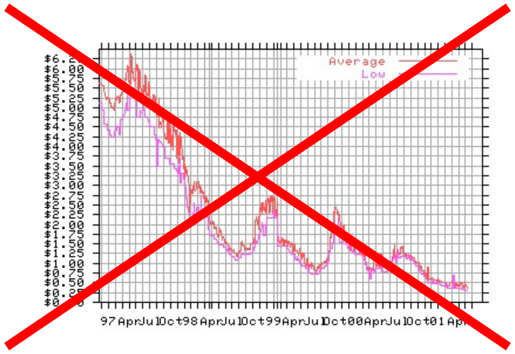
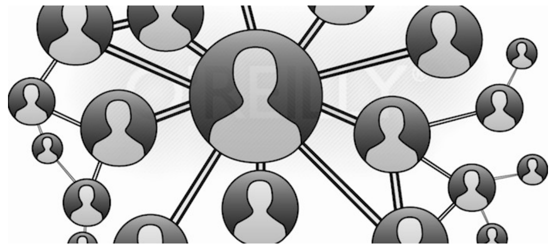
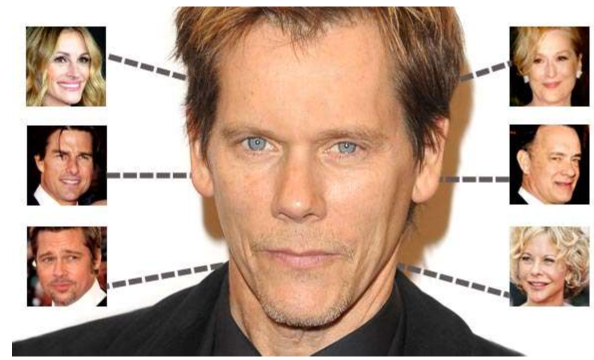
Lecture 27-28

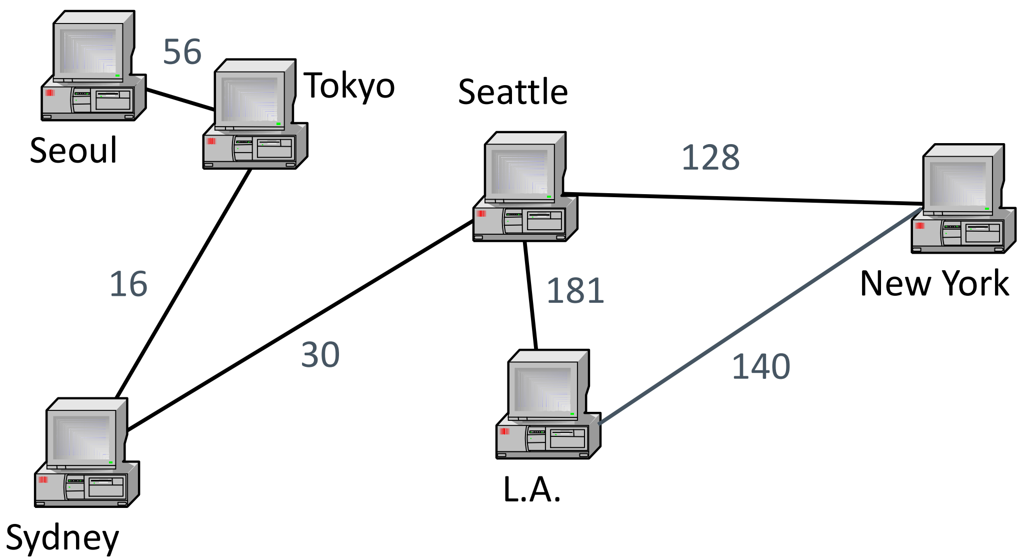
Graphs: Terminology, Representation Techniques, Adjacency Matrix, Adjacency List; Various Applications of Graphs

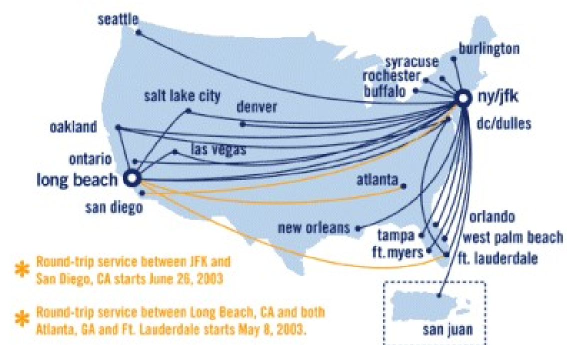
What is a graph?











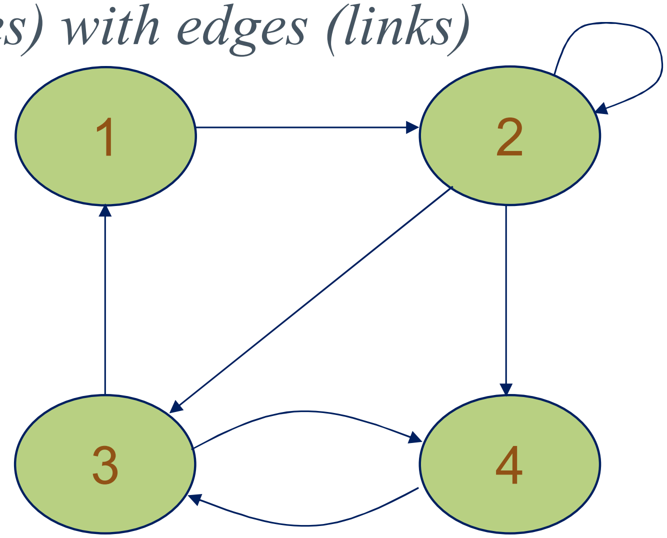
Graphs

*A graph is a pictorial representation of a set of **objects** where some pairs of objects are connected by **links**.*

*The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges***

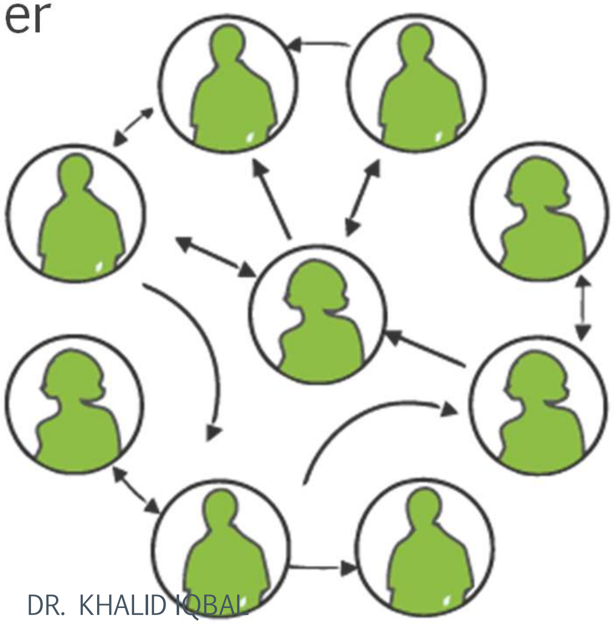
***Formal Definition** = a set of nodes (vertices) with edges (links) between them.*

- $G = (V, E)$ - graph
- V = set of vertices $\quad |V| = n$
- E = set of edges $\quad |E| = m$
  - Binary relation on V
  - Subset of $V \times V = \{(u, v): u \in V, v \in V\}$



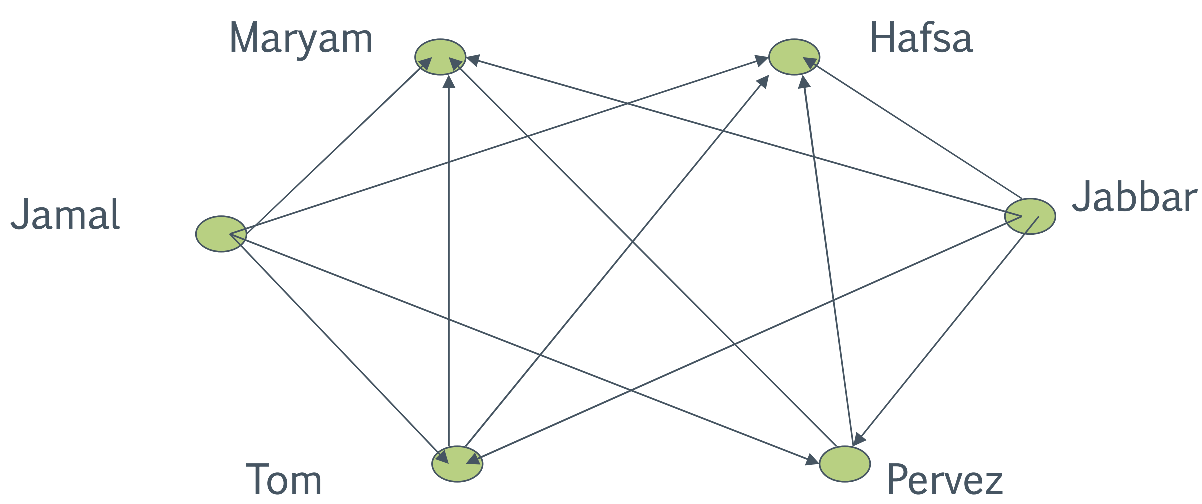
Graph examples

› For each, what are the vertices and what are the edges?
  – Web pages with links
  – Methods in a program that call each other
  – Road maps (e.g., Google maps)
  – Airline routes
  – Facebook friends
  – Course pre-requisites
  – Family trees
  – Paths through a maze



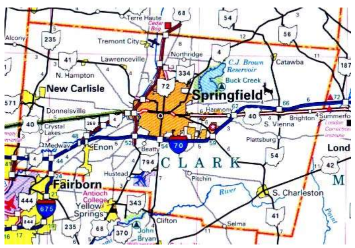
A “Real-life” Example of a Graph

- V=set of 6 people: Jamal, Maryam, Jabbar, Hafsa, Tom, and Pervez, of ages 21 20, 21, 20, 19, and 19, respectively.
- E ={(x, y) | if x is younger than y}

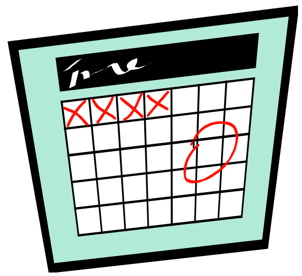


Applications

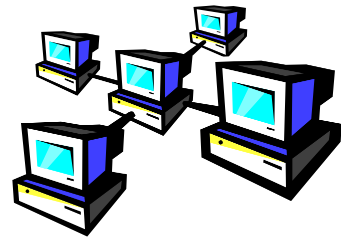
- *Applications that involve not only a set of items, but also the connections between them*



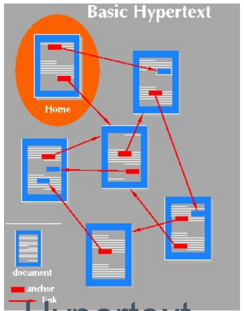
Maps



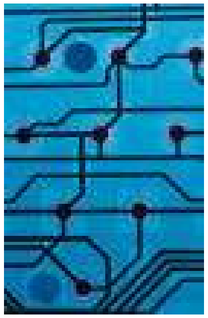
Schedules



Computer networks



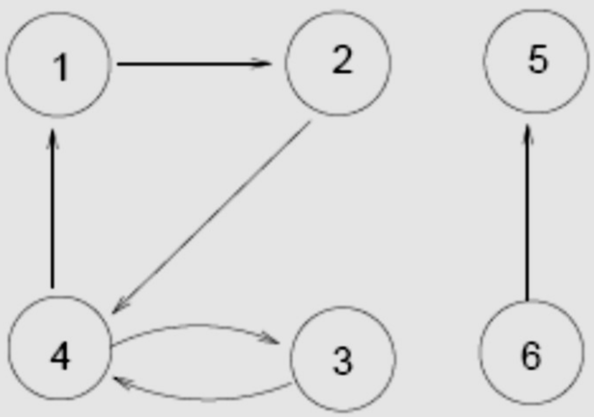
Hypertext



Circuits

Terminology *Directed vs Undirected graphs*

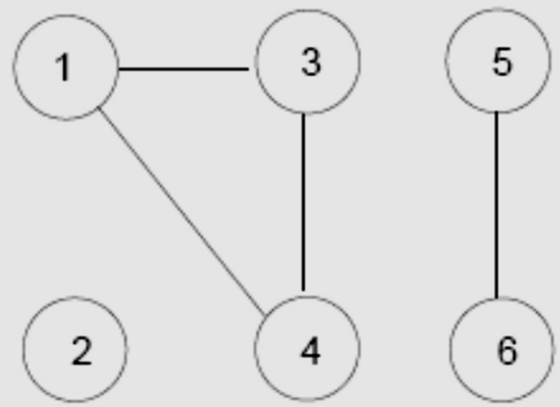
Directed graphs (digraphs)
(ordered pairs of vertices)



in-degree of v: # edges enetring v
out-degree of v: # edges leaving v

v is **adjacent** to u if there is an edge (u,v)

Undirected graphs
(unordered pairs of vertices)

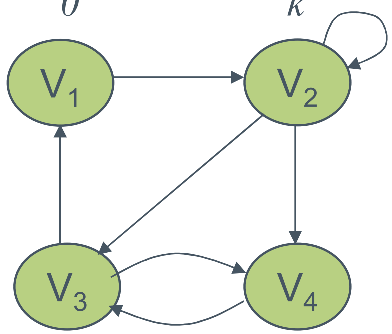


degree of v: # edges incident on v

v is **adjacent** to u and u is **adjacent** to v
if there is and edge (u,v)

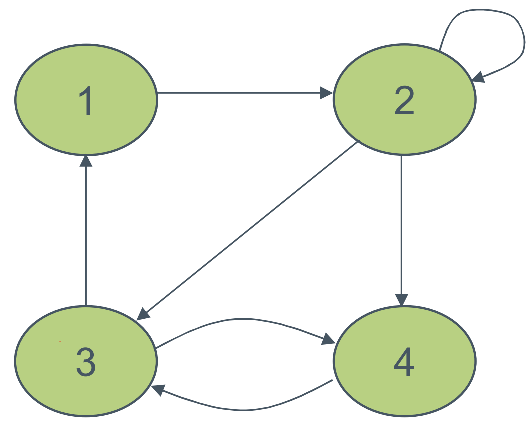
Terminology (Cont !!!)

- *Complete graph*
  - *A graph with an edge between each pair of vertices*
- *Subgraph*
  - *A graph (V', E') such that $V' \subseteq V$ and $E' \subseteq E$*
- *Path from v to w*
  - *A sequence of vertices $<v_0, v_1, \ldots, v_k>$ such that $v_0=v$ and $v_k=w$*
- *Length of a path*
  - *Number of edges in the path*



e.g., path from v_1 to v_4 $<v_1, v_2, v_4>$

Terminology (Cont !!!)

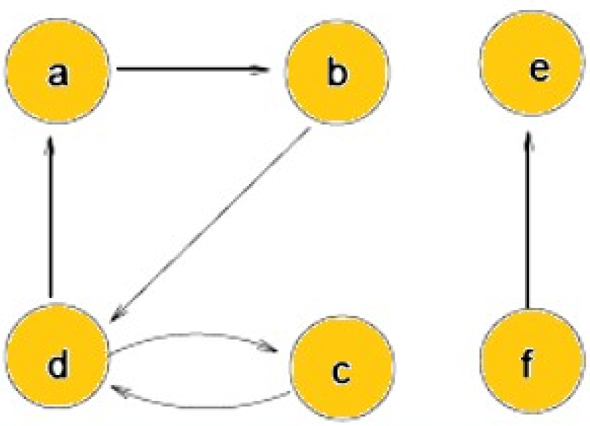


cycle from v_1 to v_1 <v_1, v_2, v_3,v_1>

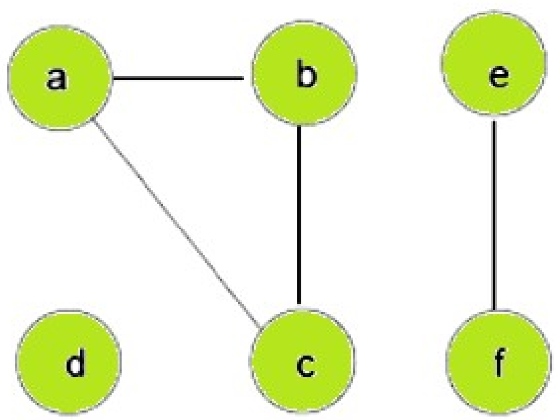
- *w is **reachable** from v*
  - *If there is a path from v to w*
- ***Simple** path*
  - *All the vertices in the path are distinct*
- *Cycles*
  - *A path* $<v_0, v_1, \ldots, v_k>$ *forms a cycle if* $v_0=v_k$ *and* $k\geq 2$
- *Acyclic graph*
  - *A graph without any cycles*

Terminology (Cont !!!) *Connected and Strongly Connected*

directed graphs	**undirected graphs**
strongly connected: every two vertices are reachable from each other	connected: every pair of vertices is connected by a path
strongly connected components : all possible strongly connected subgraphs	connected components: all possible connected subgraphs



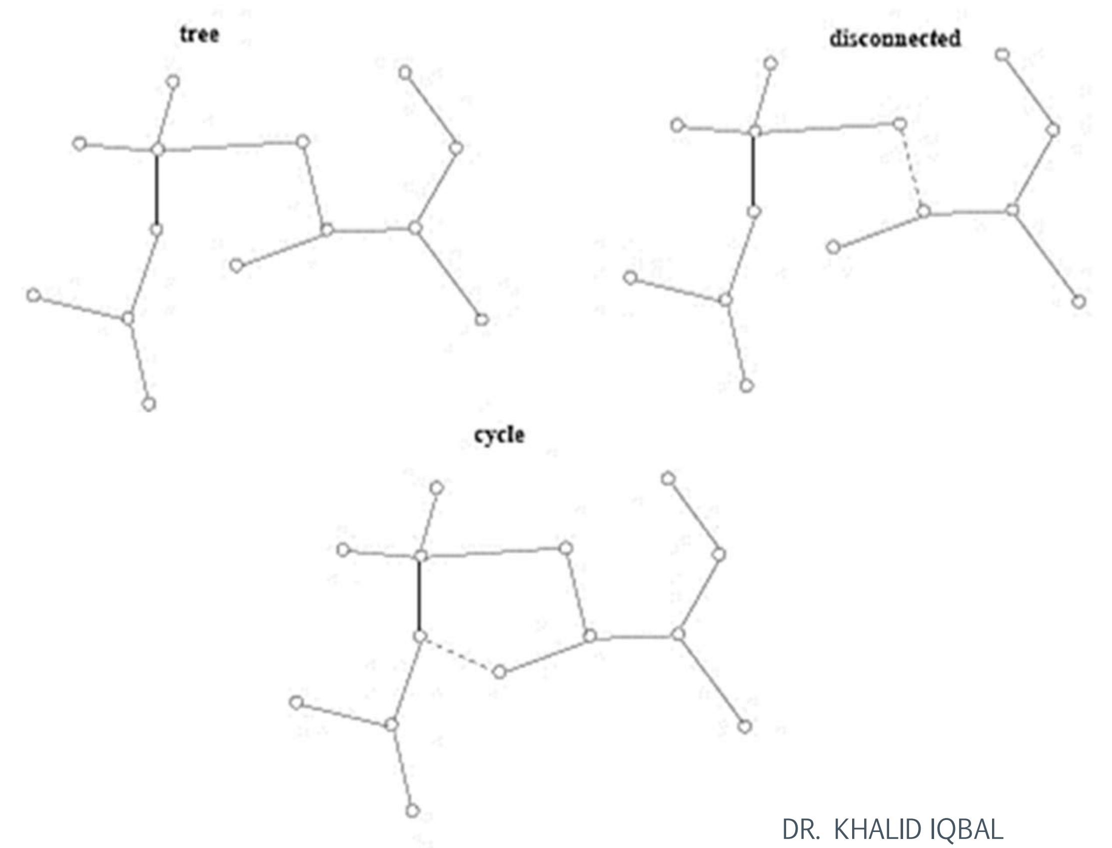
strongly connected components: {a,b,c,d} { e} {f}



connected components: {a,b,c} {d} {e,f}

Terminology (Cont !!!)

