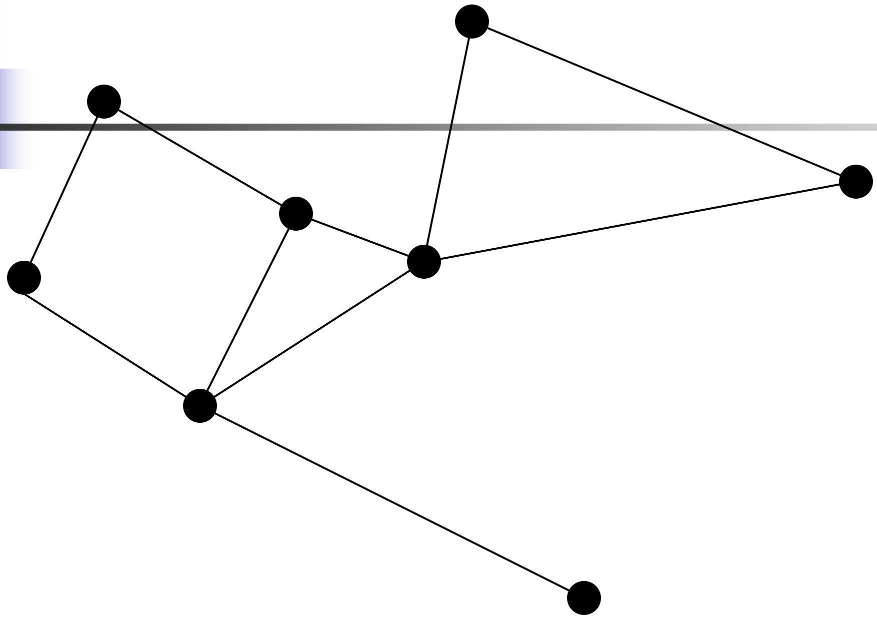
# Graph Theory



One of these!

# Varying Applications (examples)

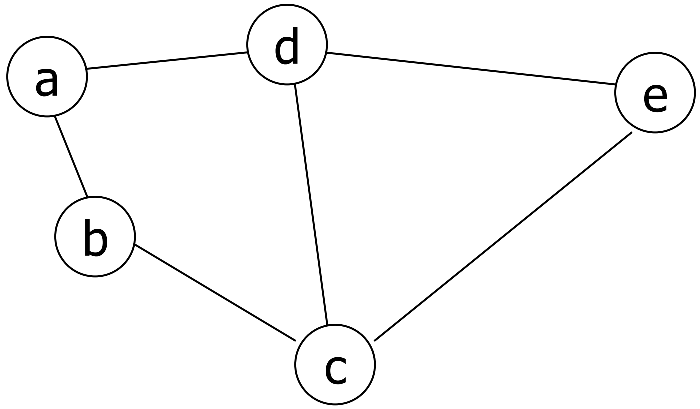
- Computer networks
- Distinguish between two chemical compounds with the same molecular formula but different structures
- Solve shortest path problems between cities
- Scheduling exams and assign channels to television stations

# Topics Covered

- **Lecture 1:** Different Types of Graphs, Subgraphs, Operations on Graphs, Walk, Path, and Circuit; Connected Graph, Disconnected Graph, and Components (algorithm);
  **Lecture 2:** Euler and Hamiltonian Graphs;
  **Lecture 3:** Planar Graph; Coloring of Graphs (algorithm); .

# Definitions - Graph

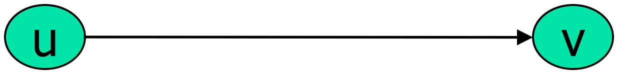
Representation: **Graph G =(V, E)** consists set of vertices denoted by V, or by V(G) and set of edges E, or E(G)



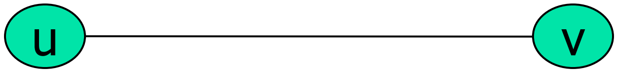

V = {a,b,c,d,e}
E = {(a,b),(a,d),(b,c),(c,d),(c,e),(d,e)}

# Definitions – Edge Type

**Directed:** **Ordered pair of vertices.** Represented as (u, v) directed from vertex u to v.

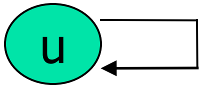



**Undirected:** **Unordered pair of vertices.** Represented as {u, v}. Disregards any sense of direction and treats both end vertices interchangeably.



# Definitions – Edge Type

- **Loop:** A loop is an edge whose endpoints are equal i.e., an edge joining a vertex to it self is called a loop. Represented as {u, u} = {u}

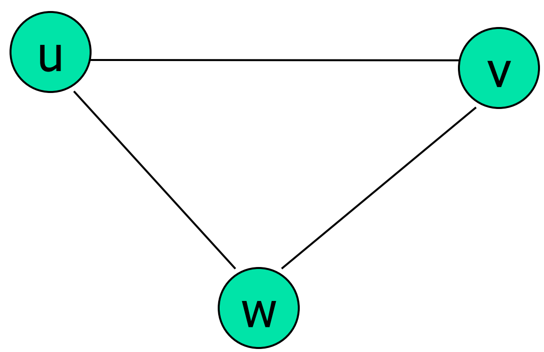



- **Multiple Edges:** Two or more edges joining the same pair of vertices.

# Definitions – Graph Type

**Simple (Undirected) Graph:** consists of V, a nonempty set of vertices, and E, a set of unordered pairs of distinct elements of V called edges (undirected)

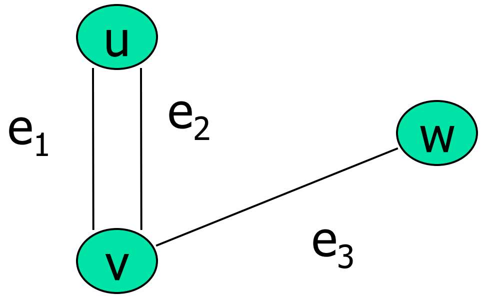
Representation Example: G(V, E), V = {u, v, w}, E = {{u, v}, {v, w}, {u, w}}



# Definitions – Graph Type

**Multigraph:** Graphs that may have multiple edges connecting the same vertices are called multigraphs

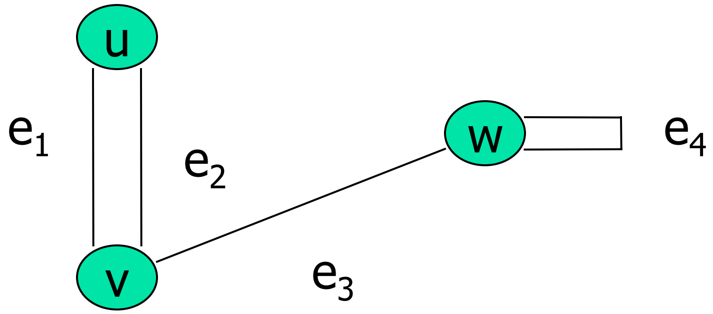
Representation Example: $V = \{u, v, w\}$, $E = \{e_1, e_2, e_3\}$



# Definitions – Graph Type

**Pseudograph:** G(V,E), consists of set of vertices V, set of Edges E and may include loops and possibly multiple edges connecting the same pair of vertices

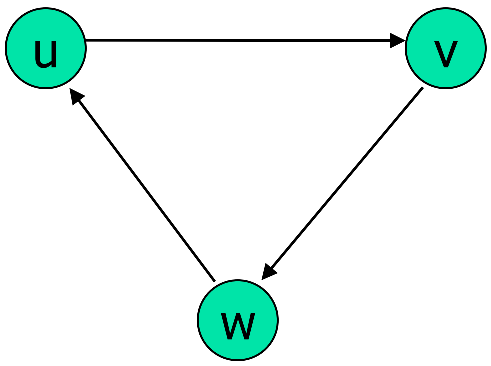
Representation Example: V = {u, v, w}, E = {$e_1$, $e_2$, $e_3$, $e_4$}



# Definitions – Graph Type

**Directed Graph:** G(V, E), set of vertices V, and set of Edges E, that are ordered pair of elements of V (directed edges)

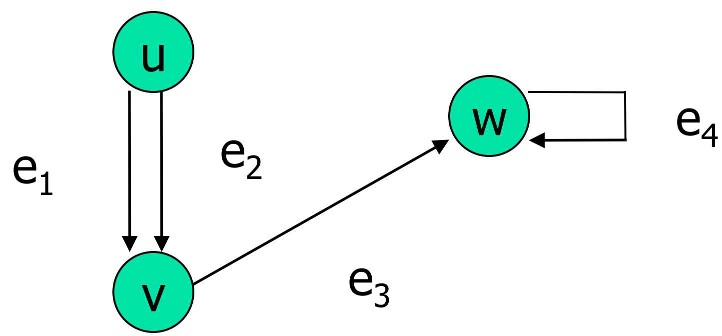
Representation Example: G(V, E), V = {u, v, w}, E = {(u, v), (v, w), (w, u)}



# Definitions – Graph Type

**Directed Multigraph:** G(V,E), consists of set of vertices V, set of Edges E and a function f from E to {{u, v}| u, v  V}. The edges e1 and e2 are multiple edges if f(e1) = f(e2)

Representation Example: V = {u, v, w}, E = {$e_1$, $e_2$, $e_3$, $e_4$}



# Definitions – Graph Type

| Type | Edges | Multiple Edges Allowed ? | Loops Allowed ? |
|---|---|---|---|
| **Simple Graph** | undirected | No | No |
| **Multigraph** | undirected | Yes | No |
| **Pseudograph** | undirected | Yes | Yes |
| **Directed Graph** | directed | No | Yes |
| **Directed Multigraph** | directed | Yes | Yes |

- Draw graph models, stating the type of graph used, to represent airline routes where every day there are four flights from Boston to Newark, two flights from Newark to Boston, three flights from Newark to Miami, two flights from Miami to Newark, one flight from Newark to Detroit, two flights from Detroit to Newark, three flights from Newark to Washington, two flights from Washington to Newark, and one flight from Washington to Miami, with

A)an edge between vertices representing cities that have a flight between them (in either direction).

B) an edge for each flight from a vertex representing a city where the flight begins to the vertex representing city where the flight ends.

Which type of graph can be used to model

Acquaintance graphs

Influence graphs

Round robin tournaments

Telephone call graphs

Web graphs

- Which type of graph can be used to model
- Acquaintance graphs (person – vertex, know each other- edge, undirected, no multiple edges, no loops)

Influence graphs (person-vertex, edge (a,b) – person a influences the person b, directed, no multiple edges, no loops )

Round robin tournaments (team-vertex, edge (a,b) – team a beats team b, directed, no multiple edges, no loops )

Telephone call graphs (tel. no.-vertex, edge (a,b) – call starts at tel no a and ends at tel no b, directed multi graph, no loops )

Web graphs (web page-vertex, edge (a,b) – starts at web page a and ends at web page b, directed graph)

# **Terminology** – Undirected graphs

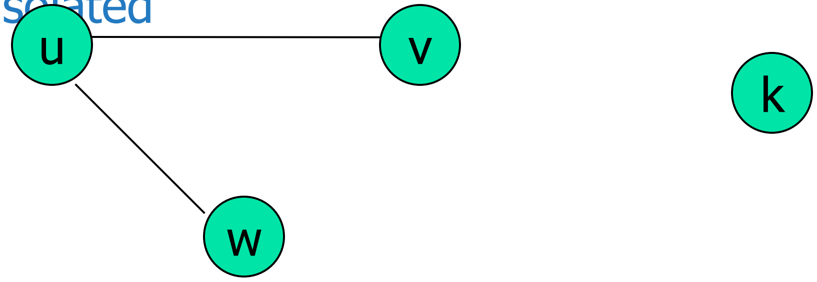
- u and v are **adjacent** if {u, v} is an edge, e is called **incident** with u and v. u and v are called **endpoints** of {u, v}

- **Degree of Vertex (deg (v)):** the number of edges incident on a vertex. A loop contributes twice to the degree (why?).

- **Pendant Vertex:** deg (v) =1

- **Isolated Vertex:** deg (v) = 0

**Representation Example:** For V = {u, v, w} , E = { {u, w}, {u, w}, (u, v) }, deg (u) = 2, deg (v) = 1, deg (w) = 1, deg (k) = 0, w and v are pendant , k is isolated



# **Terminology** – Directed graphs

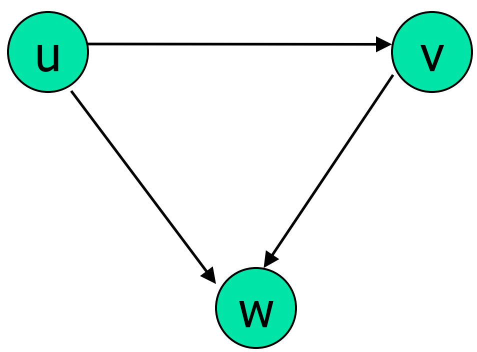
- For the edge (u, v), u is **adjacent to** v OR v is **adjacent from** u, u – **Initial vertex**, v – **Terminal vertex**

- **In-degree ($deg^-$ (u)):** number of edges for which u is terminal vertex

- **Out-degree ($deg^+$ (u)):** number of edges for which u is initial vertex

*Note: A loop contributes 1 to both in-degree and out-degree (why?)*

**Representation Example:** For V = {u, v, w} , E = { (u, w), ( v, w), (u, v) }, $deg^-$ (u) = 0, $deg^+$ (u) = 2, $deg^-$ (v) = 1,
$deg^+$ (v) = 1, and $deg^-$ (w) = 2, $deg^+$ (w) = 0



# Theorems: Undirected Graphs

## Theorem 1

**THE HANDSHAKING THEOREM** Let $G = (V, E)$ be an undirected graph with $m$ edges. Then

$$2m = \sum_{v \in V} \deg(v).$$

(Note that this applies even if multiple edges and loops are present.)

(why?) Every edge connects 2 vertices

# Theorems: Undirected Graphs

<u>**Theorem 2:**</u>

An undirected graph has even number of vertices with odd degree

*Proof:* Let $V_1$ and $V_2$ be the set of vertices of even degree and the set of vertices of odd degree, respectively, in an undirected graph $G = (V, E)$ with $m$ edges. Then

$$2m = \sum_{v \in V} \deg(v) = \sum_{v \in V_1} \deg(v) + \sum_{v \in V_2} \deg(v).$$

Because deg*(v) is even for v ∈ V1, the first term in the right-hand side of the last equality is* even.

Furthermore, the sum of the two terms on the right-hand side of the last equality is even, because this sum is 2*m.*

*Hence, the second term in the sum is also even.*

*Because all the terms in* this sum are odd, there must be an even number of such terms.

Thus, there are an even number of vertices of odd degree

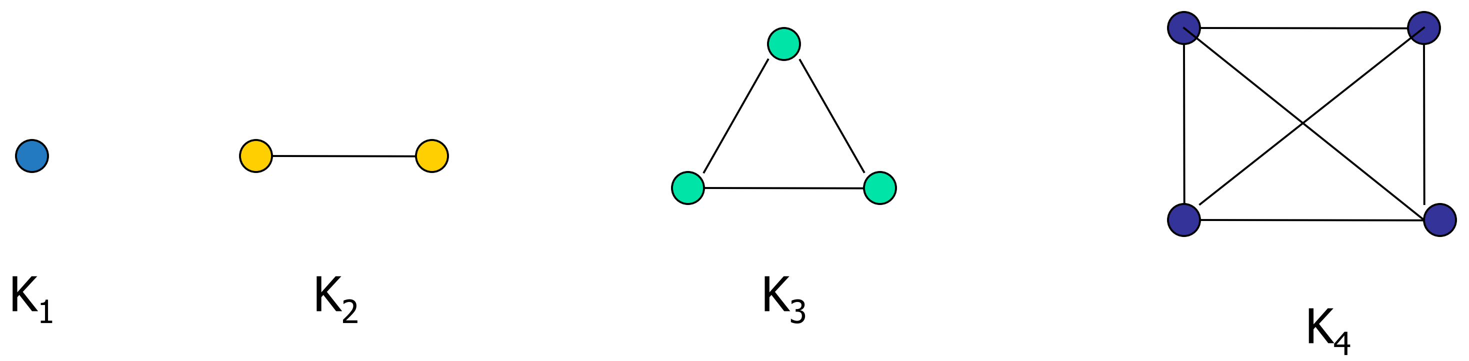
# Theorems: directed Graphs

- **Theorem 3:** $\sum$ **deg $^{+}$ (u) =** $\sum$ **deg $^{-}$ (u) = |E|**

# Simple graphs – special cases

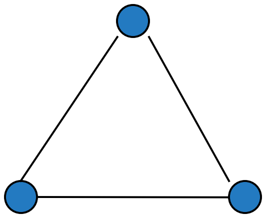
- **Complete graph:** on n vertices, $K_n$, is the simple graph that contains exactly one edge between each pair of distinct vertices.

