10 - Graphs

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

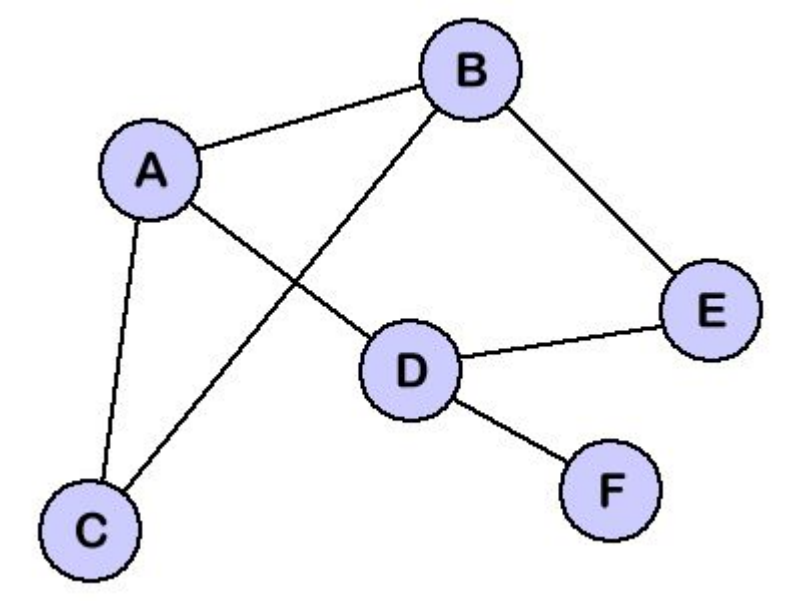
Agenda

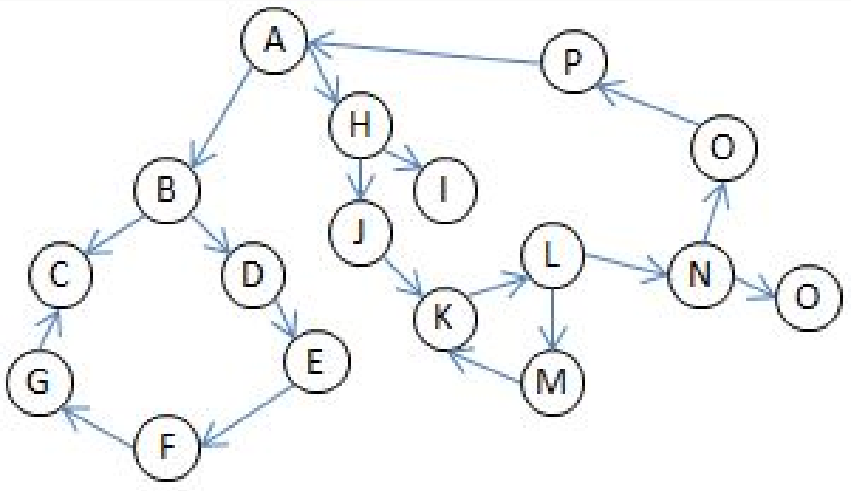
- Intro
- Terms
- Examples
- Representations
- Traversals
- BFS/DFS

Reading Assignment

- Read Chapter 28 - Graphs
  - Chapter 28 (Read about: **Examples and Terms, Traversals, DFS, BFS**)

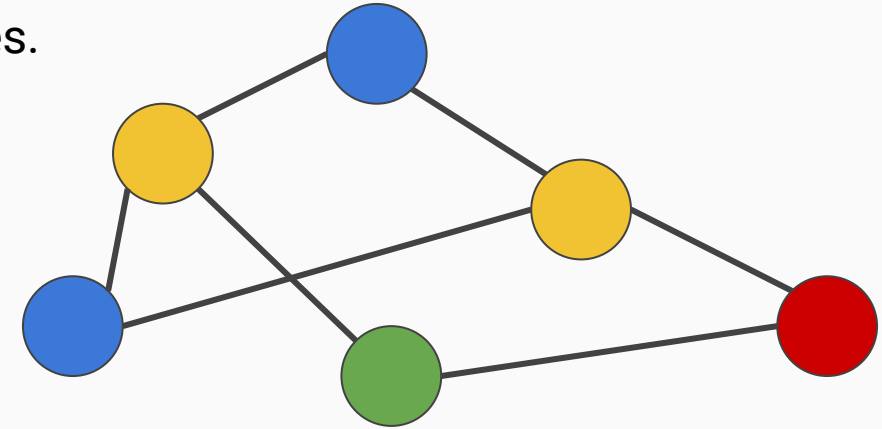
Graphs





Graphs

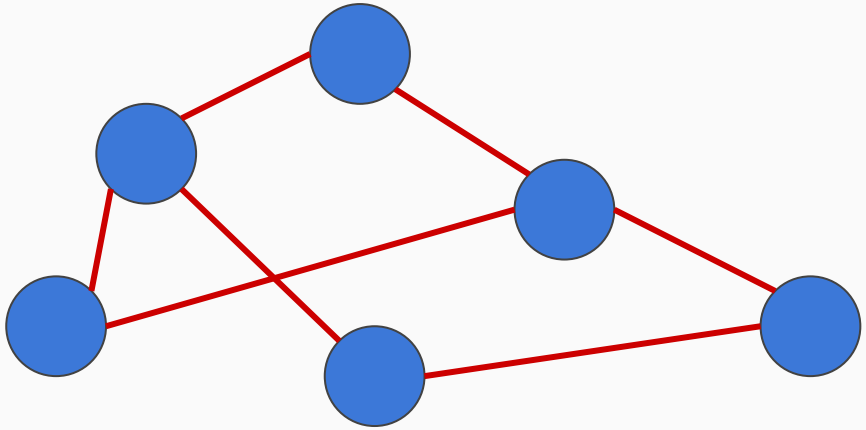
- An abstract data type that represents Mathematical concepts
- Useful in modeling certain types of problems
- A **graph** is a non-linear data structure consisting of nodes and links between nodes.