- A **tree** is a connected, acyclic undirected graph



Adjacency Matrix & Adjacency List

- An adjacency matrix is a binary matrix of size n x n
- Two possible values in each cell of the matrix: 0 and 1. Let there be an edge between vertices v_i and v_j , means that ith row and jth column of such matrix is equal to 1.
- **Importantly, if the graph is undirected then the matrix is <u>symmetric</u>.**
- Assuming the graph has n vertices, the time complexity to build such a matrix is $O(n^2)$. The space complexity is also $O(n^2)$. Given a graph, to build the adjacency matrix, we need to create a square n x n matrix and fill its values with 0 and 1. It costs us $O(n^2)$ space.

Adjacency Matrix

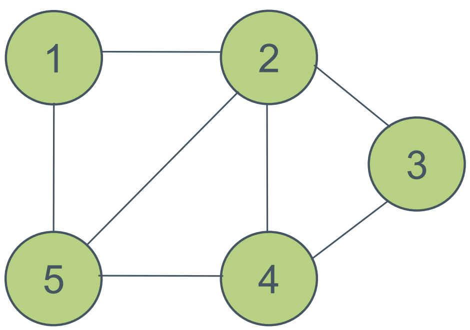
	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	0
v_2	1	0	1	0	1
v_3	0	1	0	1	0
v_4	1	0	1	0	1
v_5	0	1	0	1	0

Adjacency List

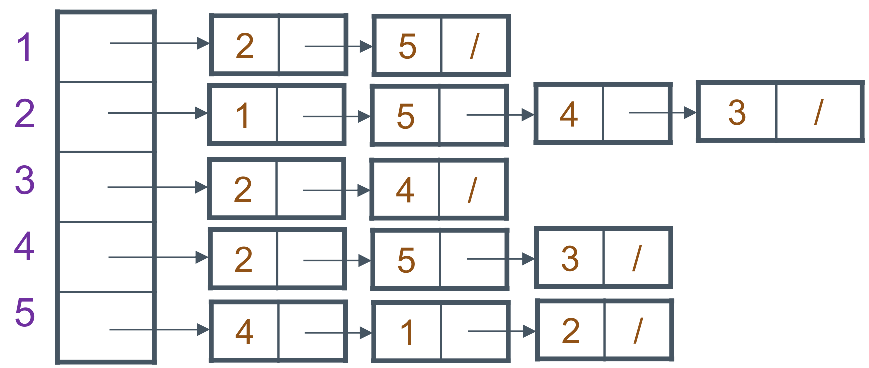
v_1	v_2, v_4
v_2	v_3, v_5
v_3	v_2, v_4
v_4	v_1, v_3, v_5
v_5	v_2, v_4

Graph Representation

- ***Adjacency list representation*** *of G = (V, E)*
  - *An array of |V| lists, one for each vertex in V*
  - *Each list Adj[u] contains all the vertices v that are adjacent to u (i.e., there is an edge from u to v)*
  - *Can be used for both directed and undirected graphs*

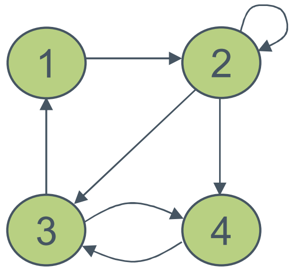


Undirected graph

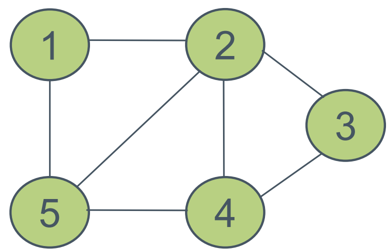


Properties of Adjacency-List Representation

- *Sum of "lengths" of all adjacency lists* |E|
  - *Directed graph:*
    - *edge (u, v) appears only once (i.e., in the list of **u**)*
  - *Undirected graph:* 2|E|
    - *edge (u, v) appears twice (i.e., in the lists of both **u** and **v**)*



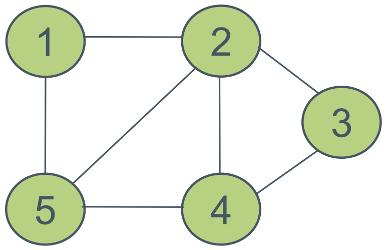
Directed graph



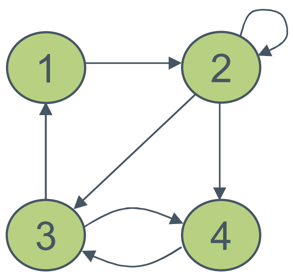
Undirected graph

Properties of Adjacency-List Representation

- *Memory required*
  - $\Theta(V + E)$
- *Preferred when*
  - *The graph is* ***sparse****:* $|E| << |V|^2$
  - *We need to quickly determine the nodes adjacent to a given node.*
- *Disadvantage*
  - *No quick way to determine whether there is an edge between node u and v*
- *Time to determine if* $(u, v) \in E$*:*
  - $O(degree(u))$
- *Time to list all vertices adjacent to u:*
  - $\Theta(degree(u))$



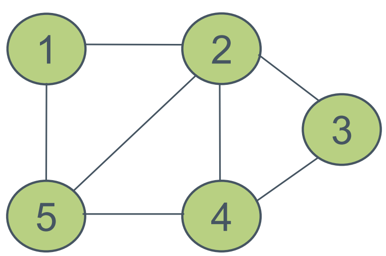
Undirected graph



Directed graph

Graph Representation

- ***Adjacency matrix representation*** *of* $G = (V, E)$
  - *Assume vertices are numbered* $1, 2, \ldots |V|$
  - *The representation consists of a matrix* $A_{|V| x |V|}$:
  - $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

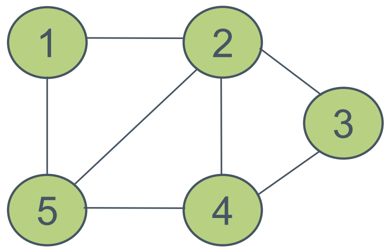
For undirected graphs, matrix A is symmetric:

$a_{ij} = a_{ji}$

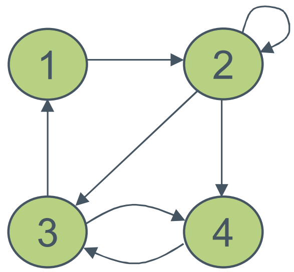
$A = A^T$

Properties of Adjacency Matrix Representation

- *Memory required: $\Theta(V^2)$, independent on the number of edges in G*
- *Preferred when*
  - *The graph is **dense:** $|E|$ is close to $|V|^2$*
  - *We need to quickly determine if there is an edge between two vertices*
- *Time to determine if $(u, v) \in E$: $\Theta(1)$*
- *Disadvantage*
  - *No quick way to determine the vertices adjacent to another vertex*
- *Time to list all vertices adjacent to u: $\Theta(V)$*

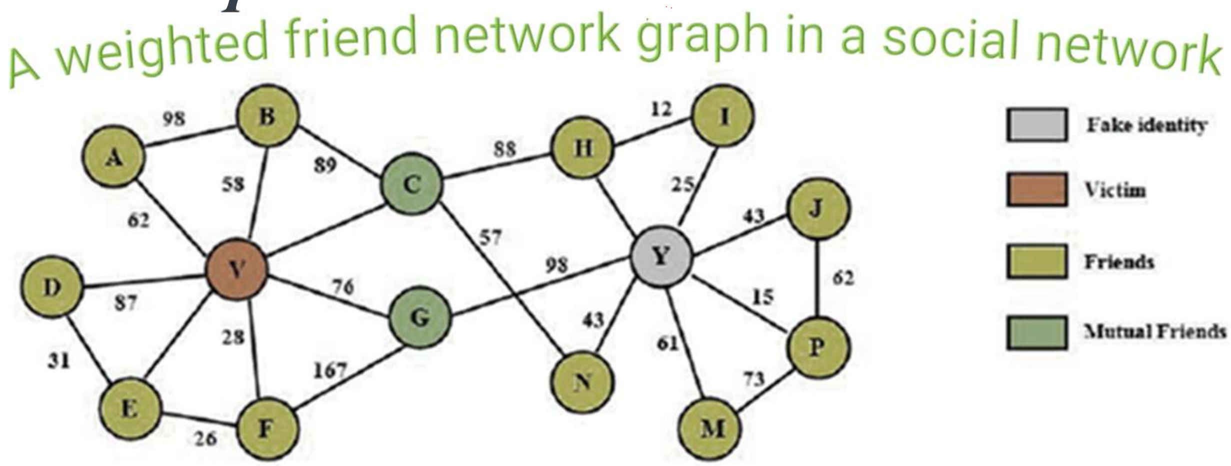


Undirected graph



Directed graph

Weighted Graphs



› *Graphs for which each edge has an associated weight w(u, v)*

$$w: E \rightarrow R, \text{ weight function}$$

› *Storing the weights of a graph*

– *Adjacency list:*

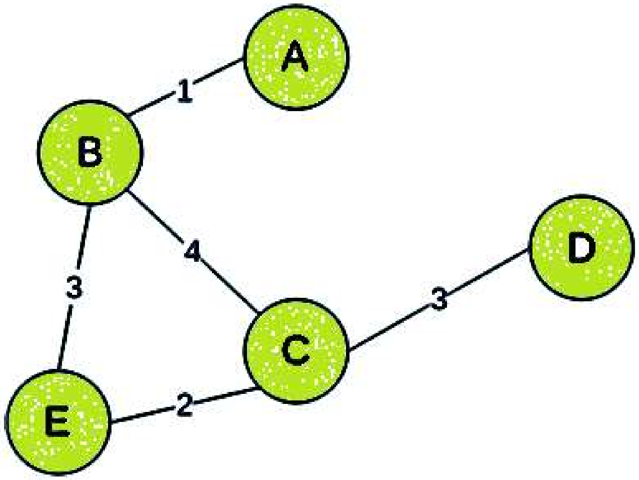
› *Store w(u, v) along with vertex v in u's adjacency list*

– *Adjacency matrix:*

› *Store w(u, v) at location (u, v) in the matrix*

Weighted Graphs: Example

- *The unweighted graphs tell us only if two nodes are linked*
  - *Is there a path between the nodes u and v?*
  - *Which nodes are reachable from u?*
  - *How many nodes are on the shortest path b/w u and v?*
- *In many applications, the edges have numerical properties that we need to exploit in our algorithms to solve the problem at hand.*
  - *For example, we must consider road lengths and traffic density while looking for the* ***shortest path*** *between two cities.*
  - *We associate each edge* ***e*** *with a real value* ***w(e)*** *that we call its weight. We call such graphs weighted.*

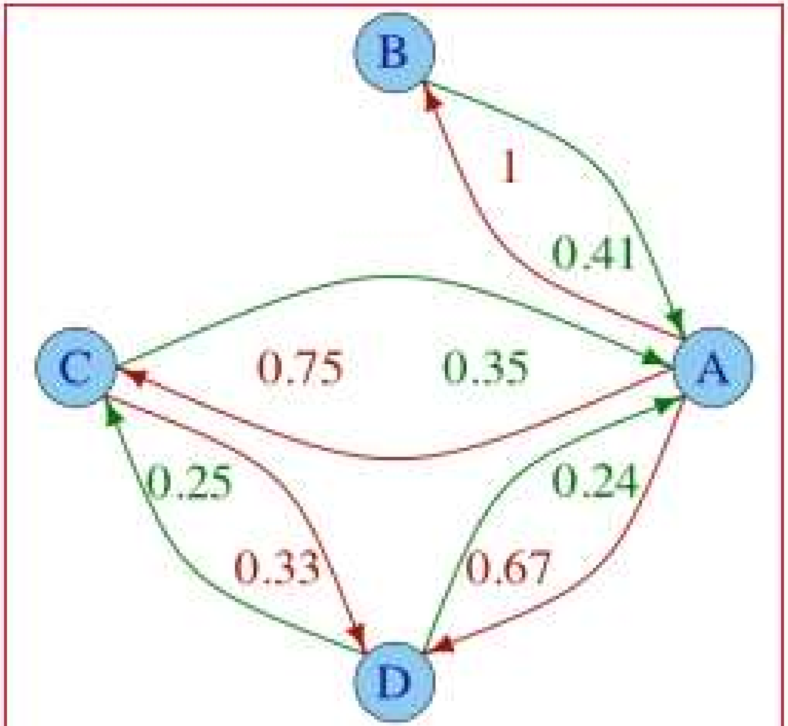


Weights can be lengths, densities, duration, costs, probabilities etc.

All Friends are not Equal: Using Weights in Social Graphs to Improve Search

- *If person B invests a lot of his time in person A, then A has high influence over B.*
- *Influence(A, B), as the proportion of B's investments on A. Let Invests(B, A) be the investment B makes on A.*

$$\text{Influence}(A, B) = \frac{\text{Invests}(B, A)}{\sum_X \text{Invests}(B, X)}$$

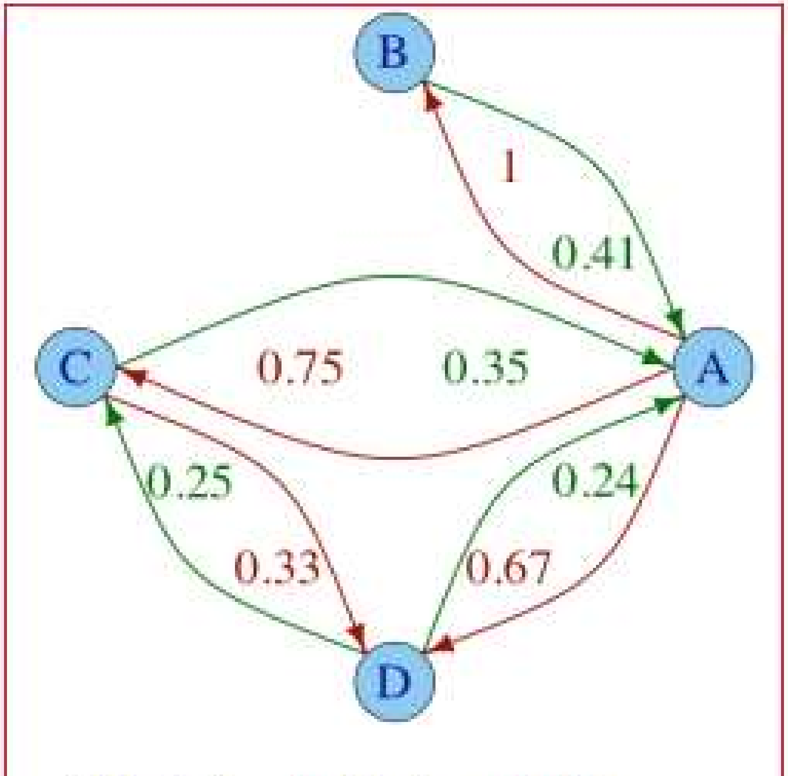


(b) After inferring influence

https://homes.cs.washington.edu/~jheer/files/snakdd.pdf

All Friends are not Equal: Using Weights in Social Graphs - Interpretation

- An intuitive interpretation of this graph runs as follows: imagine that node A is an adviser, and nodes B, C, and D are her students. The edge weights in **Figure (a)** depict the number of co-authorships between node pairs. In **Figure (b)** we see that the adviser holds more influence over her students than her students hold over her. Moreover, student D, who has authored fewer papers than student C, is more influenced by student C because a larger proportion of his total publications involve student C.