  Representation Example: $K_1$, $K_2$, $K_3$, $K_4$

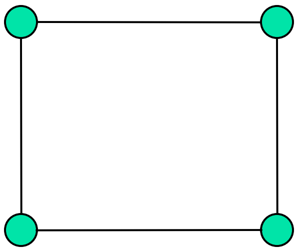


# Simple graphs – special cases

- **Cycle:** $C_n$, $n \geq 3$ consists of n vertices $v_1, v_2, v_3 \ldots v_n$ and edges $\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\} \ldots \{v_{n-1}, v_n\}, \{v_n, v_1\}$
  Representation Example: $C_3$, $C_4$



$C_3$

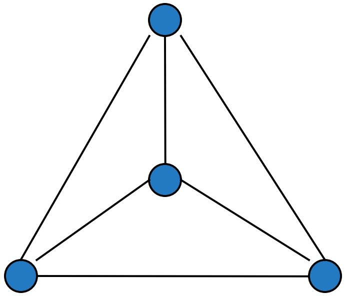


$C_4$

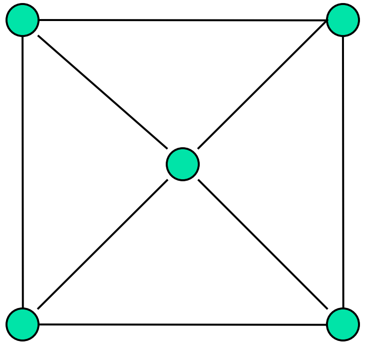
# Simple graphs – special cases

- **Wheels:** $W_n$, obtained by adding additional vertex to Cn and connecting all vertices to this new vertex by new edges.

  Representation Example: $W_3$, $W_4$



$W_3$



$W_4$

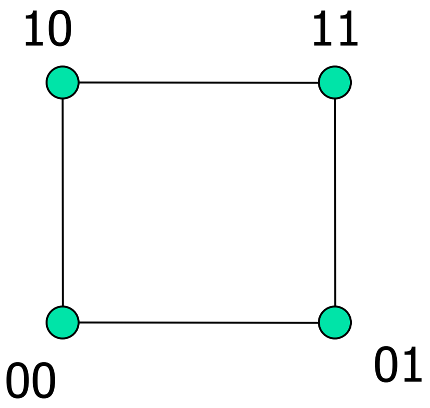
# Simple graphs – special cases

- **N-cubes:** $Q_n$, vertices represented by $2^n$ bit strings of length n. Two vertices are adjacent if and only if the bit strings that they represent differ by exactly one bit positions

  Representation Example: $Q_1$, $Q_2$

  $Q_3$ ?


$Q_1$



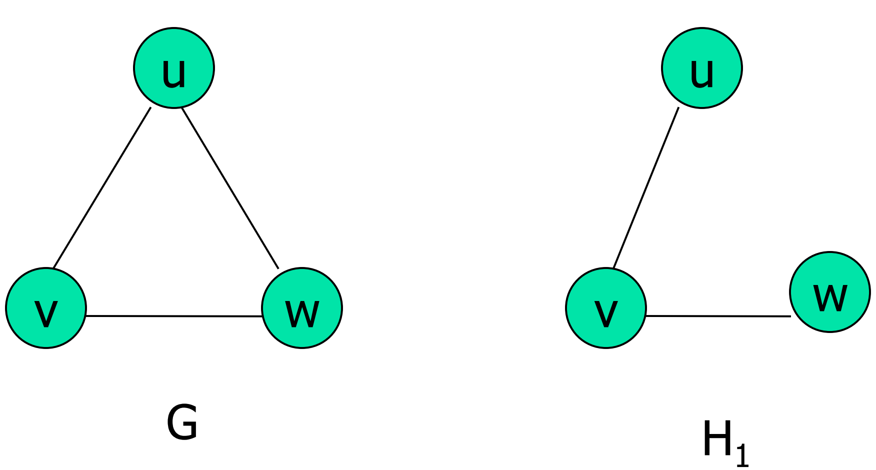

$Q_2$

# Subgraphs

- A subgraph of a graph $G = (V, E)$ is a graph $H = (V', E')$ where $V'$ is a subset of $V$ and $E'$ is a subset of $E$

  Application example: solving sub-problems within a graph

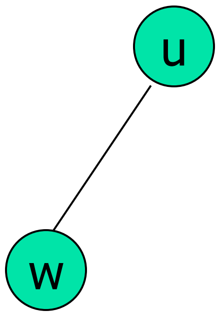
  Representation example: $V = \{u, v, w\}$, $E = (\{u, v\}, \{v, w\}, \{w, u\})$, $H_1$, $H_2$



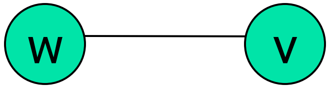
# Subgraphs

- G = G1 U G2 wherein E = E1 U E2 and V = V1 U V2, G, G1 and G2 are simple graphs of G

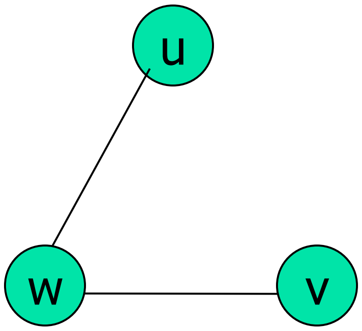
Representation example: V1 = {u, w}, E1 = {{u, w}}, V2 = {w, v}, E1 = {{w, v}}, V = {u, v ,w}, E = {{{u, w}, {{w, v}}




G1




G2



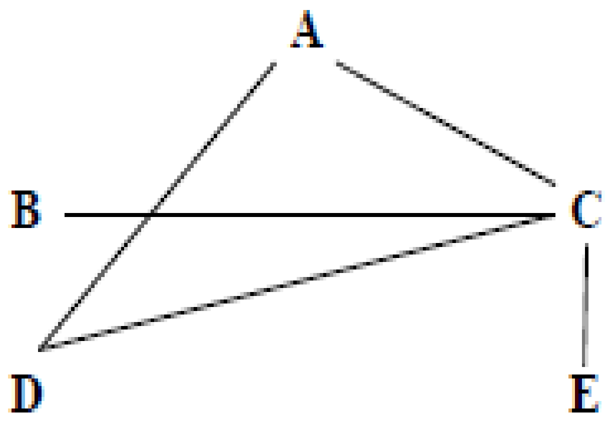

G

- Suppose that in a group of 5 people: A, B, C, D, and E, the following pairs of people are acquainted with each other.
- A and C
- A and D
- B and C
- C and D
- C and E

a) Draw a graph G to represent this situation.

b) List the vertex set, and the edge set, using set notation. In other words, show sets V and E for the vertices and edges, respectively, in G = {V, E}.

# Answer



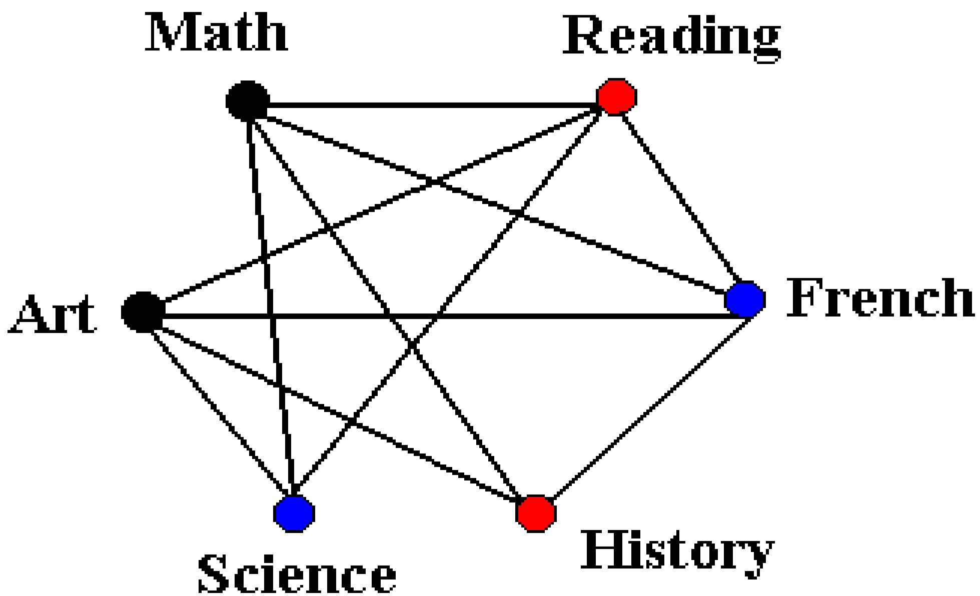

V = {A, B, C, D, E}
E = {(A, C), (A, D), (B, C), (C, D), (C, E)}

- In scheduling final exams for summer school at Central High, six different tests have to be given to seven students. The table below shows the exams that each of the students must take.

- Draw a graph that illustrates which exams have students in common with other exams.

| Exams | Amy | Ben | Charles | Debra | Ed | Frank | Georgia |
|---|---|---|---|---|---|---|---|
| Math | X | | X | | X | | X |
| Art | | X | | X | | X | |
| Science | X | X | | | | | X |
| History | | | X | | | X | |
| French | | | | | X | X | |
| Reading | X | X | | X | X | | X |

**Solution:**

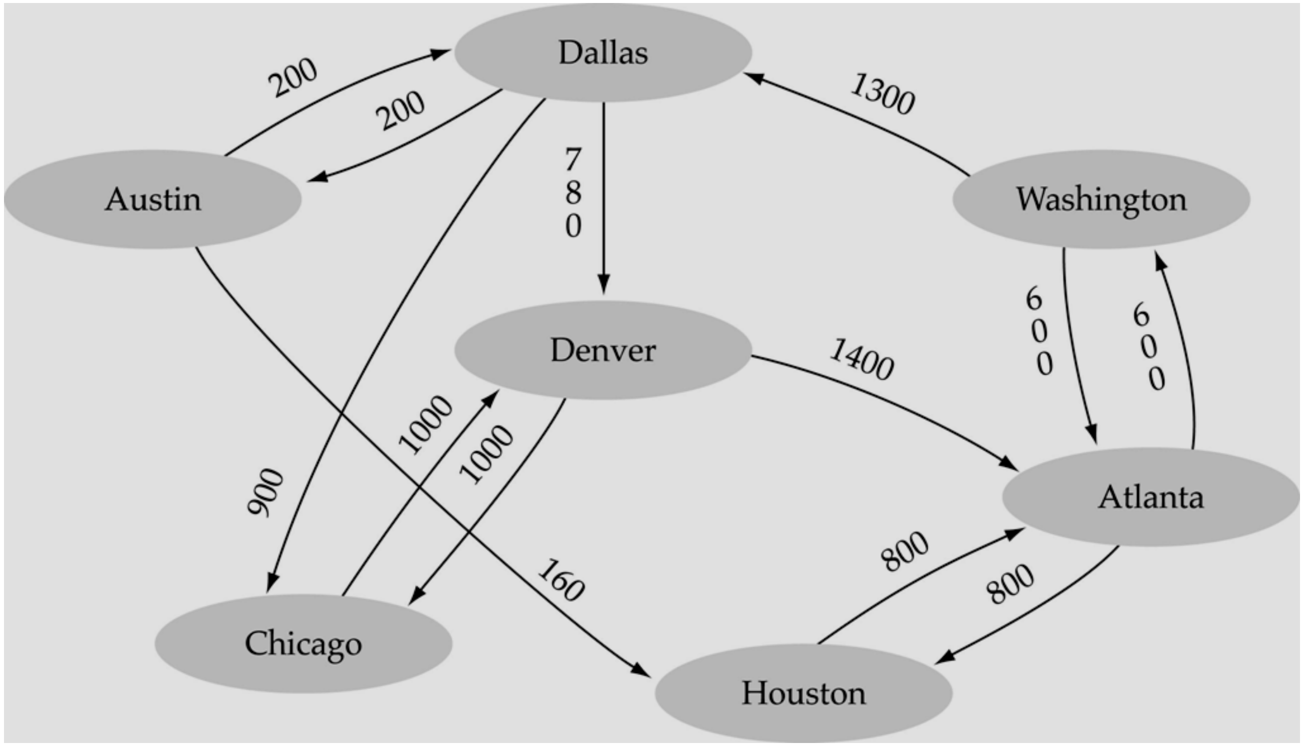


- How many more edges are there in the complete graph $K_7$ than in the complete graph $K_5$ ?

$C(7, 2) - C(5, 2) = 21 - 10 = 11$

# Graph terminology (cont.)

Weighted graph: a graph in which each edge carries a value



# Representation

- **Incidence (Matrix):** Most useful when information about edges is more desirable than information about vertices.

- **Adjacency (Matrix/List):** Most useful when information about the vertices is more desirable than information about the edges. These two representations are also most popular since information about the vertices is often more desirable than edges in most applications

# Representation- Incidence Matrix

- G = (V, E) be an unditected graph. Suppose that $v_1$, $v_2$, $v_3$, …, $v_n$ are the vertices and $e_1$, $e_2$, …, $e_m$ are the edges of G. Then the incidence matrix with respect to this ordering of V and E is the $n x m$ matrix $M = [m_{ij}]$, where

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise} \end{cases}$$

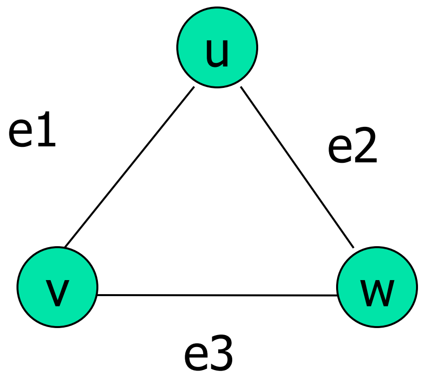
Can also be used to represent :

**Multiple edges:** by using columns with identical entries, since these edges are incident with the same pair of vertices

**Loops:** by using a column with exactly one entry equal to 1, corresponding to the vertex that is incident with the loop

# Representation- Incidence Matrix

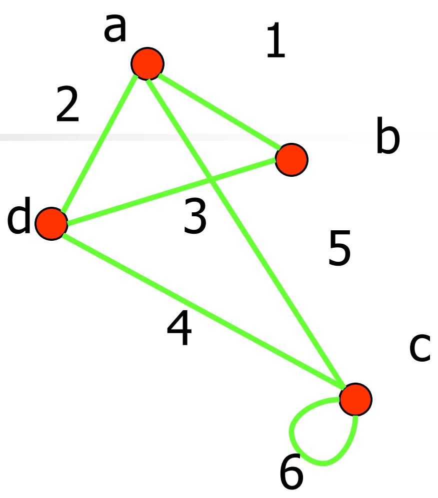
- Representation Example: G = (V, E)




| | $e_1$ | $e_2$ | $e_3$ |
|---|---|---|---|
| v | 1 | 0 | 1 |
| u | 1 | 1 | 0 |
| w | 0 | 1 | 1 |

# Representing Graphs

**Example:** What is the incidence matrix M for the following graph G based on the order of vertices a, b, c, d and edges 1, 2, 3, 4, 5, 6?




**Solution:**

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

**Note:** Incidence matrices of directed graphs contain two 1s per column for edges connecting two vertices and one 1 per column for loops.

# Representation- Adjacency Matrix

- There is an N x N matrix, where |V| = N , the Adjacenct Matrix (NxN) $A = [a_{ij}]$

**For undirected graph**

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$

- **For directed graph**

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$

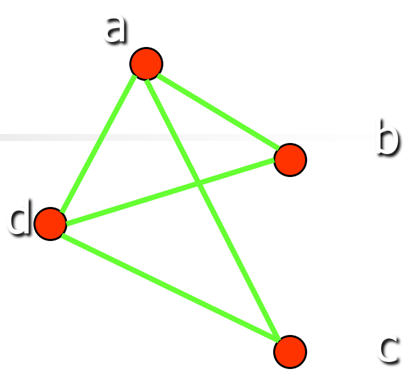
- This makes it easier to find subgraphs, and to reverse graphs if needed.

