Graph

Graph

Topics

- Graph Concepts
- Graph Terminology
- Graph Representation
- Depth First Search

What is a Graph?

- Non-linear data structure

A Graph is a data structure which consists of a set of ***vertices***, and a set of ***edges*** that connect (some of) them.

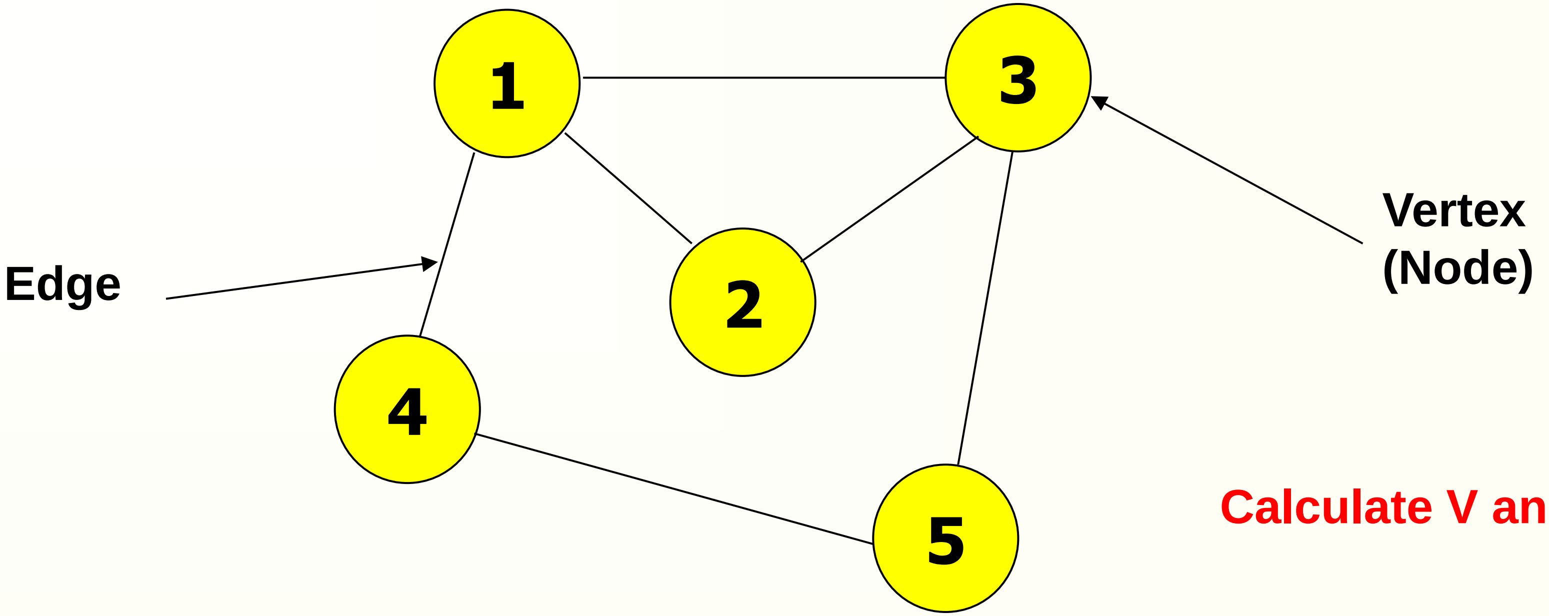
G = (V, E)

where,

V - set of vertices

E - set of edges

What is a Graph?



Calculate V and E

V = {1, 2, 3, 4, 5}

E = { (1,2), (1,3), (1,4), (2,3), (3,5), (4,5) }

A “Real-life” Example of a Graph

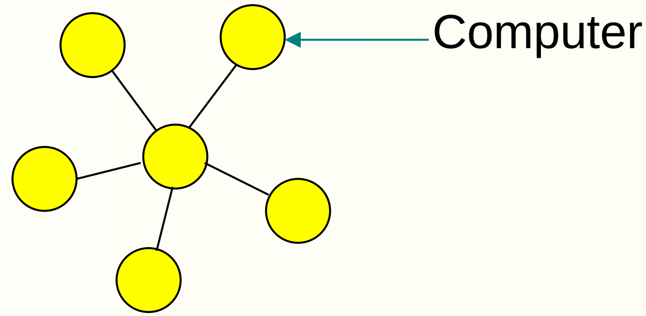
- V=set of 6 people:

  John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.

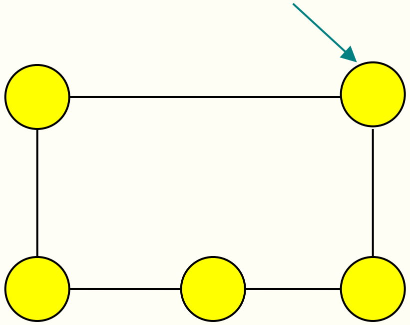
- E = {(x,y) | if x is younger than y}

Applications

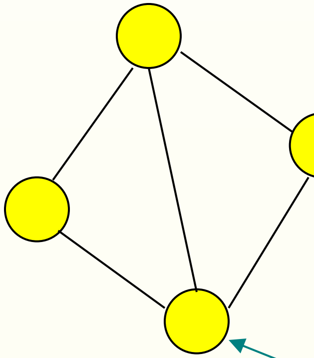
- Computer Networks



- Electrical Circuits



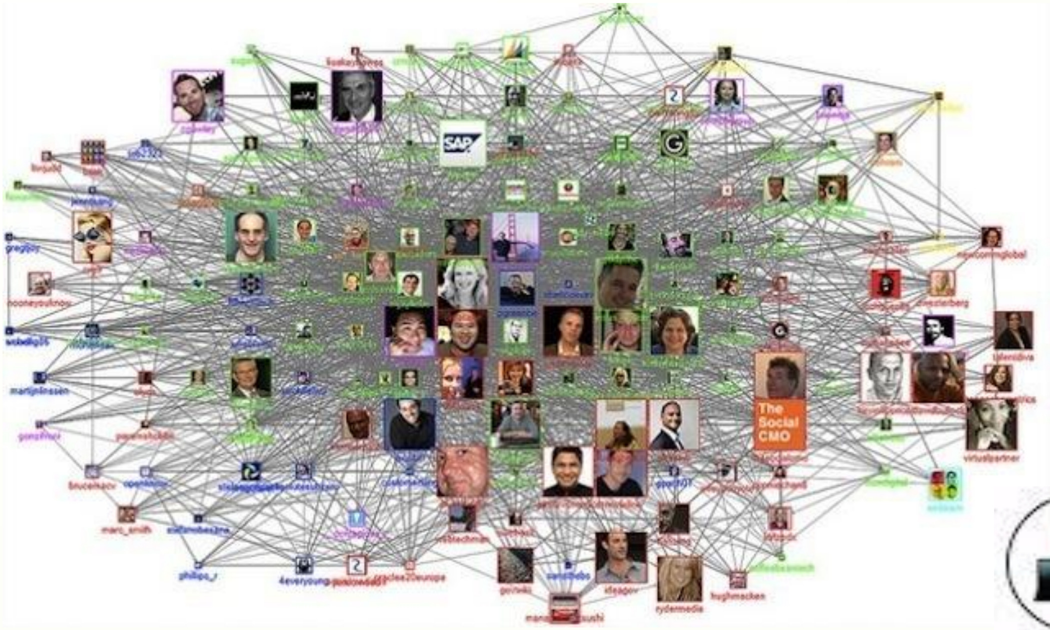
- Road Map

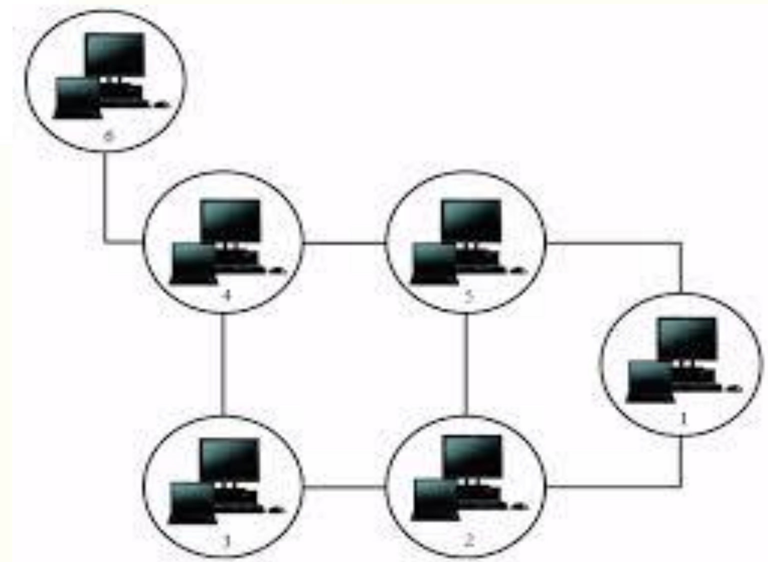


Applications

Applications

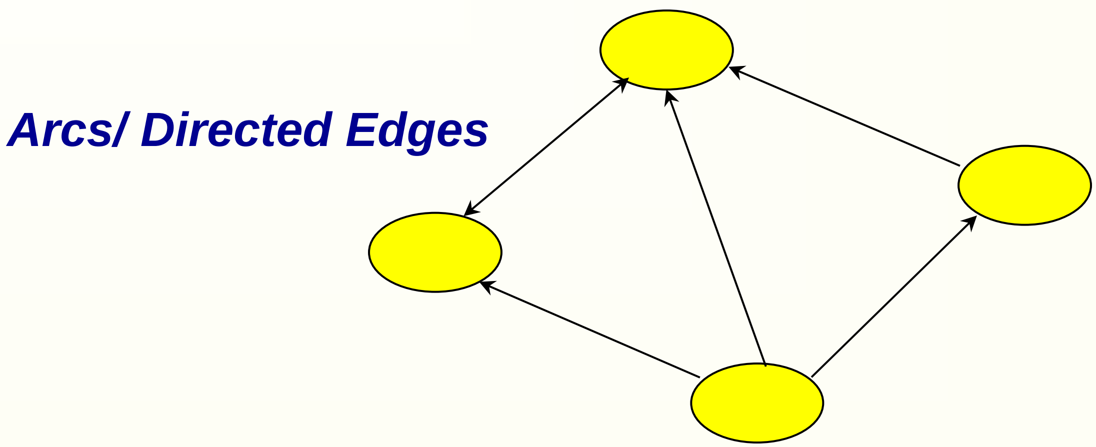
Graph





Graph Categorization: Digraph

- A *Directed Graph* or *Digraph* is a graph where each edge has a direction



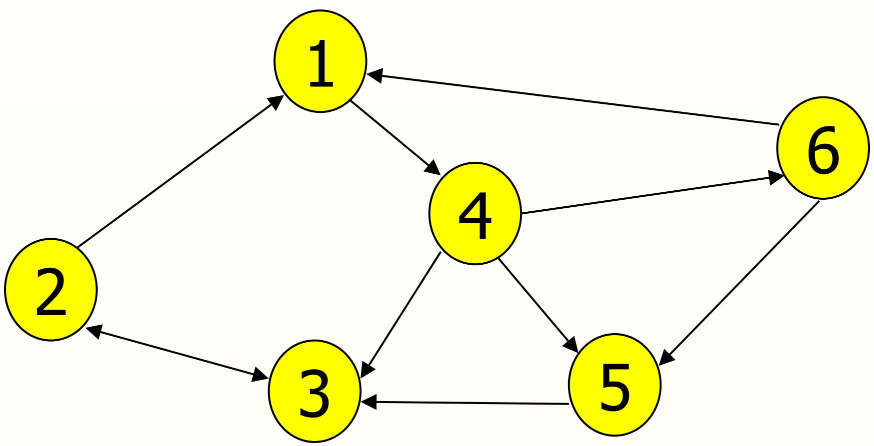
Graph Categorization: Digraph

- Digraph - Example

G = (V, E)