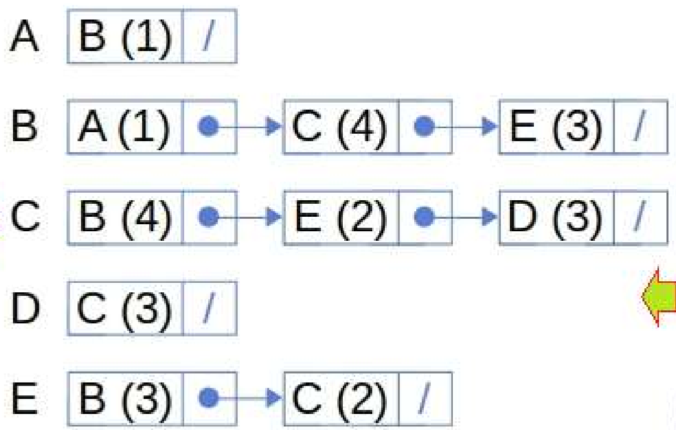
(b) After inferring influence

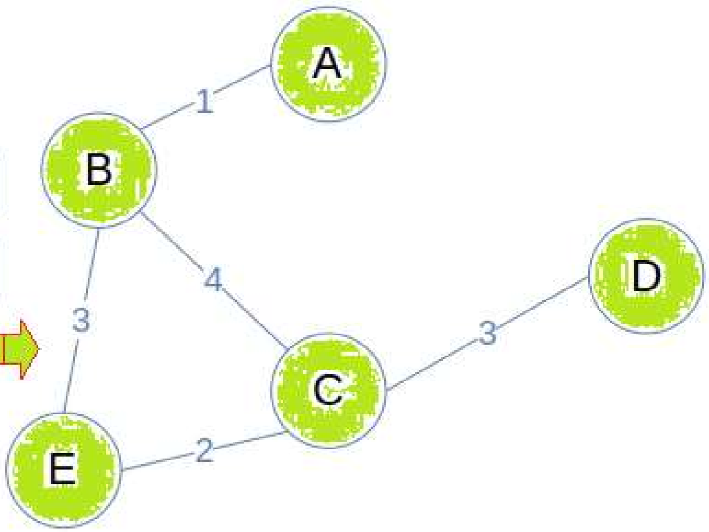
https://homes.cs.washington.edu/~jheer/files/snakdd.pdf

Weighted Graphs

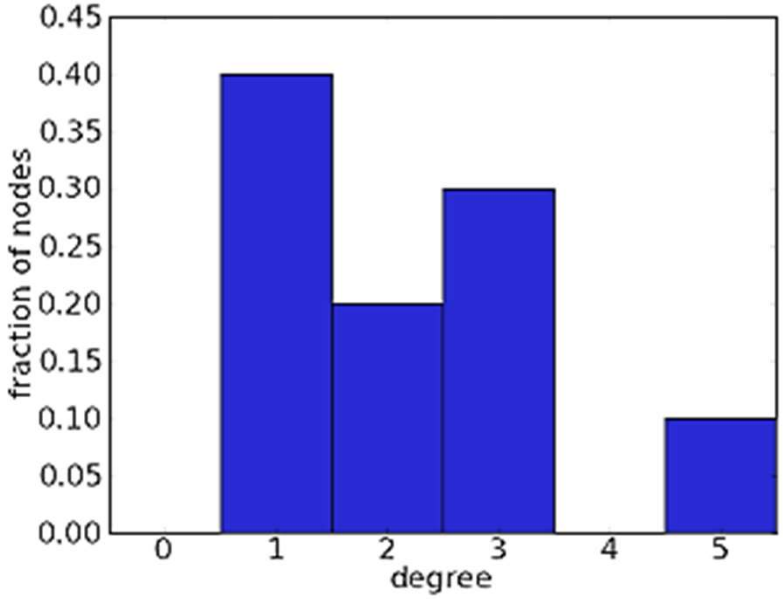
› **Representation of Weighted Graphs**

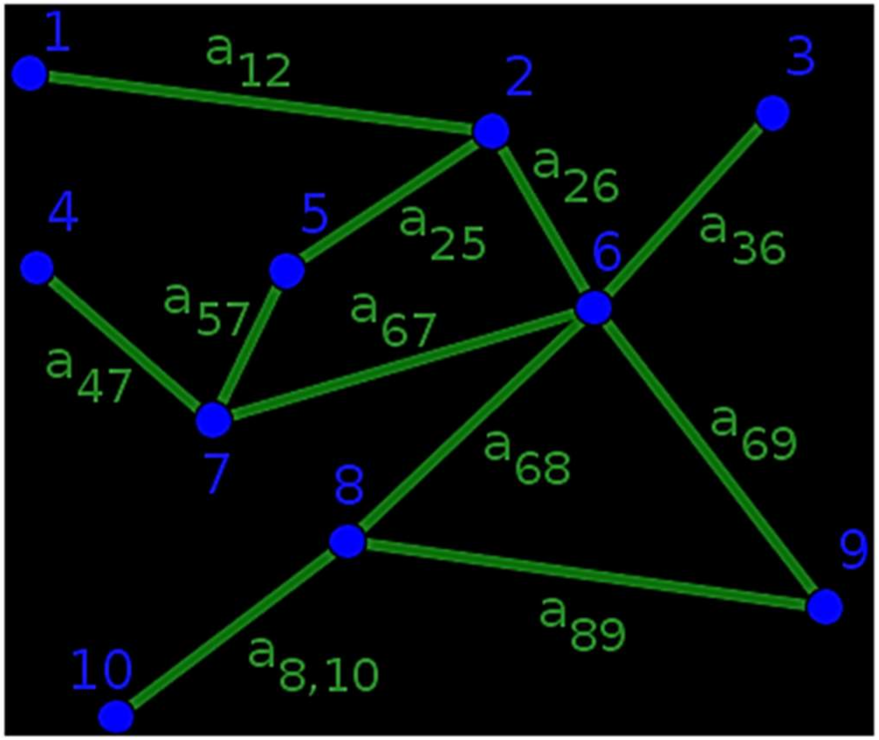
$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}
\begin{array}{c}
\begin{array}{ccccc} A & B & C & D & E \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
1 & 0 & 4 & 0 & 3 \\
0 & 4 & 0 & 3 & 2 \\
0 & 0 & 3 & 0 & 0 \\
0 & 3 & 3 & 0 & 0
\end{pmatrix}
\end{array}
$$





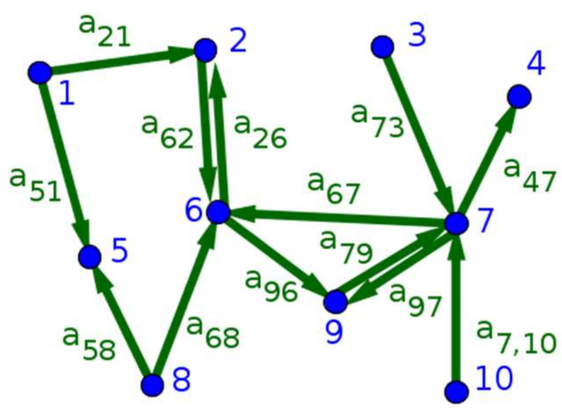
In-Degree & Out-Degree of a Graph

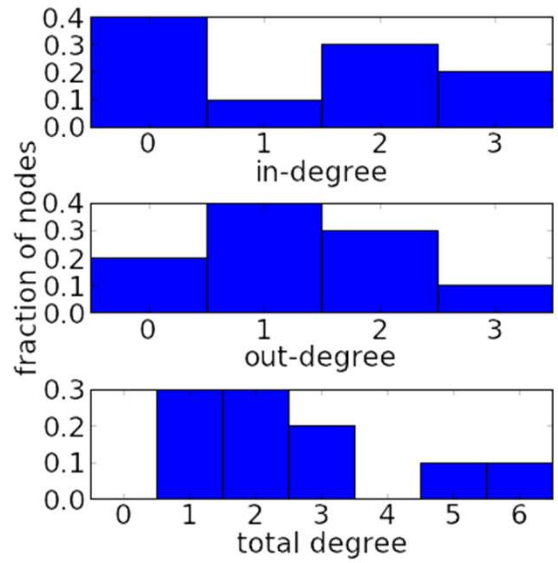


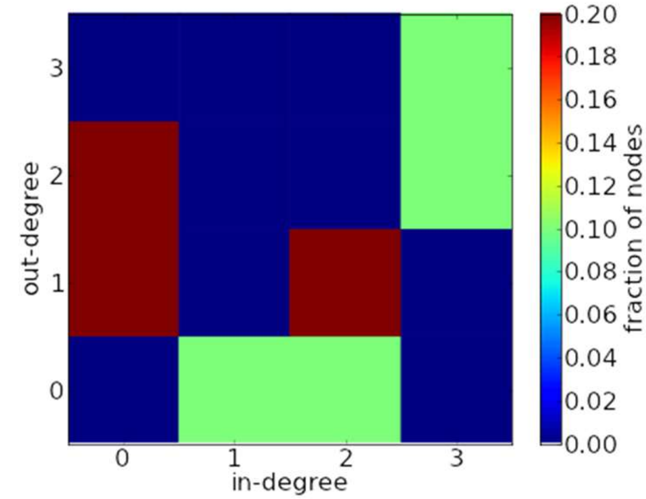


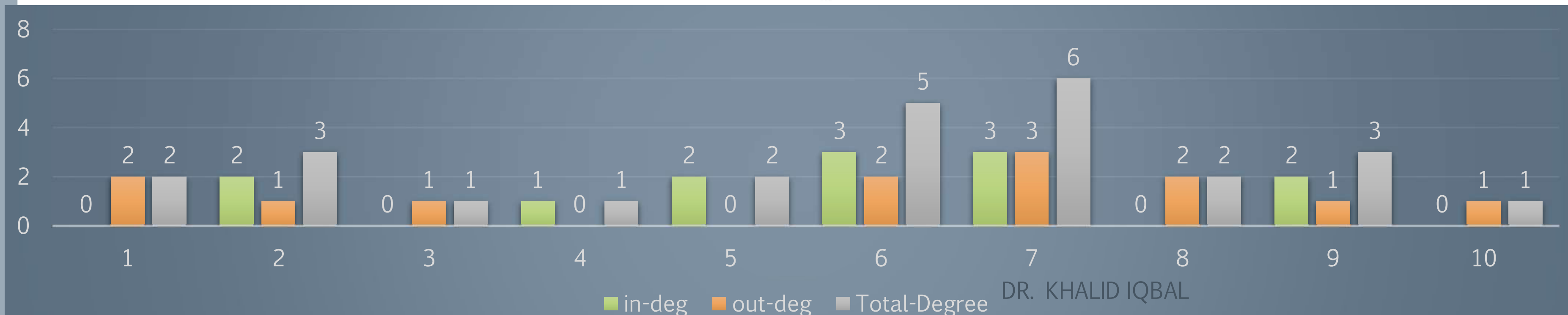
For this undirected network, the degrees are $k_1 = 1$, $k_2 = 3$, $k_3 = 1$, $k_4 = 1$, $k_5 = 2$, $k_6 = 5$, $k_7 = 3$, $k_8 = 3$, $k_9 = 2$, and $k_{10} = 1$. Its degree distribution is $P_{\text{deg}}(1) = 2/5$, $P_{\text{deg}}(2) = 1/5$, $P_{\text{deg}}(3) = 3/10$, $P_{\text{deg}}(5) = 1/10$, and all other $P_{\text{deg}}(k) = 0$.

In-Degree & Out-Degree of a Graph



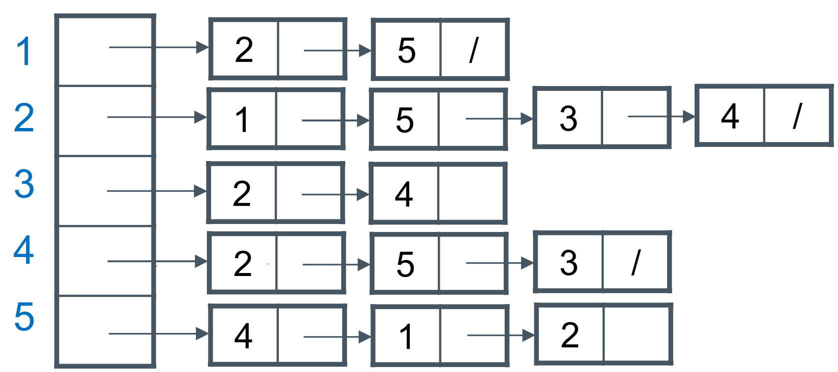






Problem 1

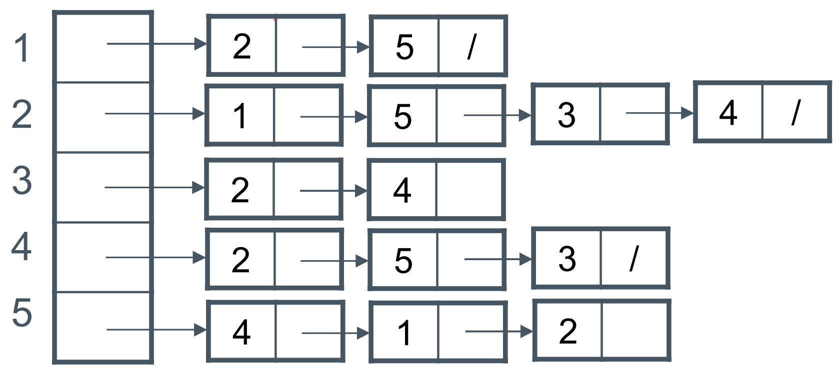
- *Given an adjacency-list representation, how long does it take to compute the out-degree of every vertex?*
  - *For each vertex u, search Adj[u]* → Θ*(E)*



Problem 2

- *How long does it take to compute the in-degree of every vertex?*
  - *For each vertex u,*

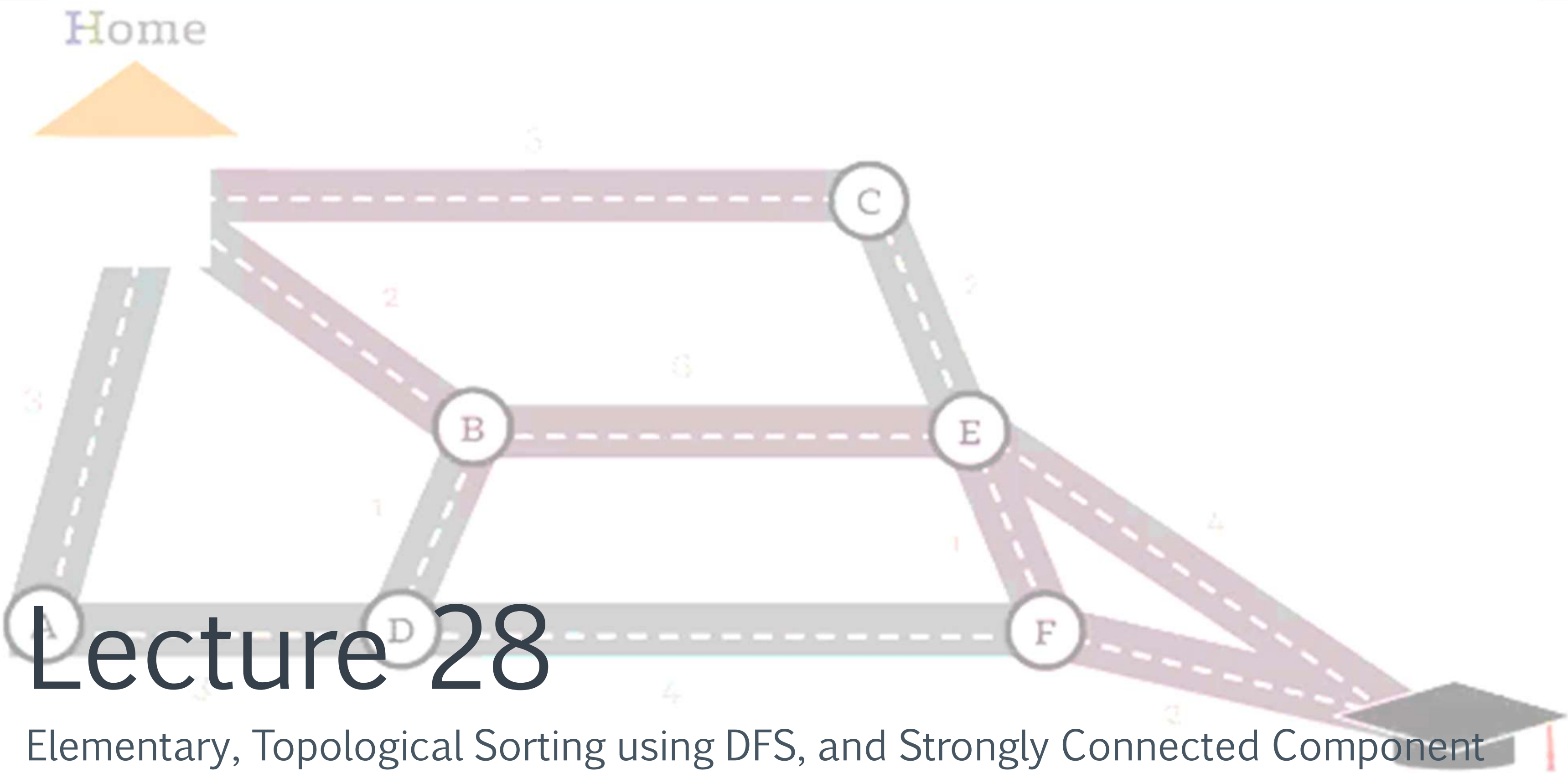
    search entire list of edges → $\Theta(VE)$



Network Models

- *Network models are used to solve a variety of problems:*

1. *the minimal-spanning tree technique,*
2. *the maximal-flow technique, and*
3. *the shortest-route technique.*



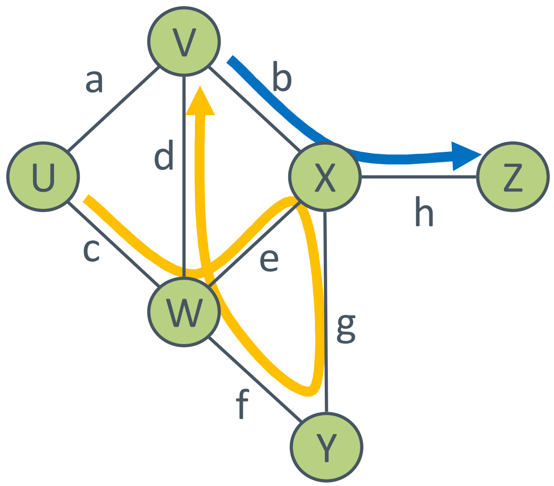
Lecture 28

Elementary, Topological Sorting using DFS, and Strongly Connected Component

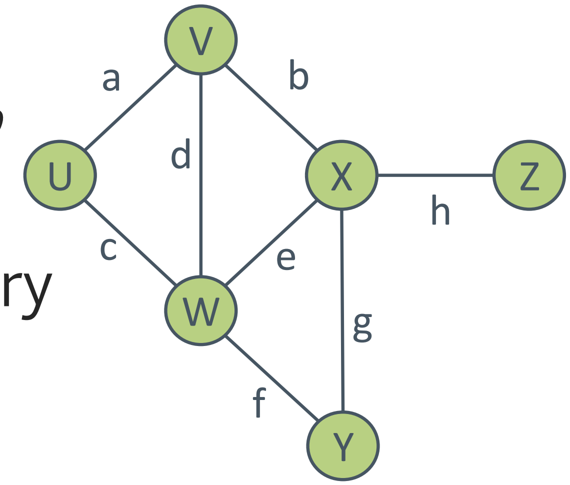
DR. KHALID IQBAL

Paths

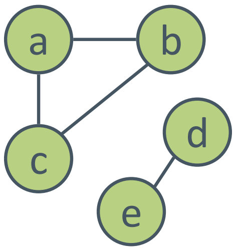
- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?
- **path length**: Number of vertices or edges contained in the path.
- **neighbor** or **adjacent**: Two vertices connected directly by an edge.
  - example: V and X

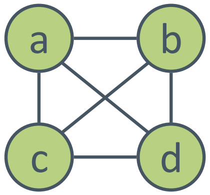


Reachability, connectedness



- reachable: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.
- connected: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?
- strongly connected: When every vertex has an edge to every other vertex.



