$.

Postcondition: a vertex is reachable from s if and only if it is marked as “explored.”

mark all vertices as unexplored

$S :=$ a stack data structure, initialized with s

while S is not empty **do**

 remove (“pop”) the vertex v from the front of S

if v is unexplored **then**

 mark v as explored

for each edge (v, w) in v 's adjacency list **do**

 add (“push”) w to the front of S





Literally just BREADTH vs DEPTH:

While

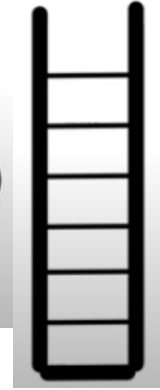
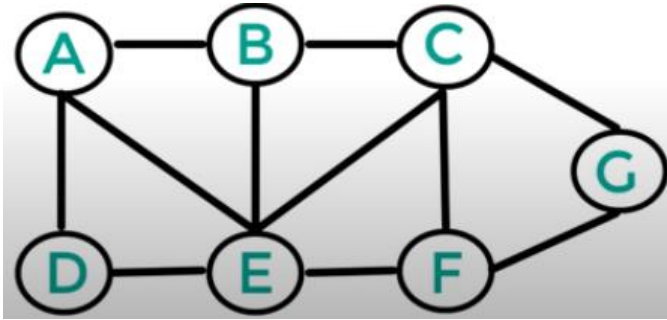
BFS first explores the nodes closest to the “source” and then moves outwards in layers,

DFS goes as far down a path as it can before it comes back to explore other options.





DFS stack example



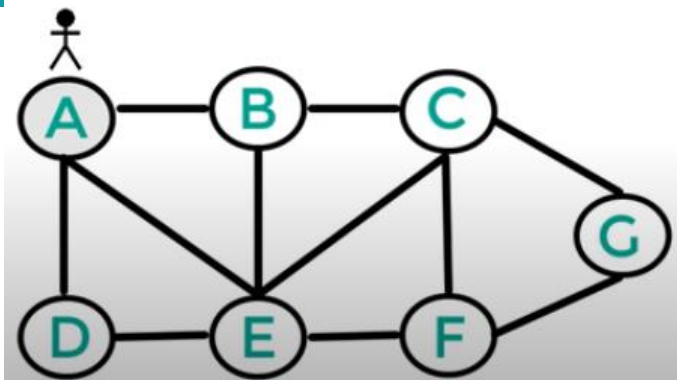
STACK

1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.





DFS stack example



STACK

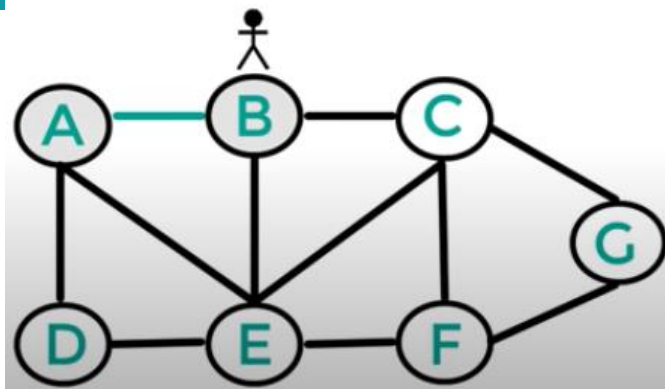


1. Define a **stack**, size = # of Vertices in graph.
2. Select **any vertex** as starting point, visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3, **until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