Terms

Nodes or Vertices

Edges or Lines

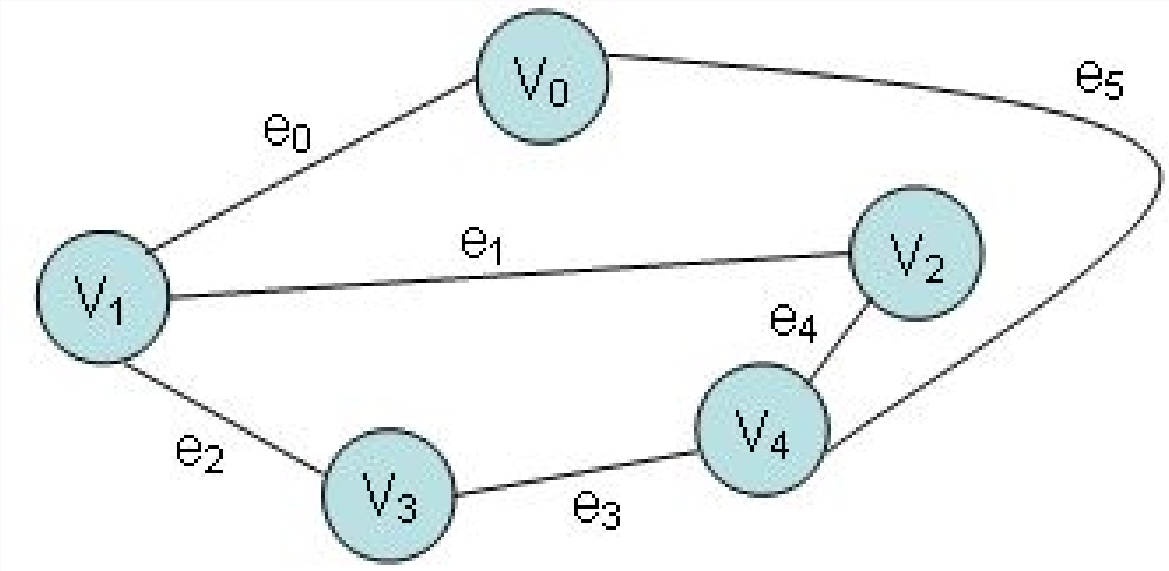


Undirected Graphs

- An **undirected graph** is a set of nodes and a set of links between the nodes.
  - The order of the two connected vertices is **unimportant**.
  - An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.

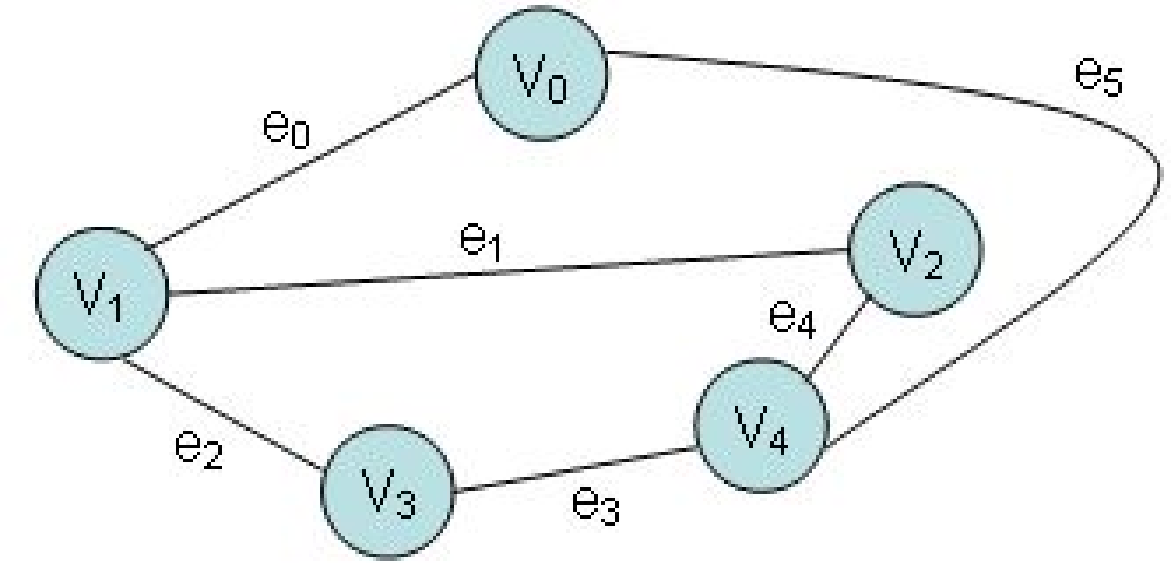
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices.

Undirected Graphs Contd.



ICE 10.1 Undirected Graphs

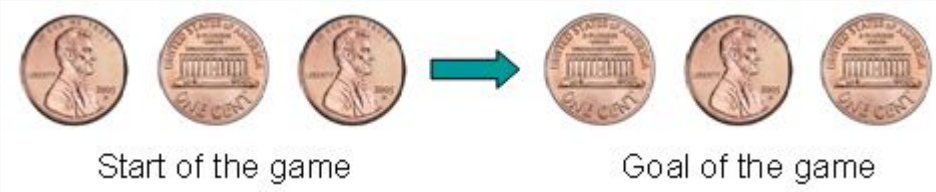
Identify all the edges, and vertices in the graph below:



Example: Coin Flip Game

Rules:

1. You may flip the middle coin whenever you want to.
2. You may flip one of the end coin only if the other two coins are the same as each other.



ICE 10.2 Coin Flip Game

Find the flip operations required to beat the coin flipping game:

Rules:

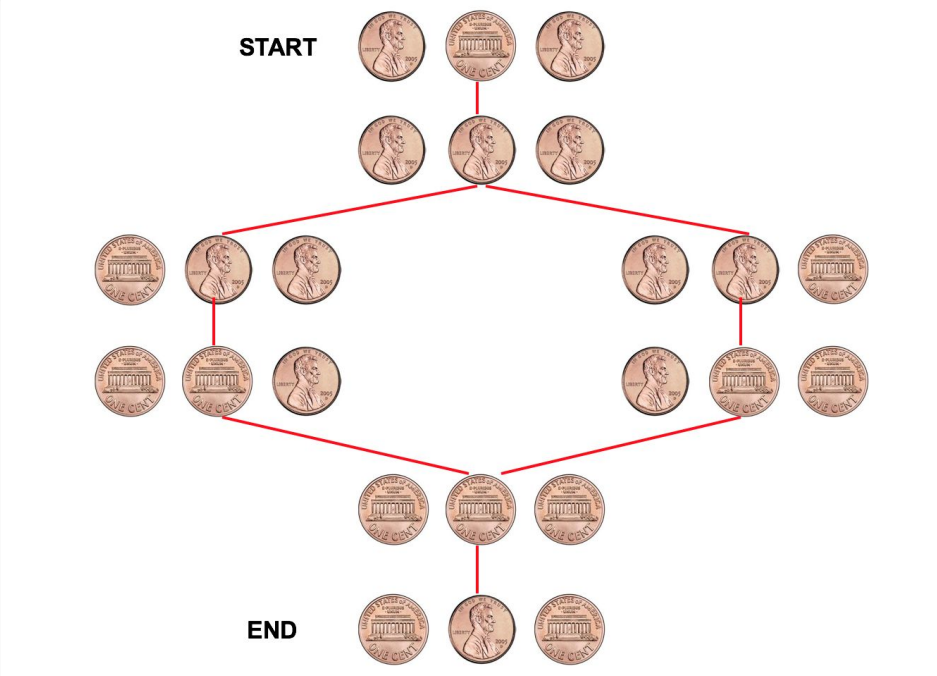
1. You may flip the middle coin whenever you want to.
2. You may flip one of the end coin only if the other two coins are the same as each other.

Graphs and Problem Solving

- Often a problem can be represented as a graph, and the solution to the problem is obtained by solving a problem on the corresponding graph.

- **Let’s look at a coin flipping game:**

Solution



Graphs and Problem Solving Contd.

- To solve the above problem, we could build an undirected state graph.
- Once we know the state graph, the game becomes a problem of finding a path from one vertex to another, where the path is allowed only to follow edges.

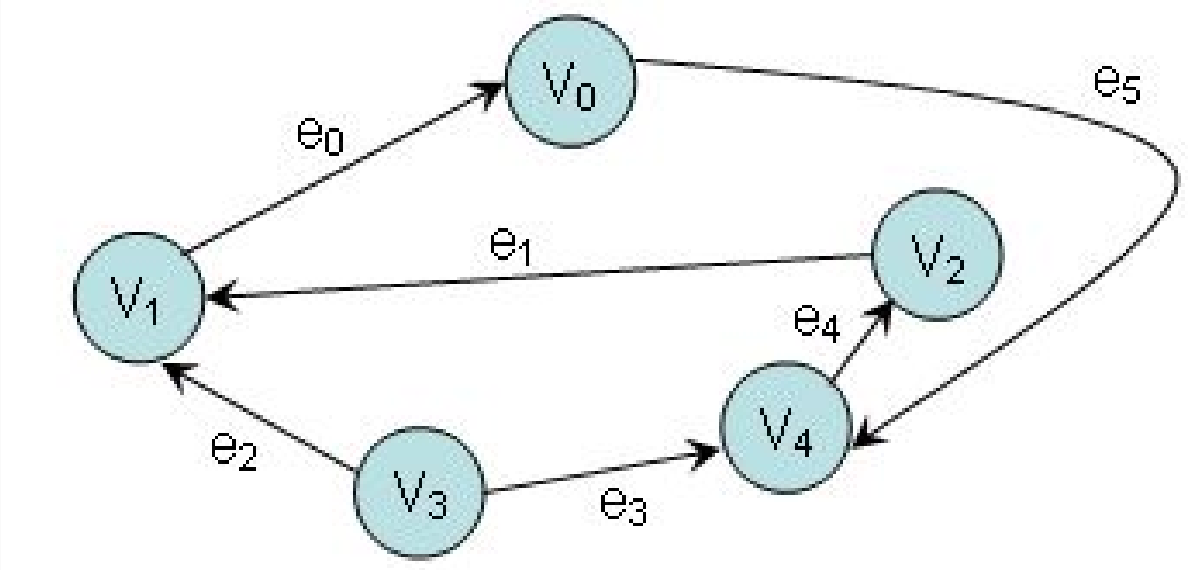
Convert Problem to Graph -> Solve Using Graph Properties

Directed Graphs

- A directed graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.

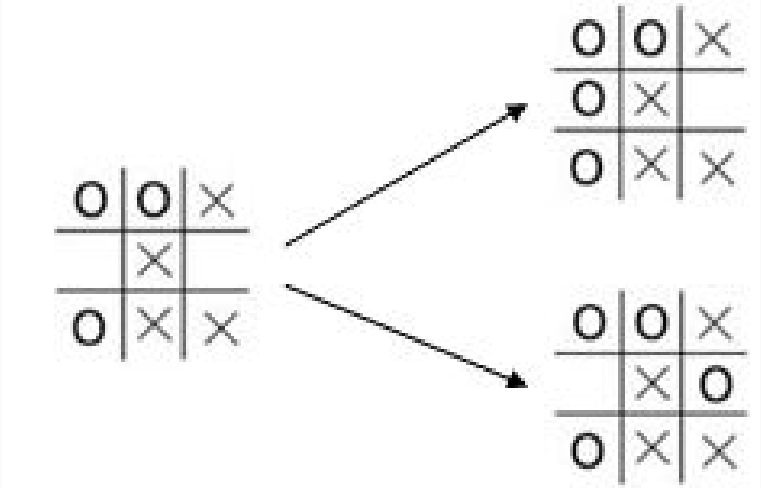
- Each edge is associated with two vertices, called its **source** and **target** vertices.
- We say that the edge connects its source to its target.
- The **order** of the two connected vertices is important.

Directed Graphs



Directed Graphs Contd.

- A state graph for a game where reversing a move is sometimes forbidden….