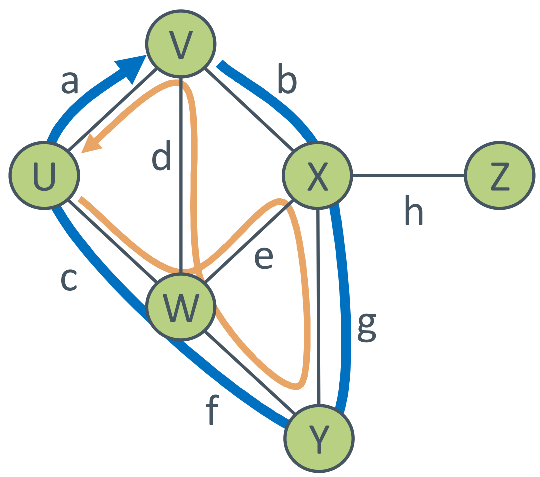


Loops and cycles

› cycle: A path that begins and ends at the same node.
  – example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  – example: {c, d, a} or {U, W, V, U}.

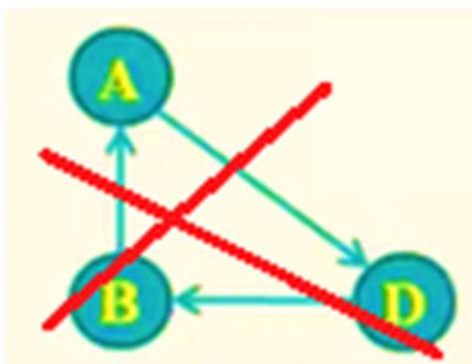
  – acyclic graph: One that does not contain any cycles.

› loop: An edge directly from a node to itself.
  – Many graphs don't allow loops.



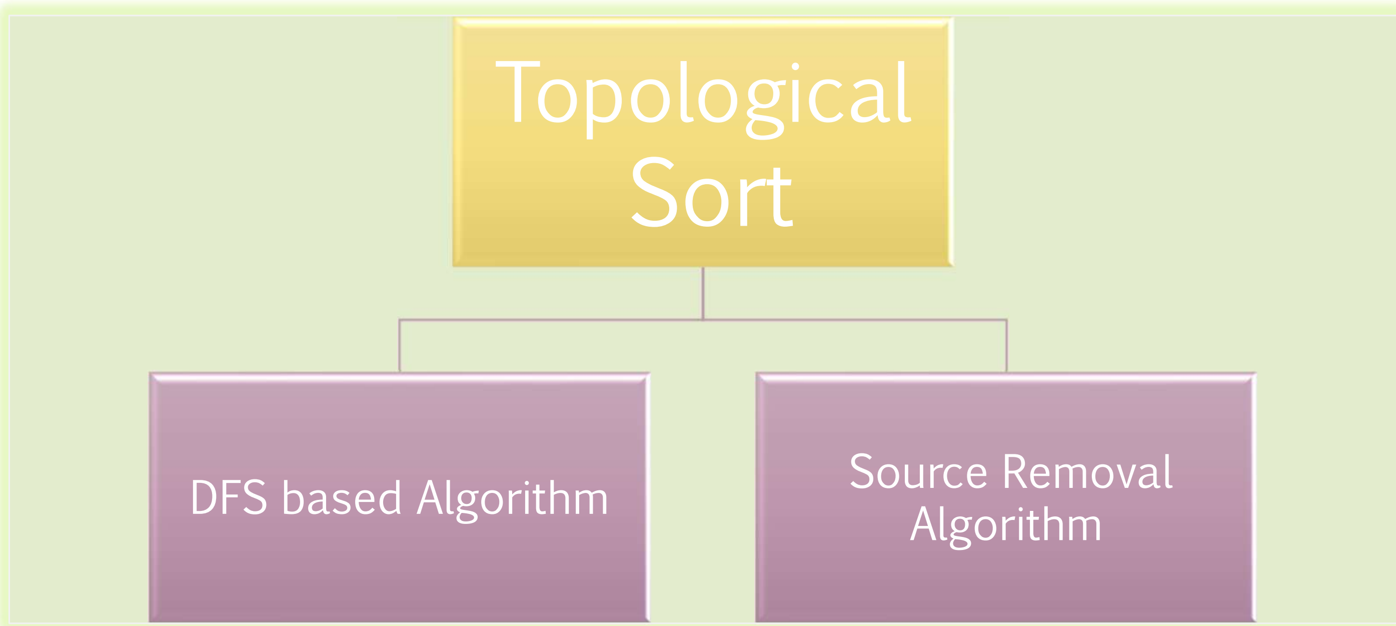
Topological Sort

- A process of designing a linear ordering to the vertices of DAG (Directed Acyclic Graph: A graph having no cycles).
- If there is an edge from vertex i to vertex j then i appears before j is linear ordering.
- Examples:



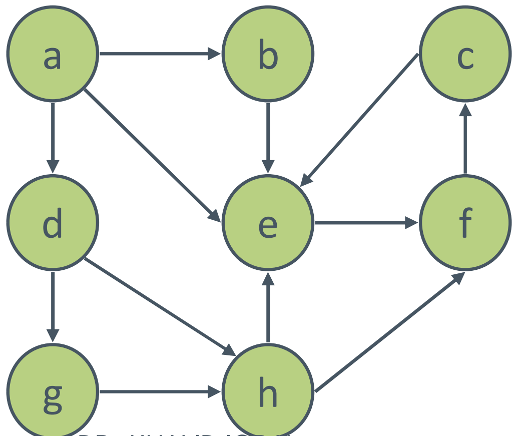
Use:
Vertices of a Graph represents the TASKS to perform.
Edges represent the constraints to perform one task before another.

Topological Sort: Methods



Depth-first search

- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.
- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

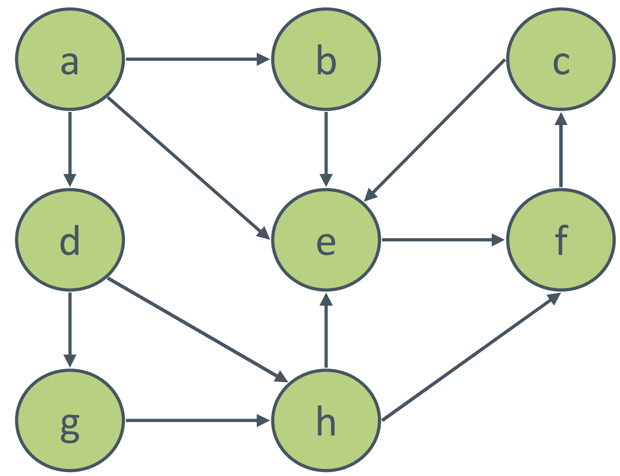
DFS pseudocode

function dfs(v_1, v_2):
    dfs(v_1, v_2, { }).

function dfs(v_1, v_2, *path*):
    path += v_1.
    mark v_1 as visited.
    if v_1 is v_2:
        a path is found!

    for each unvisited neighbor *n* of v_1:
        if dfs(*n*, v_2, *path*) finds a path: a path is found!

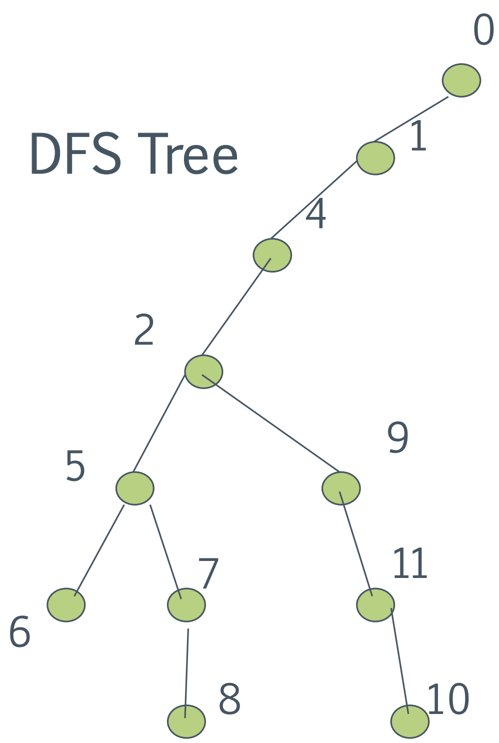
    path -= v_1.   // path is not found.

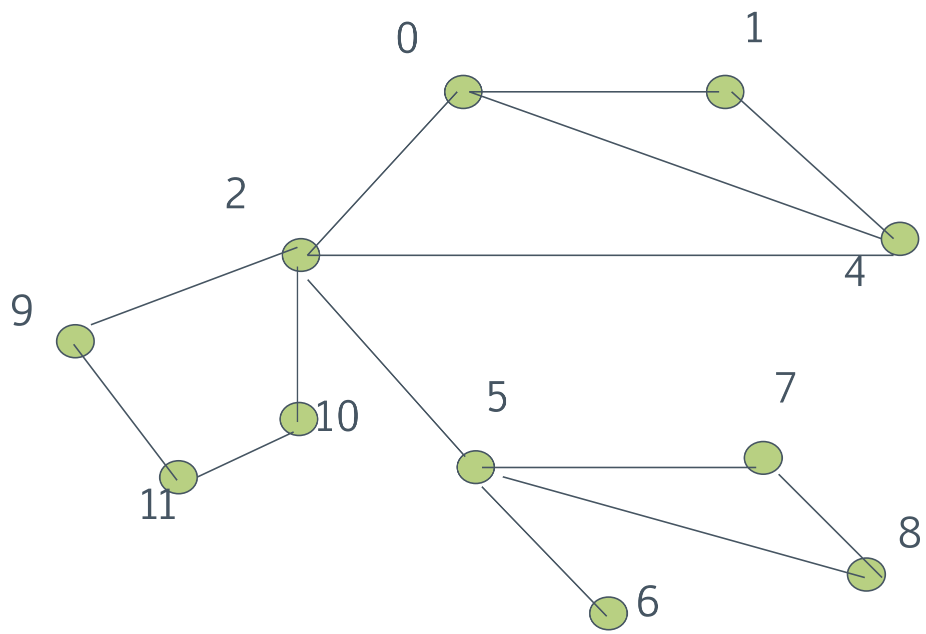


- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

Illustration of DFS

DFS Tree



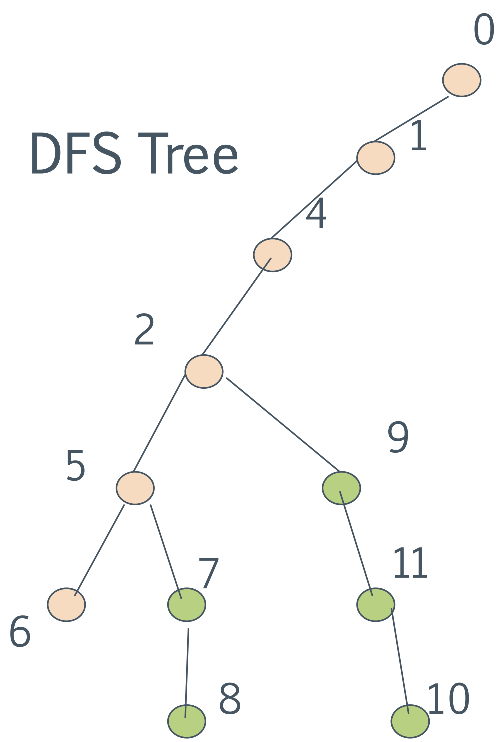


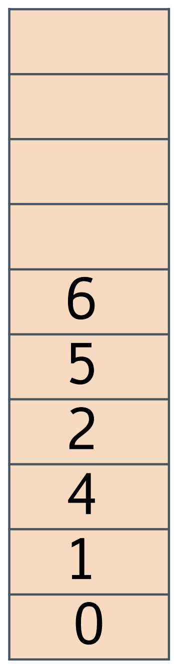
Graph G

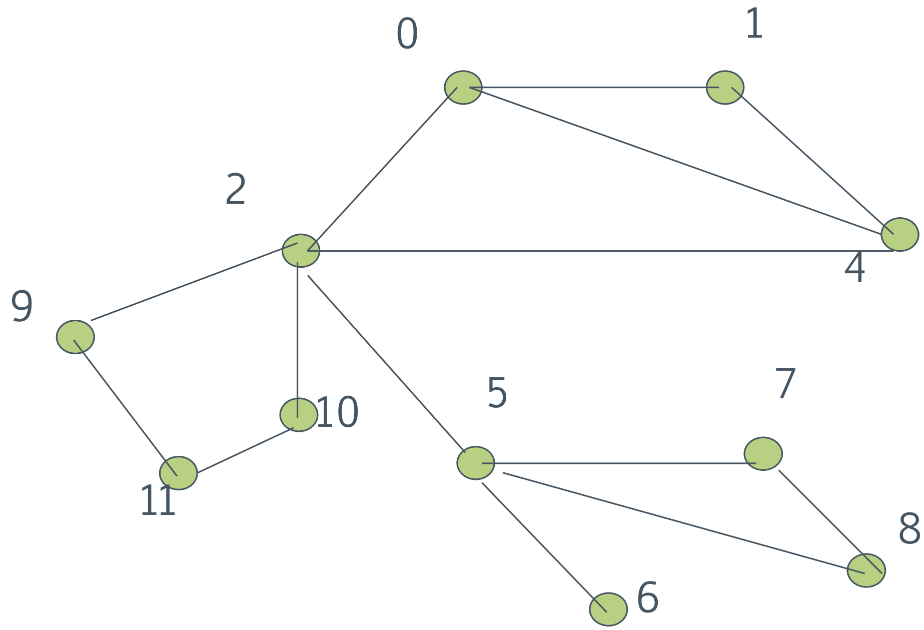
Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6