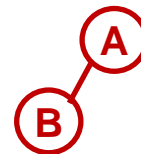




DFS stack example



STACK

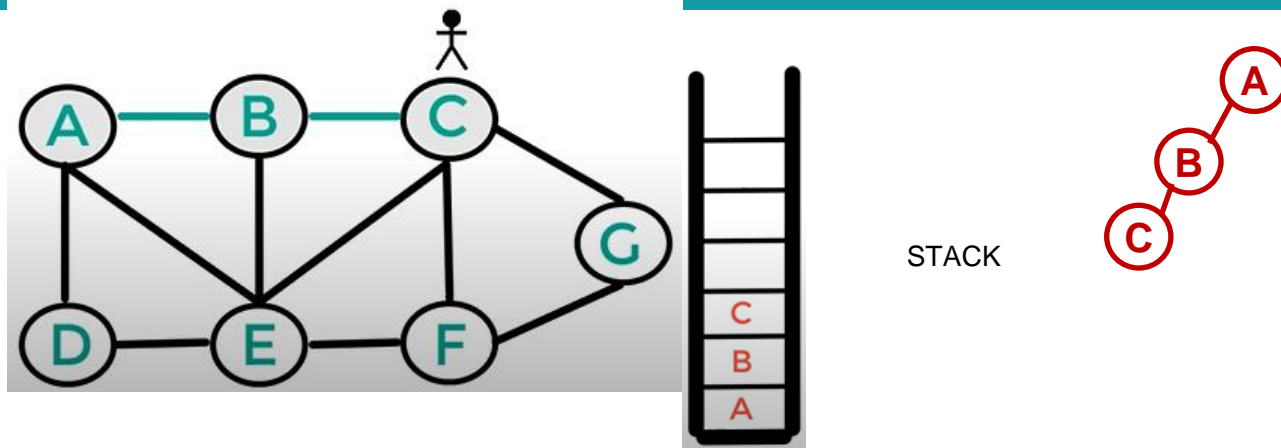


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.





DFS stack example

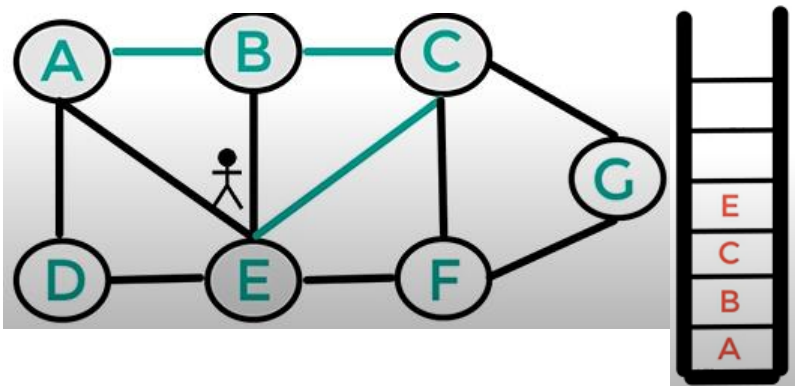


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

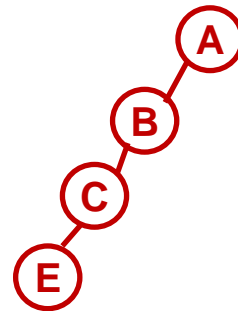




DFS stack example



STACK

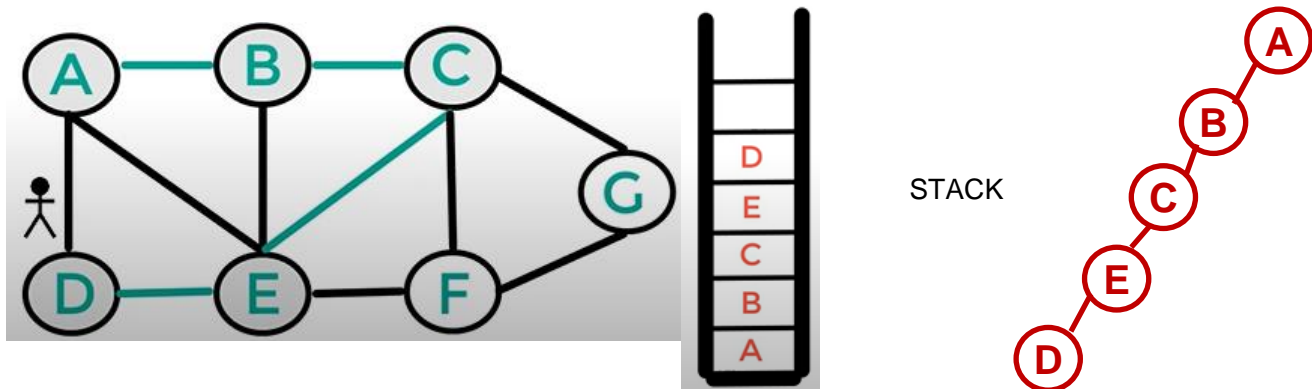


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.