V = {1, 2, 3, 4, 5, 6}

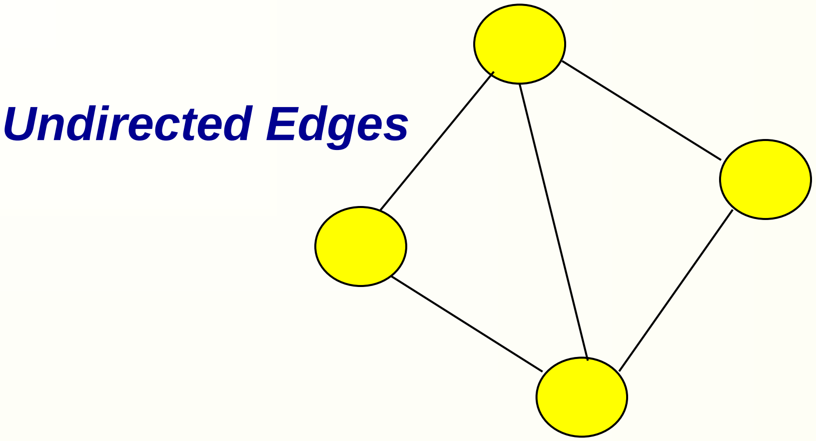
E = {(1,4), (2,1), (2,3), (3,2), (4,3),
(4,5), (4,6), (5,3), (6,1), (6,5)}



(1, 4) = 1→4 where 1 is the *tail*
and 4 is the *head*

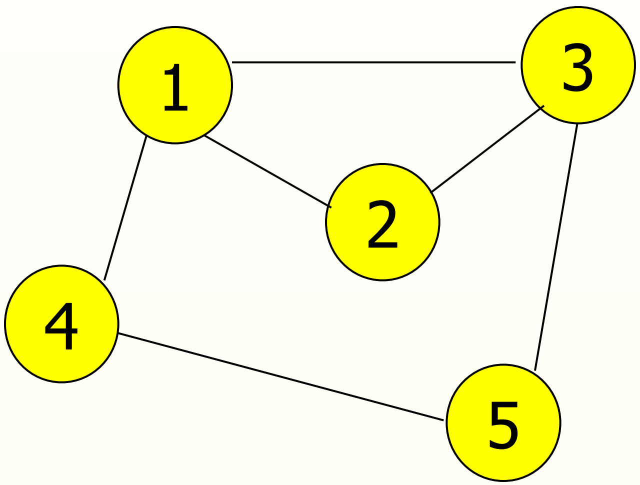
Graph Categorization: Undirected Graph

- An *Undirected Graph* is a graph where the edges have no directions



Graph Categorization

- Undirected Graph - Example



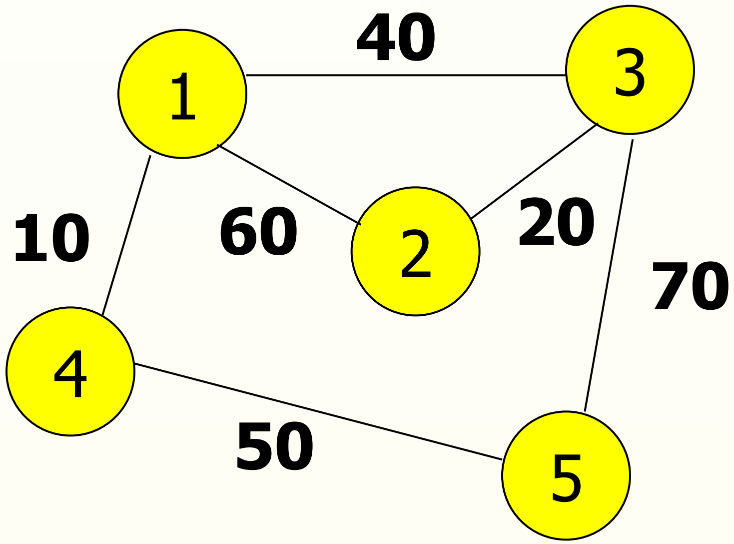
G = (V, E)

V = {1, 2, 3, 4, 5}

E = {(1,2), (1,3), (1,4), (2,3), (3,5), (4,5)}

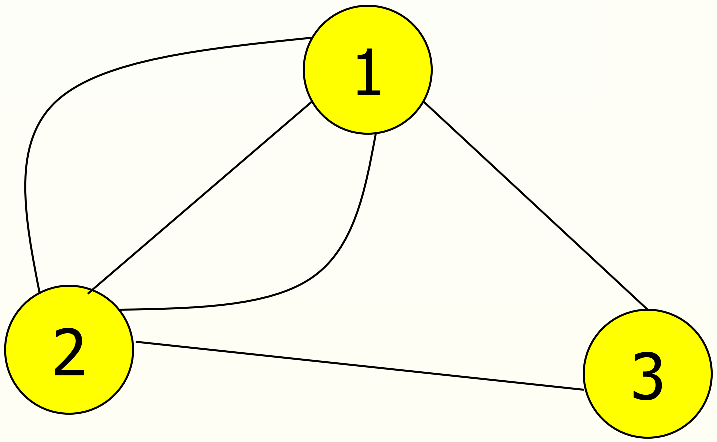
Graph Categorization: Weighted Graph

- A *Weighted Graph* is a graph where all the edges are assigned weights



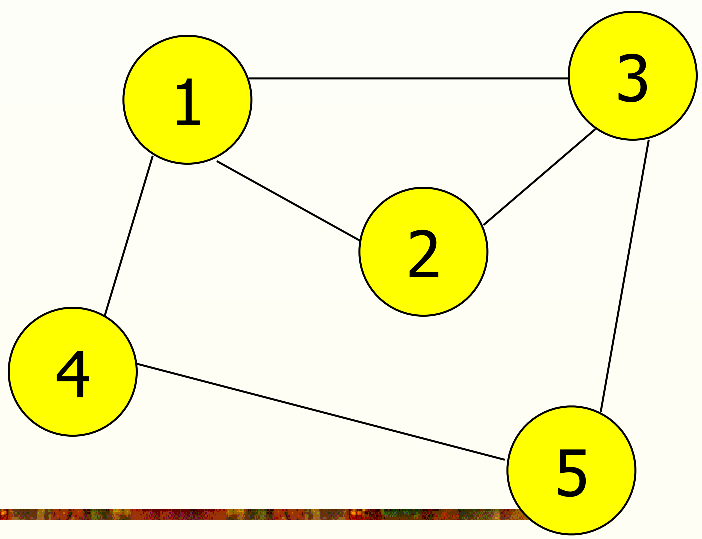
Graph Categorization: Multigraph

- If the same pair of vertices have more than one edge, that graph is called a *Multigraph*



Graph Terminology: Adjacent Vertices

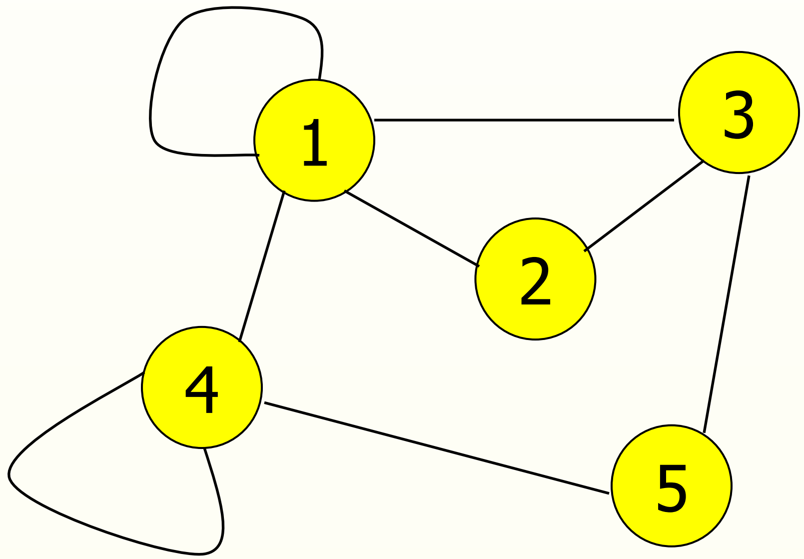
- *Adjacent vertices:* If (i,j) is an edge of the graph, then the nodes i and j are adjacent



Vertices 2 and 5 are *not* adjacent

Graph Terminology: Loop

- *Loop or Self edges:* An edge (i,i) is called a self edge or a loop
- In graphs loops are ***not*** permitted



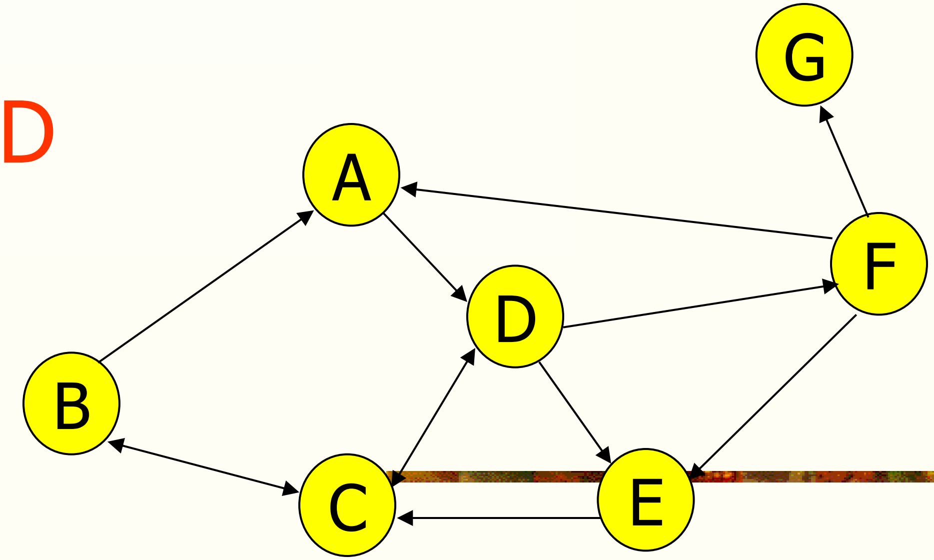
(1,1) and (4,4) are self edges

Graph Terminology: Path

- *Path:* A sequence of edges in the graph
- There can be more than one path between two vertices
- Vertex A is *reachable* from B if there is a path from A to B

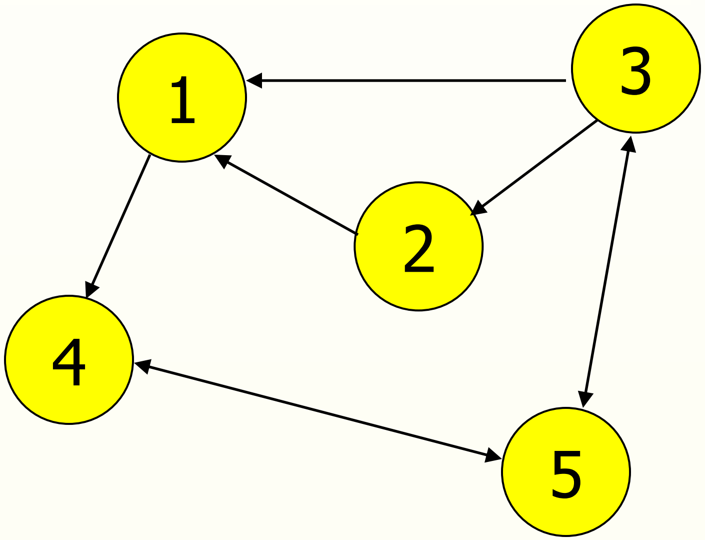
Paths from B to D

- B, A, D
- B, C, D



Graph Terminology: Simple Path

- *Simple Path:* A path where all the vertices are distinct

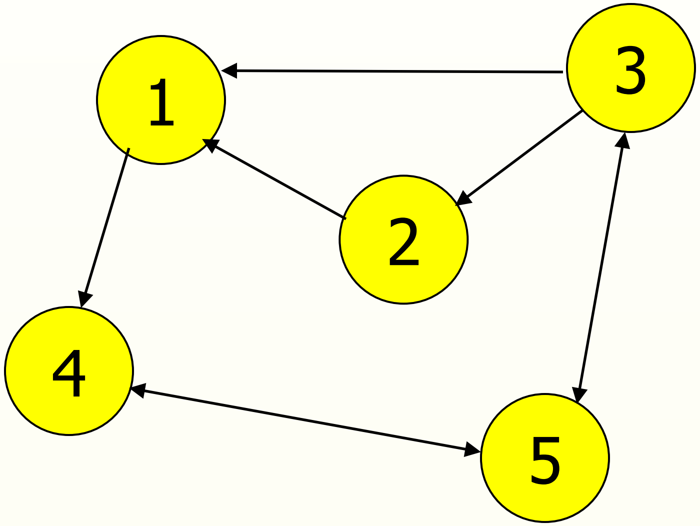


1,4,5,3 is a simple path.