# Representation- Adjacency Matrix

- Adjacency is chosen on the ordering of vertices. Hence, there as are as many as n! such matrices.
- The adjacency matrix of simple graphs are symmetric ($a_{ij} = a_{ji}$) (why?)
- When there are relatively few edges in the graph the adjacency matrix is a **sparse matrix**
- Directed Multigraphs can be represented by using aij = number of edges from $v_i$ to $v_j$

# Adjacency matrix – undirected Graphs

**Example:** What is the adjacency matrix $A_G$ for the following graph G based on the order of vertices a, b, c, d ?

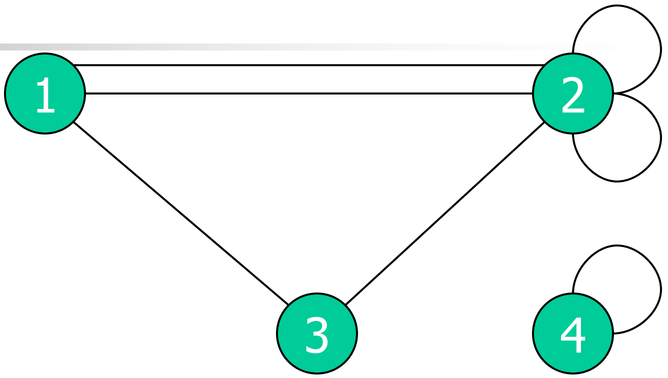



**Solution:**

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

**Note:** Adjacency matrices of undirected graphs are always symmetric.

# Adjacency matrix – undirected Graphs with multiple edges and loops

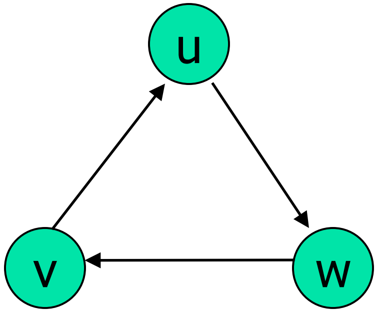



A:

$$\begin{pmatrix} 0 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

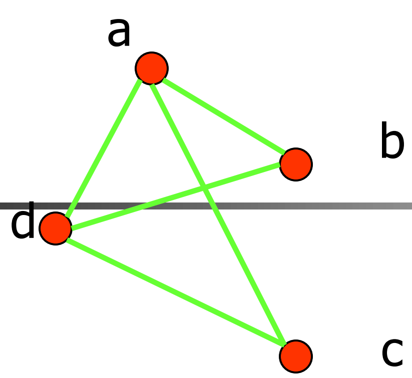
# Representation- Adjacency Matrix

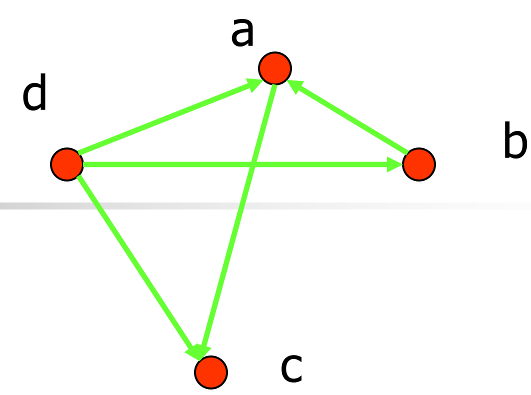
- Example: directed Graph G (V, E)




| | v | u | w |
|---|---|---|---|
| v | 0 | 1 | 0 |
| u | 0 | 0 | 1 |
| w | 1 | 0 | 0 |

# Representation- Adjacency List




| Vertex | Adjacent Vertices |
|---|---|
| a | b, c, d |
| b | a, d |
| c | a, d |
| d | a, b, c |

Undirected graph

| Initial Vertex | Terminal Vertices |
|---|---|
| a | c |
| b | a |
| c | |
| d | a, b, c |

Directed graph

# Representing graphs

- Adjacency matrix:
  - When graph is dense
- Adjacency lists:
  - When graph is sparse

# Connectivity

- Basic Idea: In a Graph, Reachability among vertices by traversing the edges

  Application Example:

  - In a city to city road-network, if one city can be reached from another city.

  - Problems if determining whether a message can be sent between two computer using intermediate links

  - Efficiently planning routes for data delivery in the Internet

A path in a graph is a sequence of (not necessarily distinct) vertices $v_1$, $v_2$, . . . , $v_k$ such that $v_i v_{i+1} \in E$ for $i = 1, 2, . . ., k - 1$.
The path is a circuit if it begins and ends at the same vertex.

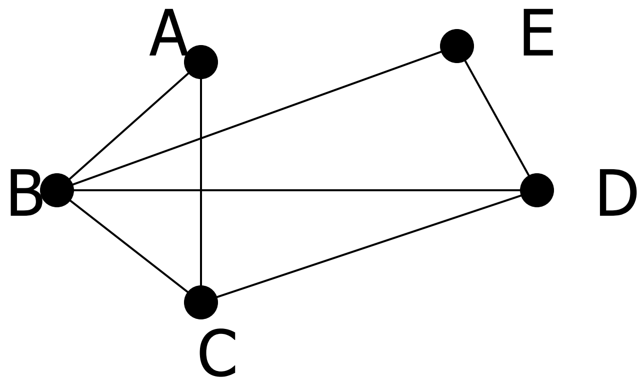
**Circuit/Cycle:** $u = v$, length of path $> 0$

In some books, a path is called a $v_1$–$v_k$ walk, and $v_1$ and $v_k$ are the end vertices of the walk. If the edges in a walk are distinct, then the walk is called a trail or simple path. Circuit is called as closed walk.

In directed multigraphs when it is not necessary to distinguish between their edges, we can use sequence of vertices to represent the path

# Example: Classifying Walks

Using the graph, classify each sequence as a walk, a path or a circuit.




a) $E \to C \to D \to E$

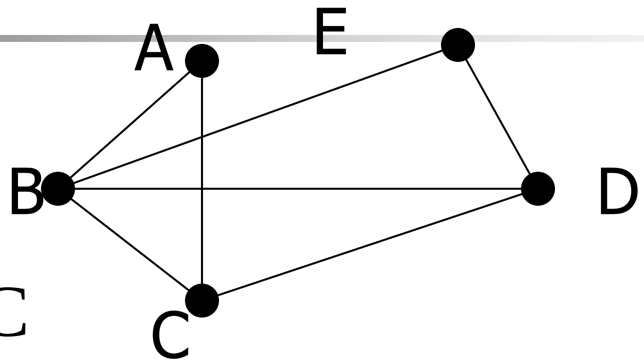
b) $A \to C \to D \to E \to B \to A$

c) $B \to D \to E \to B \to C$

d) $A \to B \to C \to D \to B \to A$

# Example: Classifying Walks

## Solution




a) $E \rightarrow C \rightarrow D \rightarrow E$

c) $B \rightarrow D \rightarrow E \rightarrow B \rightarrow C$

d) $A \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow A$

b) $A \rightarrow C \rightarrow D \rightarrow E \rightarrow B \rightarrow A$

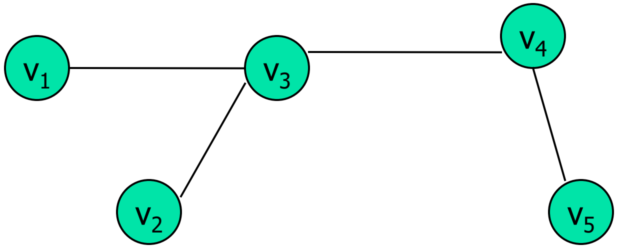
| **Walk** | **Path** | **Circuit** |
|---|---|---|
| a) No | No | No |
| b) Yes | Yes | Yes |
| c) Yes | No | No |
| d) Yes | No | No |

# Connectivity – Connectedness

## Undirected Graph

An undirected graph is connected if there exists a simple path between every pair of vertices

Representation Example: G (V, E) is connected since for $V = \{v_1, v_2, v_3, v_4, v_5\}$, there exists a path between $\{v_i, v_j\}$, $1 \leq i, j \leq 5$

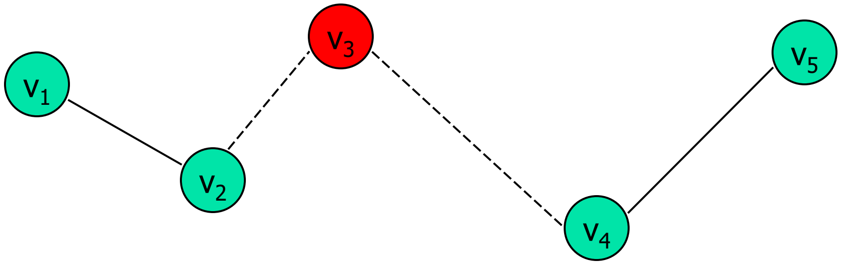


# Connectivity – Connectedness

## Undirected Graph

- **Articulation Point (Cut vertex):** removal of a vertex produces a subgraph with more connected components than in the original graph. The removal of a cut vertex from a connected graph produces a graph that is not connected
- **Cut Edge:** An edge whose removal produces a subgraph with more connected components than in the original graph.

  Representation example: G (V, E), $v_3$ is the articulation point or edge $\{v_2, v_3\}$, the number of connected components is 2 (> 1)



# Connectivity – Connectedness

**Directed Graph**

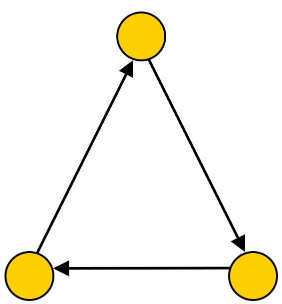
- A directed graph is **strongly connected** if there is a path from a to b and from b to a whenever a and b are vertices in the graph
- A directed graph is **weakly connected** if there is a (undirected) path between every two vertices in the underlying undirected path

A strongly connected Graph can be weakly connected but the vice-versa is not true.

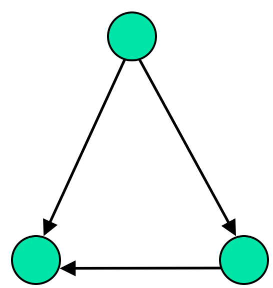
# Connectivity – Connectedness

**Directed Graph**

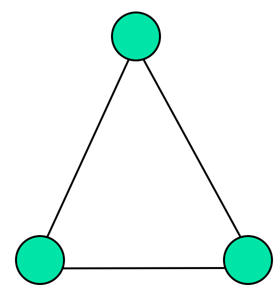
Representation example: G1 (Strong component), G2 (Weak Component), G3 is undirected graph representation of G2 or G1



G1



G2



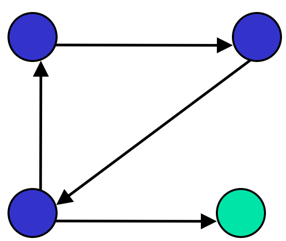
G3

# Connectivity – Connectedness

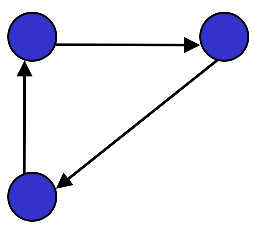
- **Directed Graph**

  **Strongly connected Components:** subgraphs of a Graph G that are strongly connected

  Representation example: G1 is the strongly connected component in G



G



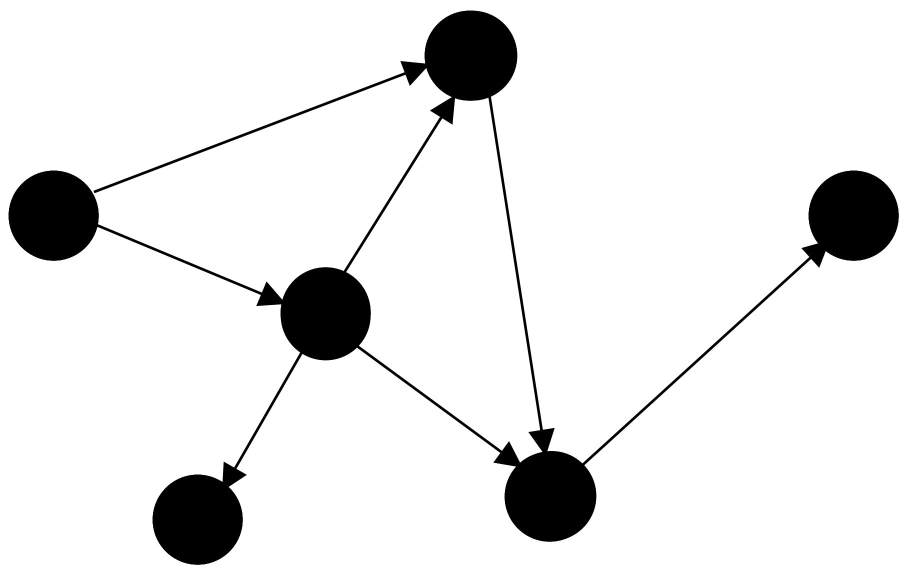
G1

# Connected Components

- Connected component of an undirected graph is a maximal connected subgraph of the graph.
- Every vertex of the graph lies in a connected component that consists of all the vertices that can be reached from that vertex, together with all the edges that join those vertices.
- If an undirected graph is connected, there is only one connected component.
- We can use a traversal algorithm, either depth-first or breadth- first, to find the connected components of an undirected graph.

# Depth-First Search (DFS)

Strategy: Go as far as you can (if you have not visited there), otherwise, go back and try another way

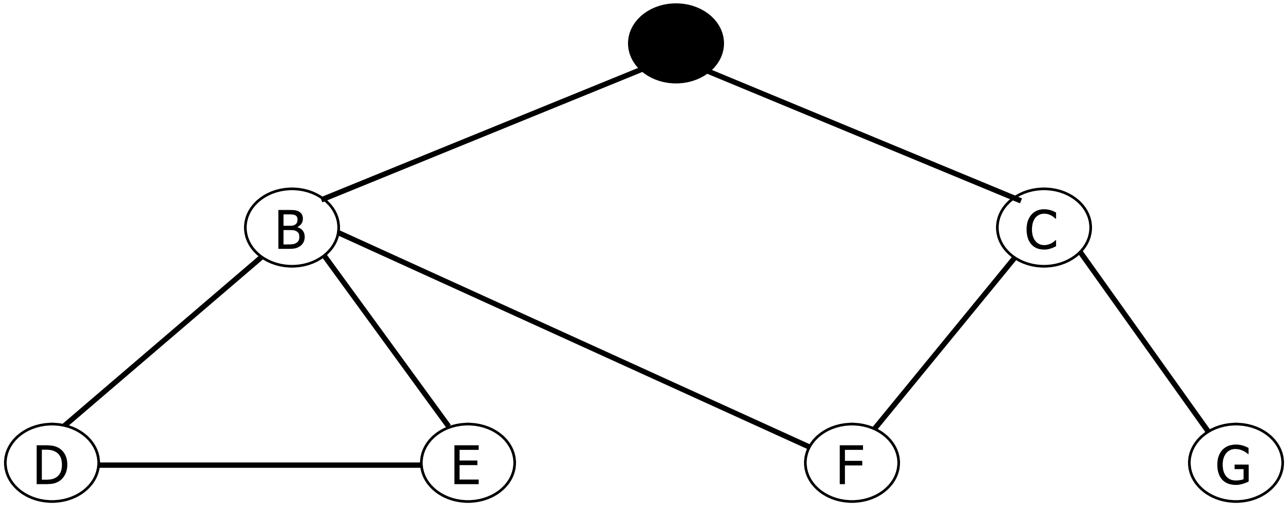


# Implementation

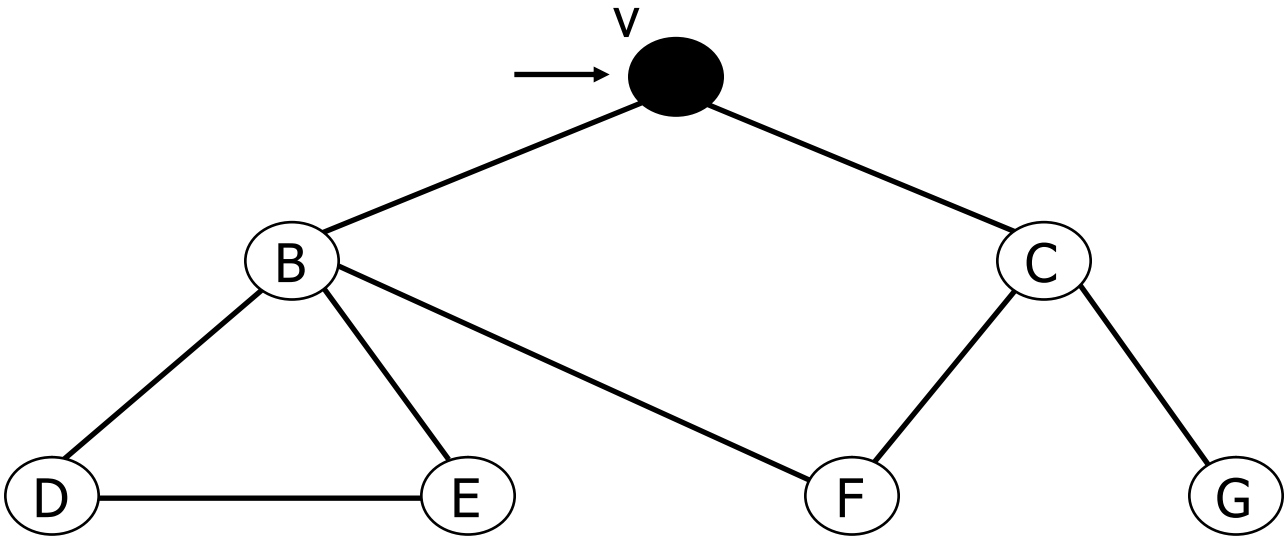
```
DFS (vertex u) {
    mark u as visited
    for each vertex v directly reachable from u
     if v is unvisited
         DFS (v)
}
```