DFS stack example

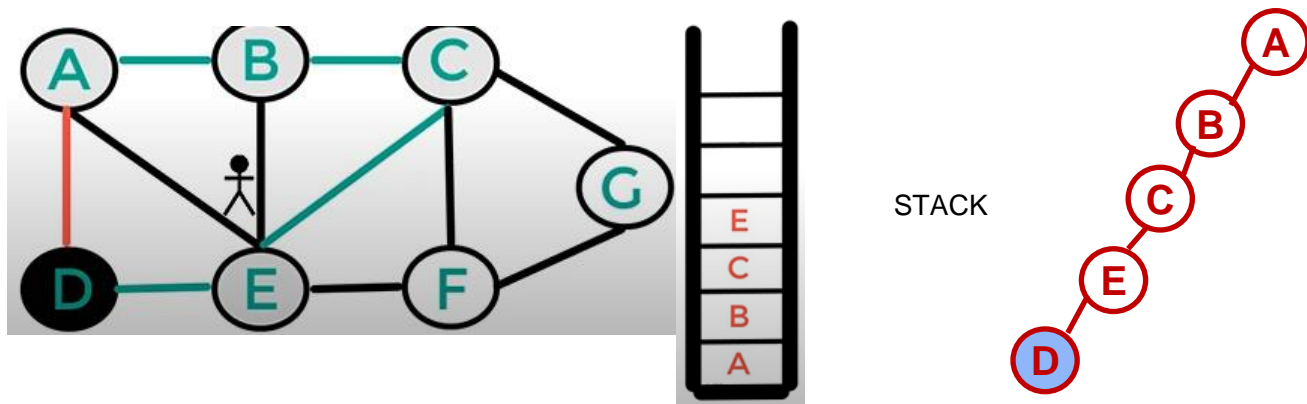


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.