DFS Tree



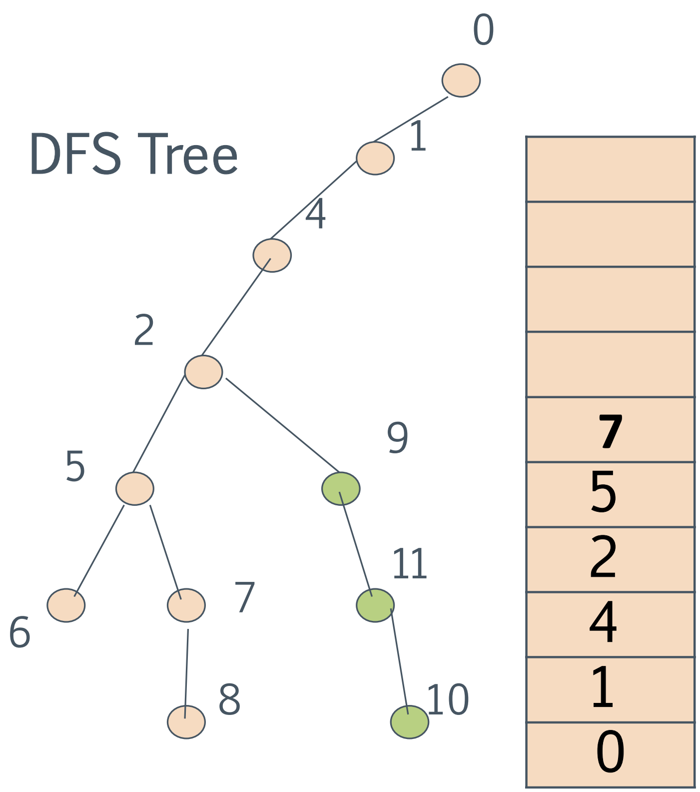


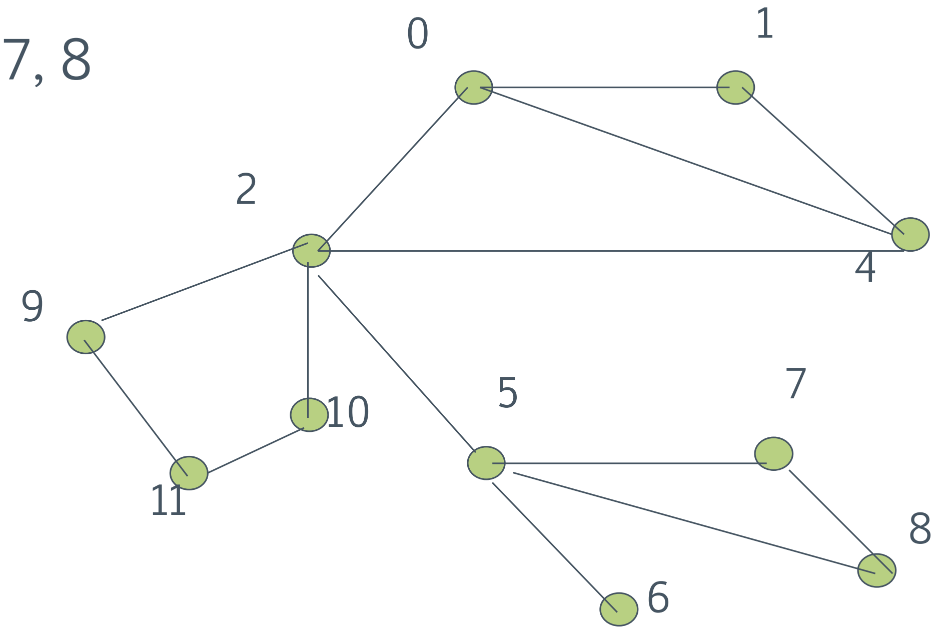


Graph G

Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6, 7, 8





Graph G

Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10

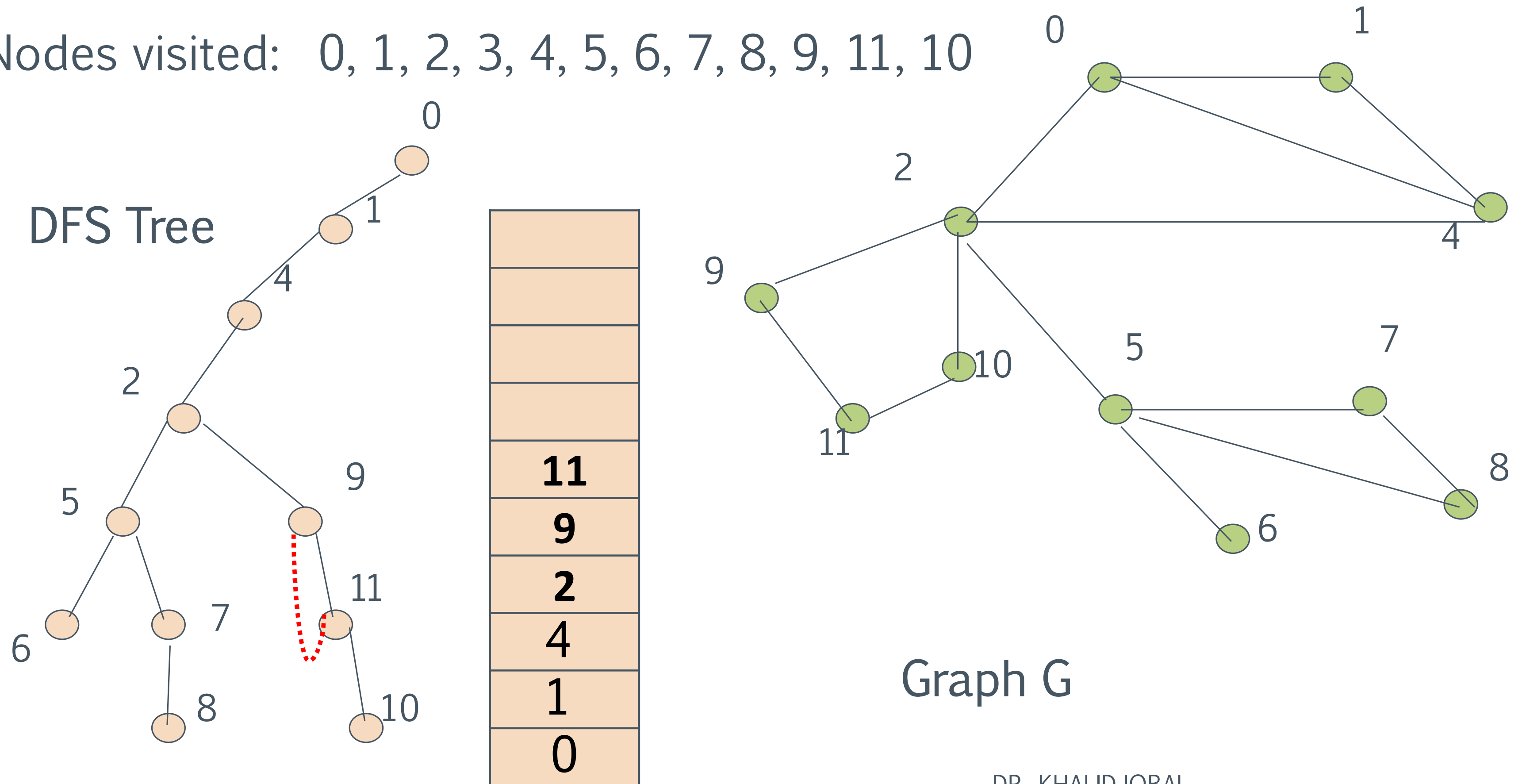
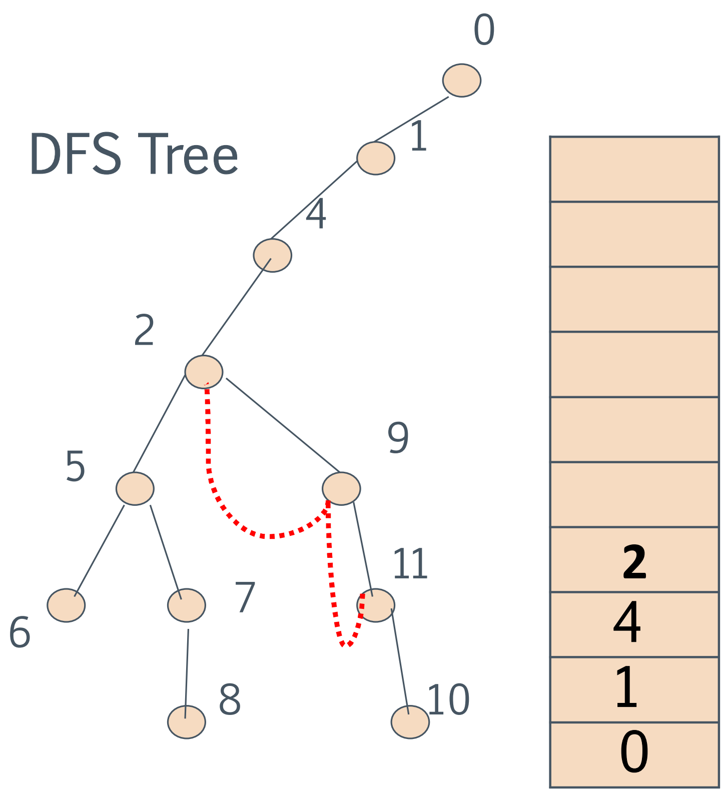
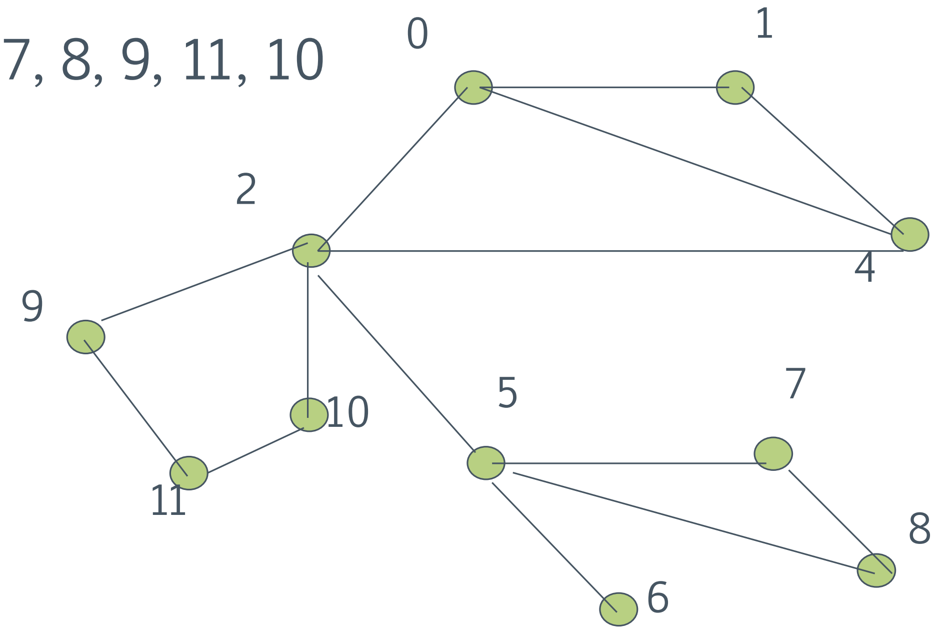


Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10

DFS Tree

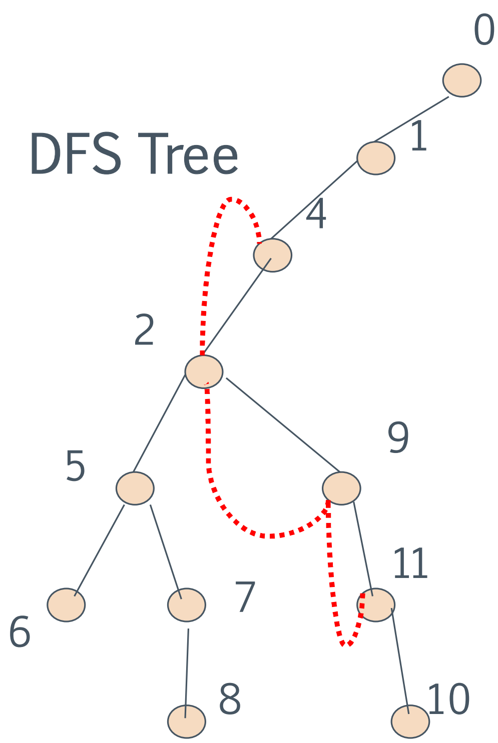


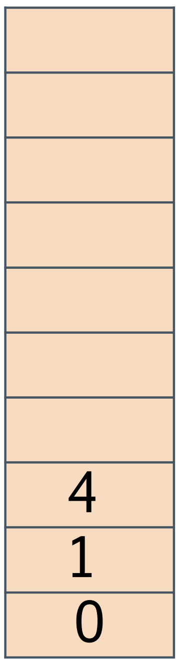


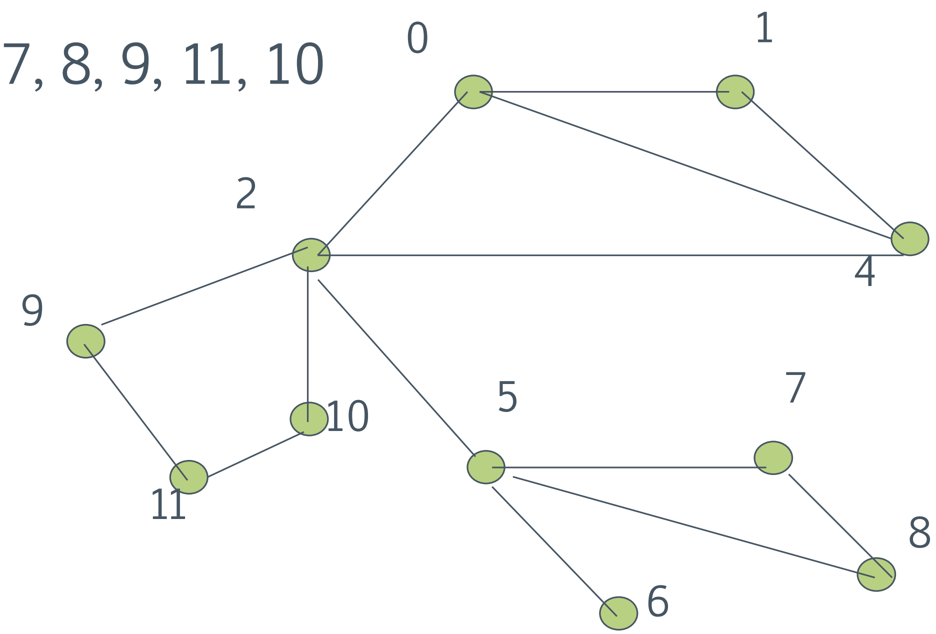
Graph G

Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10



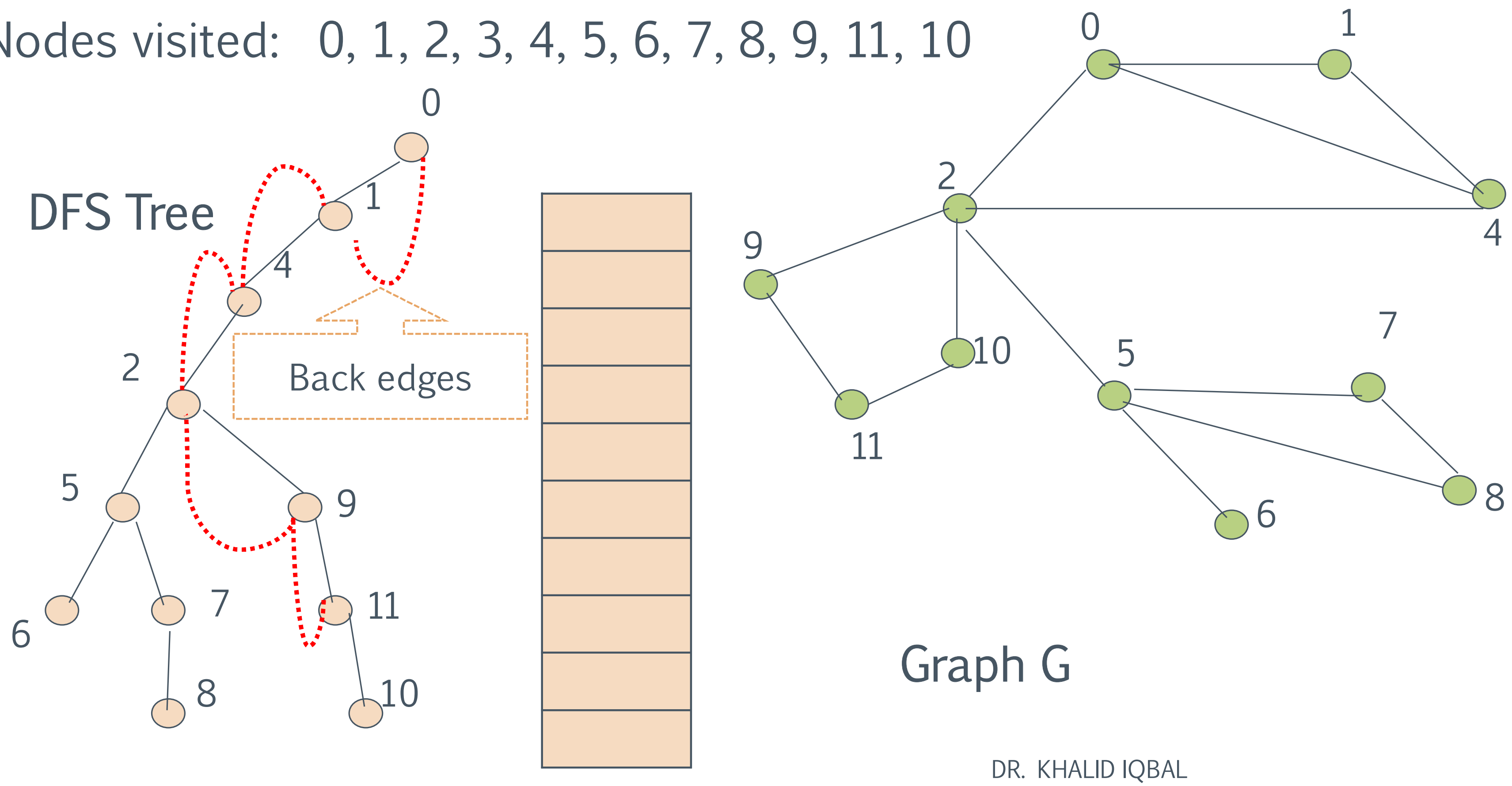




Graph G

Illustration of DFS: Stack Operation

Nodes visited: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 10



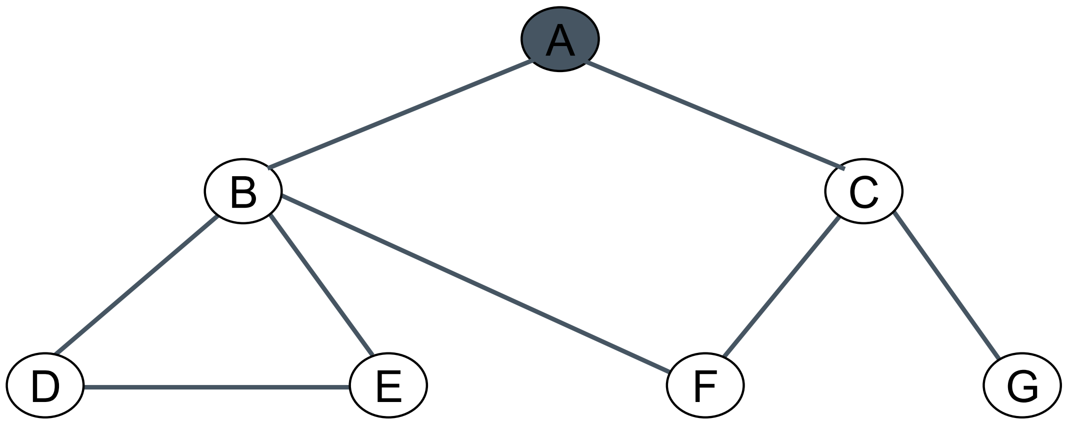
Depth-First Traversal Algorithm

- In this method, After visiting a vertex v, which is adjacent to w_1, w_2, w_3, …; Next, we visit one of v's adjacent vertices, w_1 say. Next, we visit all vertices adjacent to w_1 before coming back to w_2, etc.
- Must keep track of vertices already visited to avoid cycles.
- The method can be implemented using recursion or iteration.
- The iterative preorder depth-first algorithm is:

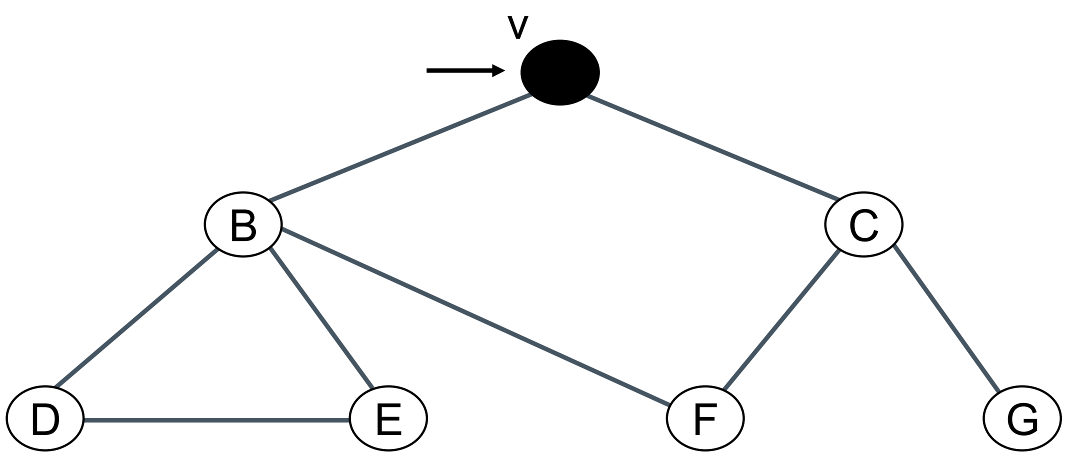
```
push the starting vertex onto the stack
while(stack is not empty){
     pop a vertex off the stack, call it v
     if v is not already visited, visit it
     push vertices adjacent to v, not visited, onto the stack
 }
```

- Note: Adjacent vertices can be pushed in any order; but to obtain a unique traversal, we will push them in reverse alphabetical order.