But 1,4,5,4 is not a simple path.

Graph Terminology: Length

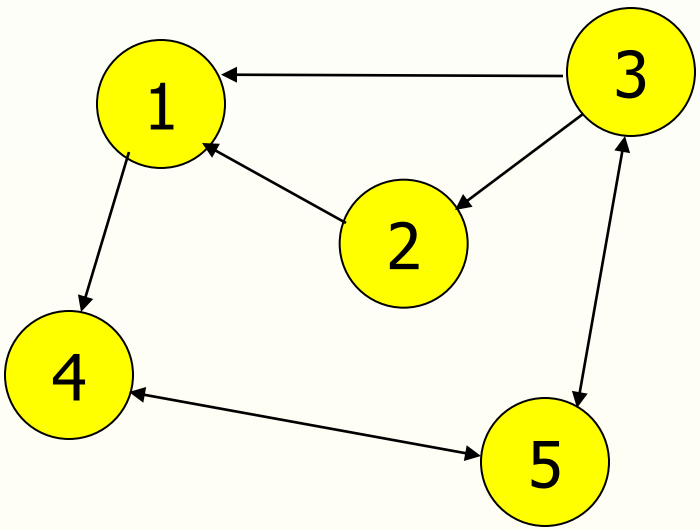
- *Length :* Sum of the lengths of the edges on the path.



Length of the path 1,4,5,3 is 3

Graph Terminology: Circuit

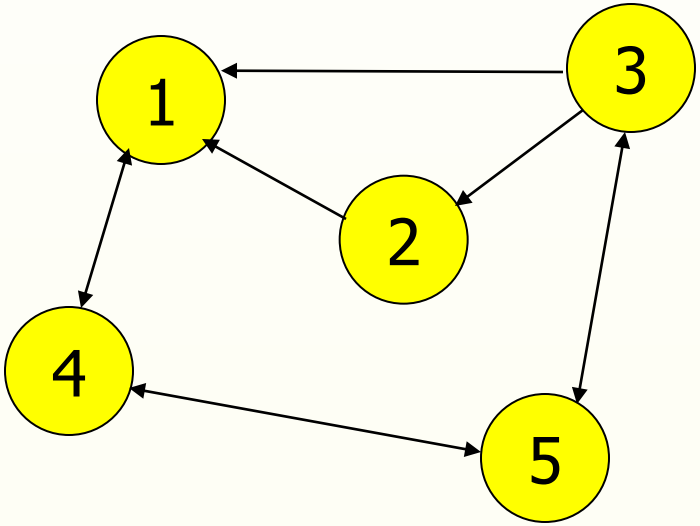
- *Circuit:* A path whose first and last vertices are the same



The path 3,2,1,4,5,3 is a circuit

Graph Terminology: Cycle

- *Cycle*: A circuit where all the vertices are distinct except for the first (and the last) vertex

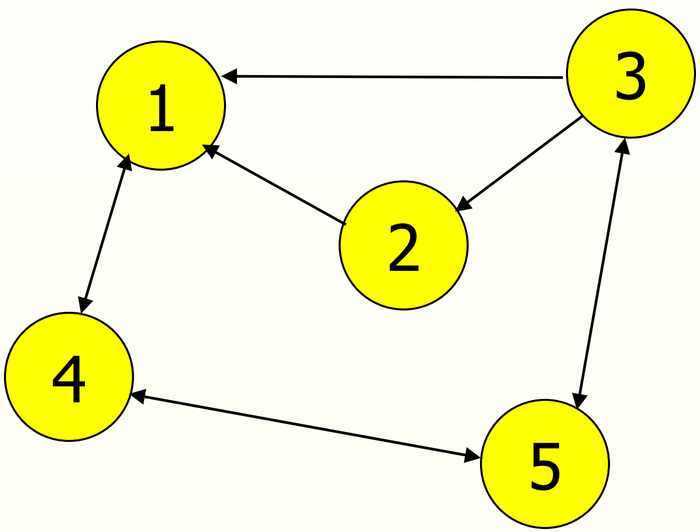


1,4,5,3,1 is a cycle

1,**4**,5,**4**,1 is not a cycle

Graph Terminology: Hamiltonian Cycle

- *Hamiltonian Cycle*: A Cycle that contains all the vertices of the graph

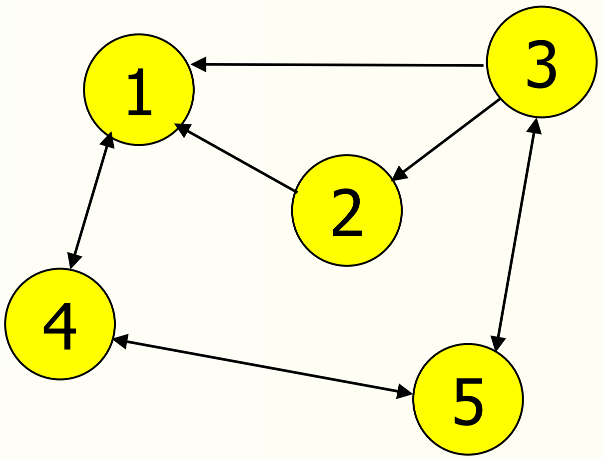


1,4,5,3,2,1 is a Hamiltonian Cycle

Graph Terminology: Degree

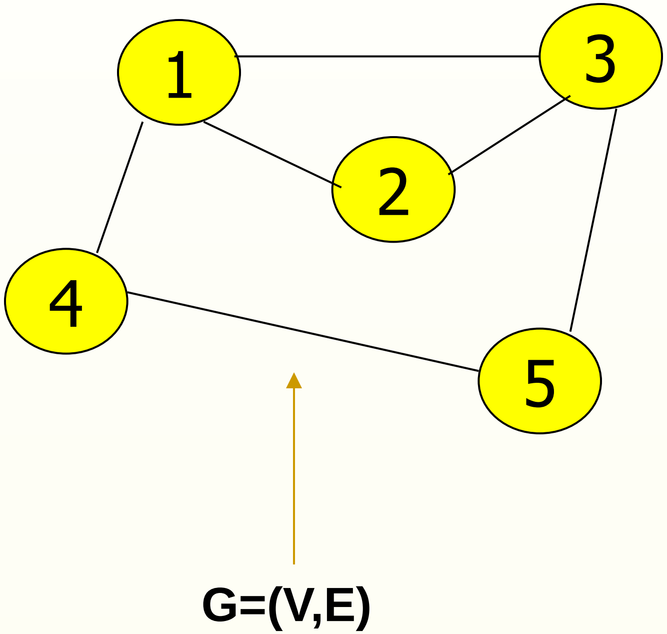
- *Degree of a Vertex :* In an directed graph, the no. of edges incident to the vertex
- *In-degree: indeg(N)*
- *Out-Degree: outdeg(N)*

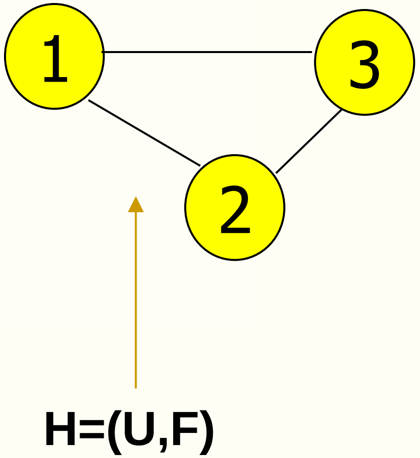
Calculate indeg(1) and outdeg(5)



Graph Terminology: Subgraph

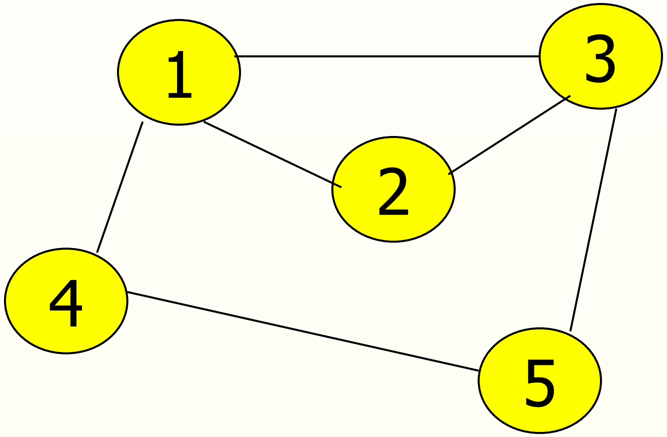
- A *Subgraph* of graph G=(V,E) is a graph H=(U,F) such that U Є V and F Є E



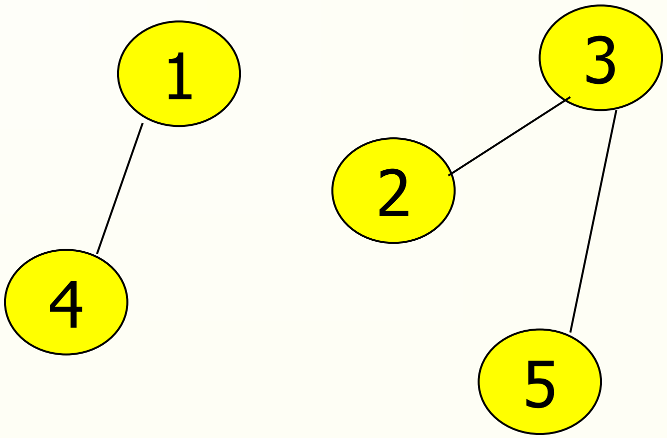


Graph Terminology

- A graph is said to be *Connected* if there is at least one path from every vertex to every other vertex in the graph



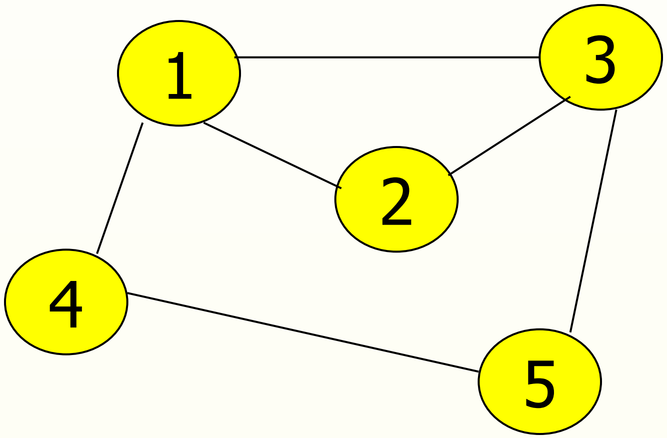
Connected



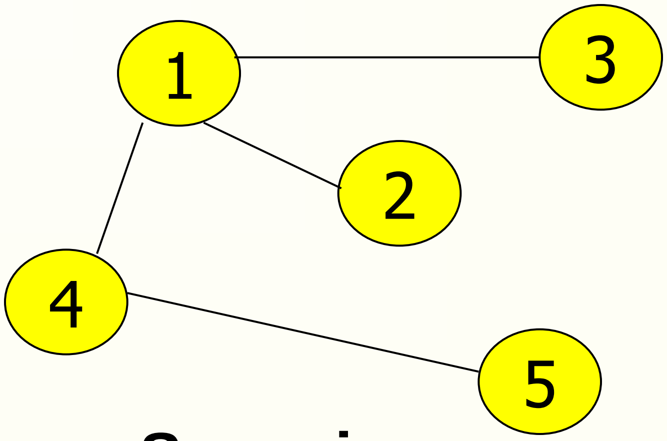
Unconnected

Graph Terminology

- The *Spanning Tree* of a Graph G is a subgraph of G that is a tree and contains all the vertices of G



Graph



Spanning Tree

Graph Representation

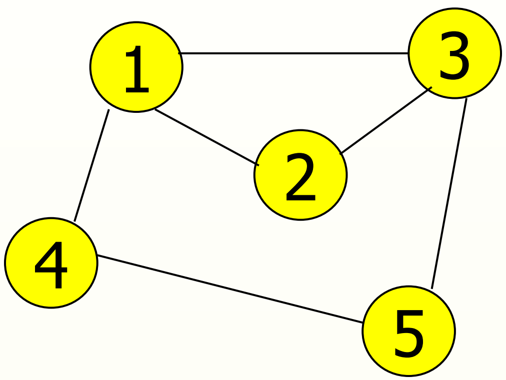
Representation of Graphs

- *Adjacency Matrix (A)*
  - The Adjacency Matrix $A=(a_{i,j})$ of a graph G=(V,E) with *n nodes* is an *nXn* matrix