DFS stack example

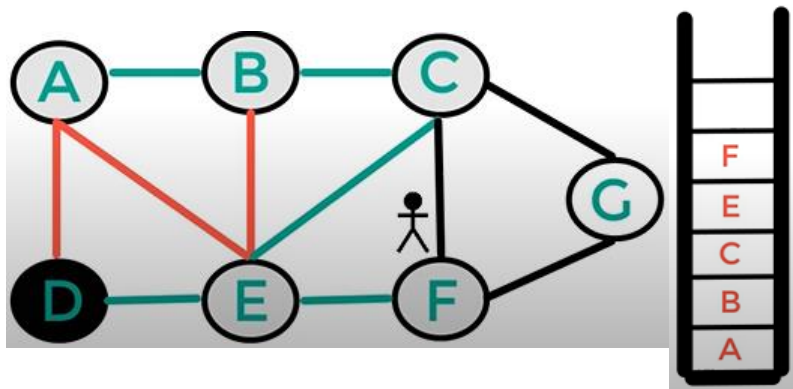


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

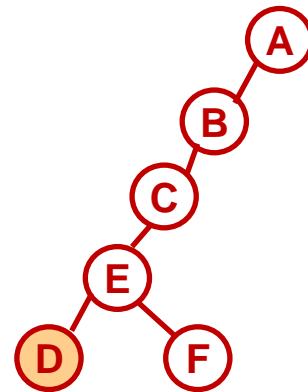




DFS stack example



STACK

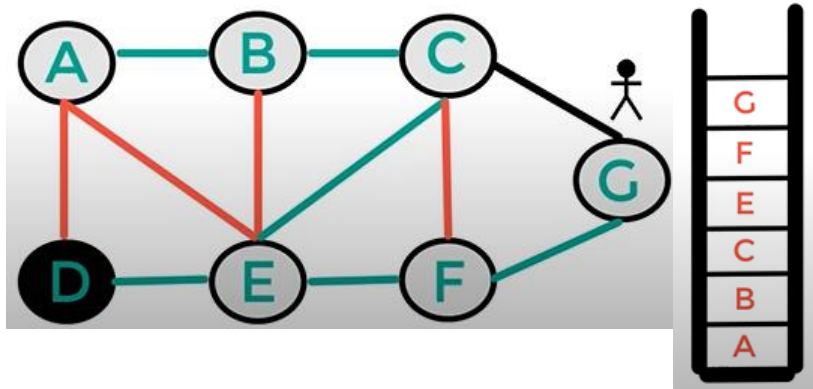


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

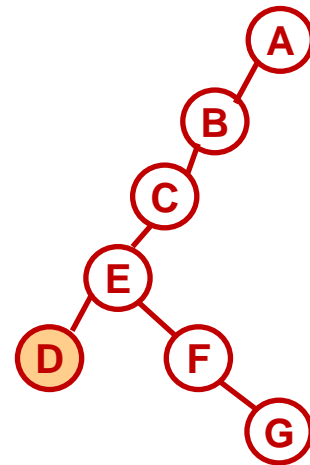




DFS stack example



STACK

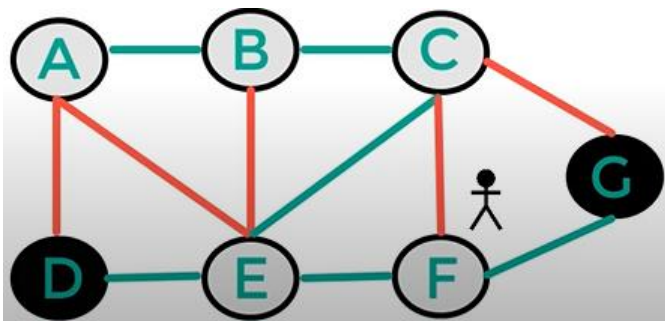


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

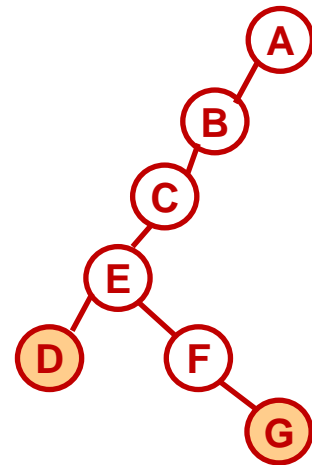




DFS stack example