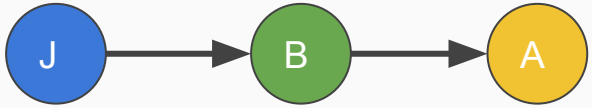
Directed vs. Undirected

Directed Graph - Twitter Followers

➔ Direction of edge holds meaning

Ex)

- Joseph follows Bob, but not Alice
- Bob follows Alice, but not Joseph
- Alice doesn't follow anyone

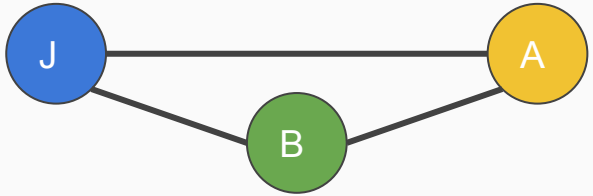


Undirected Graph - Facebook Friends

➔ Edge is bidirectional

Ex)

- Joseph is friends with Bob
- Bob is friends with Alice
- Alice is friends with Joseph



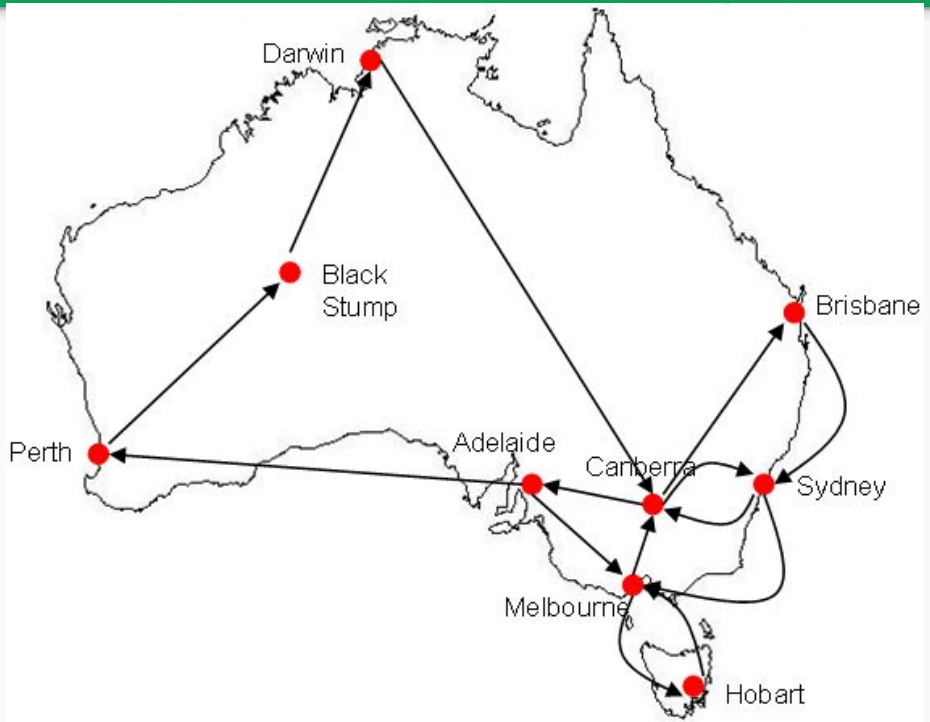
Additional Terms

- **Loop**: an edge that connects a vertex to itself.
- **Path**: a sequence of vertices, p_0, p_1, ..., p_m, such that each adjacent pair of vertices p_i and p_{i+1} are connected by an edge.
- **Cycle**: a simple path with no repeated vertices or edges other than the starting and ending vertices. A cycle in a directed graph is called a directed cycle.

- **Multiple edges**: a graph can have two or more edges connecting the same two vertices in the same direction.
- **Simple graphs**: the graphs that have no loops and no multiple edges. Many applications require only simple directed graphs or even simple undirected graphs.

ICE 10.3 Identification

1. How many vertices and edges does the graph have? How many loops?
2. Is it a simple graph? Why or why not?
3. What is the shortest path from "Black Stump" to "Melbourne"? -- The shortest path problem.

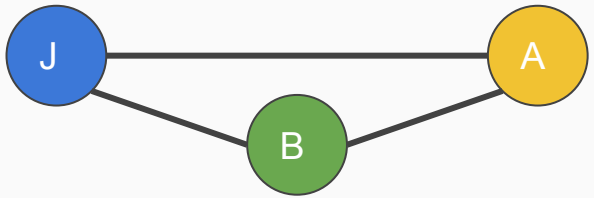


Cyclic vs. Acyclic

Cyclic Graph - Facebook Friends

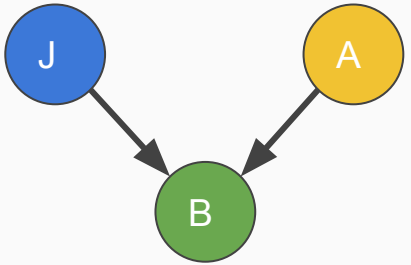
- ➔ Relationships between nodes can form cycles

Ex) Previous Graph



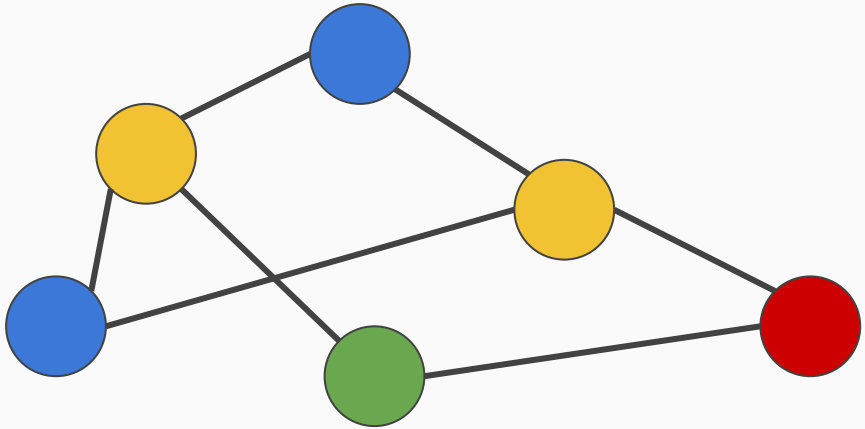
Acyclic Graph - Family Tree

- ➔ Relationships between nodes cannot form loops or cycles
- ➔ DAG (Directed Acyclic Graph) - Directed graph without cycles



How is this useful?