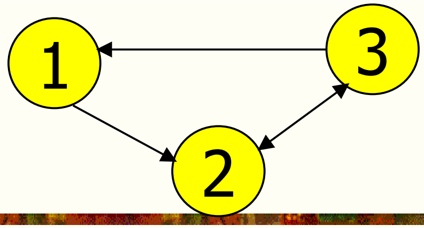
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Representation of Graphs

- *Eg:* Find the adjacency matrices of the following graphs



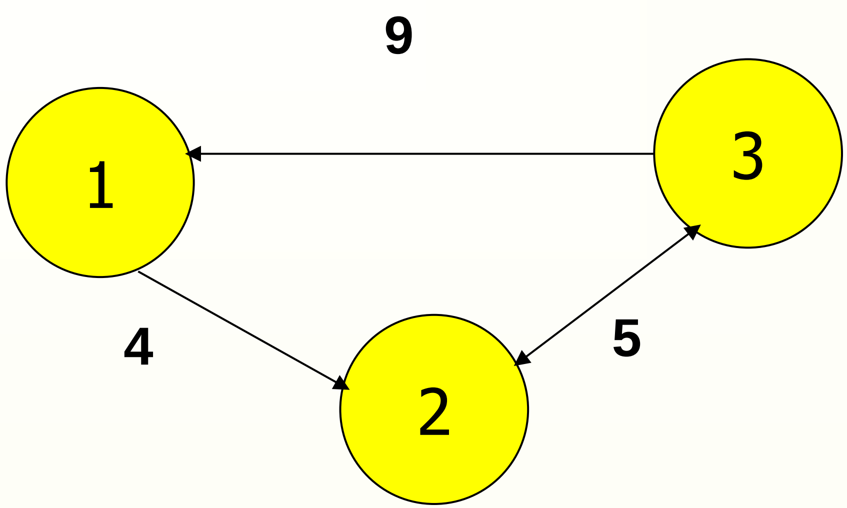
0	1	1	1	0
1	0	1	0	0
1	1	0	0	1
1	0	0	0	1
0	0	1	1	0



0	1	0
0	0	1
1	1	0

Representation of Graphs

- *Adjacency Matrix* of a *Weighted Graph*



INF	4	INF
INF	INF	5
9	5	INF

Pros and Cons of Adjacency Matrices

- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

Example of Linked Representation

L[0]: empty

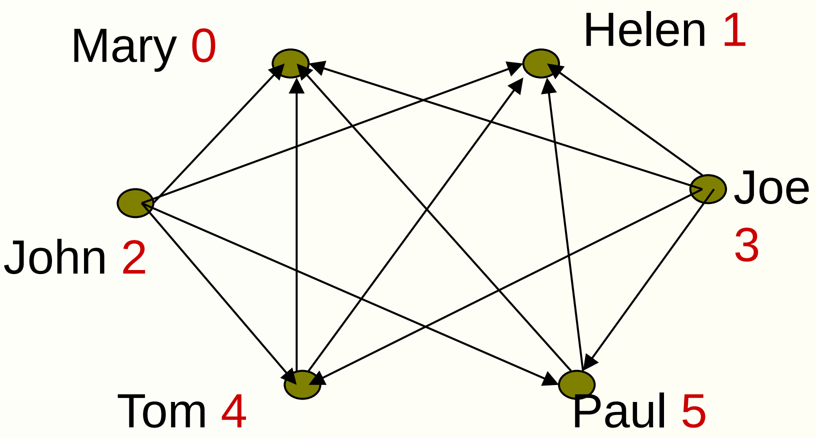
L[1]: empty

L[2]: 0, 1, 4, 5

L[3]: 0, 1, 4, 5

L[4]: 0, 1

L[5]: 0, 1



Pros and Cons of Adjacency Lists

- Pros:
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.
- Cons:
  - It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

Example of Representations

Linked Lists:

L[0]: 1, 2, 3

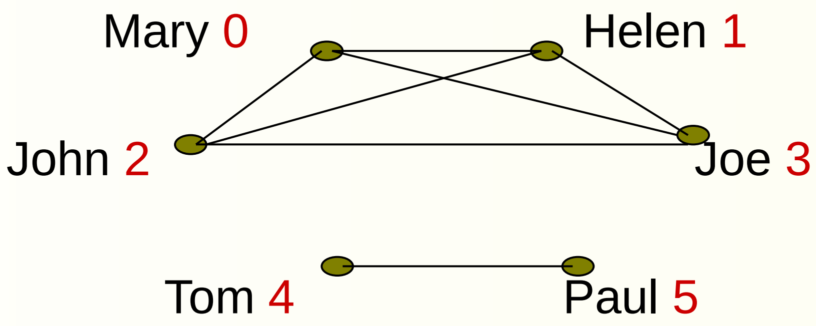
L[1]: 0, 2, 3

L[2]: 0, 1, 3

L[3]: 0, 1, 2

L[4]: 5

L[5]: 4



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Graph Traversal

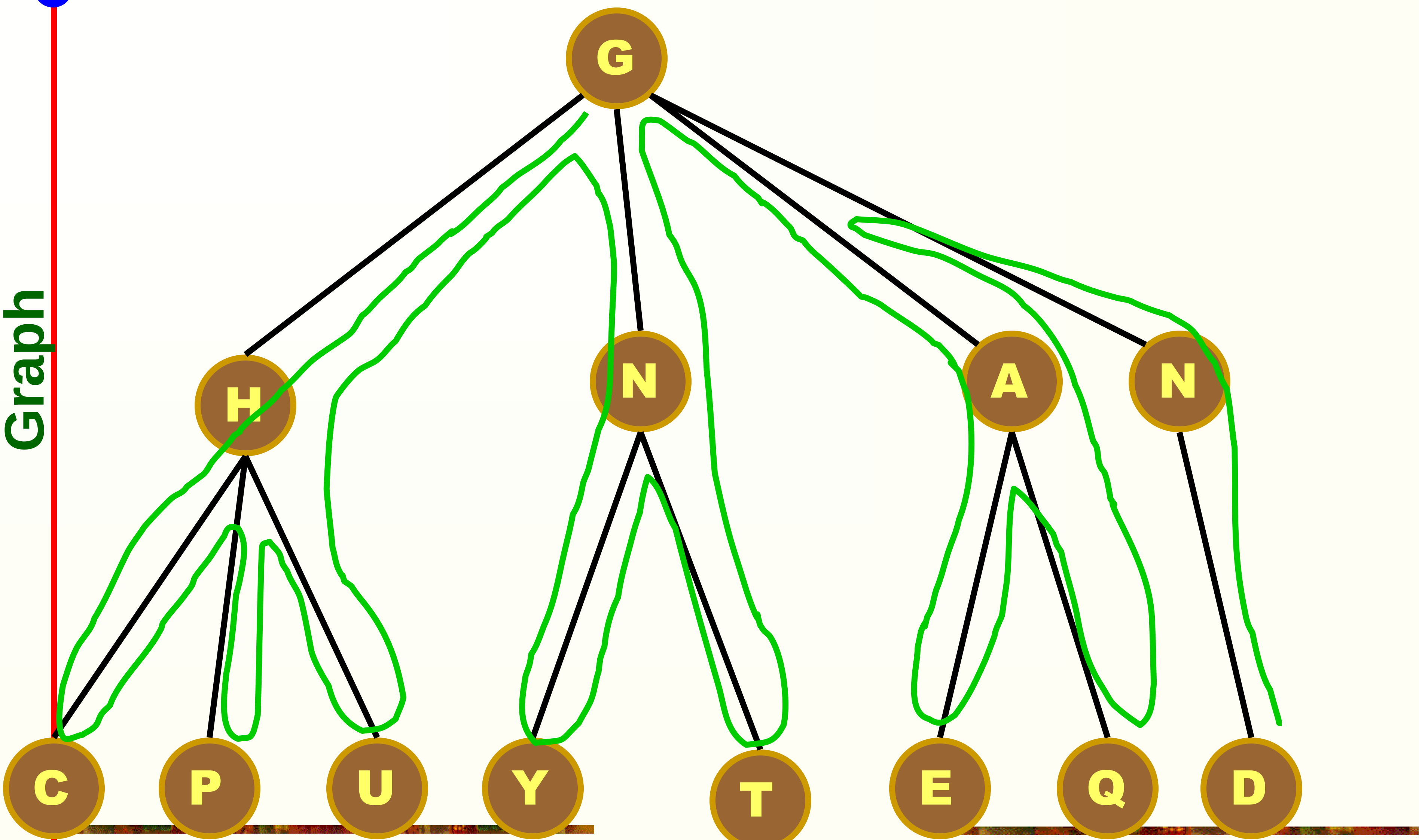
Searching Graphs

- Why do we need to search graphs
  - To find paths
  - To look for connectivity

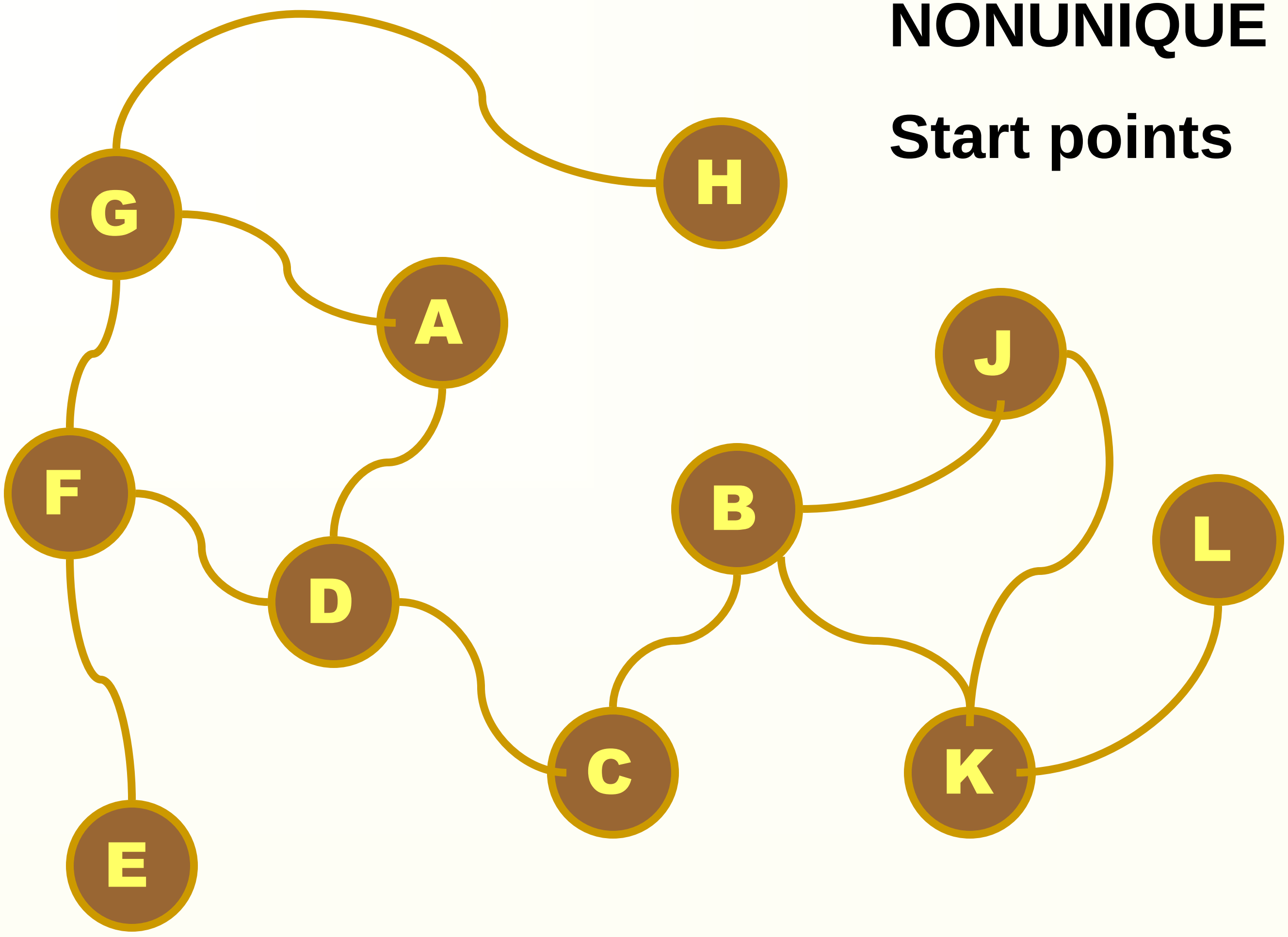
- Two Strategies
  - Depth-First Search (DFS) – Use **STACK**
  - Breadth-First Search (BFS) – Use **QUEUE**