Depth-First Search: Example

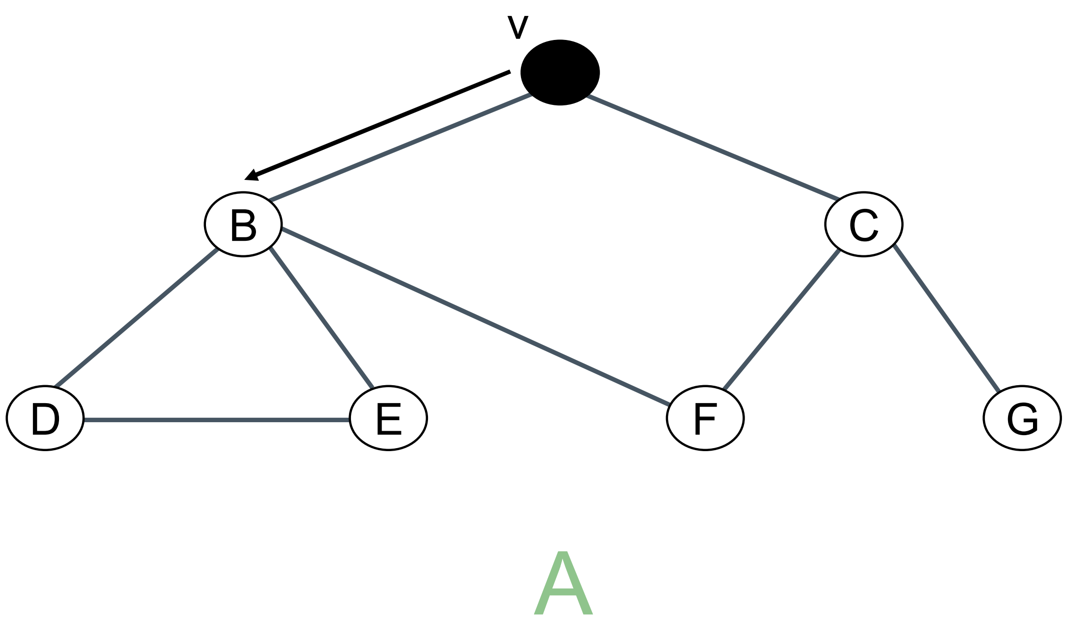


Depth-First Search: Example

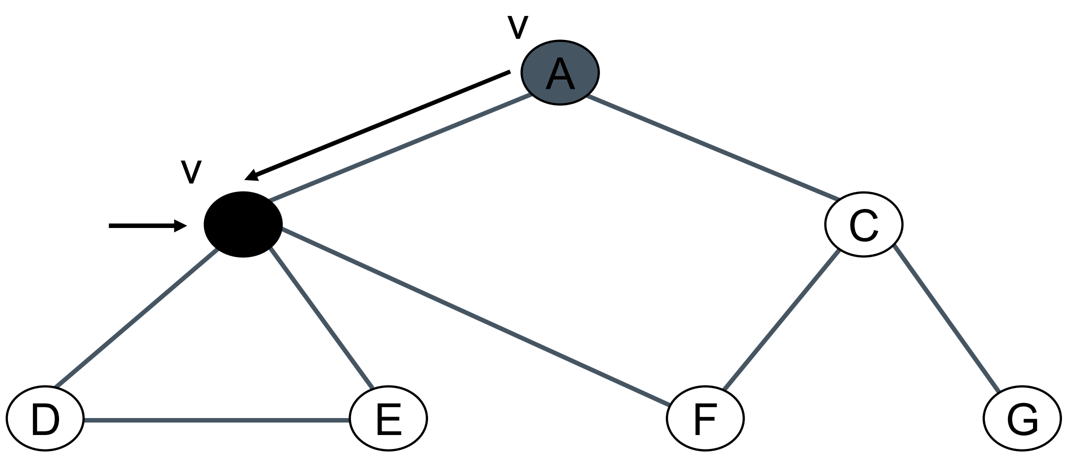


A

Depth-First Search: Example

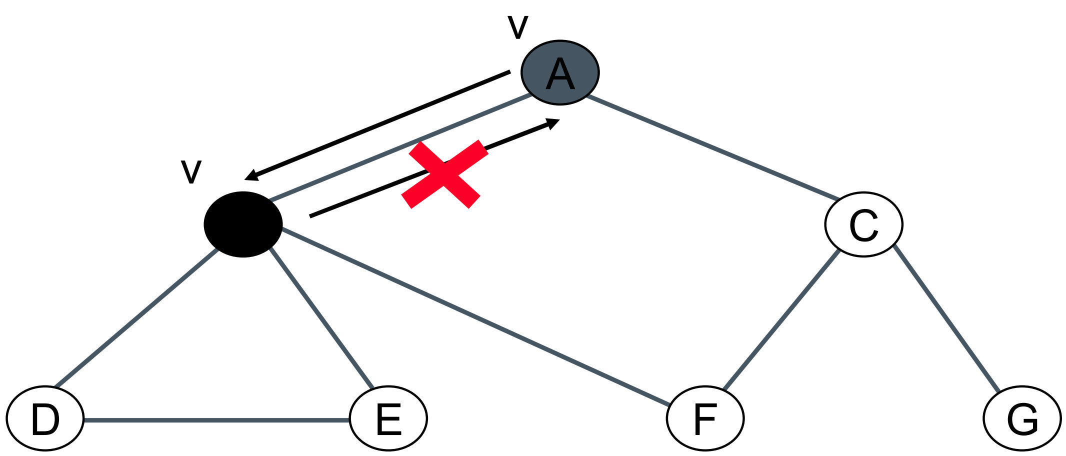


Depth-First Search: Example



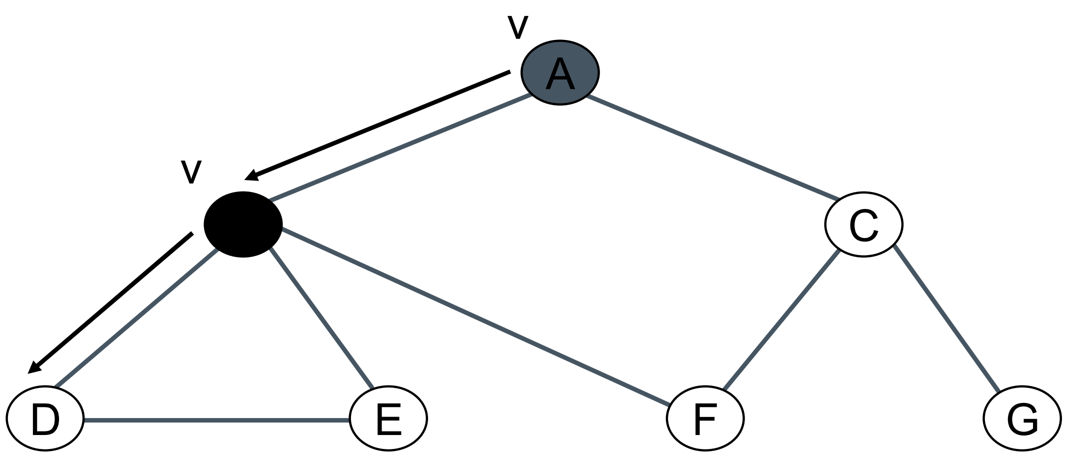
A B

Depth-First Search: Example



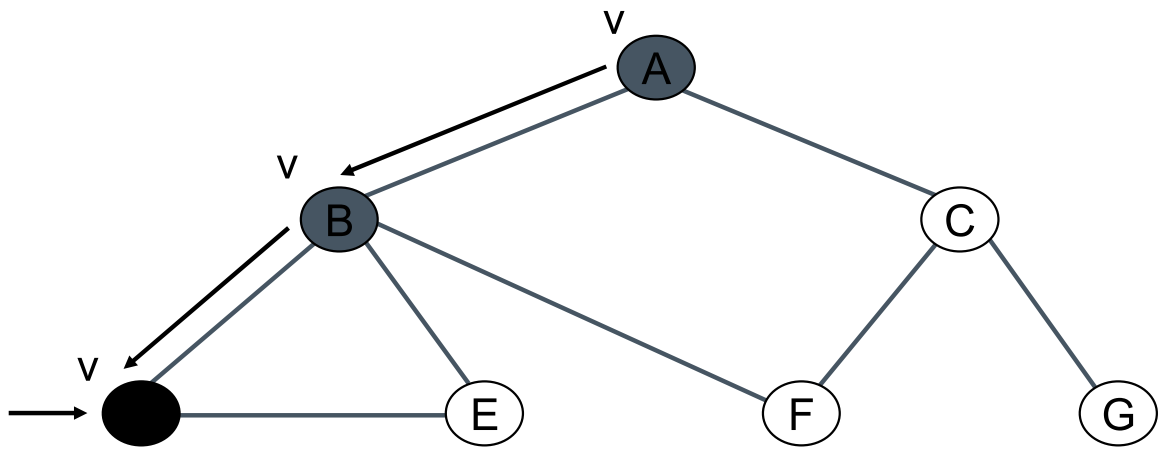
A B

Depth-First Search: Example



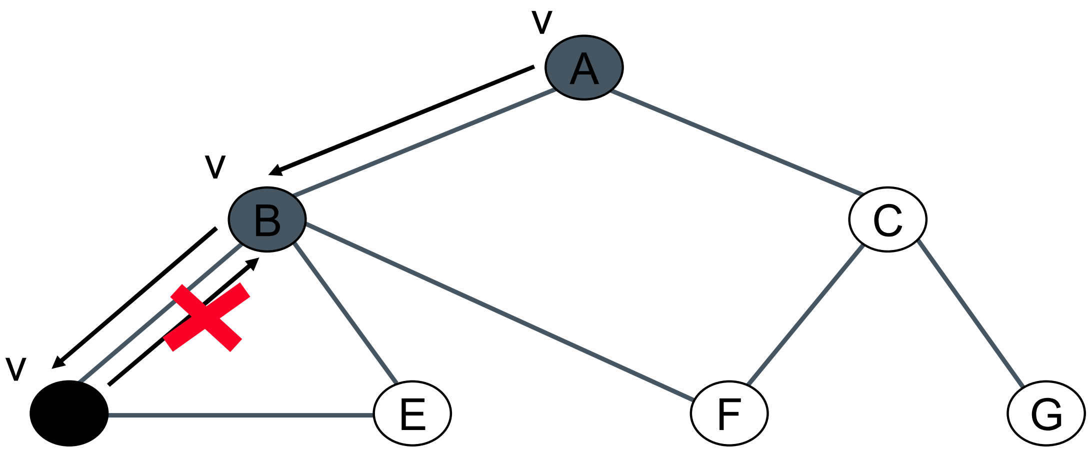
A B

Depth-First Search: Example



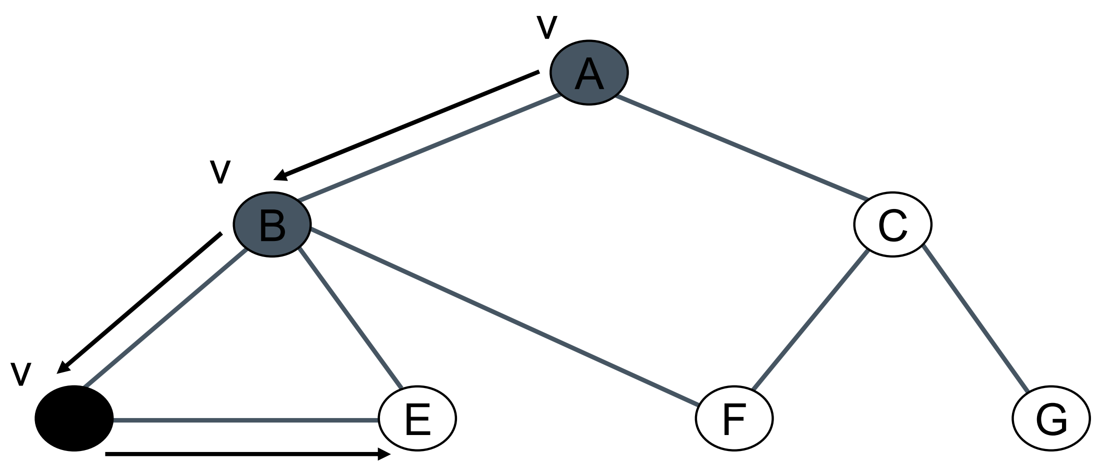
A B D

Depth-First Search: Example



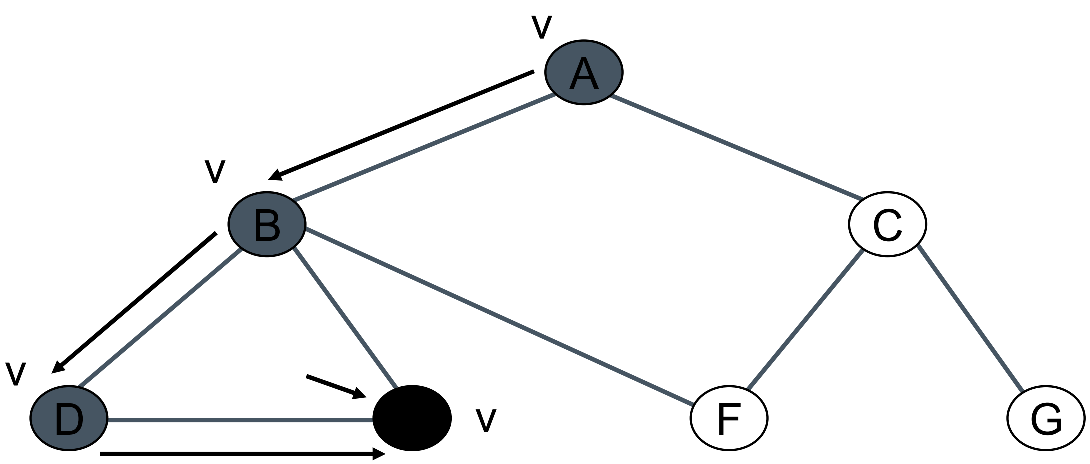
A B D

Depth-First Search: Example



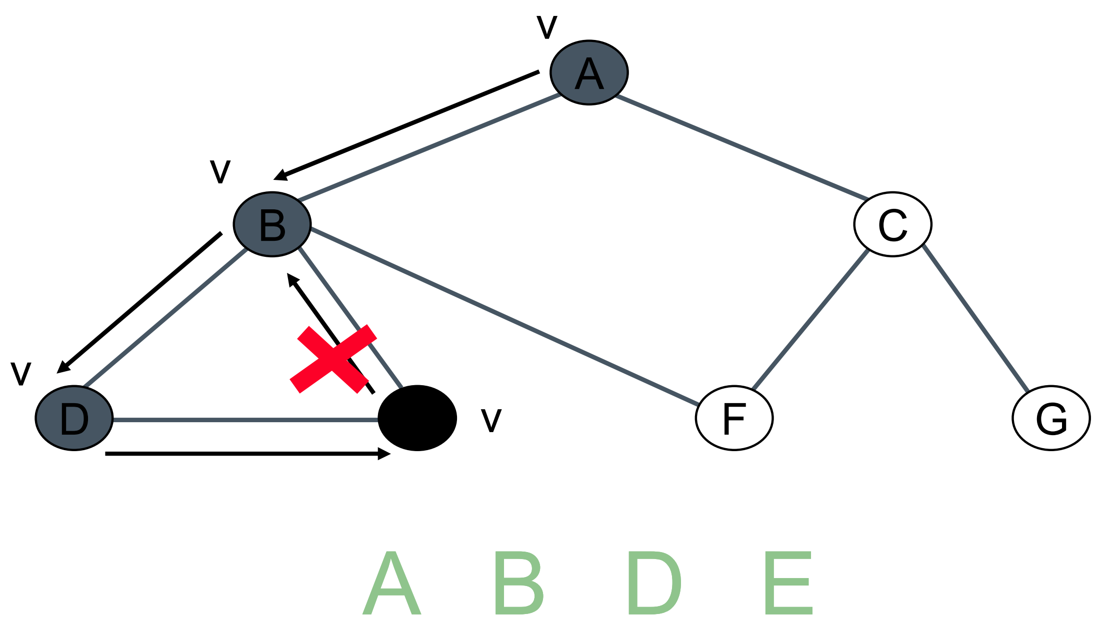
A B D

Depth-First Search: Example

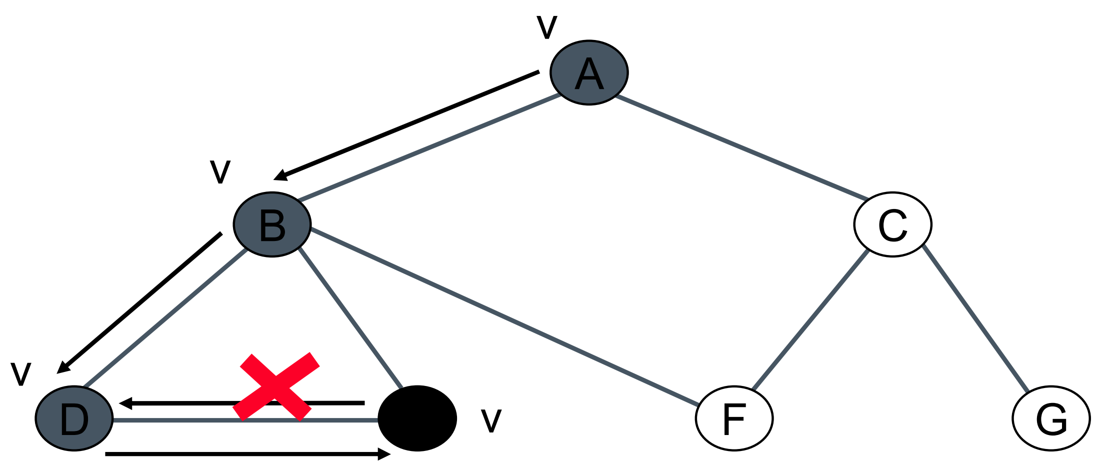


A B D E

Depth-First Search: Example

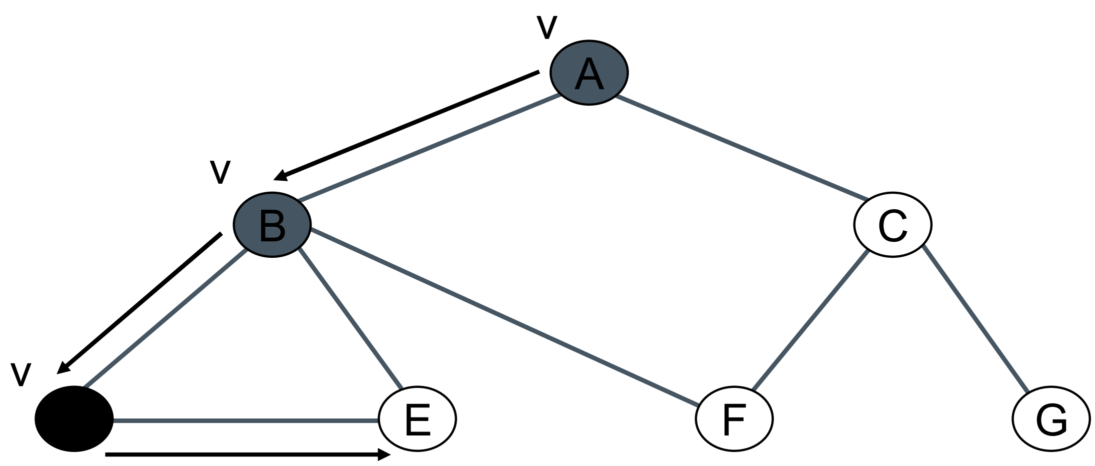


Depth-First Search: Example



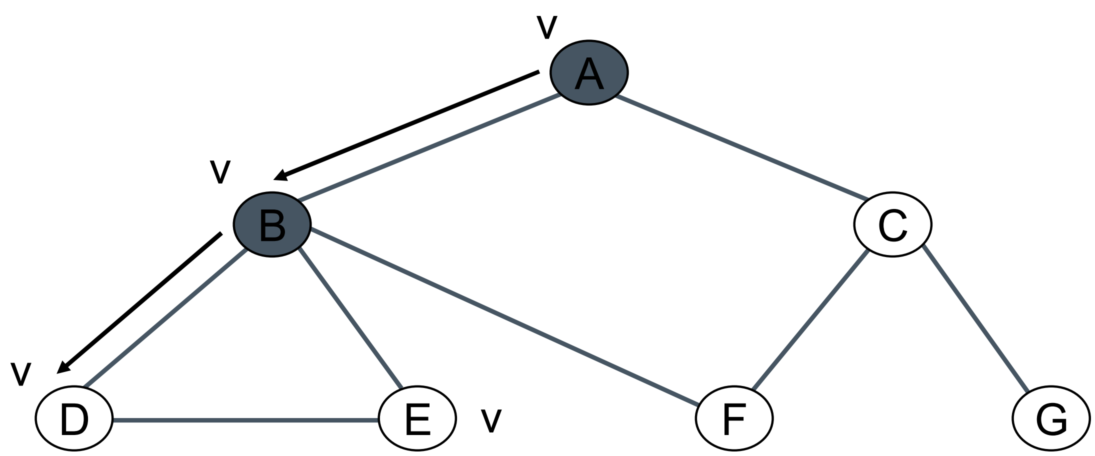
A B D E