- Modeling Networks
- Complex Relationships
  - when trees are too inflexible
- Distance Routing
  - Pathfinding
- Graph Coloring
  - Creating timetables
  - Compiler - register allocation



Graph Implementation

- Different kinds of graphs require different kinds of implementations:
- **Representations:**
  - Adjacency Matrix
  - Edge Lists

Note: *We are focusing on directed graphs in which loops are allowed*

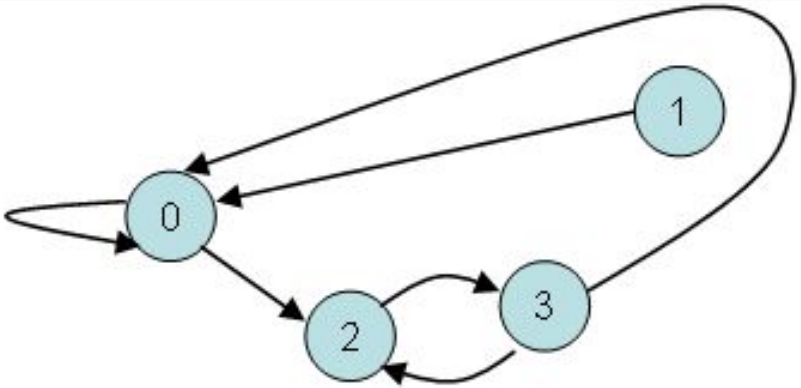
Adjacency Matrix

- Represent a graph as a two-dimensional array.
- The placement of values in the array represent edges of the graph

- **How big is the array?**
  - Graph of **n** vertices needs an array of **n rows and n columns**

Adjacency Matrix

- **How it works?**
  - Let's say we have two vertices: ***i and j***
  - The value at ***row i and column j*** is **true** if there is an edge from ***vertex i -> vertex*** j otherwise, the component is **false**.



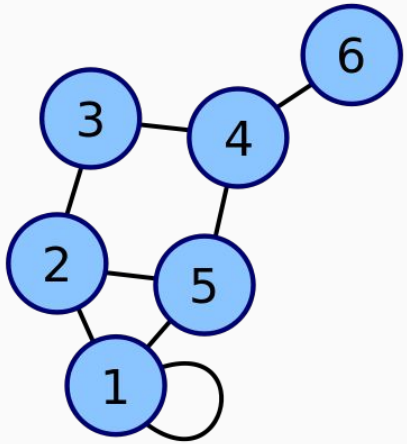
	0	1	2	3
0	true	false	true	false
1	true	false	false	false
2	false	false	false	true
3	true	false	true	false

Adjacency Matrix

- **Benefits:**
  - Once created, we have a matrix that can be examined to represent the graph
  - Matrix representation is compact and can serve as a good plain-text representation of the graph.
  - You will see adjacency matrices used in some graph algorithms for operations (ex. dijkstra's)
  - **You can answer interesting questions:**
    - Which vertices link to a specific vertex *(Look at the column)*

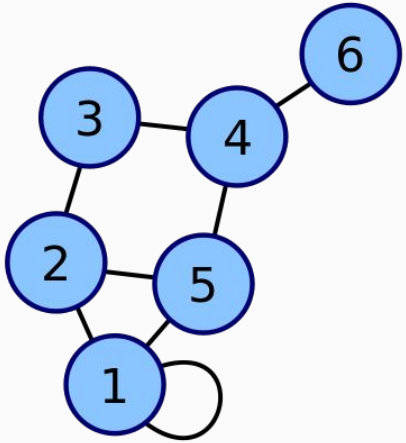
ICE 10.4 Adjacency Matrix

Build the Adjacency Matrix for the following Graph:



ICE 10.4 Adjacency Matrix

Build the Adjacency Matrix for the following Graph:



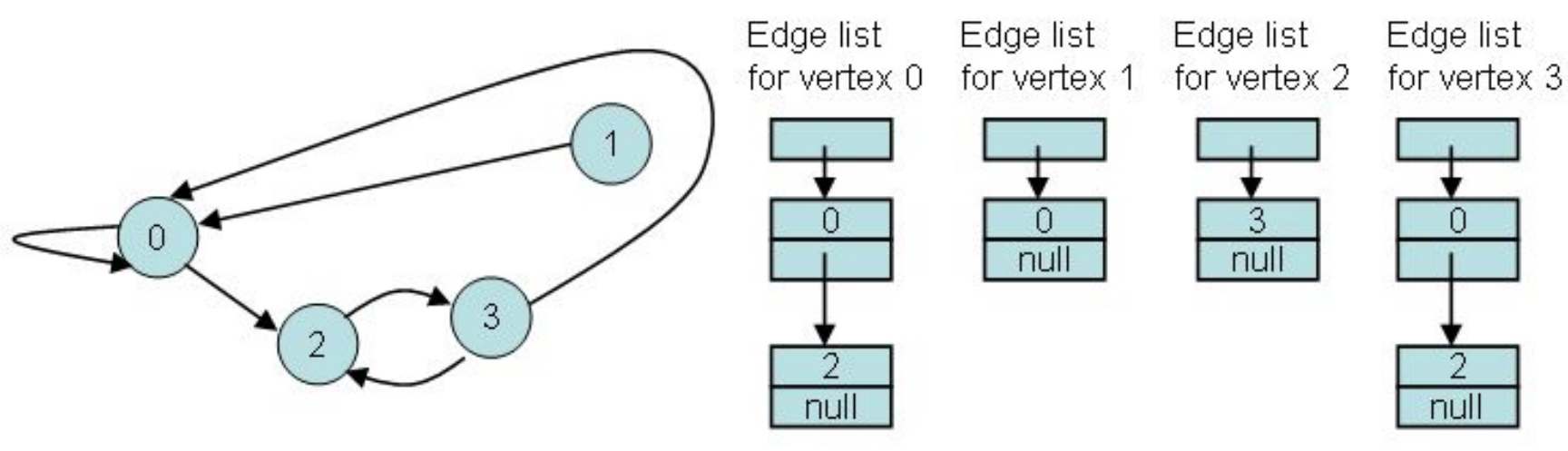
$$
\begin{array}{c}
\\
1 \\
2 \\
3 \\
4 \\
5 \\
6
\end{array}
\begin{pmatrix}
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
$$

Edge Lists

Definition:

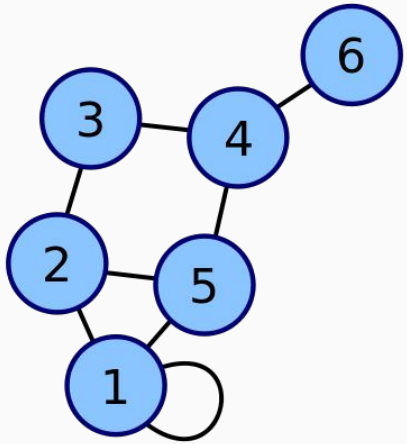
- A directed graph with ***n* vertices** can be represented by ***n* different linked lists**.
- List number ***i*** provides the connections for vertex ***i***.
- For each entry ***j*** in list number ***i***, there is an edge from ***i*** to ***j***.