Tree - Depth First Traversal

Graph



Traversal - Depth First

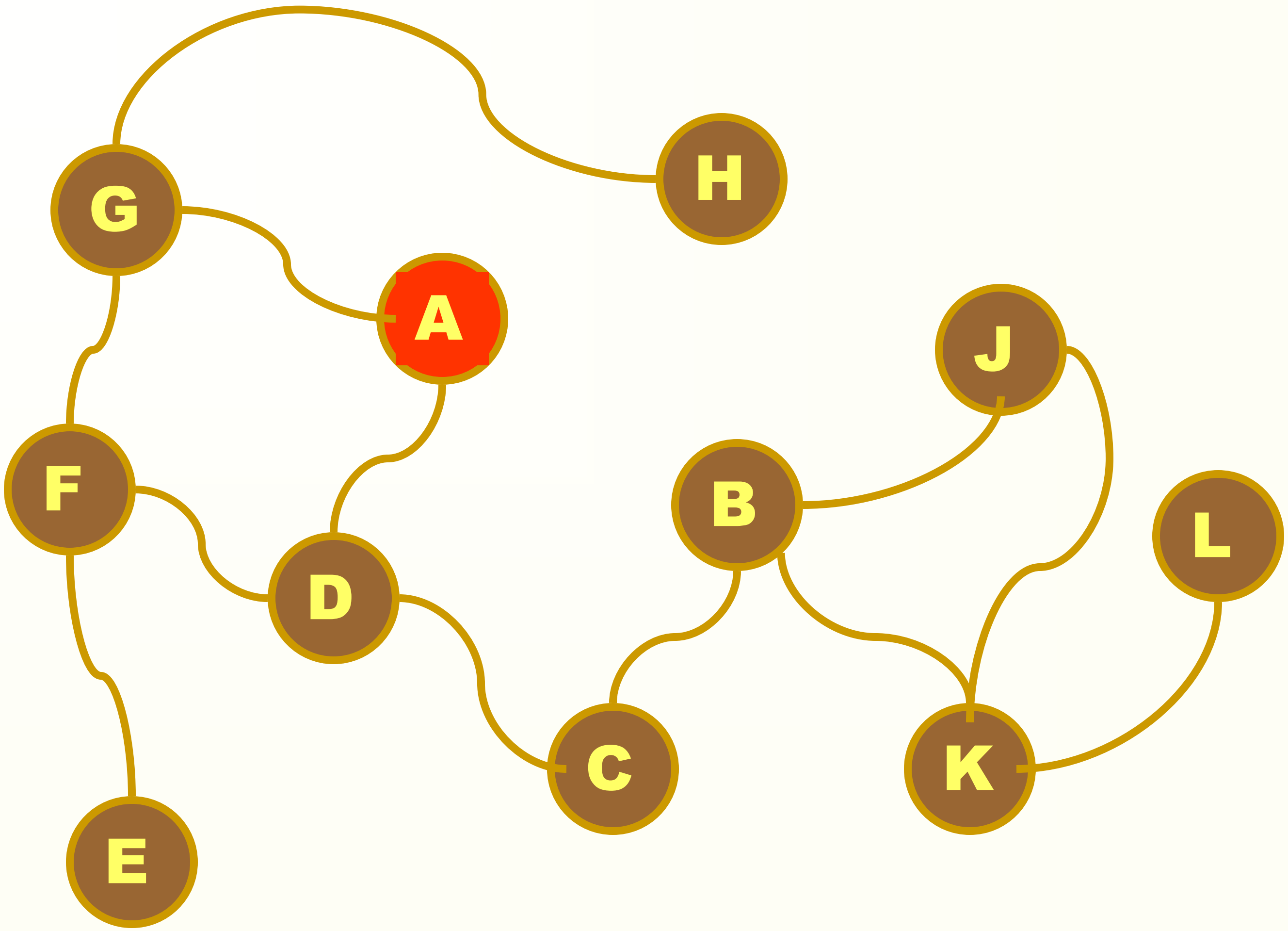


NONUNIQUE

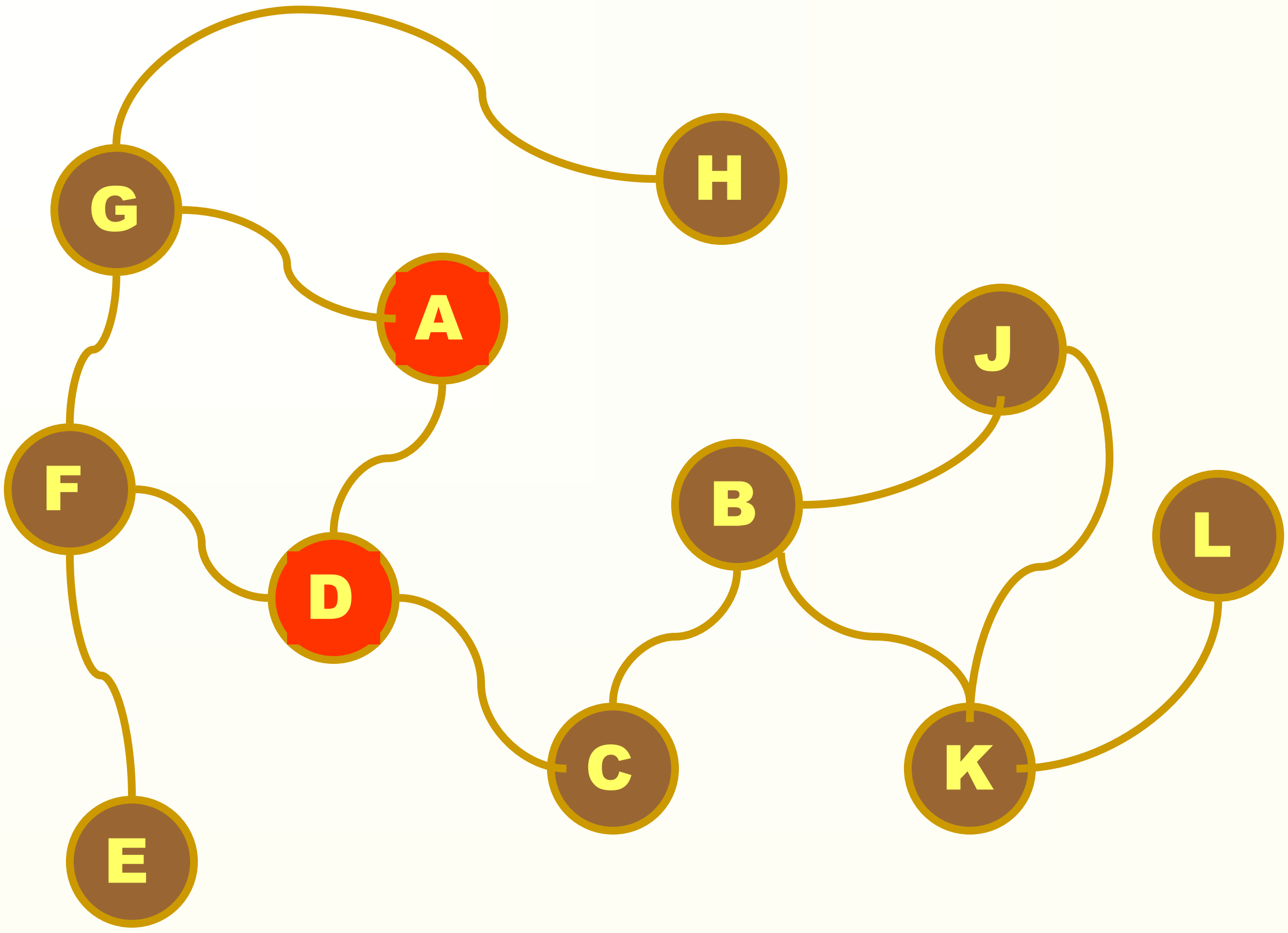
Start points

Traversal - Depth First

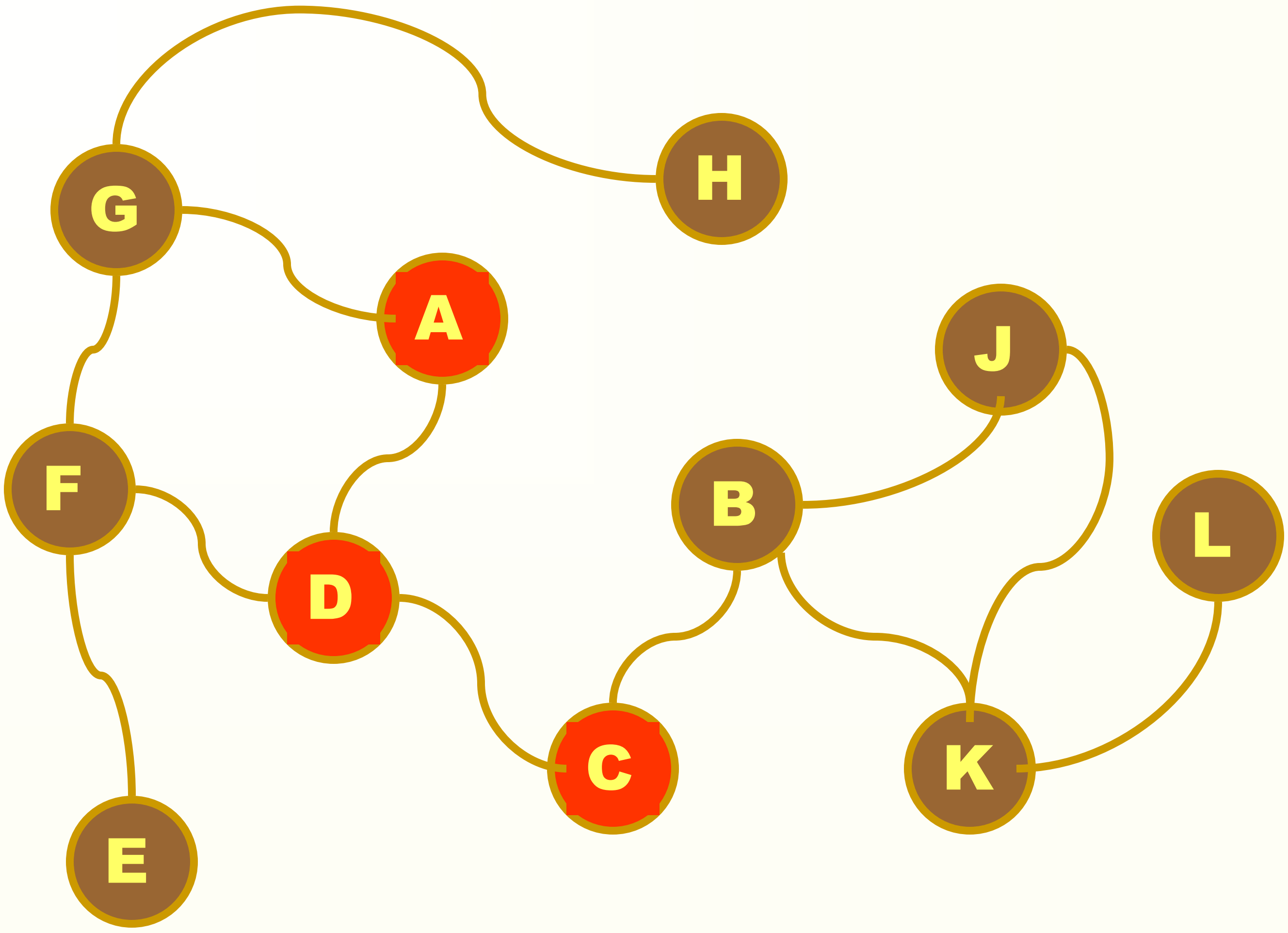
Graph



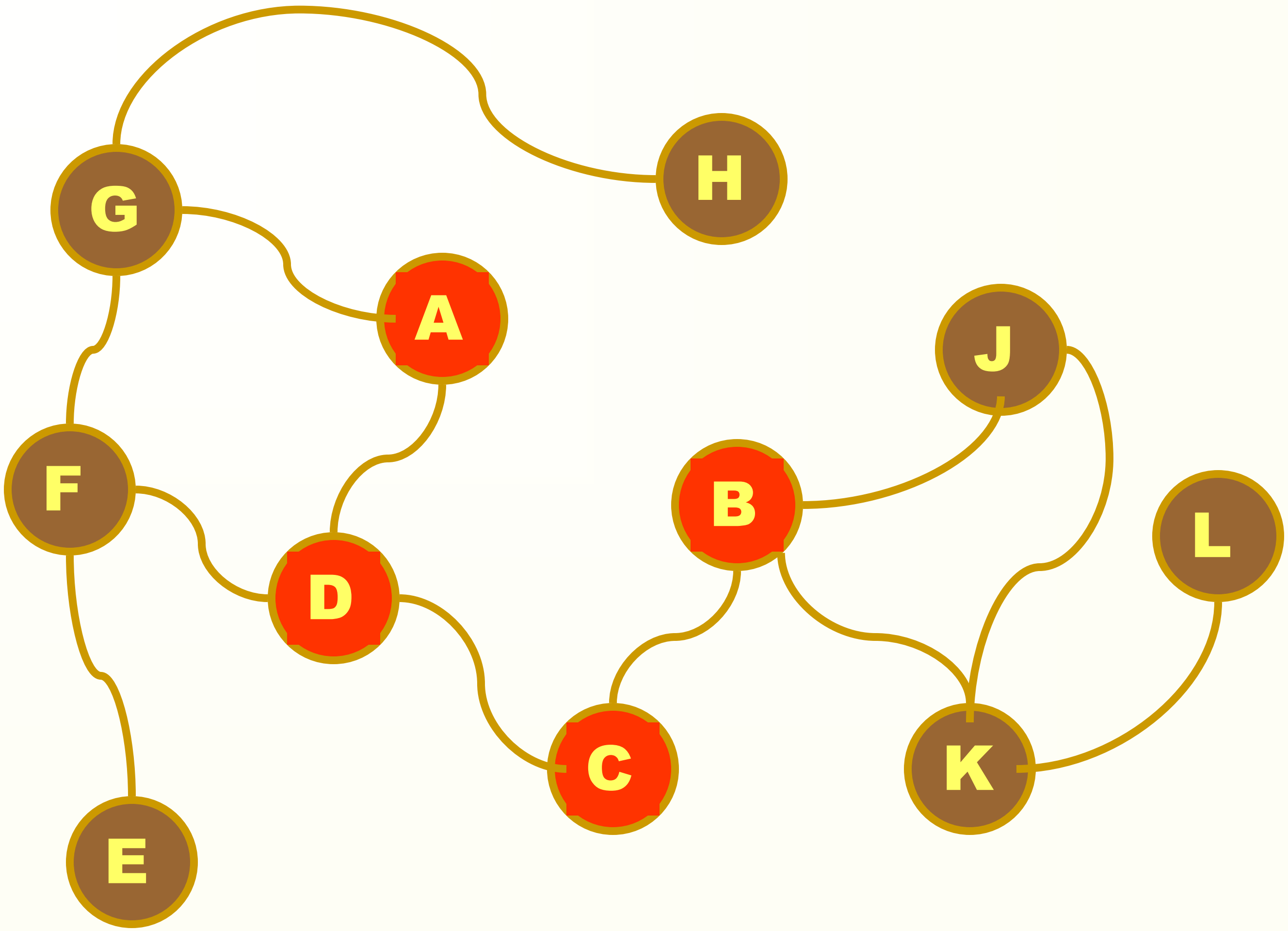
Traversal - Depth First



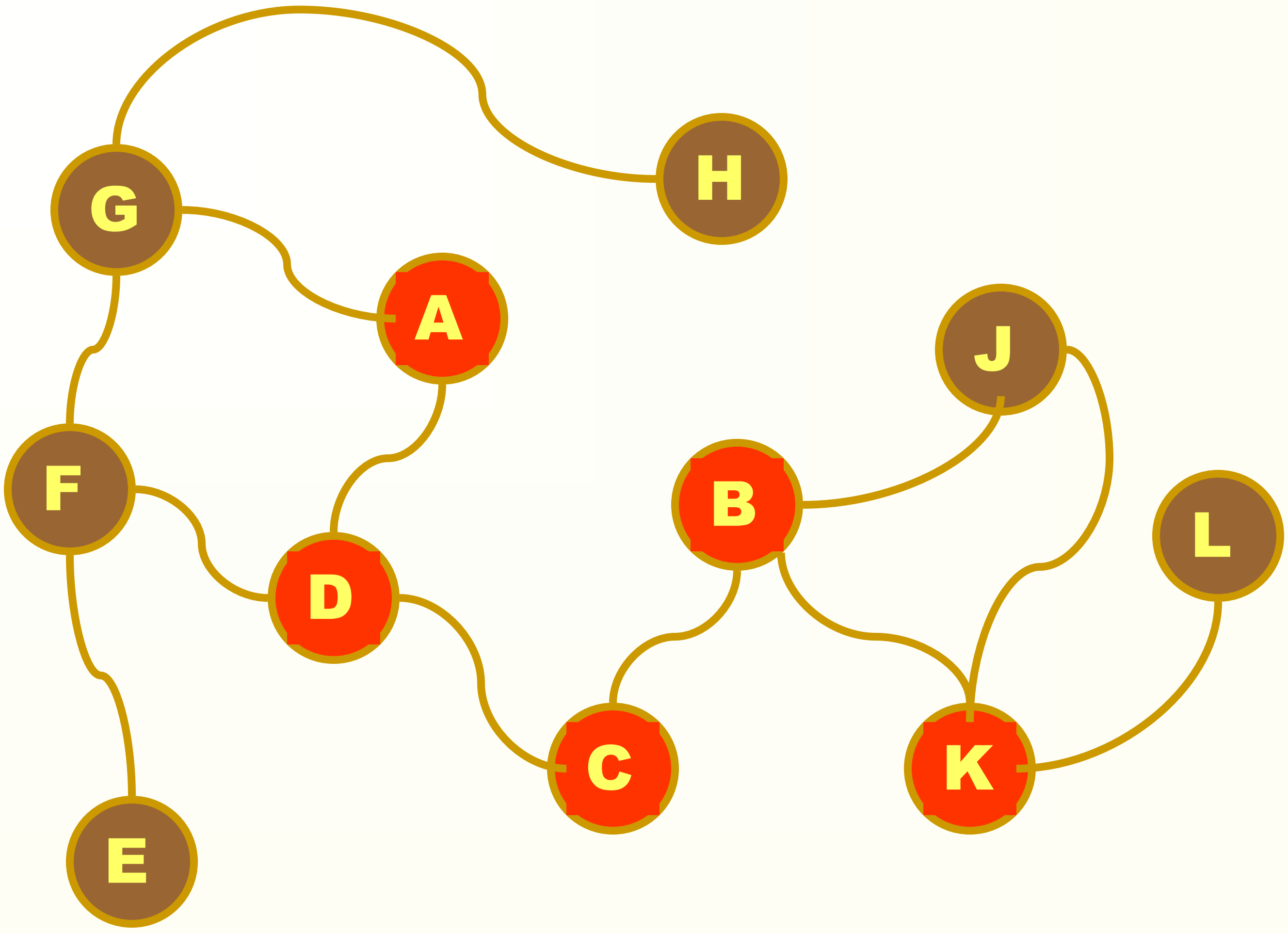
Traversal - Depth First