STACK

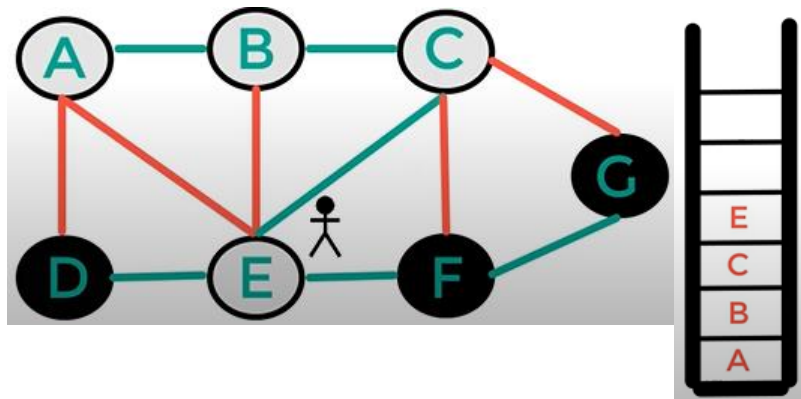


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

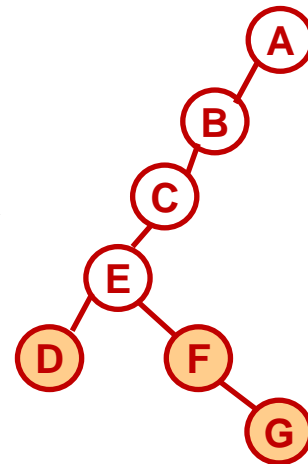




DFS stack example



STACK

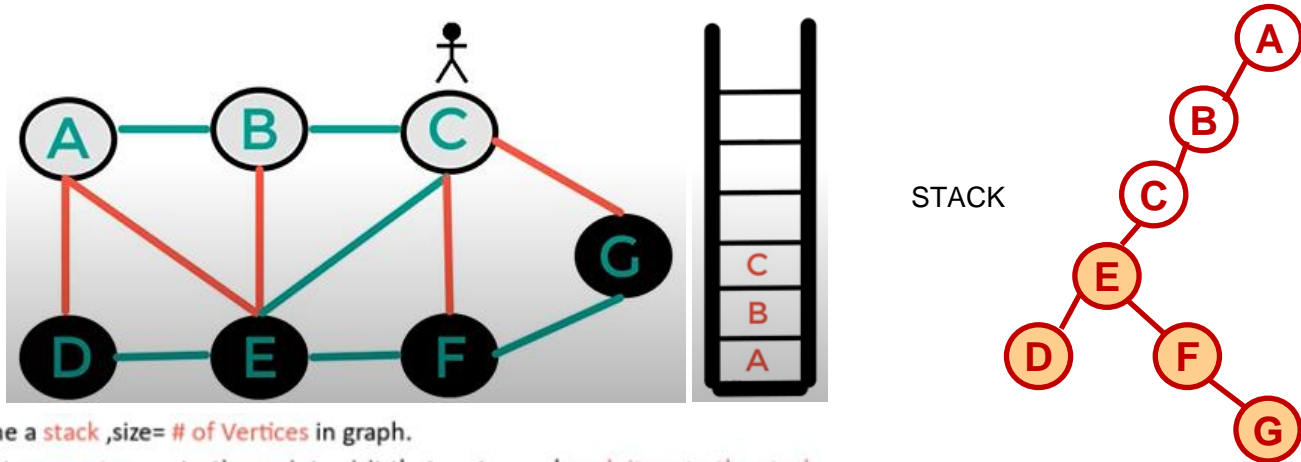


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.





DFS stack example

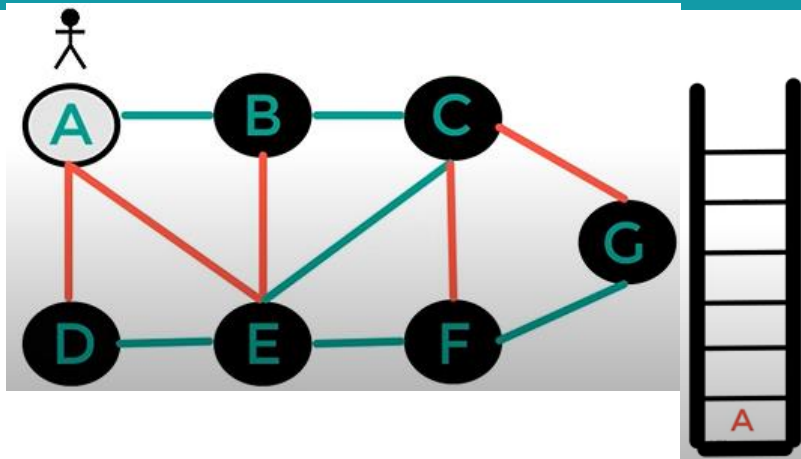


1. Define a **stack** ,size= # of Vertices in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top of the stack**, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