Edge List Example



ICE 10.5 Edge Lists

Build the Edge Lists for the following Graph:



Adjacency Matrix vs Edge List

- Consider a sparse graph (most vertices are not connected by edges) - **spacial complexity**
  - **Edge List**: space is proportional to the number of edges and vertices in the graph.
  - **Adjacency Matrix:** space is proportional to the number of vertices squared **(n x n matrix)**.
- Consider finding if two vertices are connected? - **lookup**
  - **Edge List:** O(1) * O(d) ….. d = # of adjacent vertices to vertex v
  - **Adjacency matrix:** O(1)

Note: V = # of vertices **E = # of edges (connections)**

Adjacency Matrix vs Edge List

If the space is available, then an adjacency matrix is easier to implement and is generally easier to use than edge lists or edge sets.

There are also other considerations:

1. Adding or removing edges
2. Checking whether a particular edge is present
3. Iterating a loop that executes one time for each edge with a particular source vertex

ICE 10.4 Adjacency Matrix vs Edge List

How would the runtime of the following operations compare between edge lists and adjacency matrices?

Considerations:

1. Adding or removing edges
2. Checking whether a particular edge is present
3. Iterating a loop that executes one time for each edge with a particular source vertex

ICE 10.4 Adjacency Matrix vs Edge List

How would the runtime of the following operations compare between edge lists and adjacency matrices?

Considerations:

1. Adding or removing edges
2. Checking whether a particular edge is present
3. Iterating a loop that executes one time for each edge with a particular source vertex

- Both (1) and (2) require O(1) for adjacency matrix
- Both (1) and (2) require $O(V)$ operations for edge list.
- With (3), edge lists ($O(E)$, E = # of edges that have vertex *i* as their source).
- With (3), require O(V) for adjacency matrix ($O(n)$).

Graph Traversals

There are two common ways of traversing a graph:

- **Depth-first search** uses a stack … .
- **Breadth-first search** uses a queue to keep track of vertices that still need to be visited.

Graph Traversals Contd.

Goal:

1. Start at one vertex of a graph (the "**start" vertex**) and process the information contained at that vertex.
2. Move along an edge to process a neighbor.
3. When the traversal finishes, all of the vertices that can be reached from the **start vertex** have been processed.

Note:

- A traversal processes only those vertices that can be **reached from the start vertex**.
- The algorithm must **not** enter a repetitive cycle. To prevent this, the algorithm needs to mark each vertex as it is processed.

Depth First Search (DFS)

Step 1: Initialize a stack with the start vertex

Step 2: Visit the adjacent unvisited vertex.

a. Mark the vertex as visited.
b. Log the vertex.
c. Push it in a stack.
d. Goto Step 2 until no adjacent vertex is found