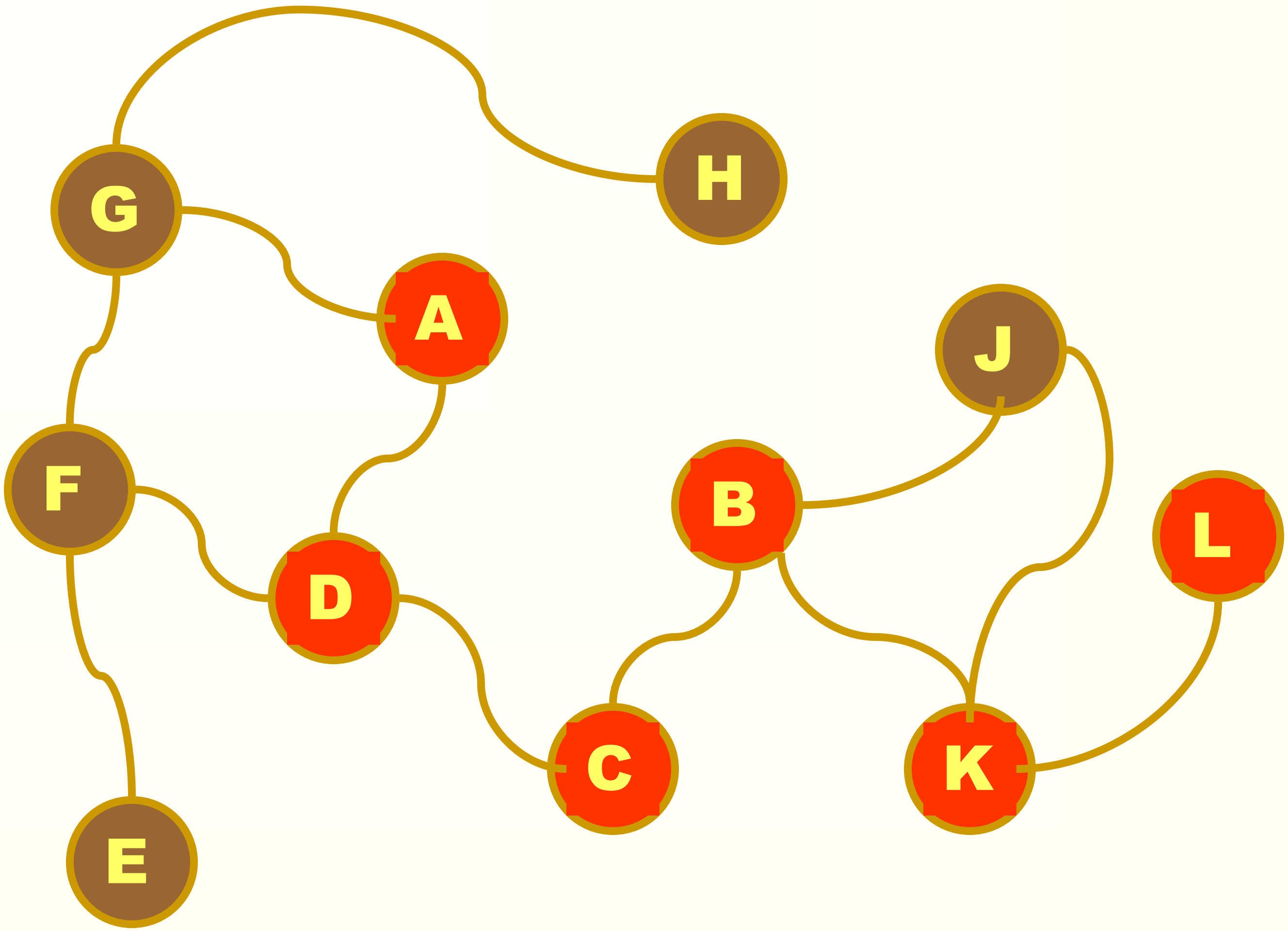
Traversal - Depth First



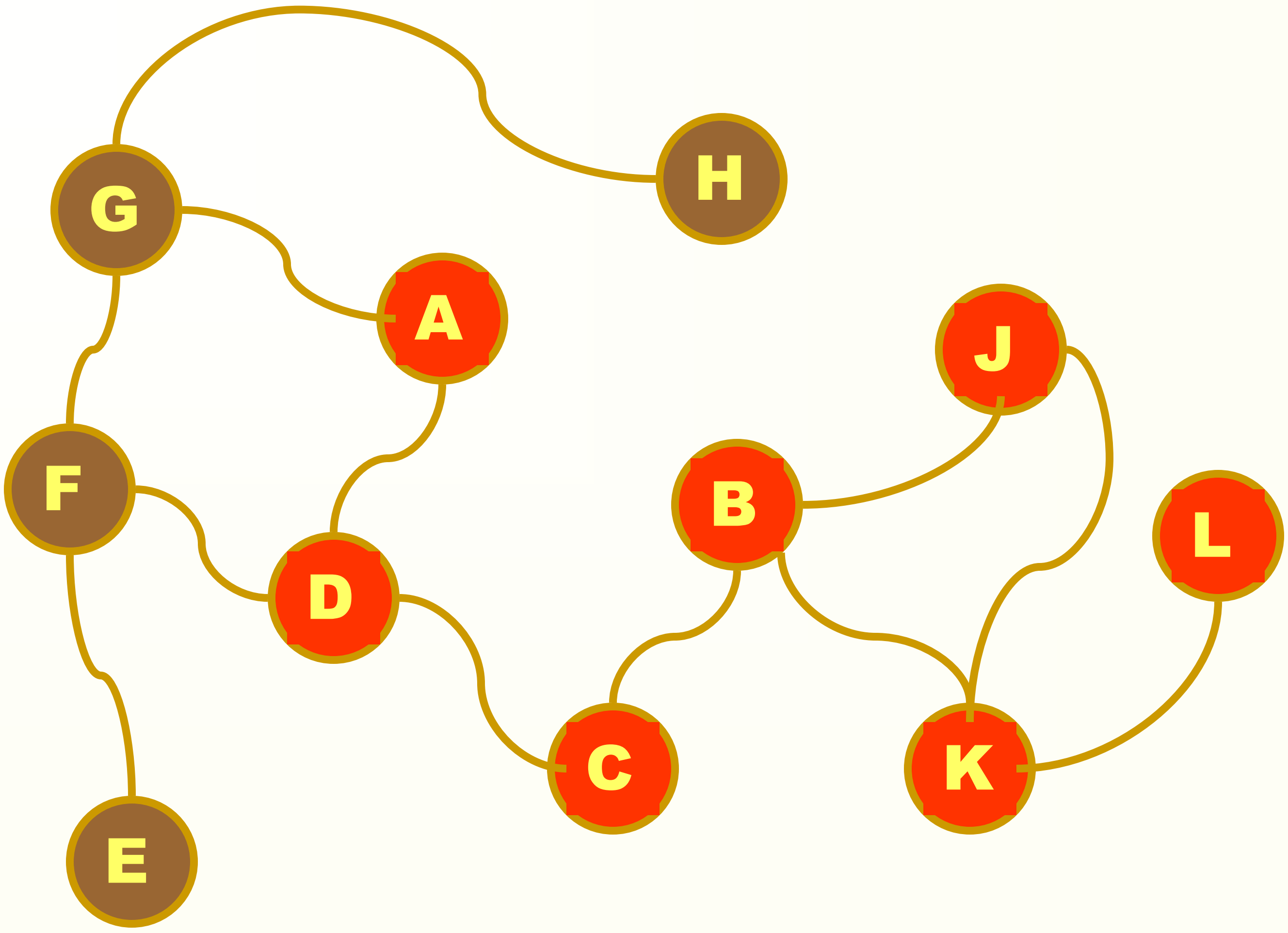
Traversal - Depth First



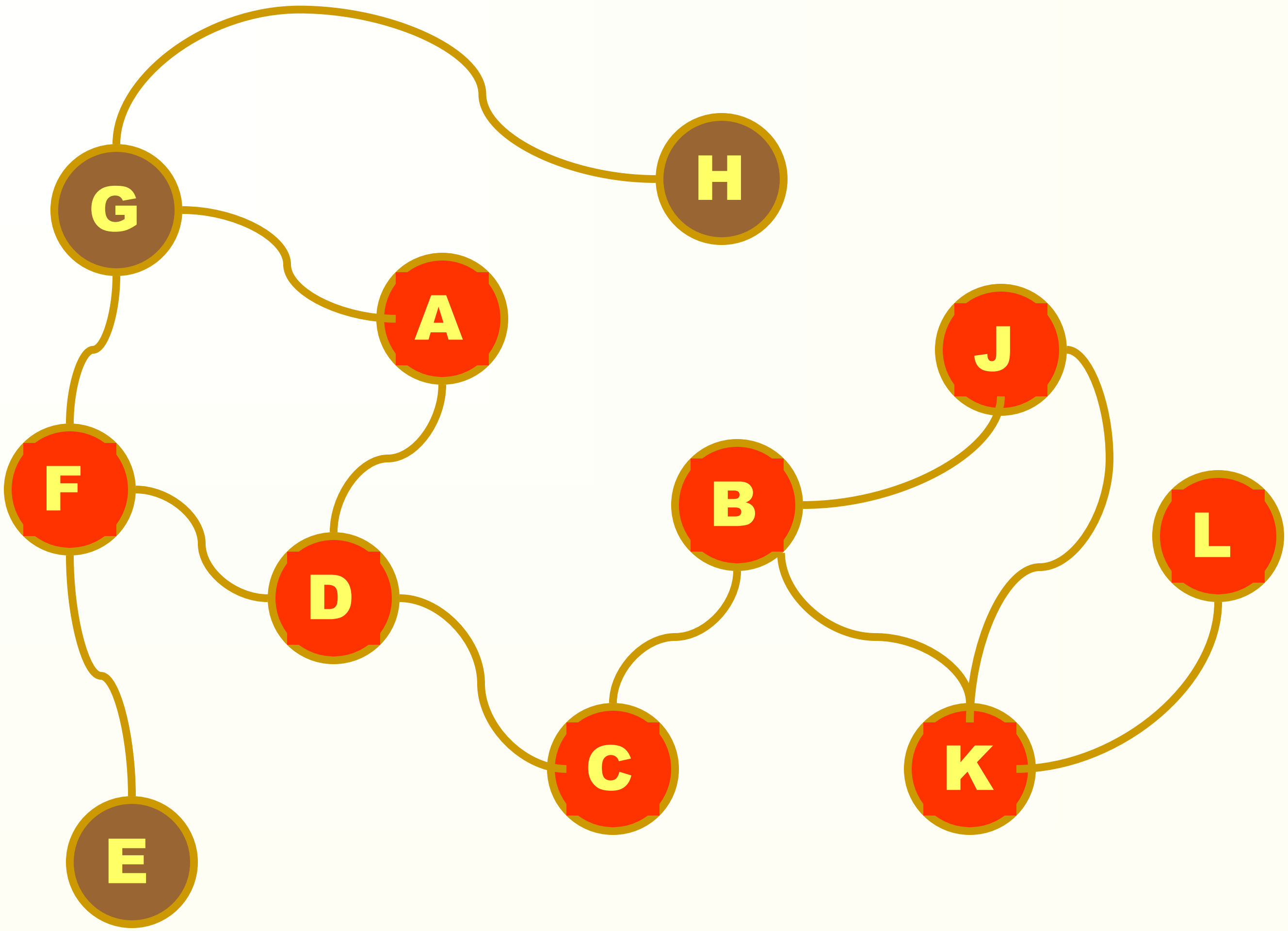
Traversal - Depth First



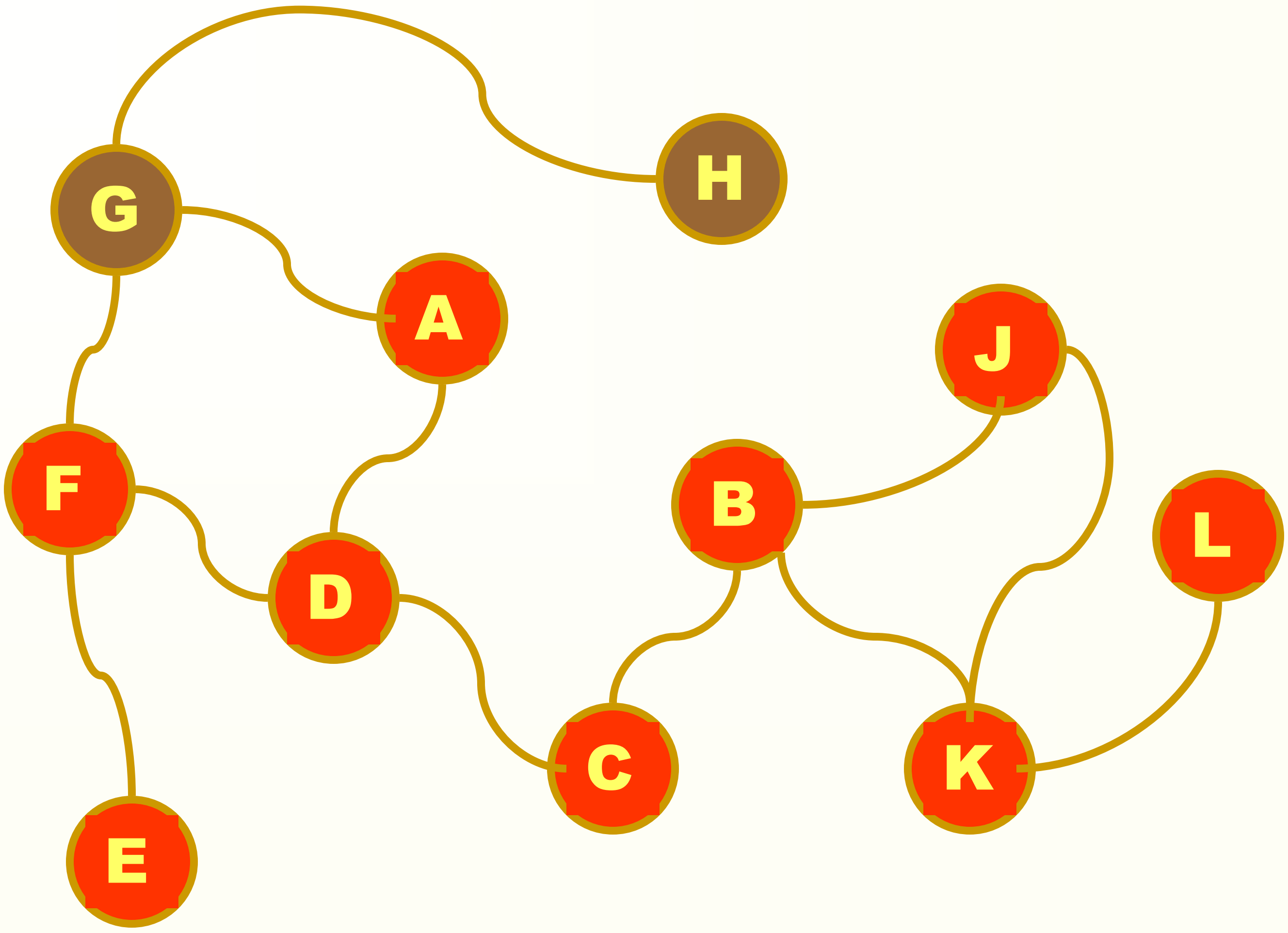
Traversal - Depth First



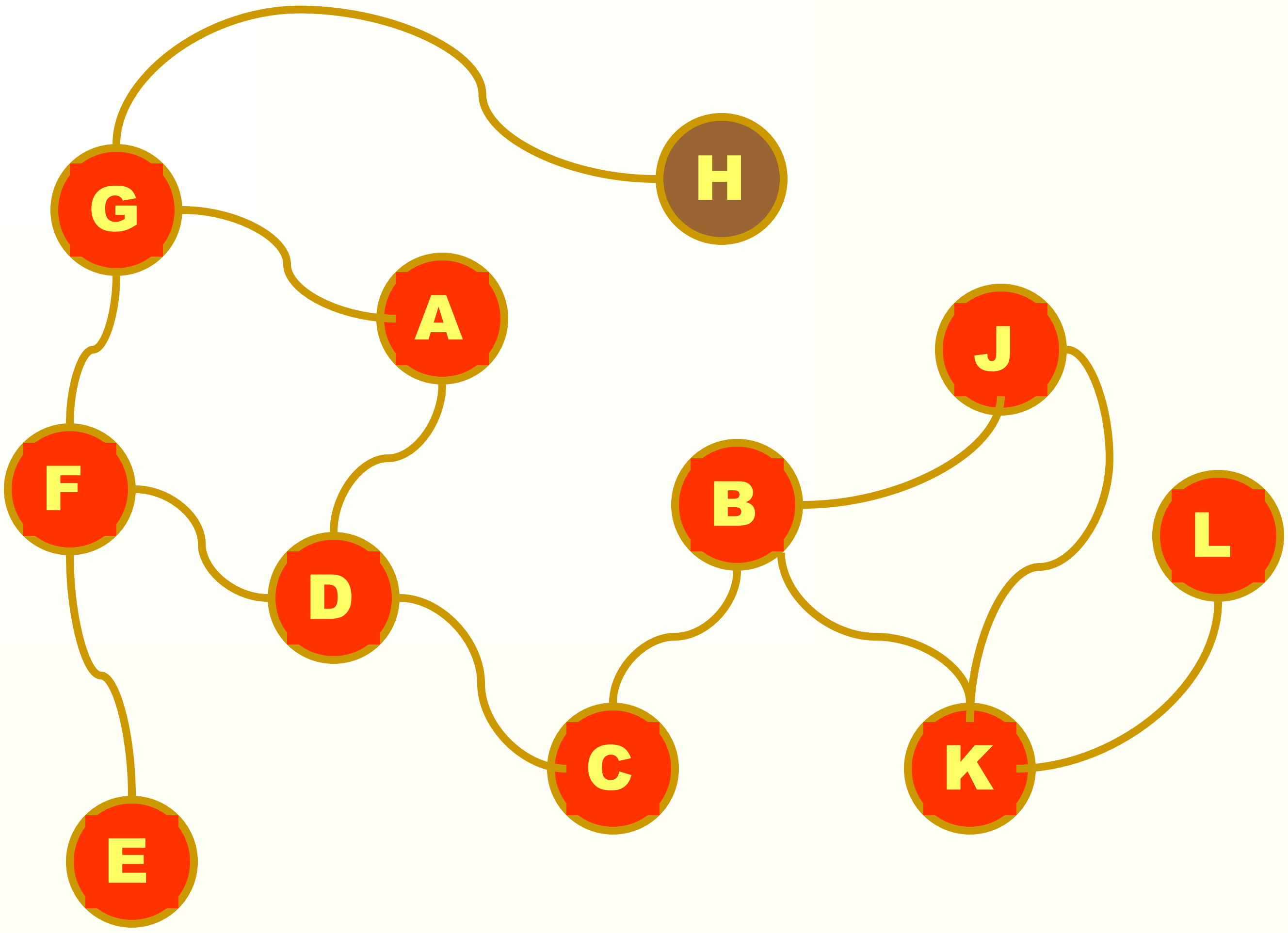
Traversal - Depth First



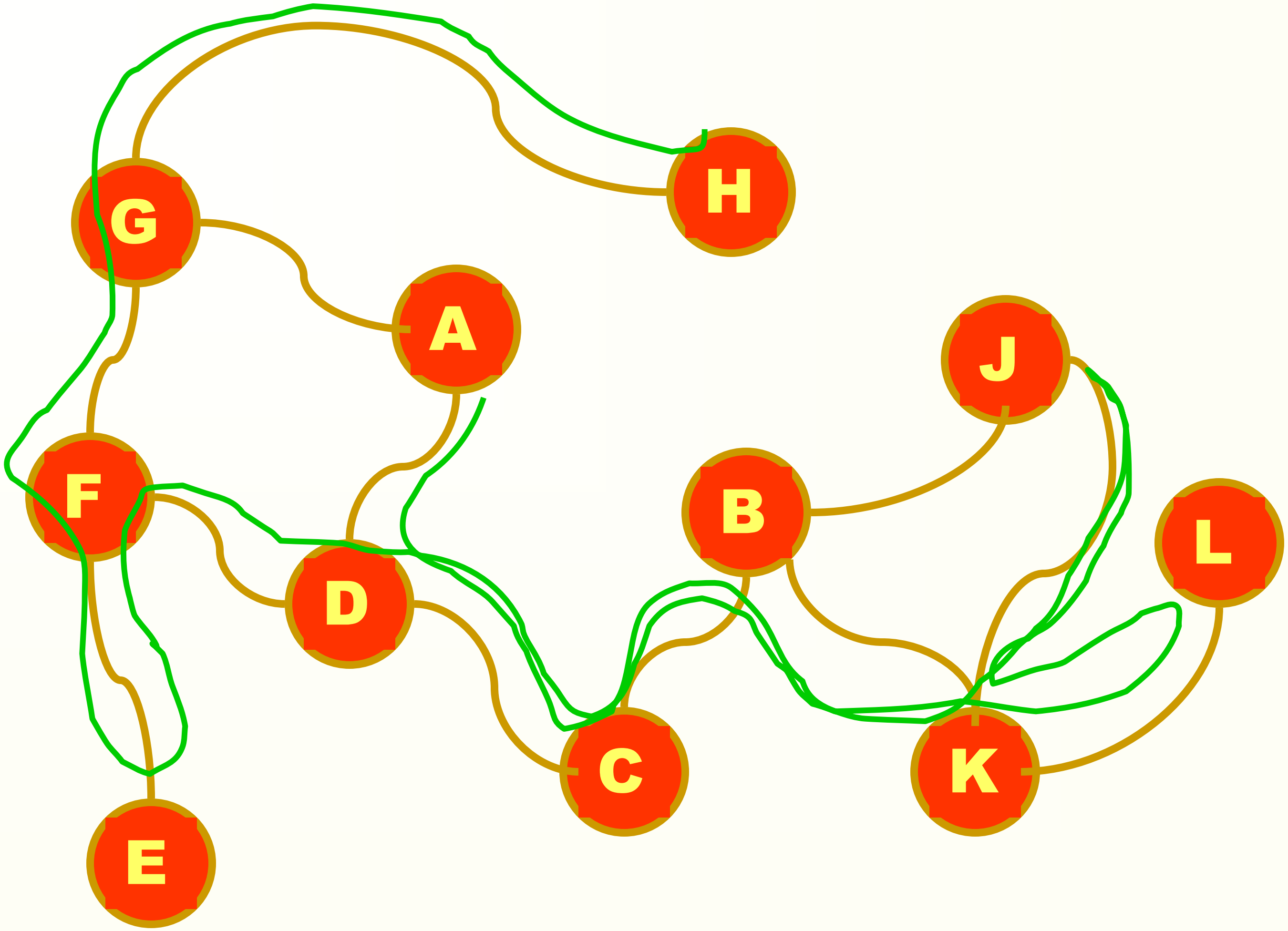