Initially all vertices are marked as *unvisited*

# Depth-First Search

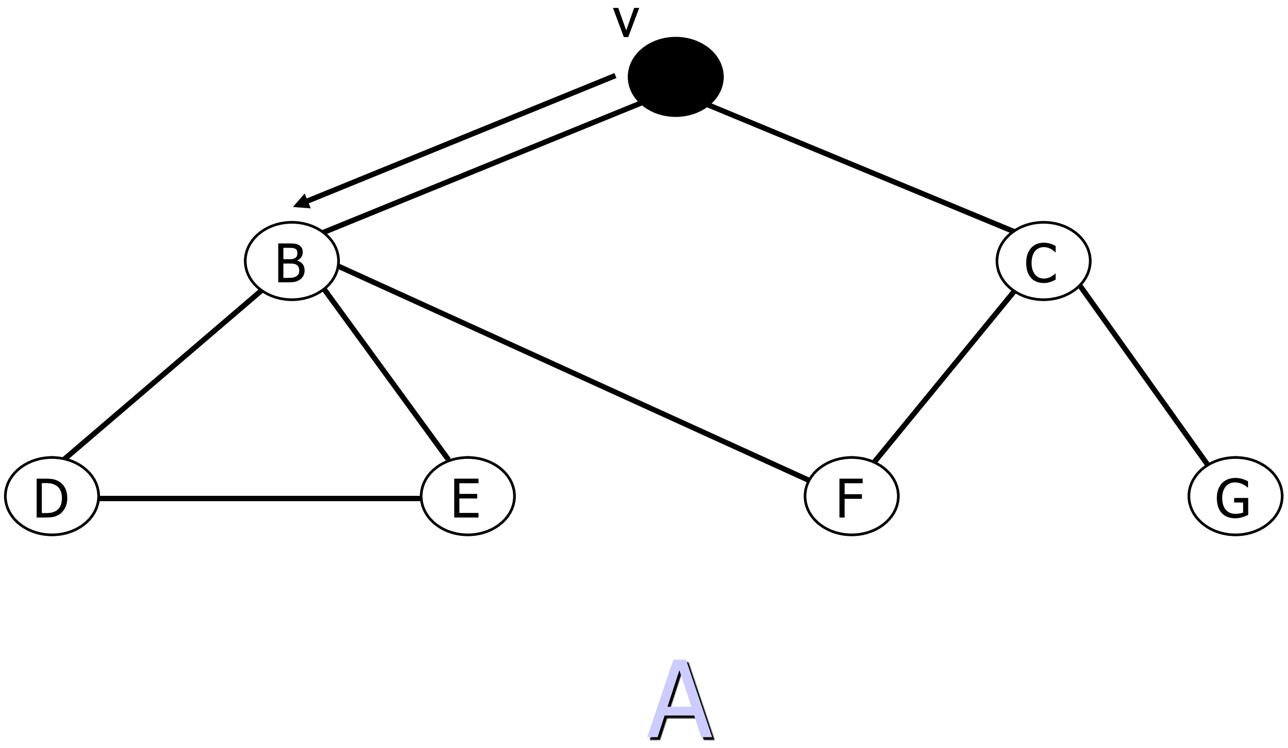


# Depth-First Search

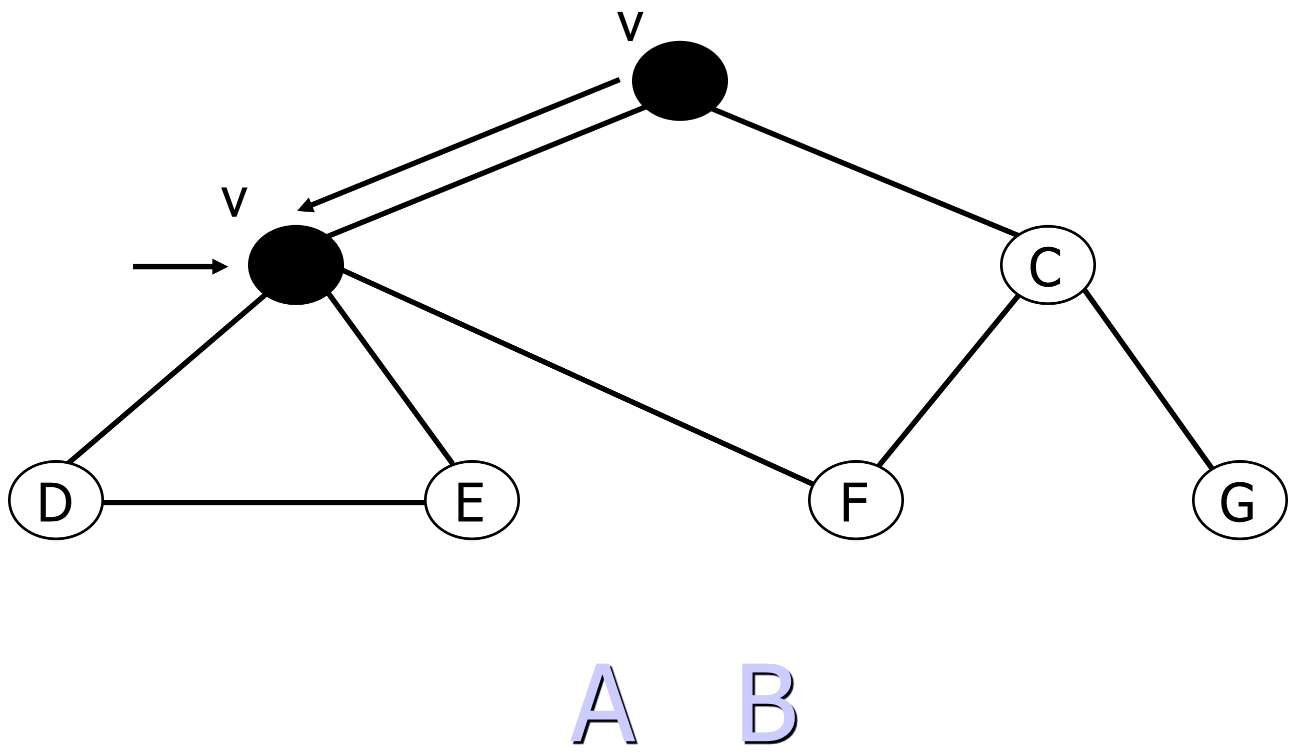



A

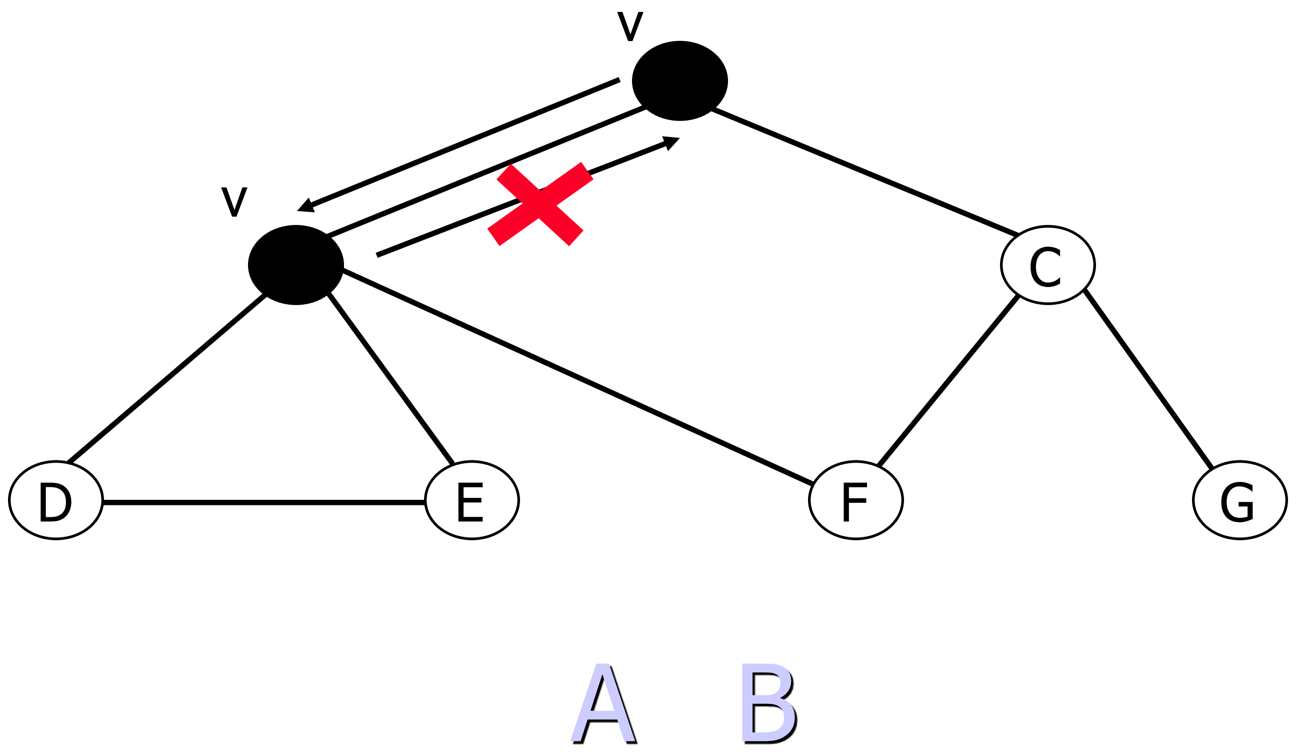
# Depth-First Search



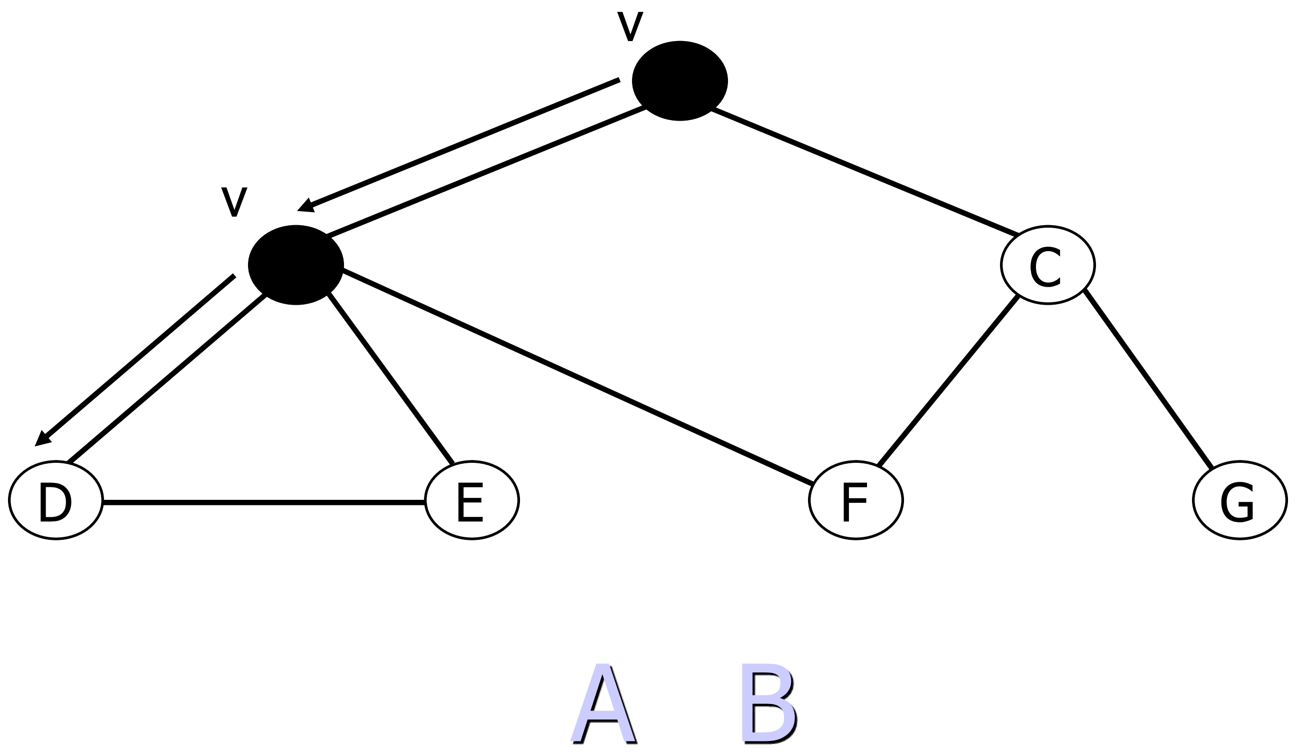
# Depth-First Search



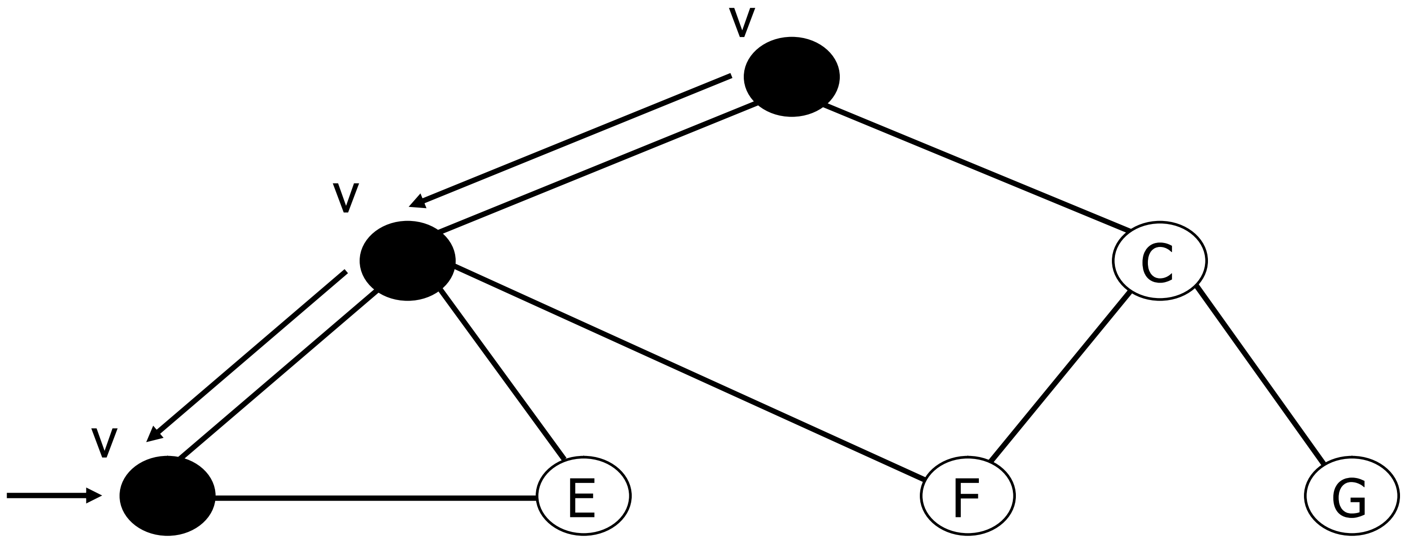
# Depth-First Search



# Depth-First Search

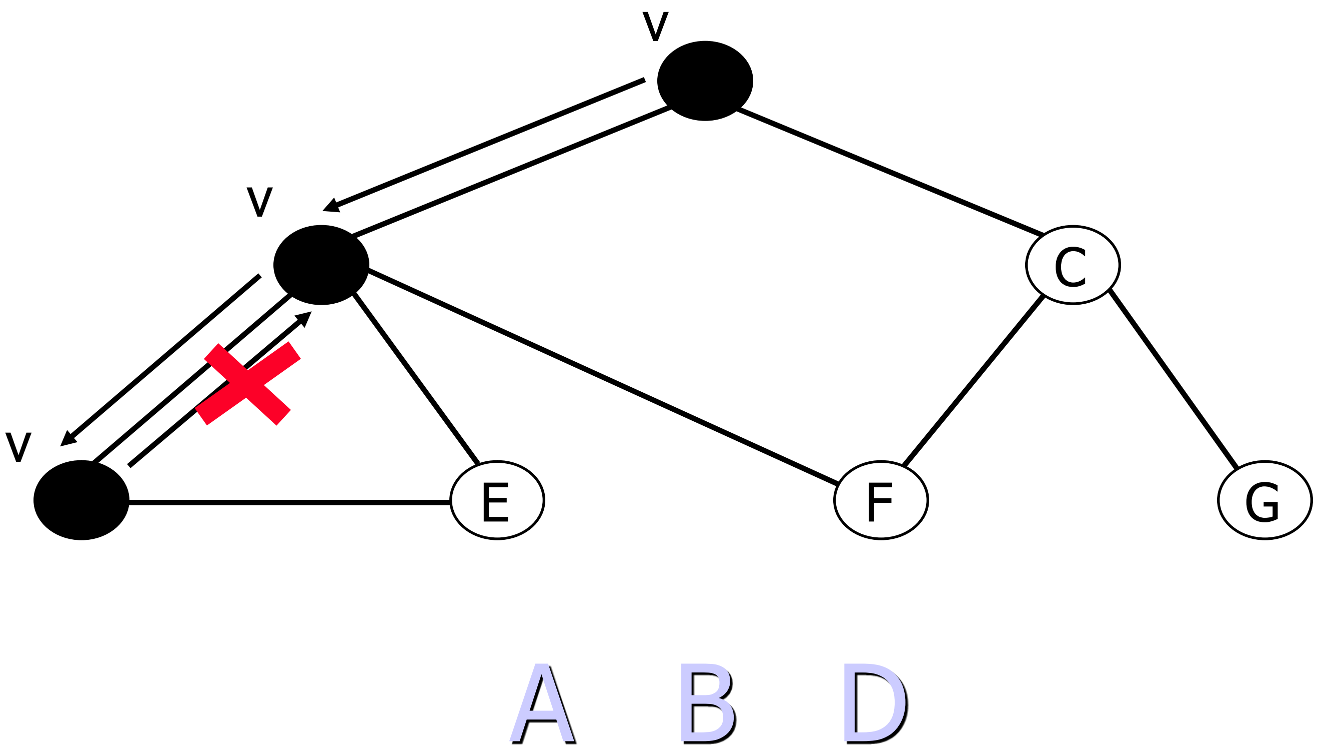


# Depth-First Search

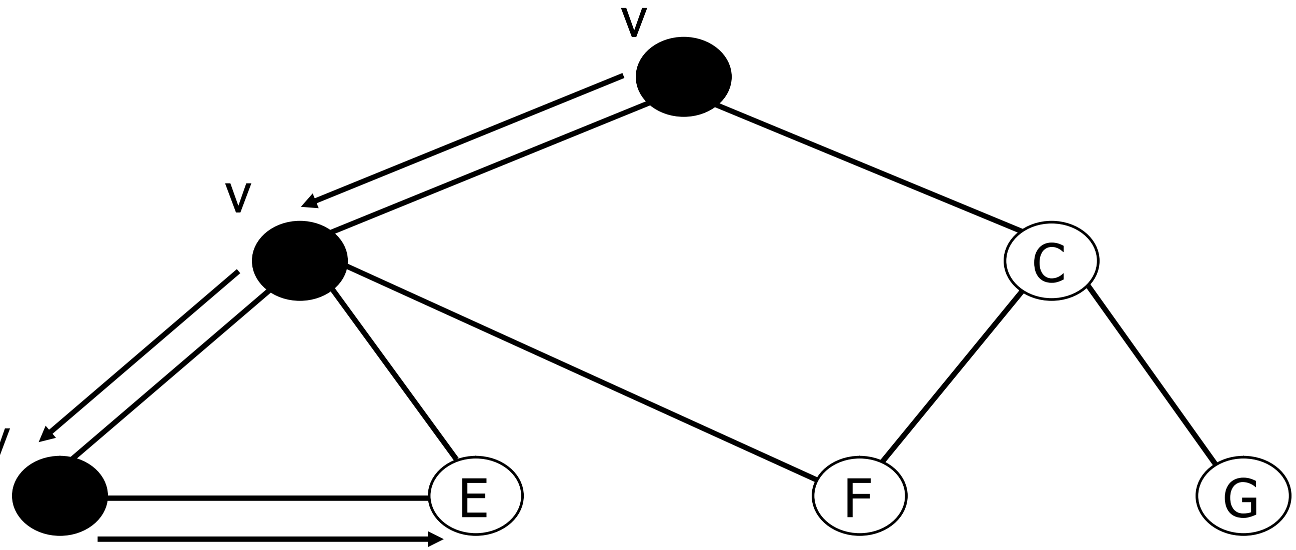



A B D

# Depth-First Search

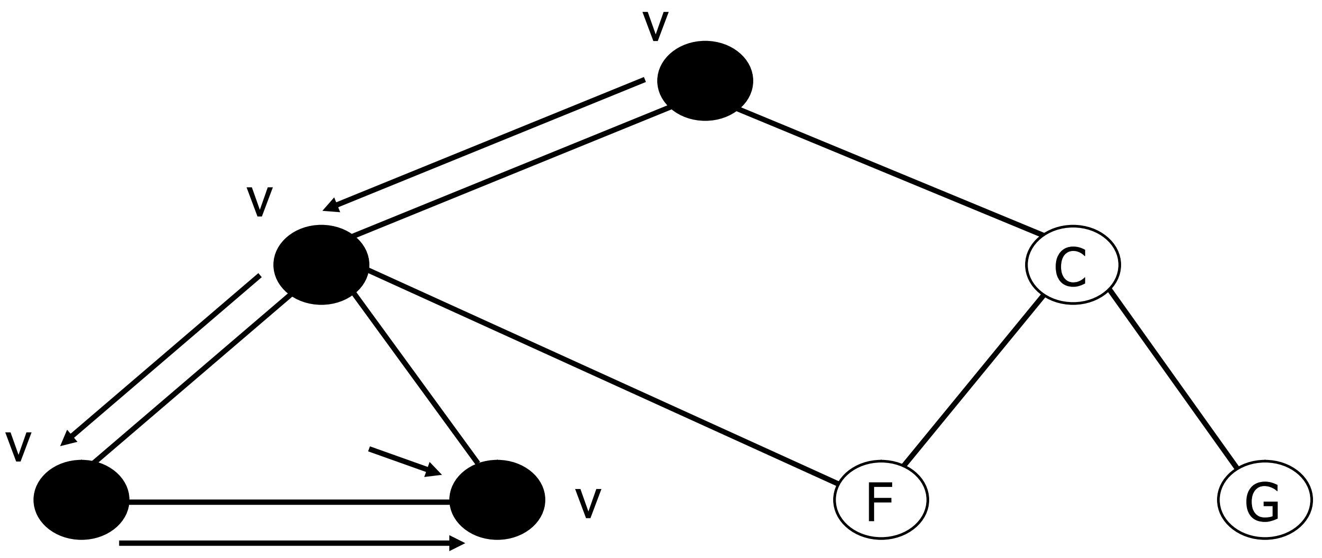


# Depth-First Search



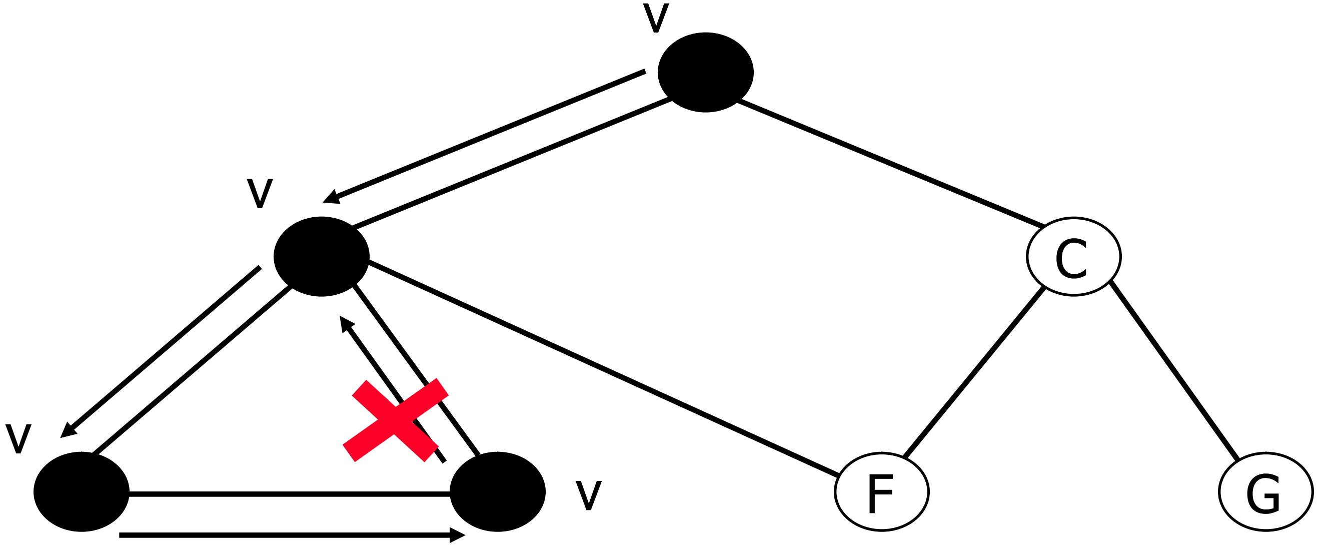

A B D

# Depth-First Search



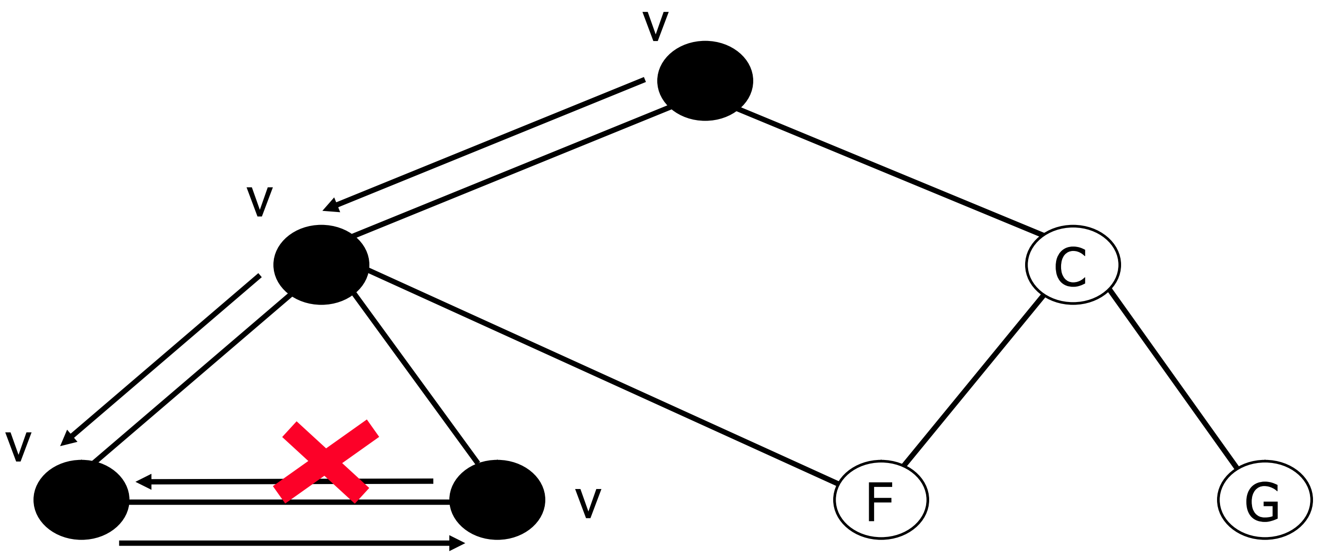

A B D E

# Depth-First Search



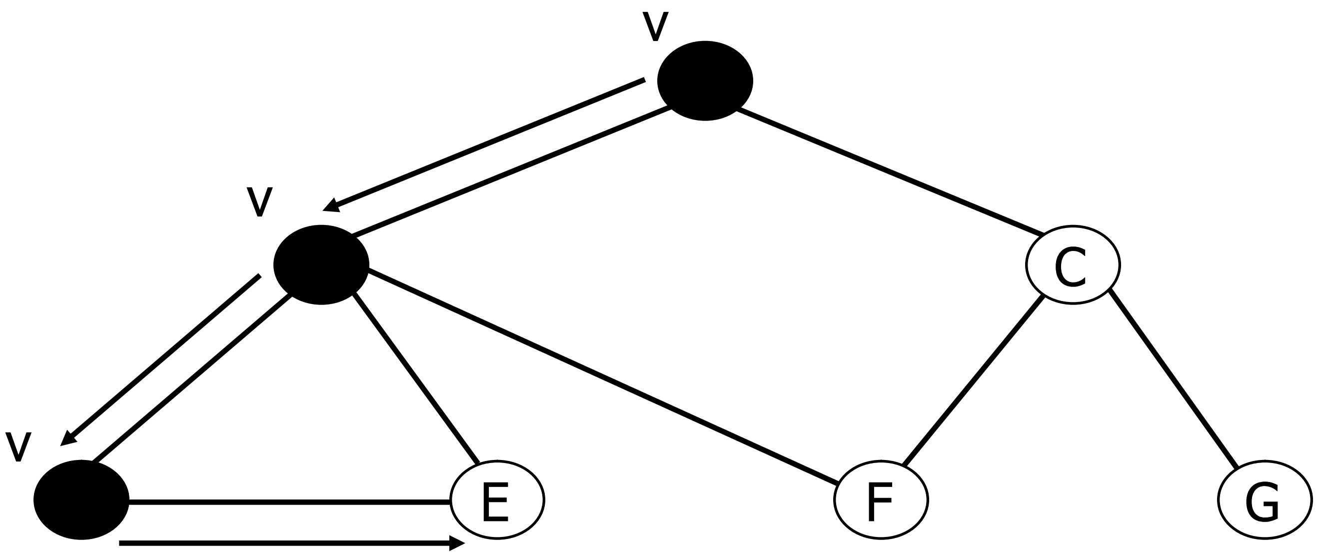

A B D E

# Depth-First Search



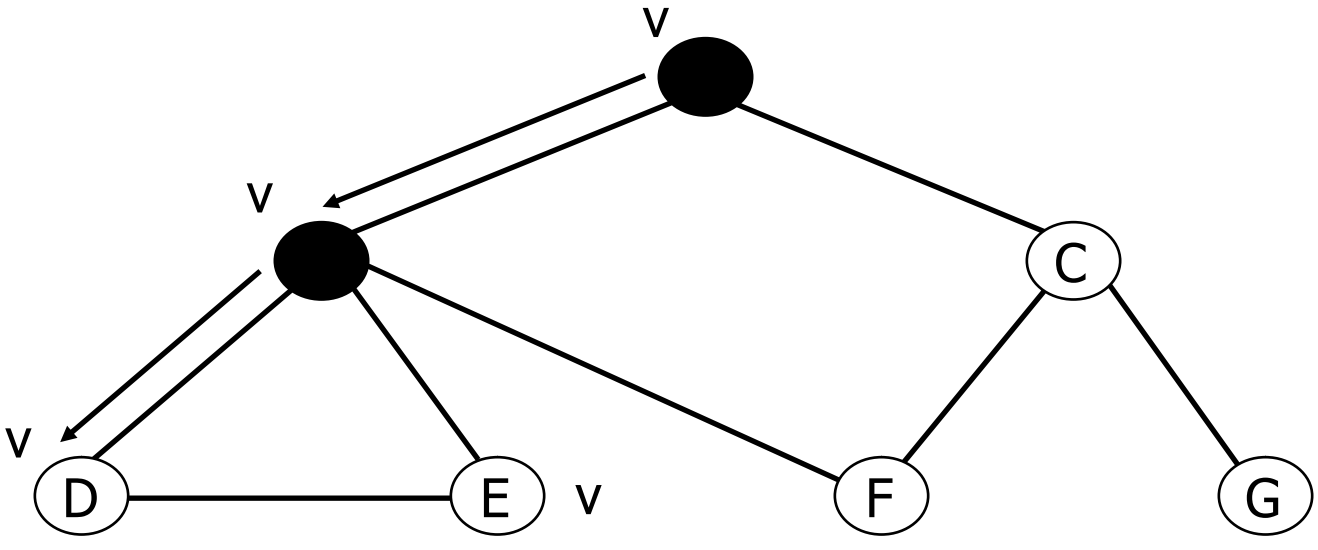

A B D E

# Depth-First Search



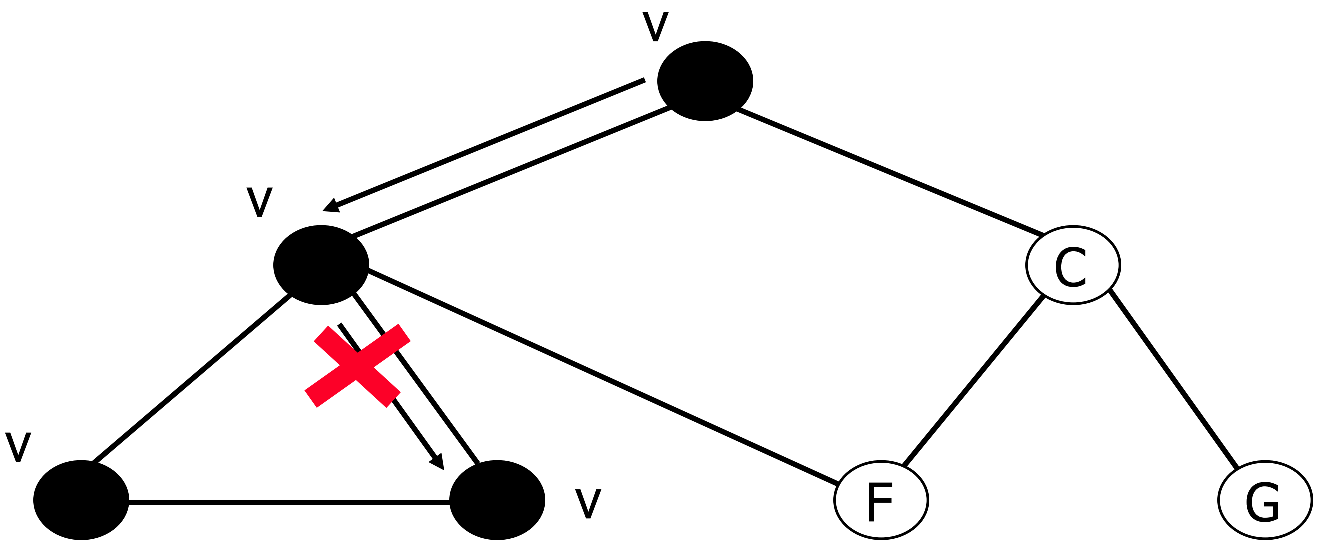