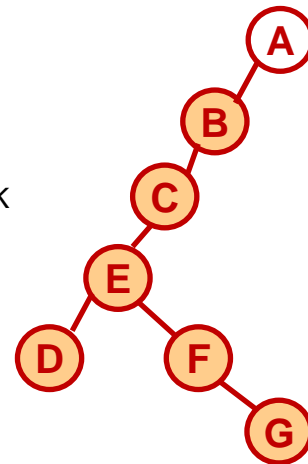




DFS stack example



STACK

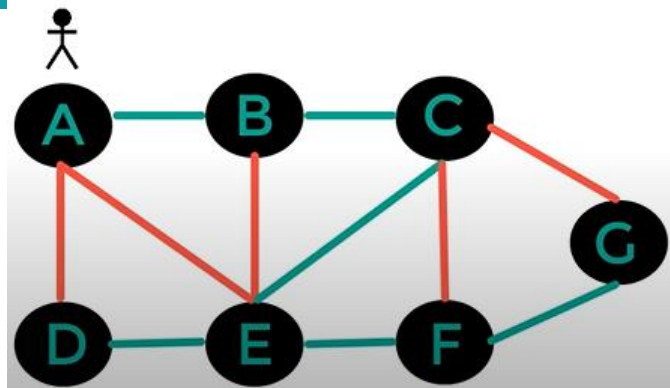


1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.

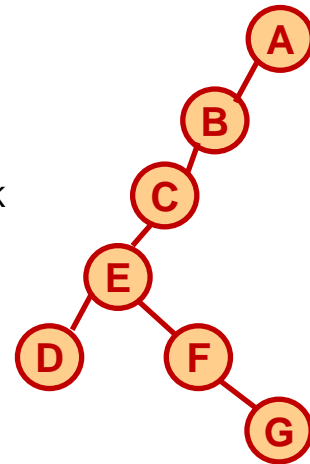




DFS stack example



STACK



1. Define a **stack** ,size= **# of Vertices** in graph.
2. Select **any vertex** as starting point , visit that vertex and **push it on to the stack**.
3. Visit **any one of the adjacent vertex** of the vertex which is **at top** of the stack, Which **is not visited**, and **push it on the stack**.
4. Repeat step 3 ,**until there are no new vertex** to be Visit from the vertex on top of the stack.
5. When there **is no new vertex** to be visit then use **backtracking** and **pop one vertex** from the stack.
6. Repeat steps 3,4,5 **until stack be comes empty**.



Advantages and Disadvantages of Depth First Search



Advantages of Depth First Search:

- Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth d and branching factor b (the number of children at each node, the outdegree) is $O(bd)$.

Advantages of Depth First Search:

- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
- DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

Disadvantages of Depth First Search:

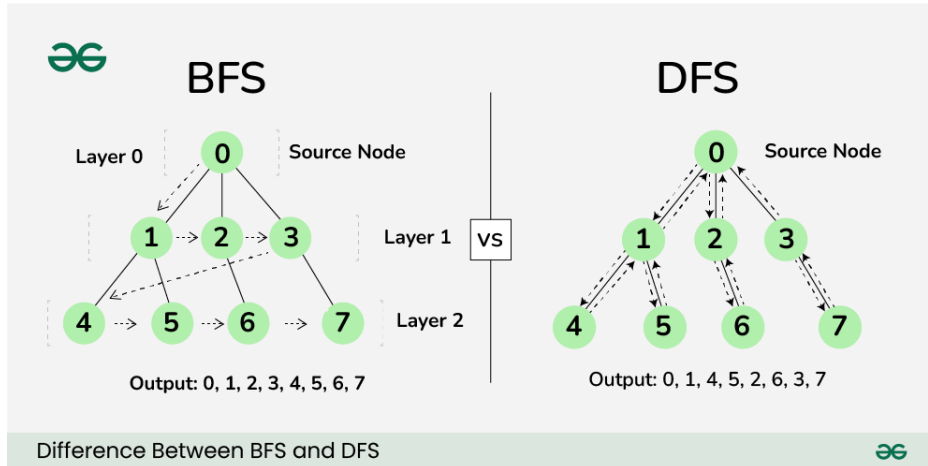
- The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever.
- Depth-First Search is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution, if more than one solution.

Difference between BFS and DFS



Difference between BFS and DFS:

- Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental algorithms used for traversing or searching graphs and trees.



Breadth-First Search (BFS):

BFS, Breadth-First Search, is a vertex-based technique for finding the shortest path in the graph.

It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

Depth First Search (DFS):

DFS, Depth First Search, is an edge-based technique.

It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

Difference between BFS and DFS:

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.

Difference between BFS and DFS:

Parameters	BFS	DFS
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc.

Thank You