Traversal - Depth First



Traversal - Depth First



Traversal - Depth First



DFS Algorithm

- STEP 1: SET status = 1 (**ready state**) for each node/vertices of G.
- STEP 2: PUSH the starting node (let it be 'A') on the stack and set STATUS = 2 (**waiting**)
- STEP 3: Repeat steps 4 and 5 until STACK is empty

DFS Algorithm

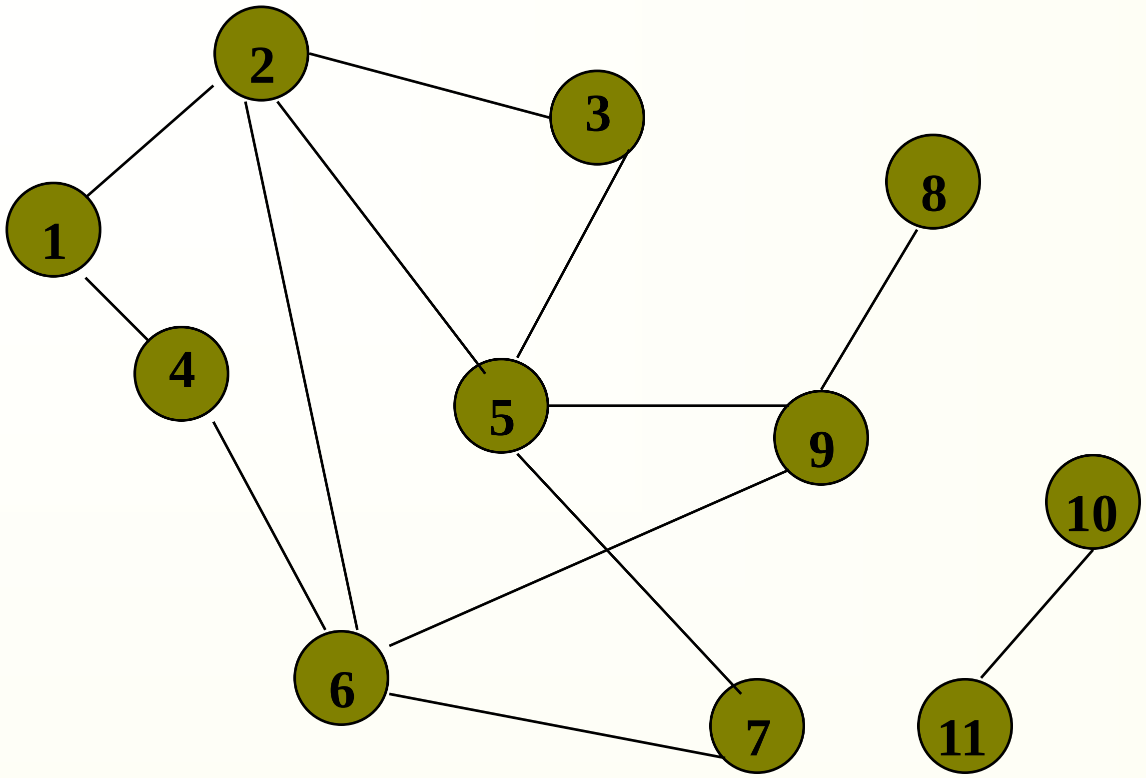