A B D E

# Depth-First Search



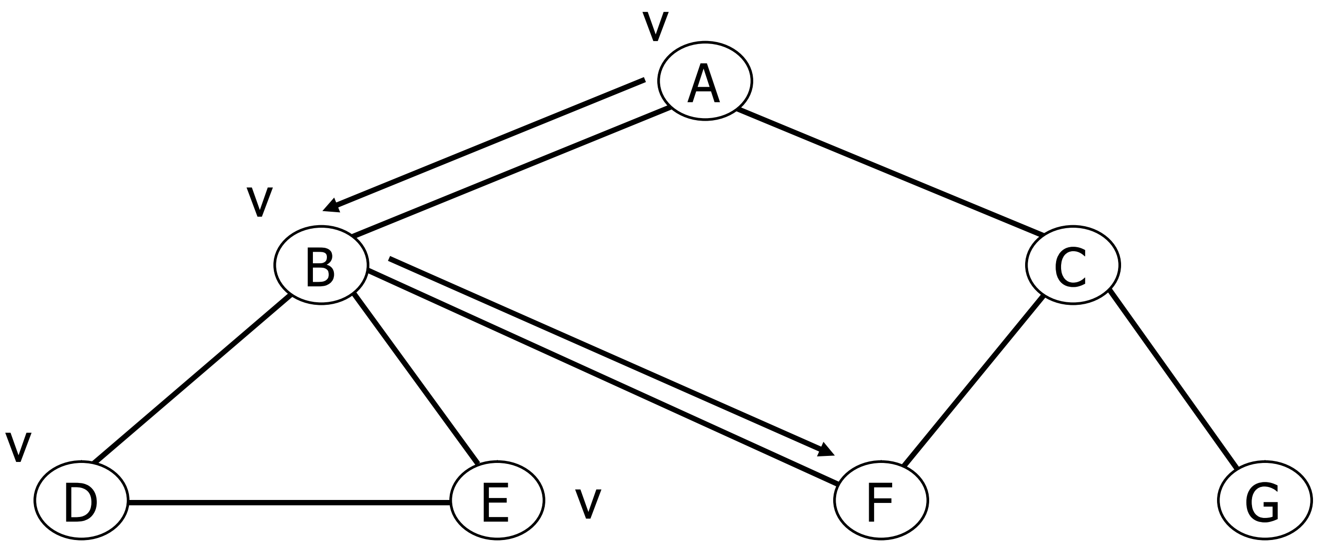

A B D E

# Depth-First Search



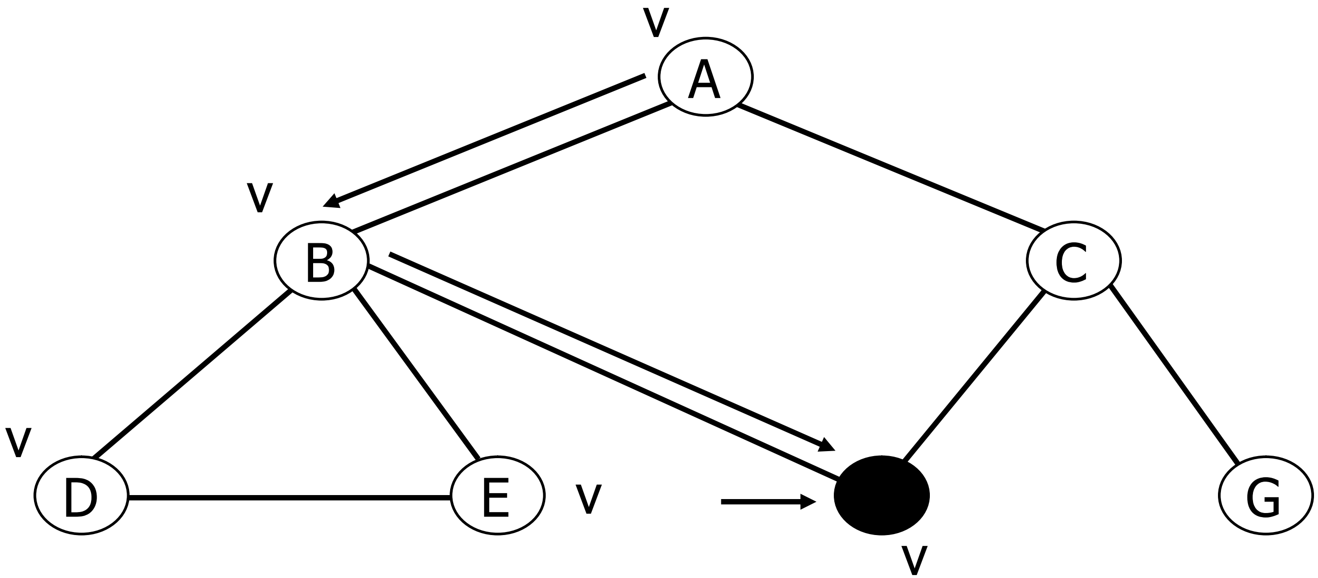

A B D E

# Depth-First Search



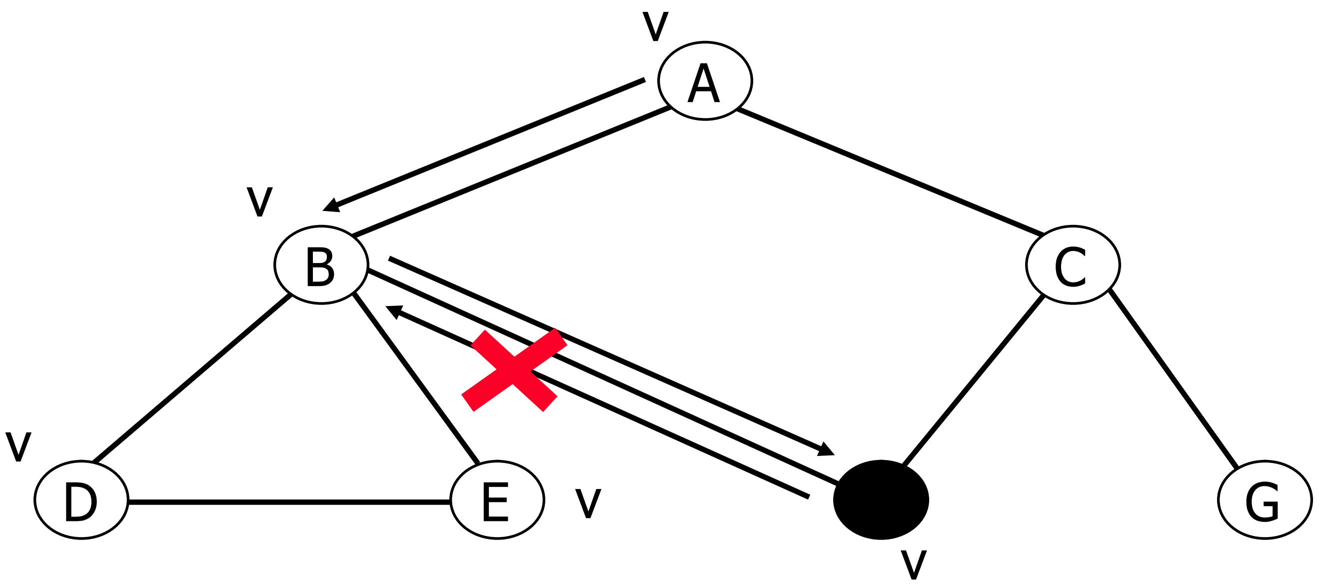

A B D E

# Depth-First Search



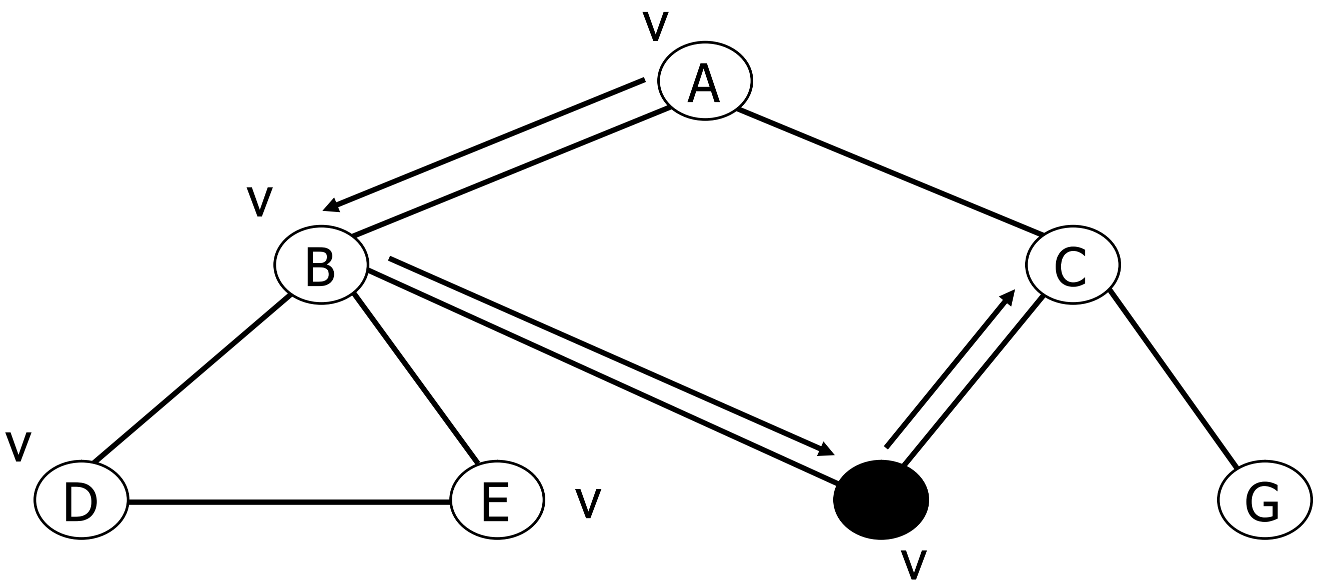

A B D E F

# Depth-First Search



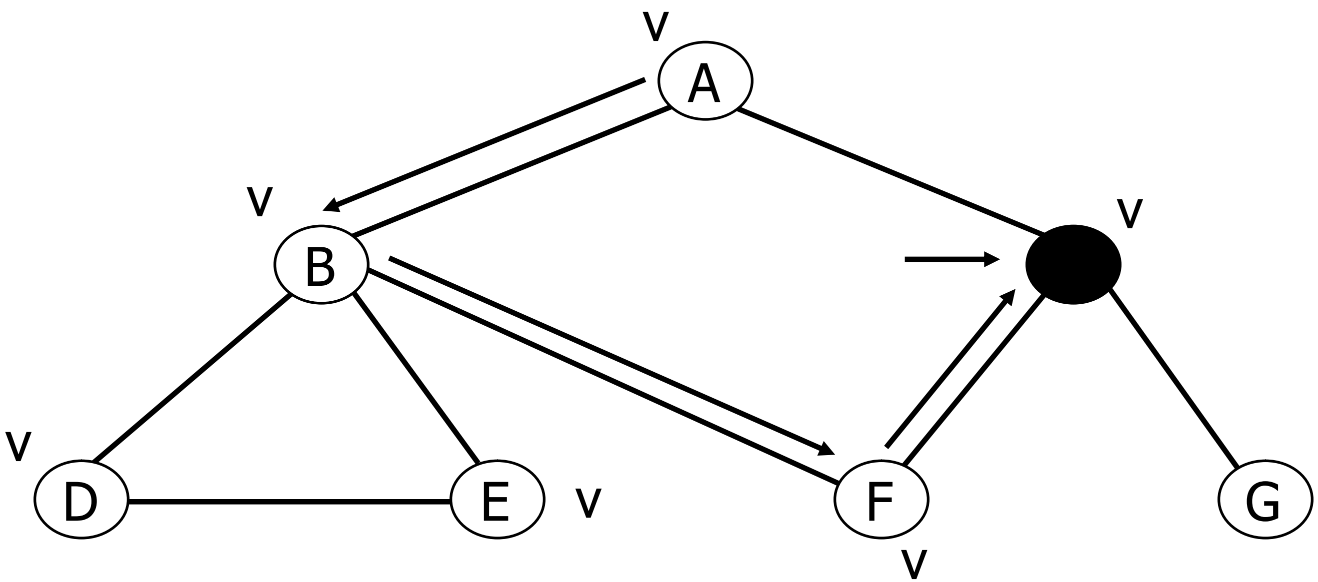

A B D E F

# Depth-First Search



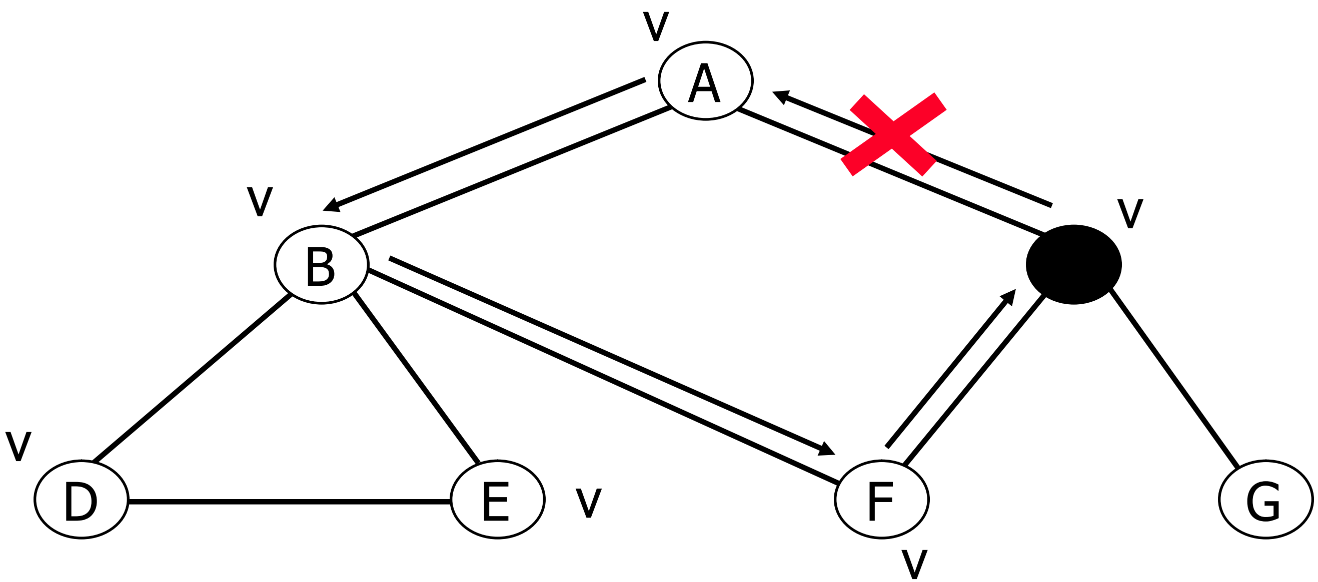

A B D E F

# Depth-First Search



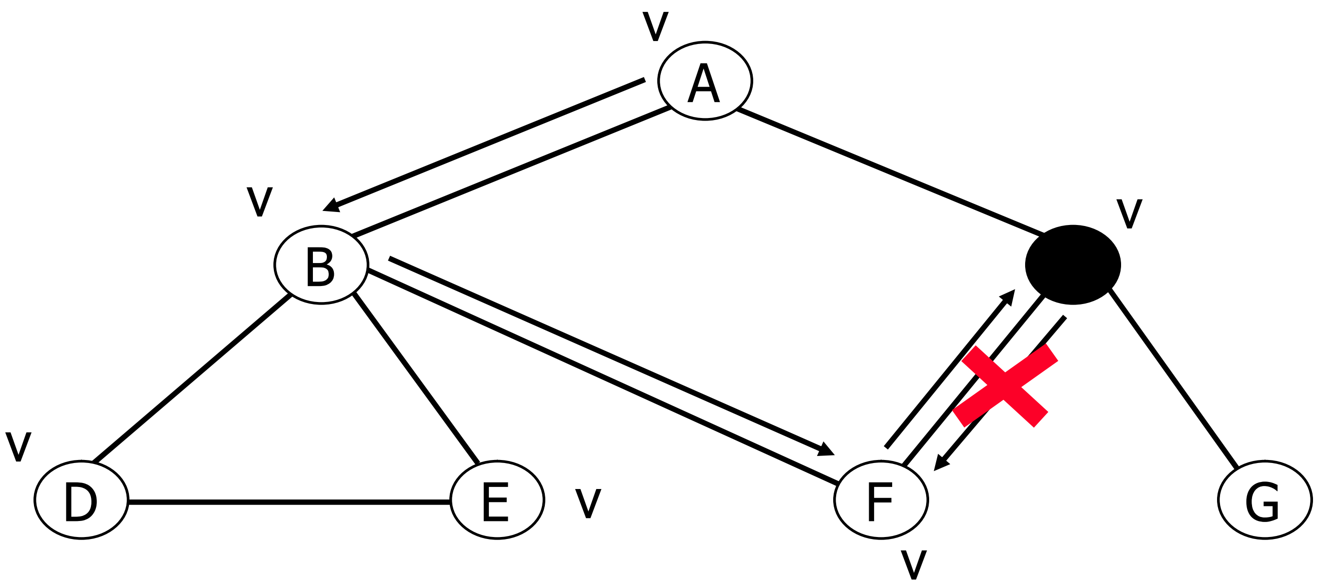

A B D E F C

# Depth-First Search



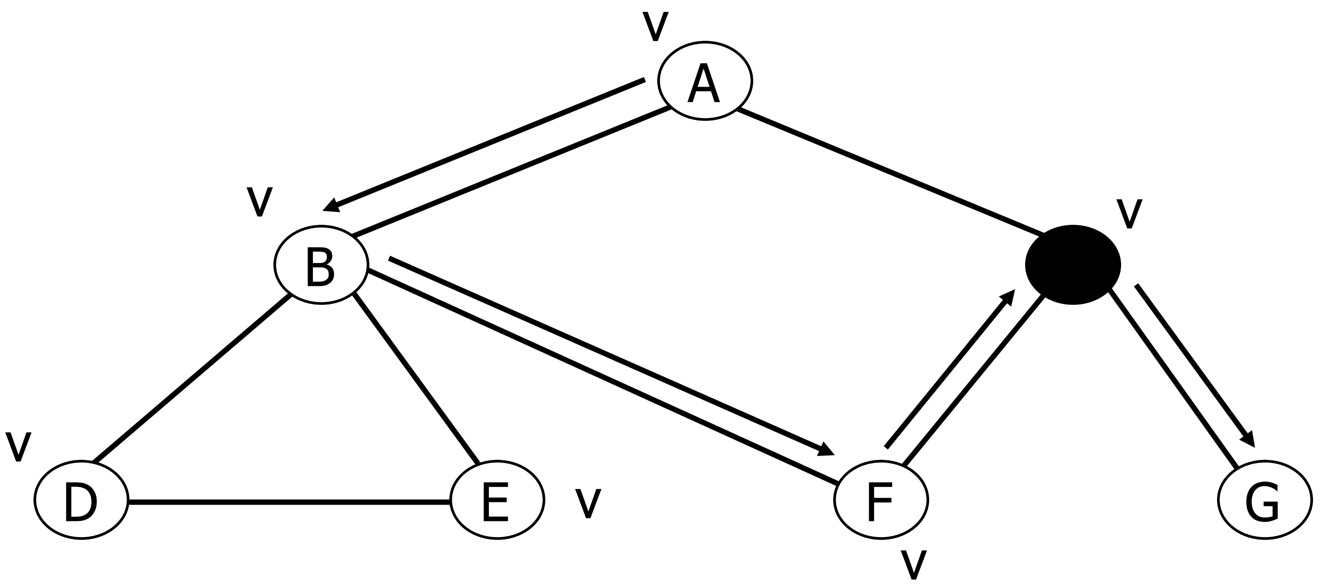

A B D E F C

# Depth-First Search



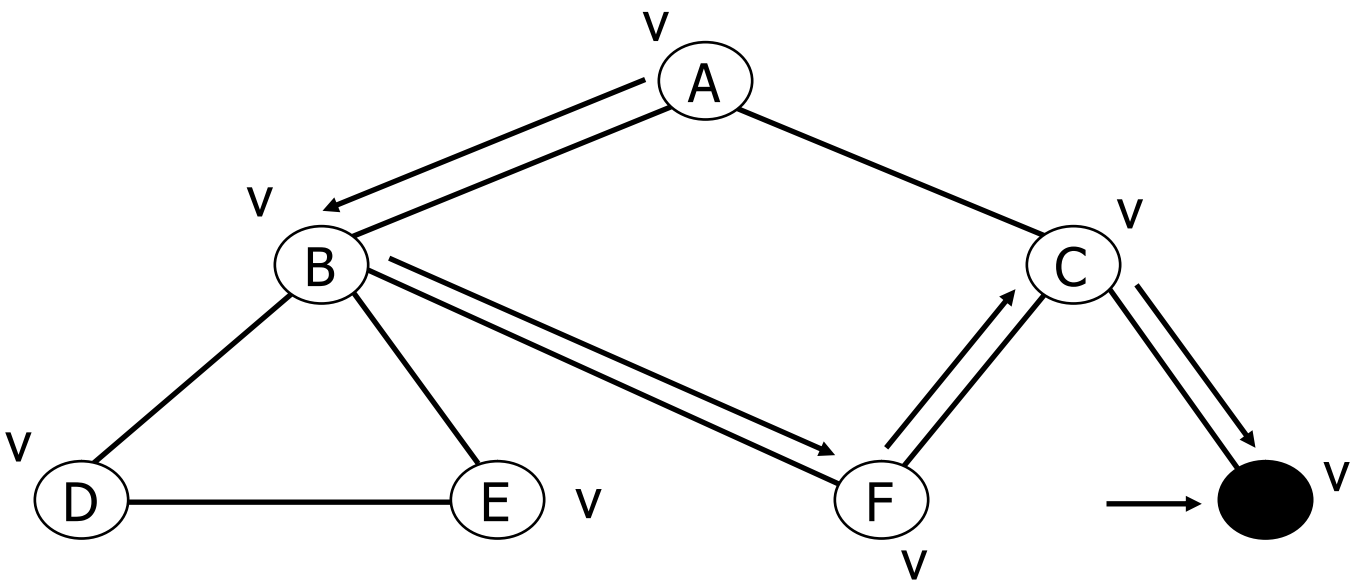

A B D E F C

# Depth-First Search



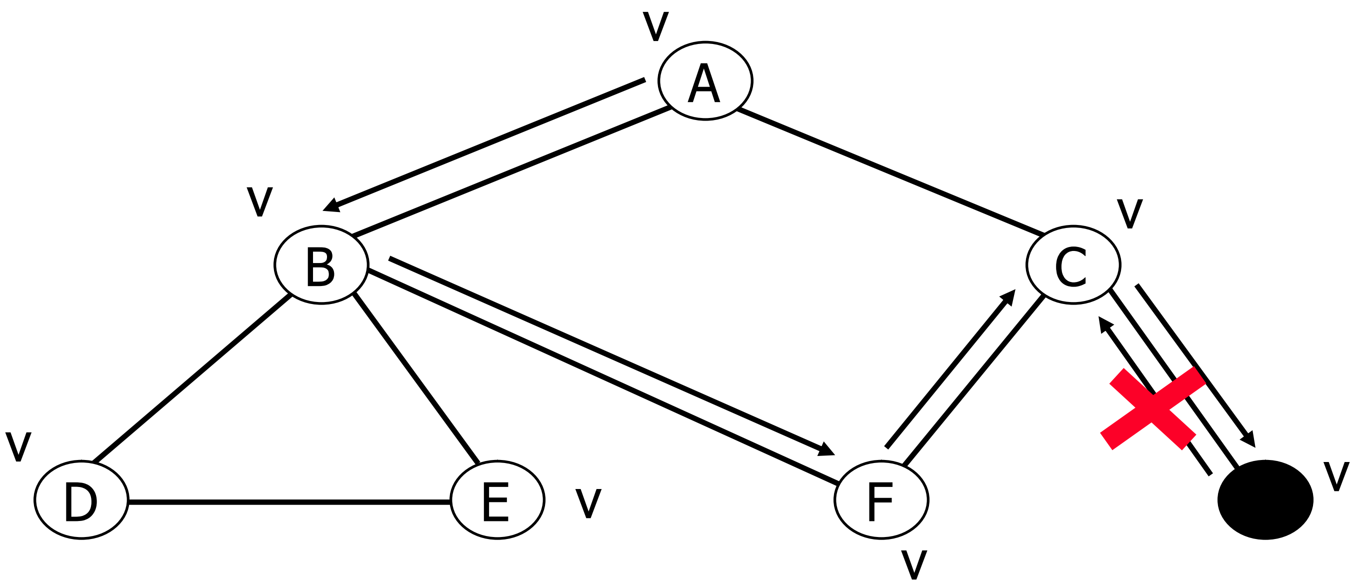

A B D E F C

# Depth-First Search



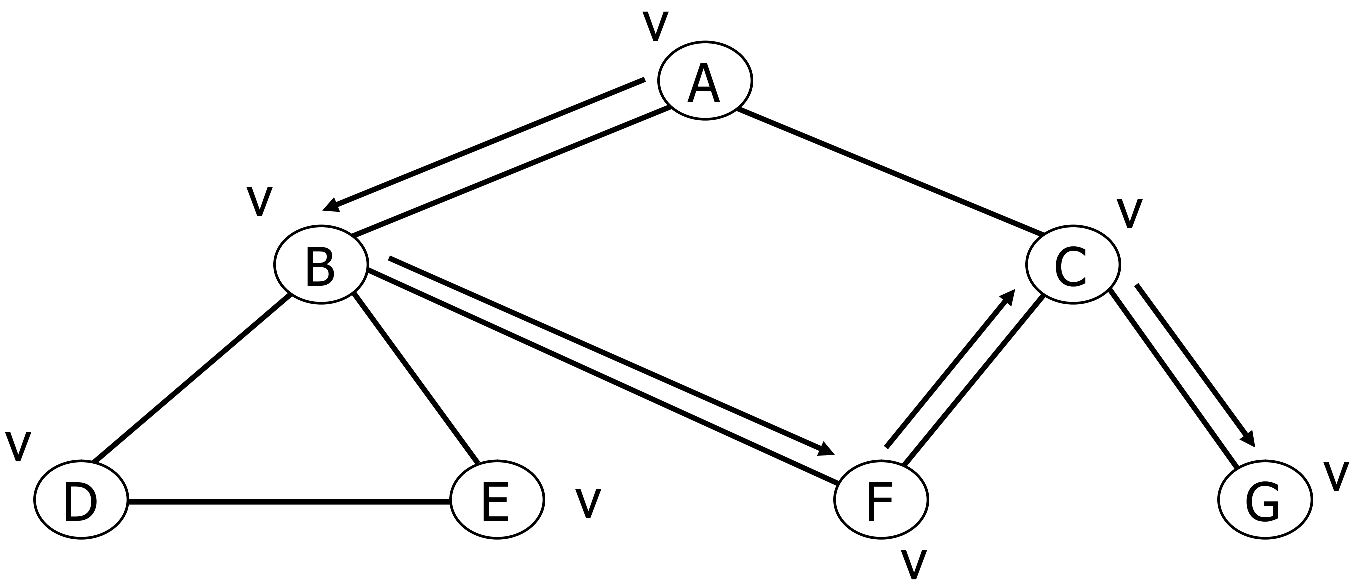

A B D E F C G

# Depth-First Search



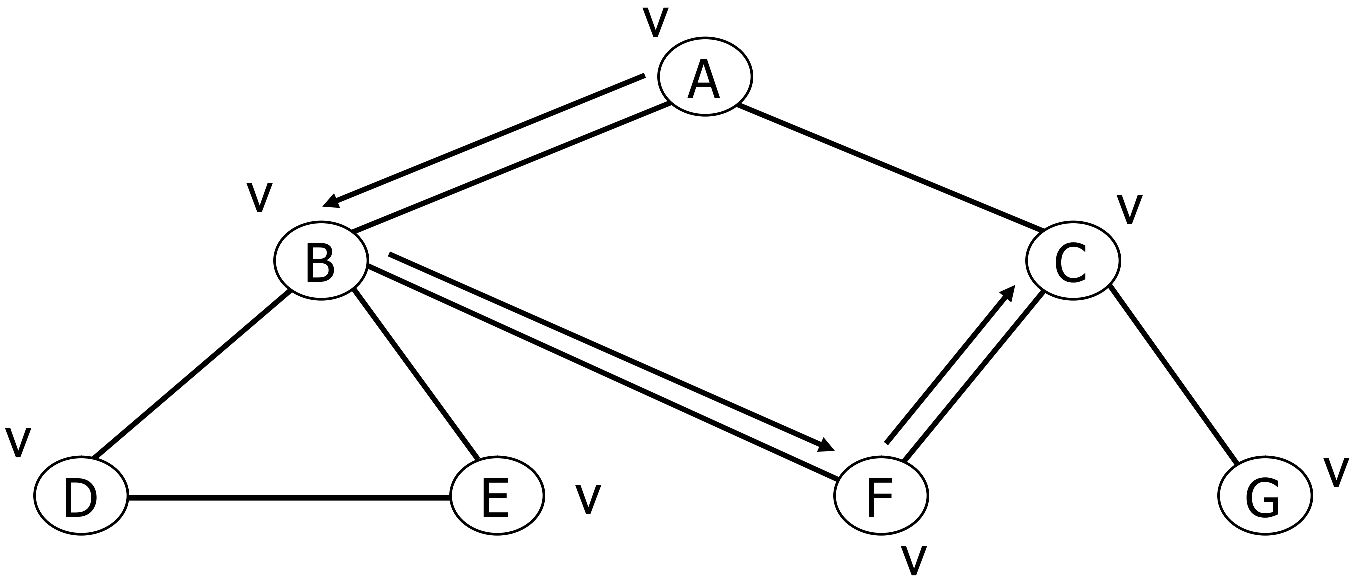