Step 3 - If no adjacent vertex found

e. Pop the current vertex off the stack
f. If stack == empty

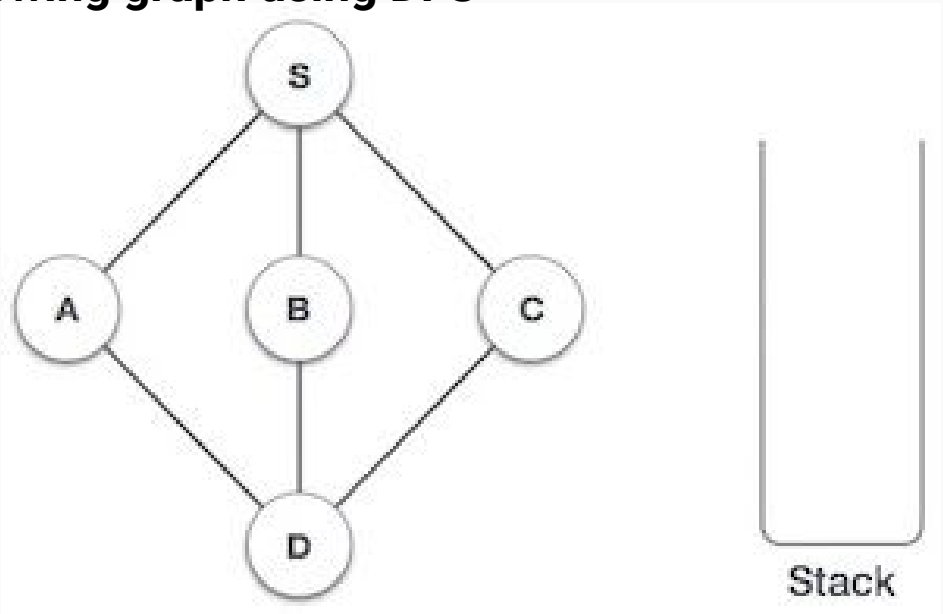
DONE

Else

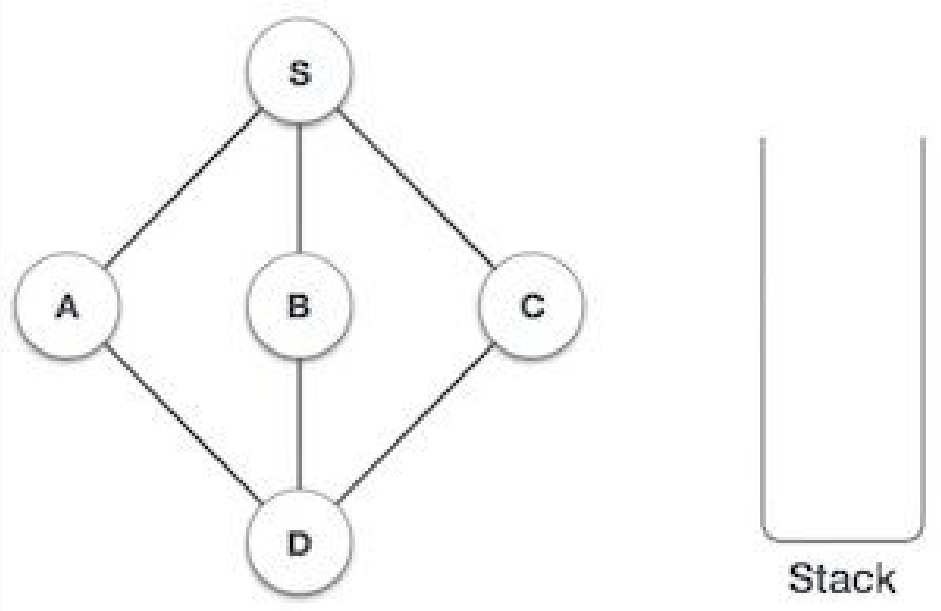
Goto Step 2

ICE 10.5 DFS

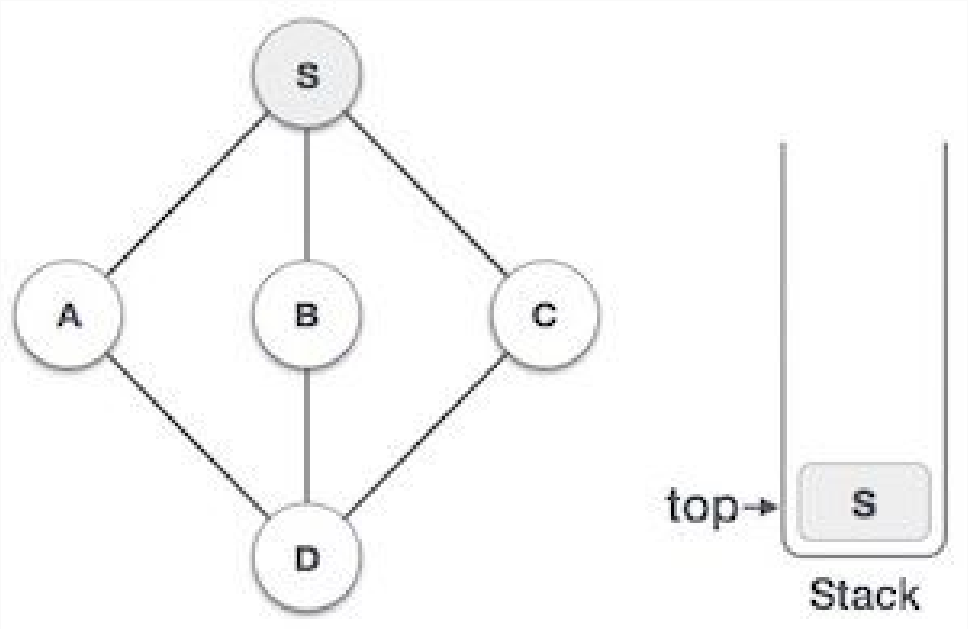
Traverse the following graph using DFS



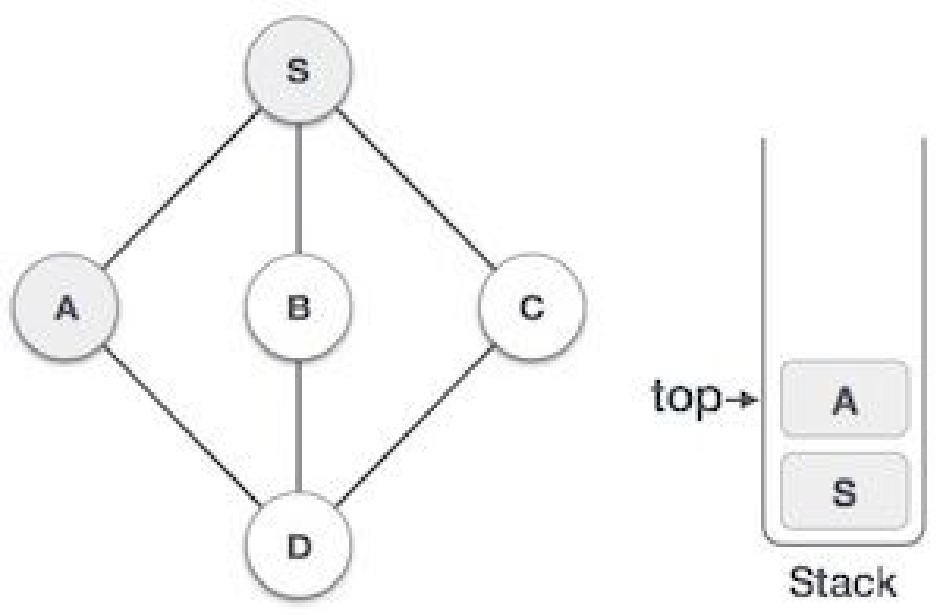
ICE 10.5 DFS



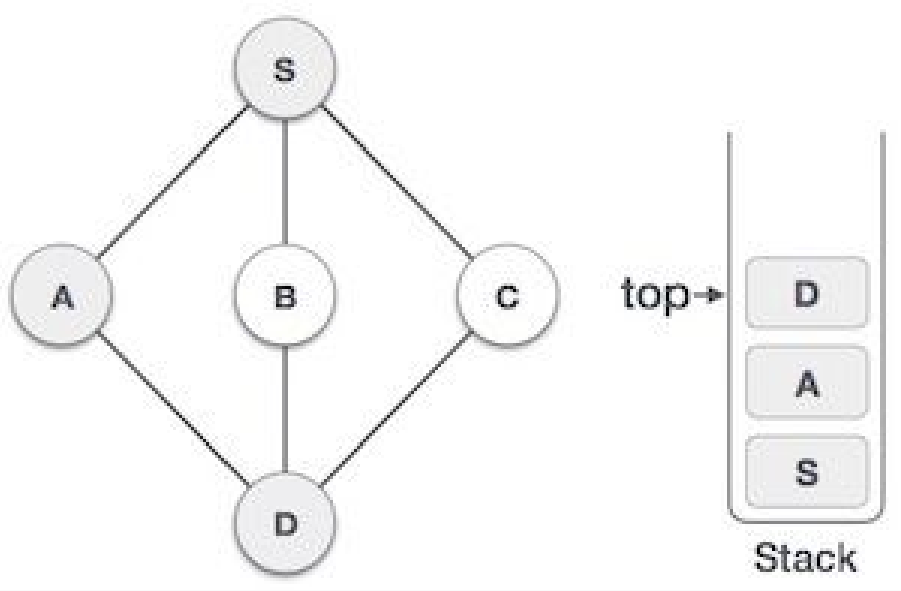
DFS Contd.



