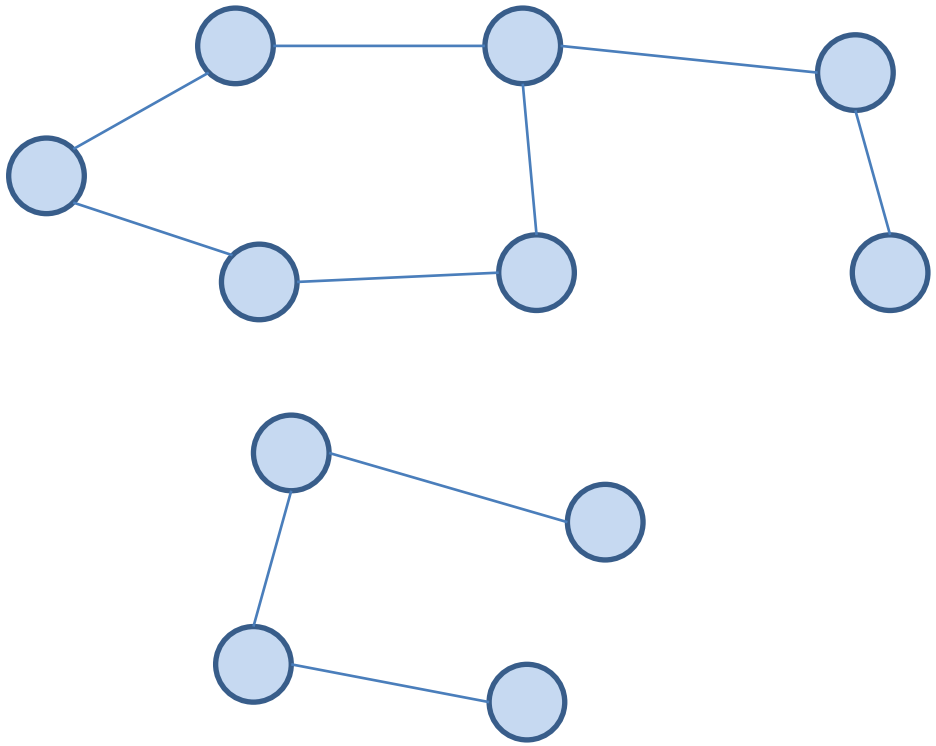
Graphs (Γράφοι)

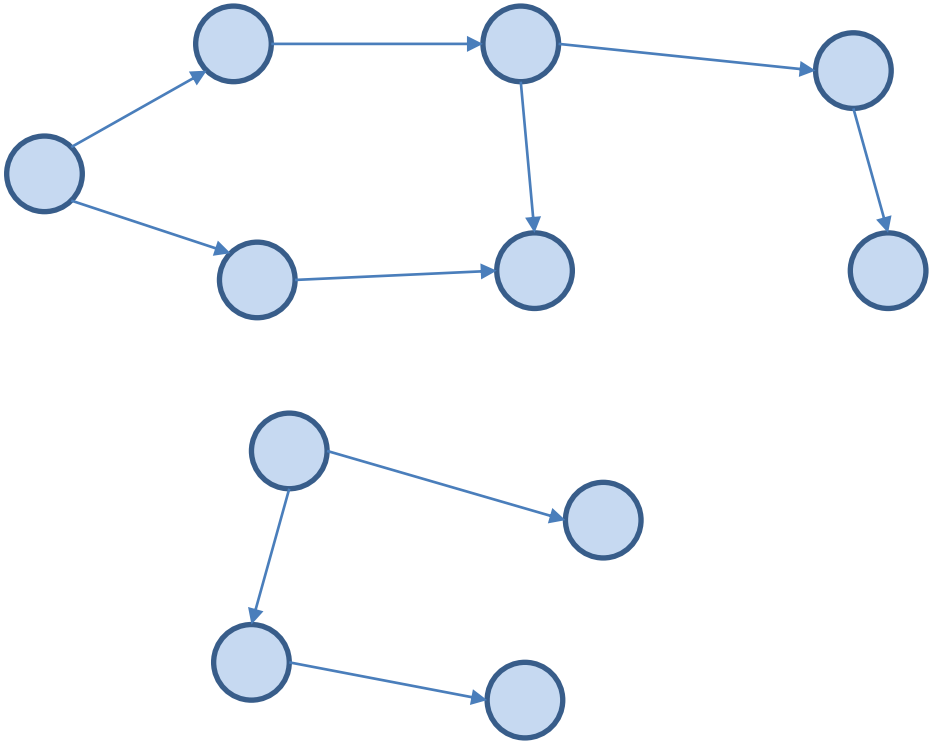
Graphs

- **Graphs** are collections of nodes in which various pairs are connected by line segments. The nodes are usually called **vertices** (**κορυφές**) and the line segments **edges (ακμές).**
- Graphs are **more general than trees**. Graphs are allowed to have cycles and can have more than one connected component.
- Some authors use the terms **nodes (κόμβοι)** and **arcs (τόξα)** instead of vertices and edges.
- Graphs can be **undirected (μη κατευθυνόμενοι)** or **directed (κατευθυνόμενοι).**

Examples of Graphs (Undirected)



Examples of Graphs (Directed)