A B D E F C G

# Depth-First Search



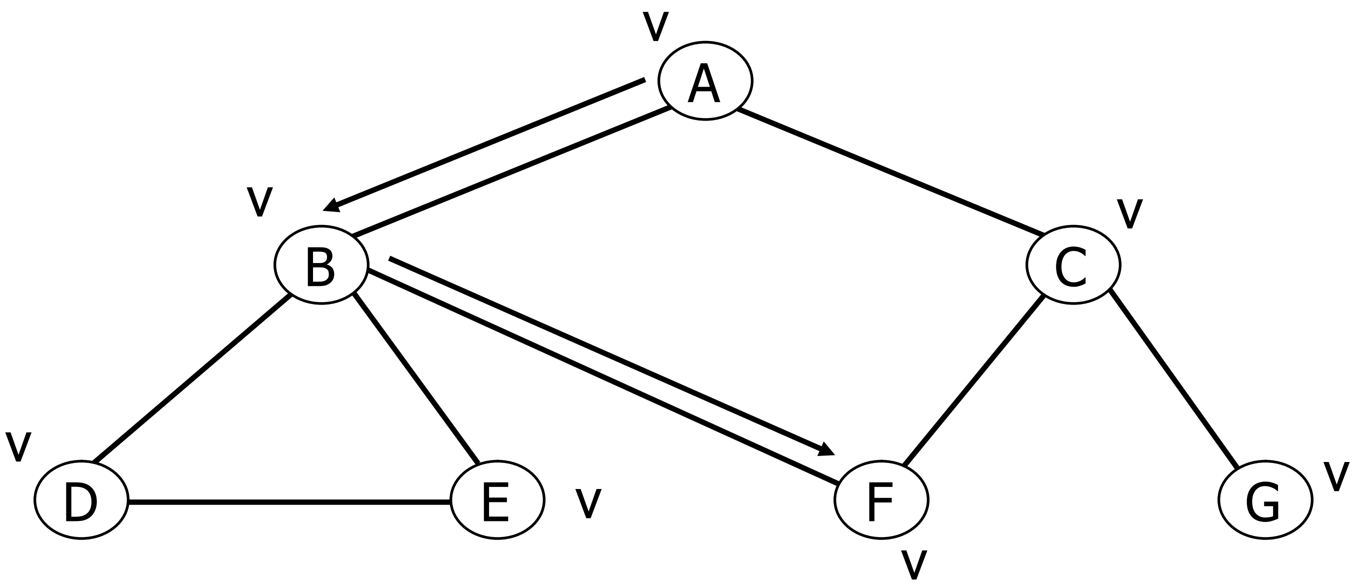

A B D E F C G

# Depth-First Search



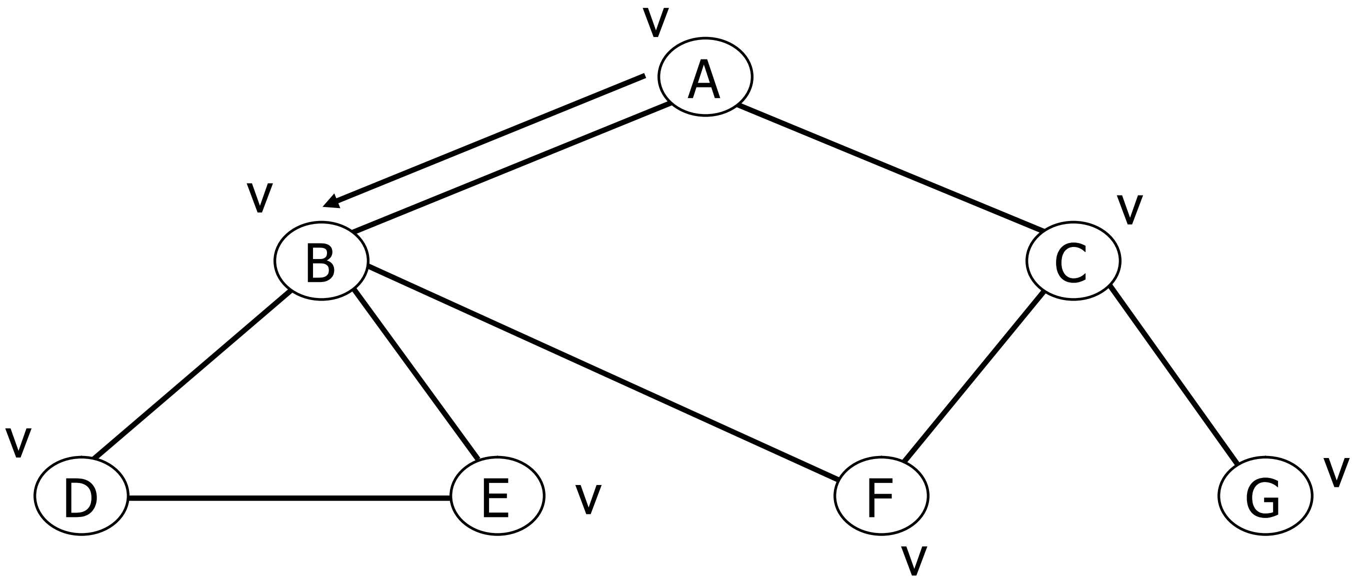

A B D E F C G

# Depth-First Search



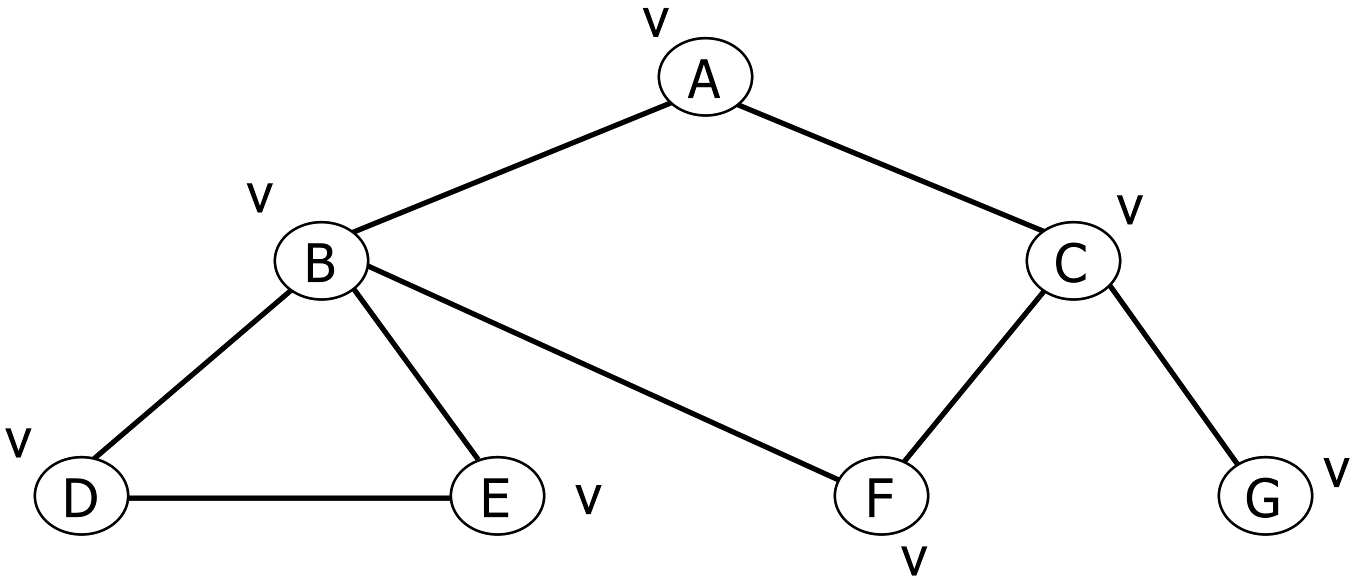

A B D E F C G

# Depth-First Search



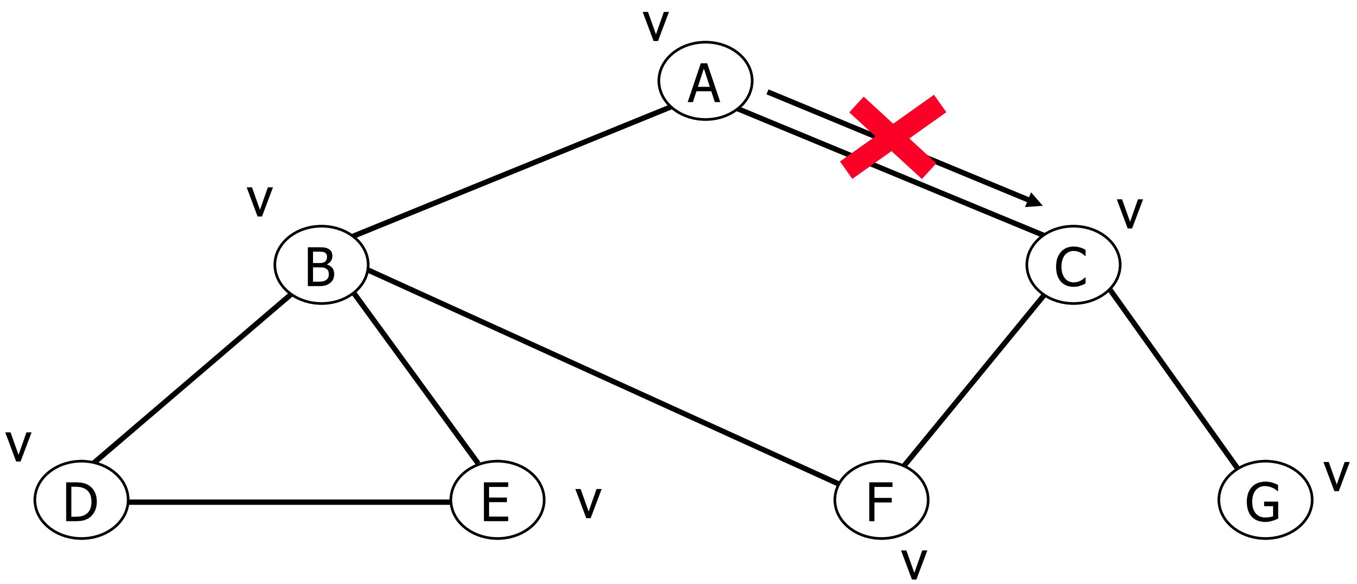

A B D E F C G

# Depth-First Search



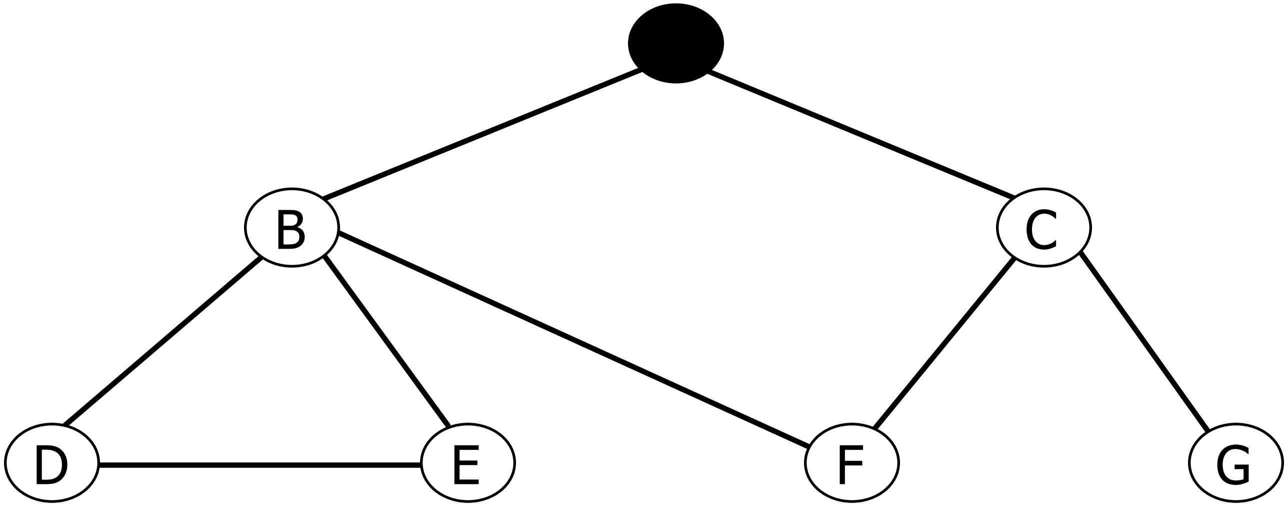

A B D E F C G

# Depth-First Search




A B D E F C G

# Depth-First Search




A B D E F C G

# Breadth-First Search (BFS)

Instead of going as far as possible, BFS tries to search all paths.

BFS makes use of a queue to store visited (but not dead) vertices, expanding the path from the earliest visited vertices.

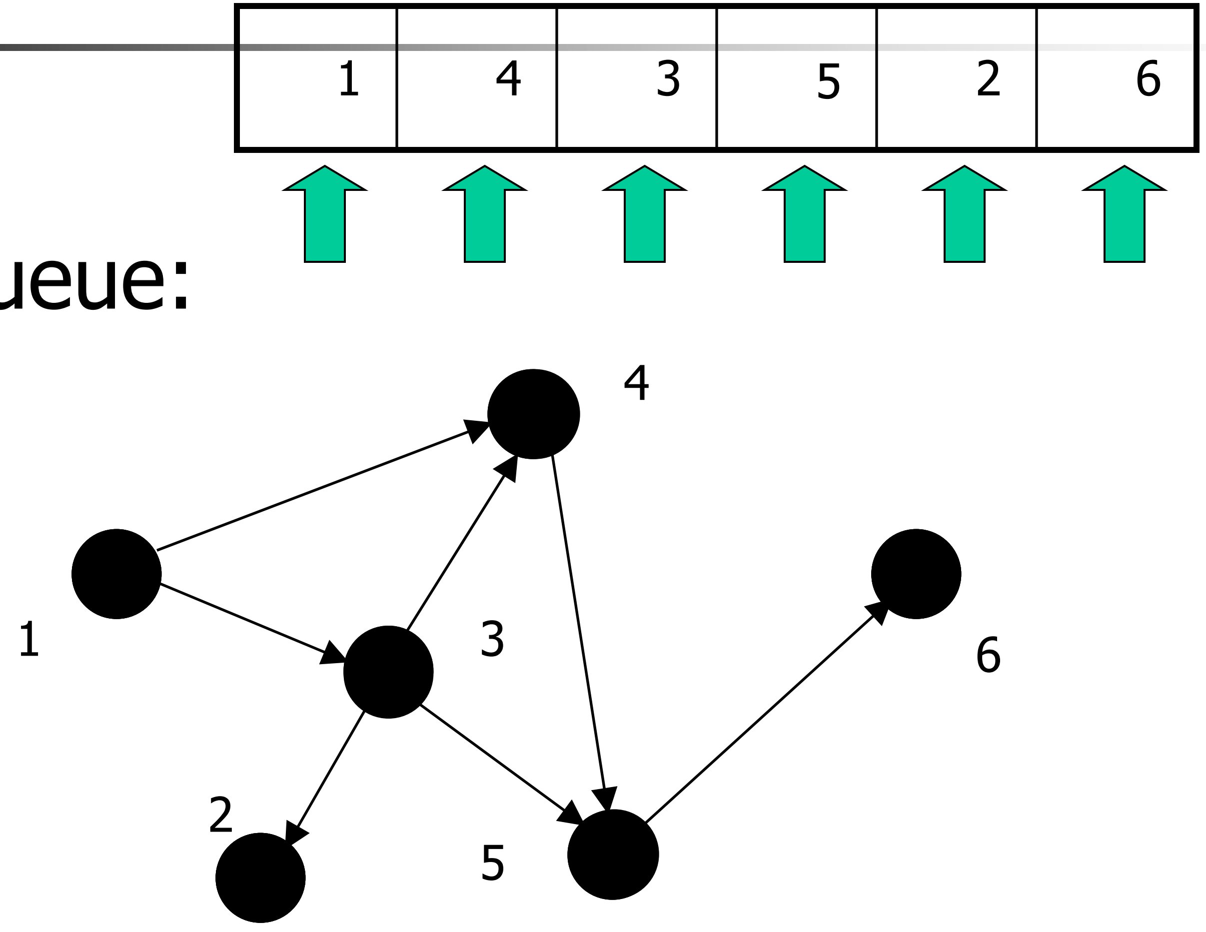
# Implementation

```
while queue Q not empty
    dequeue the first vertex u from Q
    for each vertex v directly reachable from u
     if v is unvisited