Depth-First Search: Example



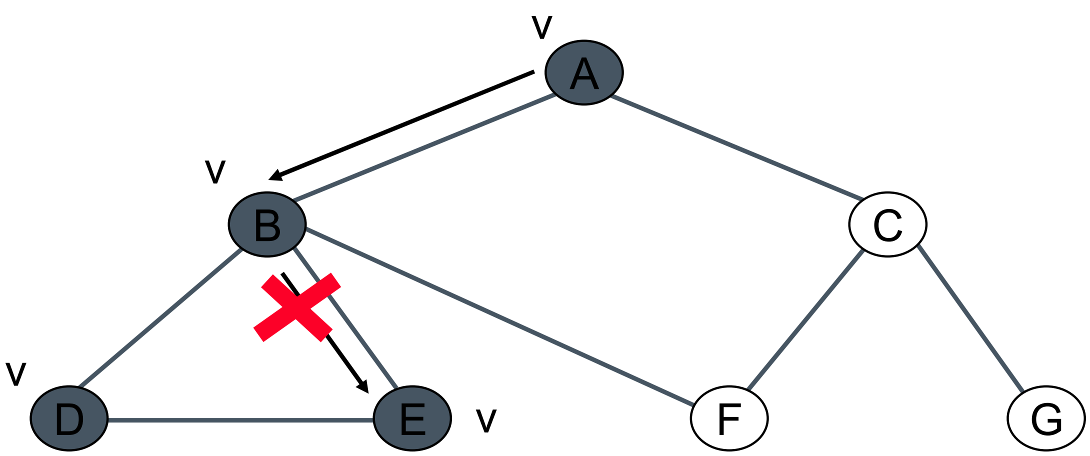
A B D E

Depth-First Search: Example



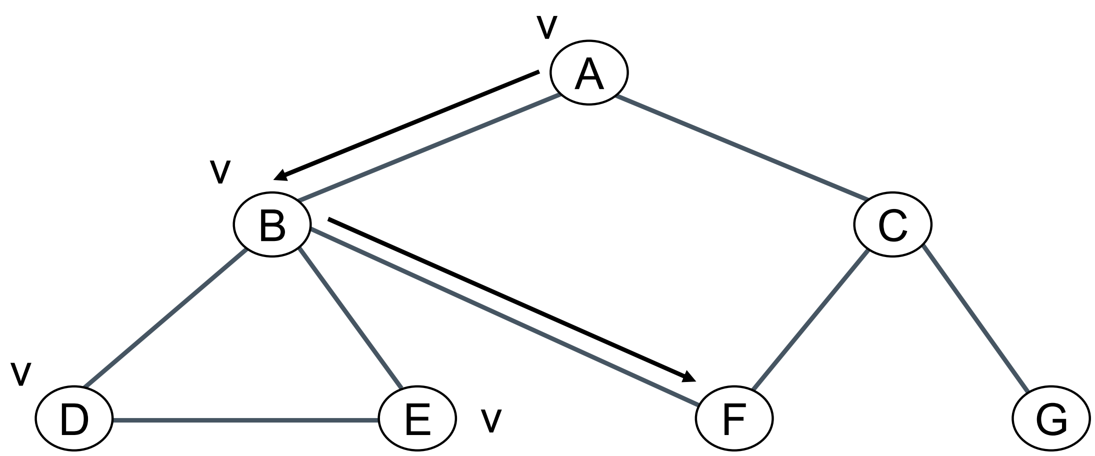
A B D E

Depth-First Search: Example



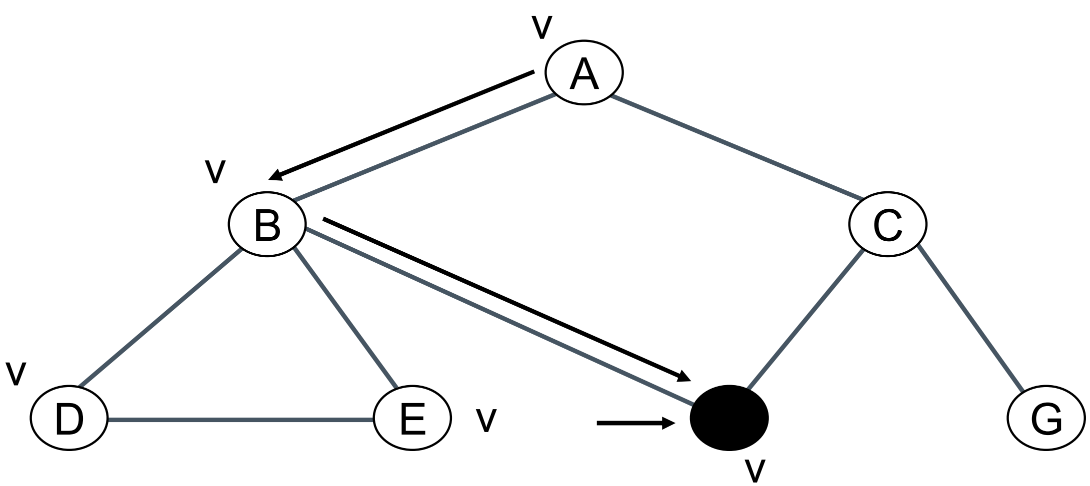
A B D E

Depth-First Search: Example



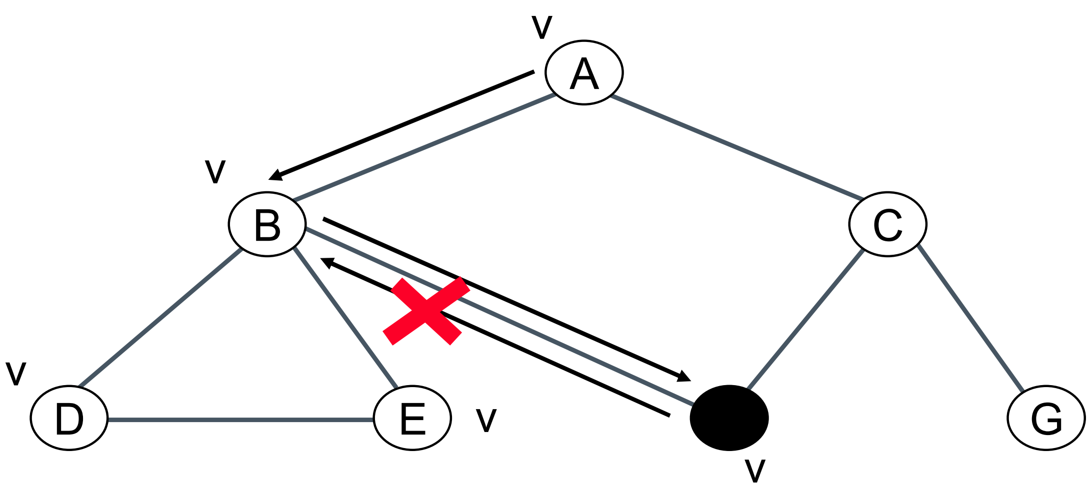
A B D E

Depth-First Search: Example



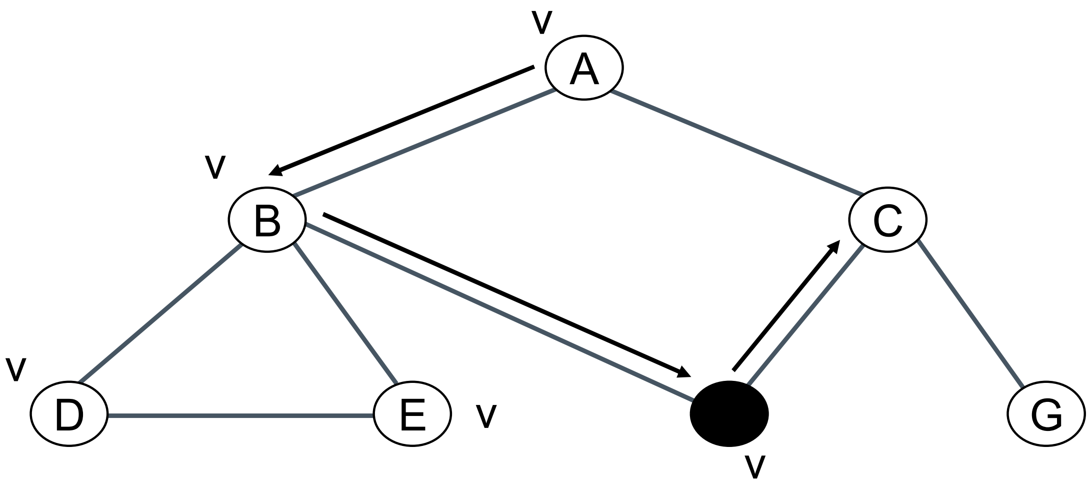
A B D E F

Depth-First Search: Example



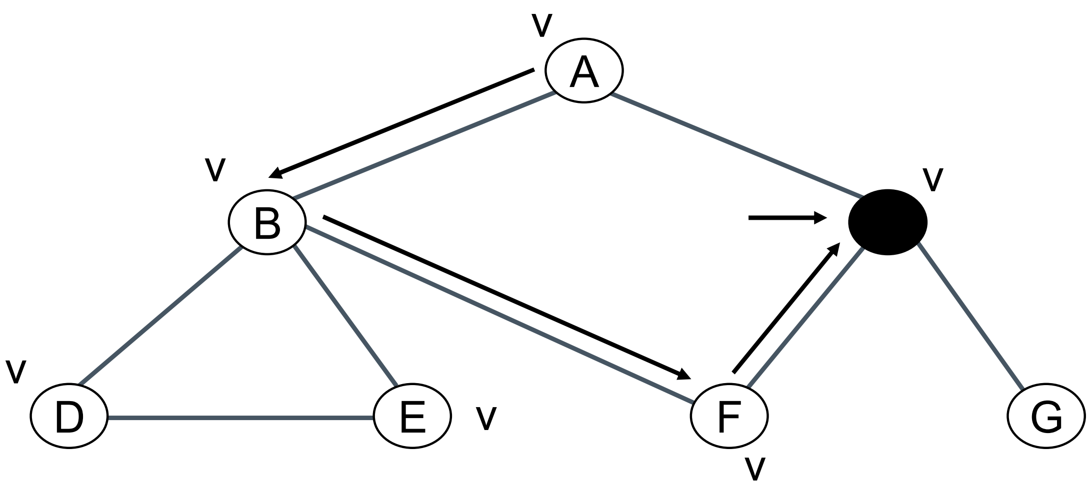
A B D E F

Depth-First Search: Example



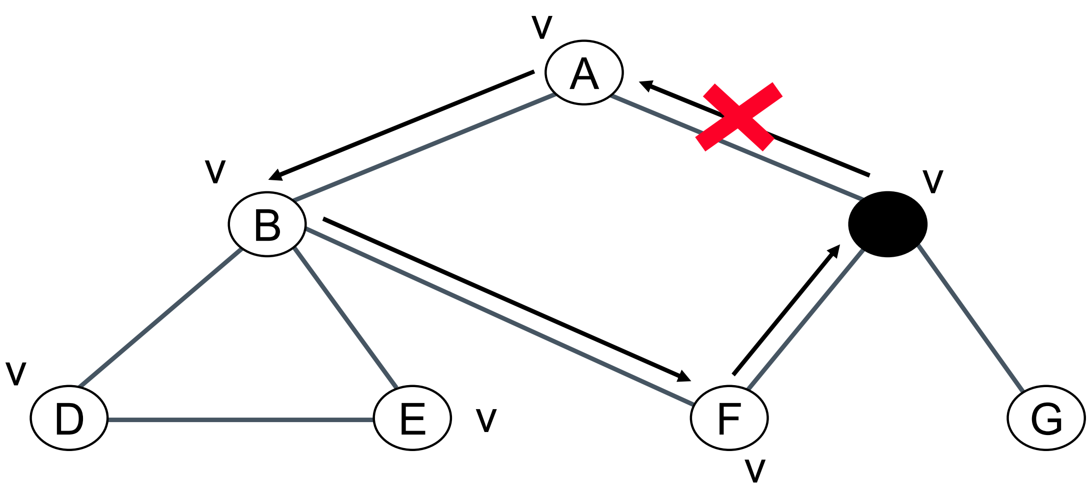
A B D E F

Depth-First Search: Example



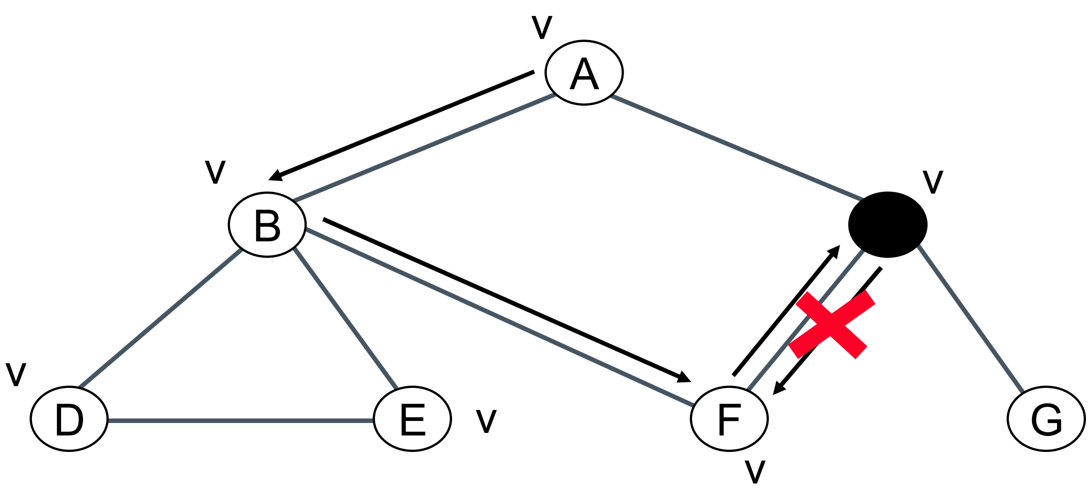
A B D E F C

Depth-First Search : Example



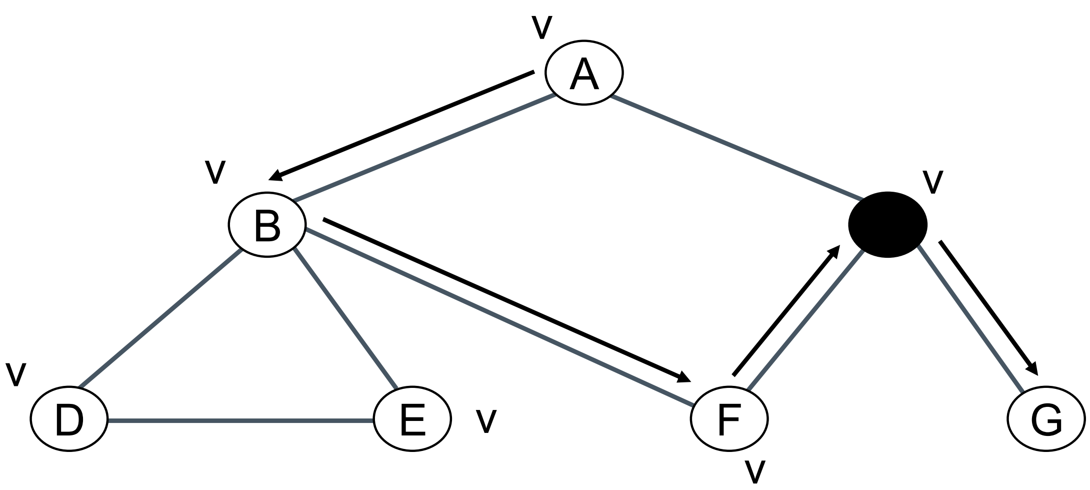
A B D E F C

Depth-First Search : Example



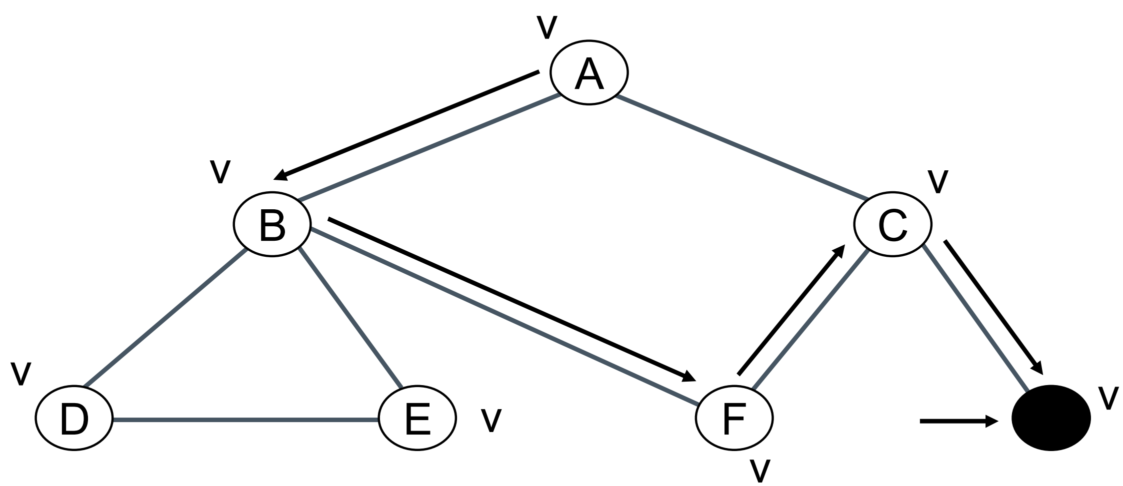
A B D E F C

Depth-First Search : Example



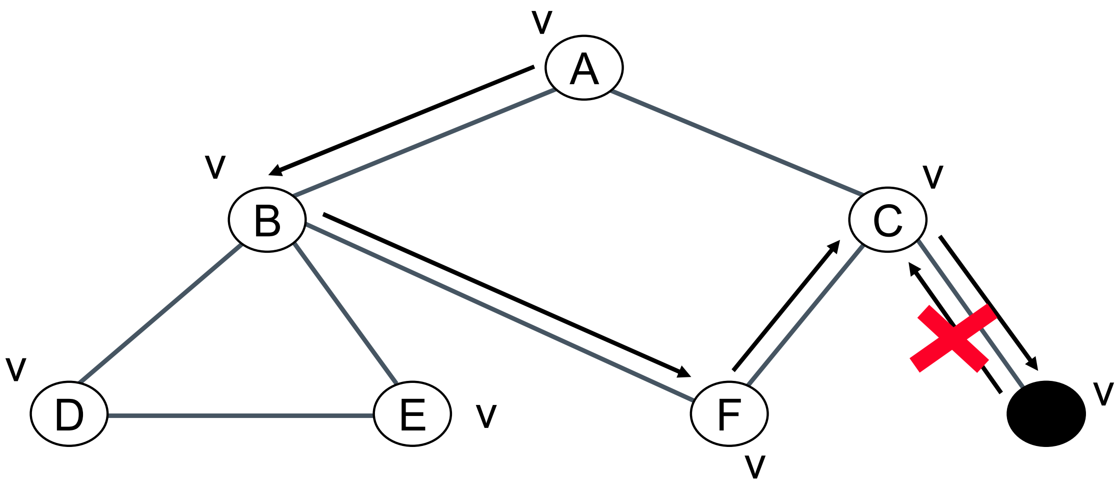
A B D E F C

Depth-First Search : Example