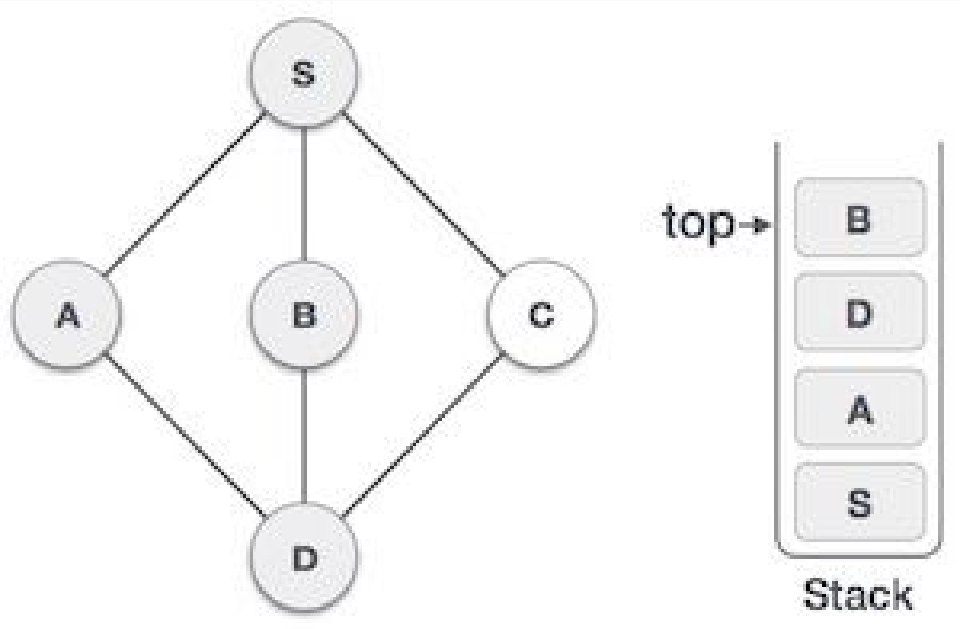
DFS Contd.



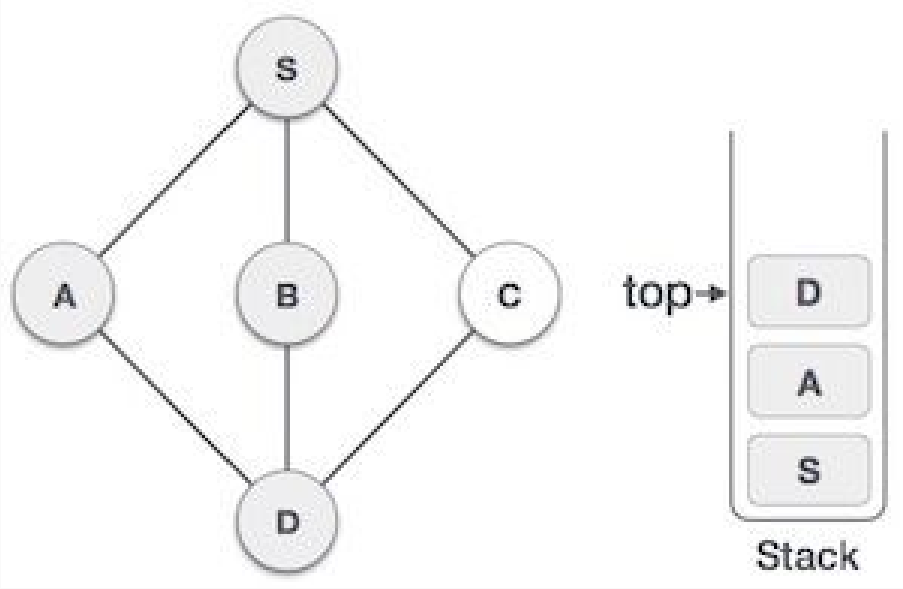
DFS Contd.



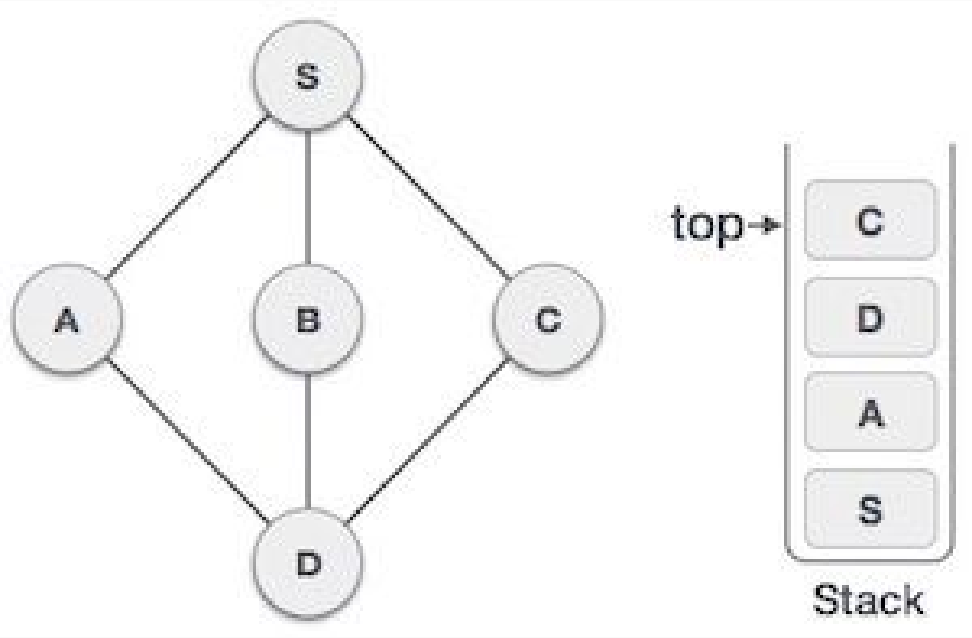
DFS Contd.



DFS Contd.



DFS Contd.



Breadth First Search (DFS)

Step 1: Initialize a queue with the start vertex

Step 2: Visit the adjacent unvisited vertex.

a. Mark the vertex as visited.
b. Log the vertex.
c. Add the vertex to the queue (insert at the end).
d. Goto Step 2 until no adjacent vertex is found (for the target vertex)

Step 3 - If no adjacent vertex found

e. Remove the first vertex from the queue
f. If queue == empty

DONE

Else

Goto Step 2

References

https://www.cpp.edu/~ftang/courses/CS241/notes/graph.htm

https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs

http://www.geeksforgeeks.org/applications-of-depth-first-search/

https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

References

http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/

http://www.geeksforgeeks.org/applications-of-breadth-first-traversal/

http://www.slideshare.net/Krish_ver2/21-graph-basic