          enqueue v to Q
          mark v as visited
```

- Initially all vertices except the start vertex are marked as *unvisited* and the queue contains the start vertex only

# Simulation of BFS

Queue:



# Connected Components

- If we do a traversal starting from a vertex v , then we will visit all the vertices that can be reached from v.
- These are the vertices in the connected component that contains v.
- If there are other connected components, then there will still be unvisited vertices after the traversal is complete.
- We can do a traversal starting from one of those vertices to find another connected component. If we continue in this way until all vertices have been visited, then we will have discovered all the connected components
- Algorithm

```
 for (int i = 0; i < |V|; i++)
    visited[i] = false;   // Mark all nodes as unvisited.
int compNum = 0;          // For counting connected components.
for (int v = 0; v < |V|; v++) {

      // If v is not yet visited, it's the start of a newly
      // discovered connected component containing v.

   if ( ! visited[v] ) { // Process the component that contains v.
      compNum++;
      cout << "Component " << compNum << ":  ";
      IntQueue q;
      q.enqueue(v); // Start the traversal from vertex v.
      visited[v] = true;
      while ( ! q.isEmpty() ) {
         int w = q.dequeue();  // w is a node in this component.
         cout << w << " ";
         for each edge from w to some vertex k {  // ***
            if ( ! visited[k] ) {
                  // We've found another node in this component.
               visited[k] = true;
               q.enque(k);
            }
         }
      }
      cout << endl << endl;
   }
}

if (compNum == 1)
   cout << "The graph is connected.";
else
   cout << "There are " << compNum << " connected components.";
```