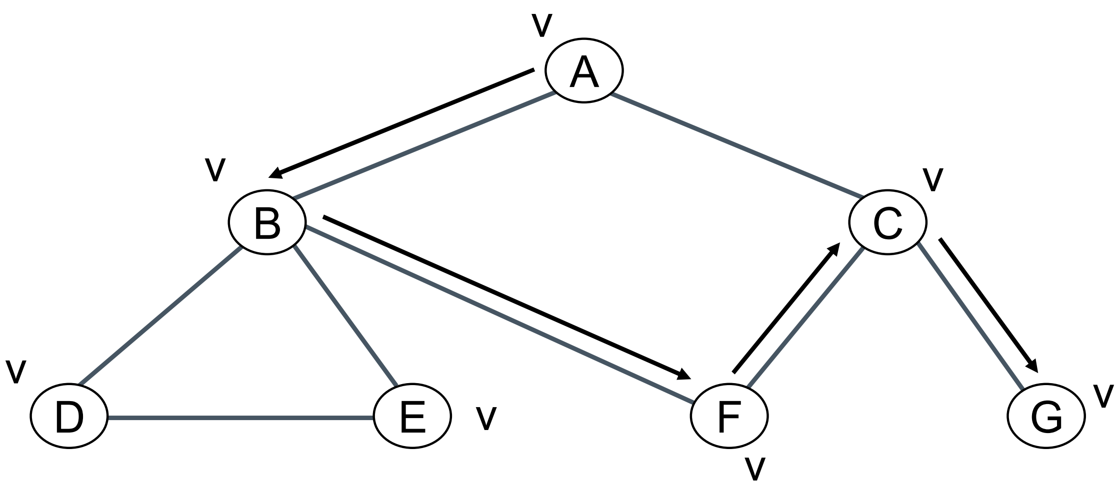
A B D E F C G

Depth-First Search



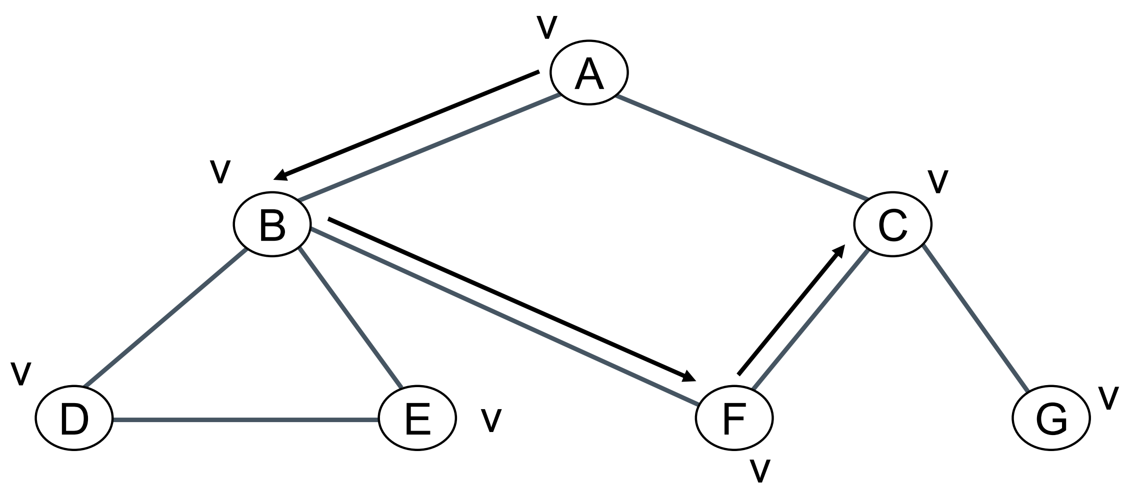
A B D E F C G

Depth-First Search : Example



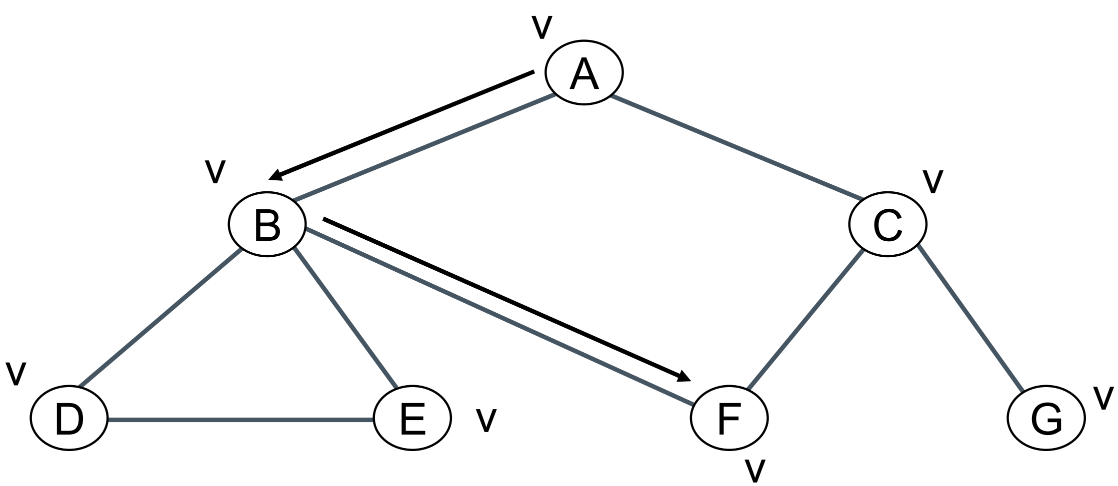
A B D E F C G

Depth-First Search : Example



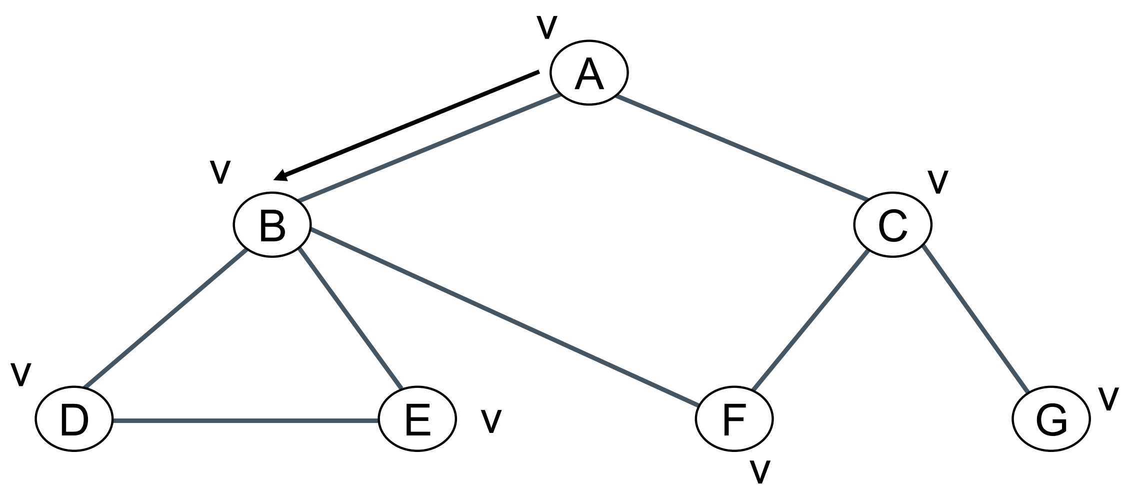
A B D E F C G

Depth-First Search : Example



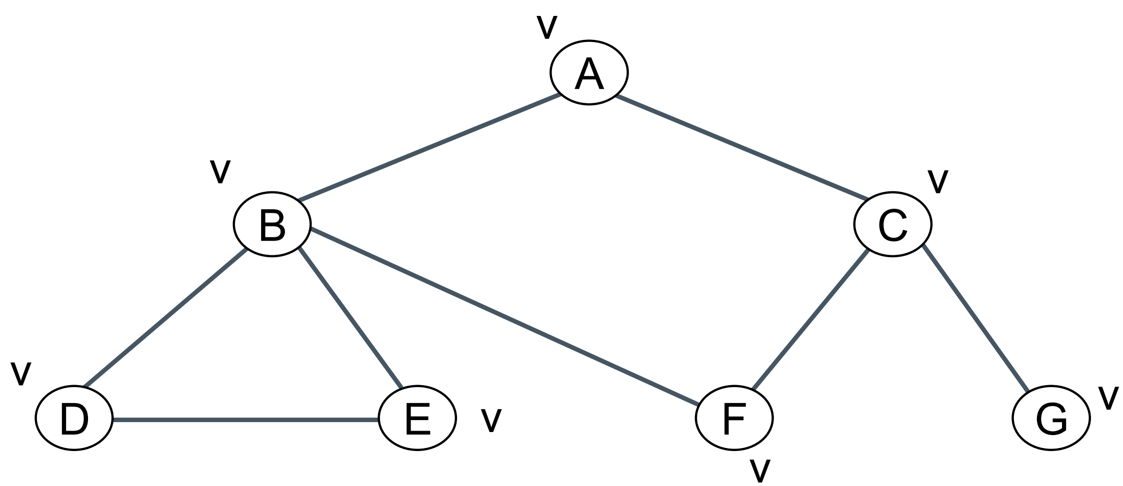
A B D E F C G

Depth-First Search : Example



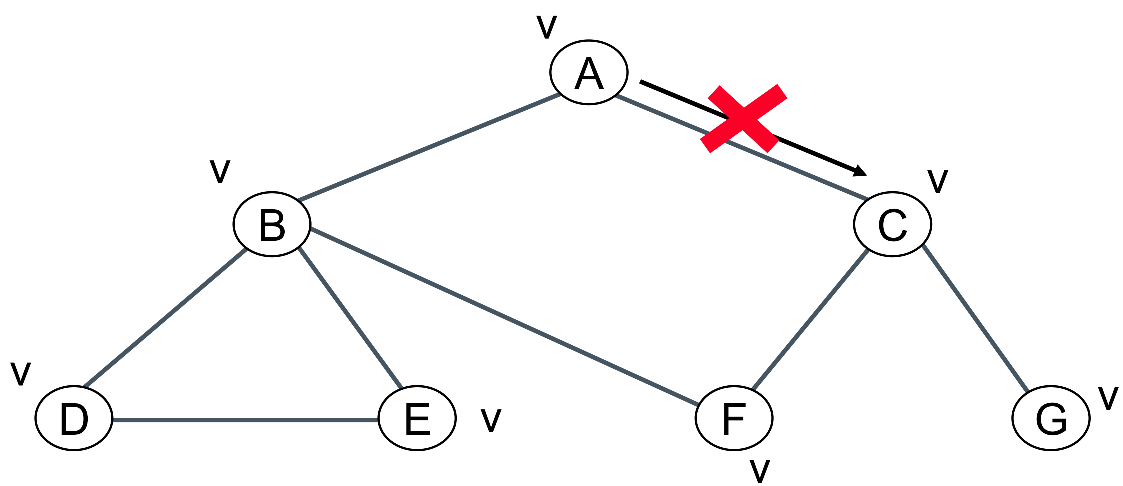
A B D E F C G

Depth-First Search : Example



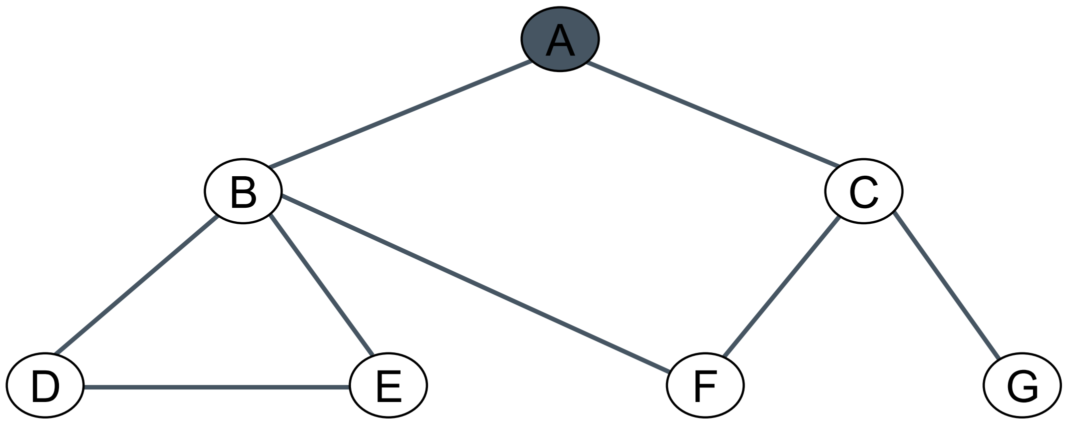
A B D E F C G

Depth-First Search : Example



A B D E F C G

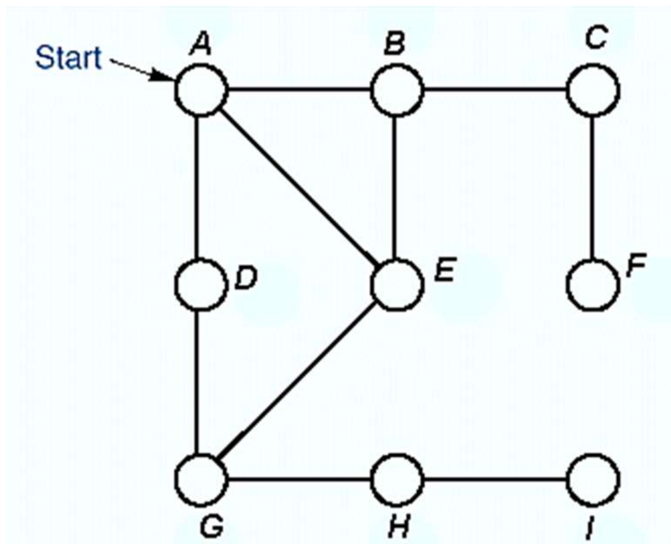
Depth-First Search : Example



A B D E F C G

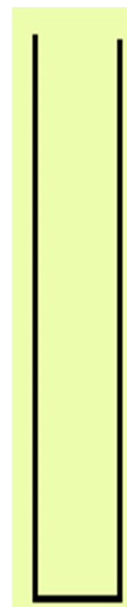
Example

› Demonstrates depth-first traversal using an explicit stack.



Order of Traversal

1	2	3	4	5	6	7	8	9
A	B	C	F	E	G	D	H	I



Stack

Thank You!!!

Have a good day