- STEP 4: POP the top node (let it be ‘N’), process it and set its STATUS = 3 (**processed**)

- STEP 5: PUSH on the STACK all the neighbors of ‘N’ that are in the ready state (STATUS = 1) and set their STATUS = 2 (**waiting**)

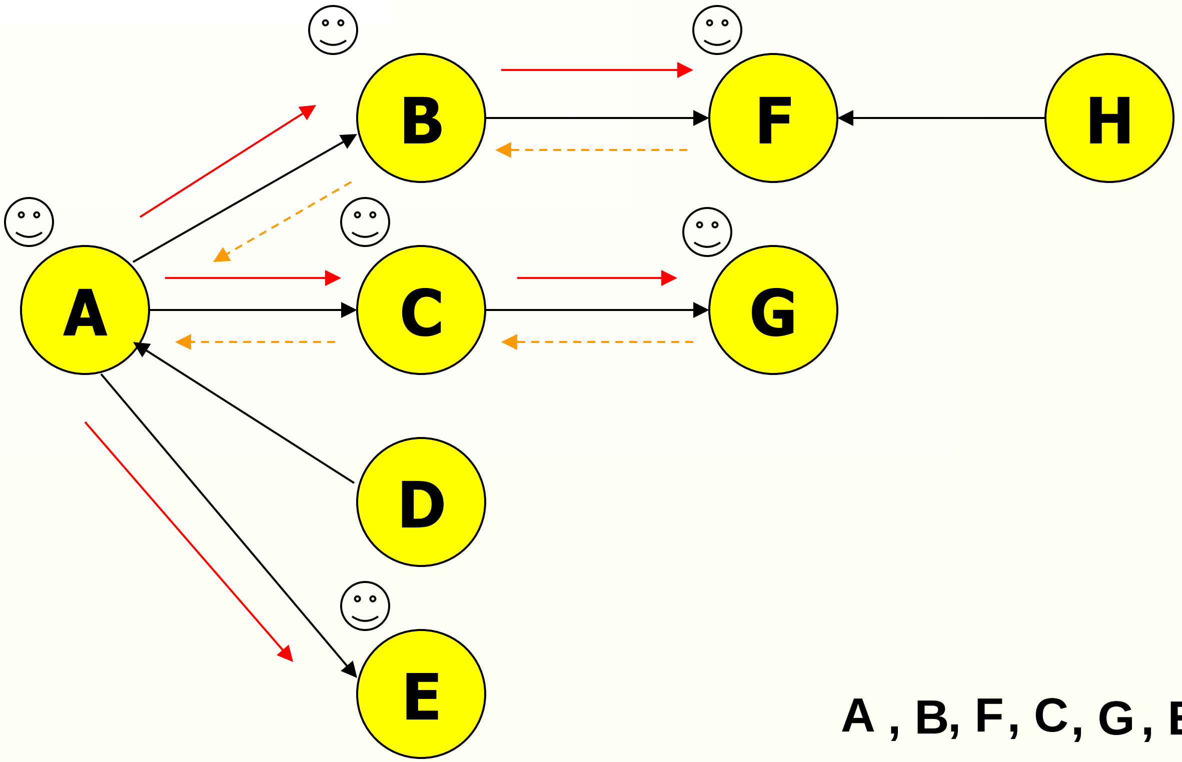
[END LOOP]

STEP 6: Exit

Depth-First Search Example



Depth-First Search



A , B, F, C, G , E

THANK YOU

Graph