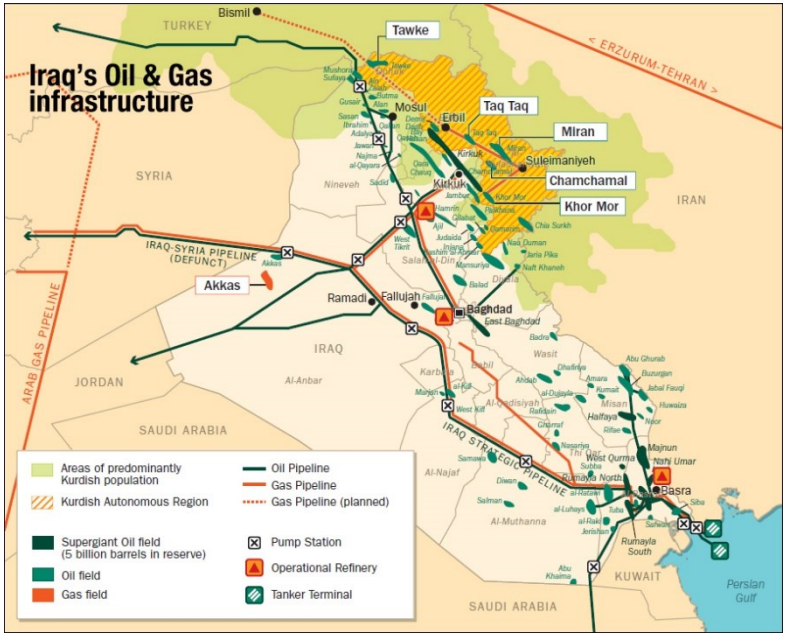


Examples of Graphs

- Transportation networks
- **Interesting problem**: What is the path with one or more stops of shortest overall distance connecting a starting city and a destination city?

Examples (cont'd)

- A network of oil pipelines
- **Interesting problem**: What is the maximum possible overall flow of oil from the source to the destination?

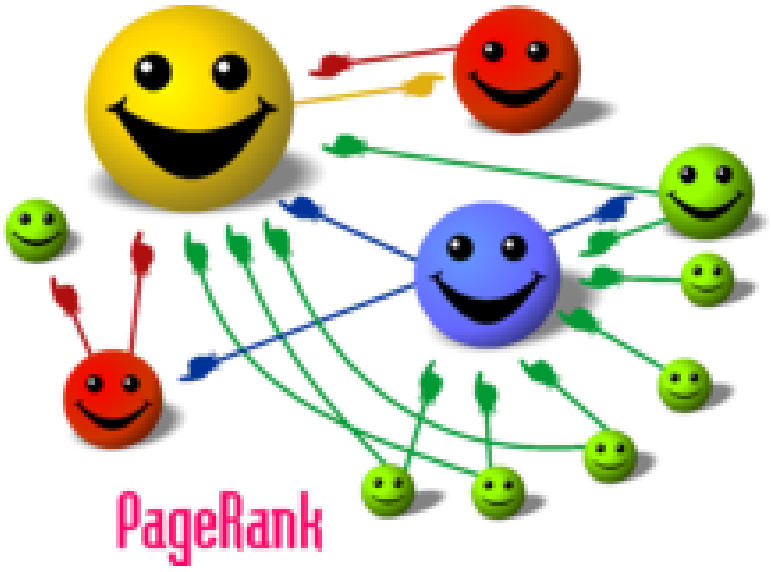


Examples (cont'd)

- The Internet
- **Interesting problem**: Deliver an e-mail from user A to user B

Examples (cont'd)

- The Web
- **Interesting problem**: What is the PageRank of a Web site?



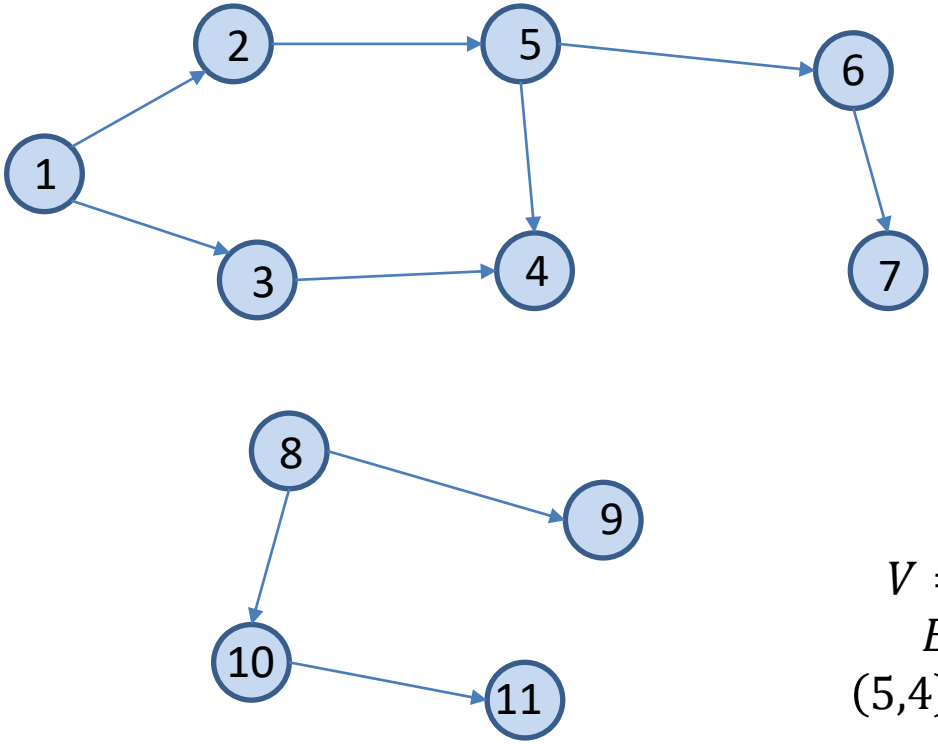
Examples (cont'd)

- The Facebook social network
- **Interesting problem**: Are John and Mary connected? What interesting clusters exist?

Formal Definitions

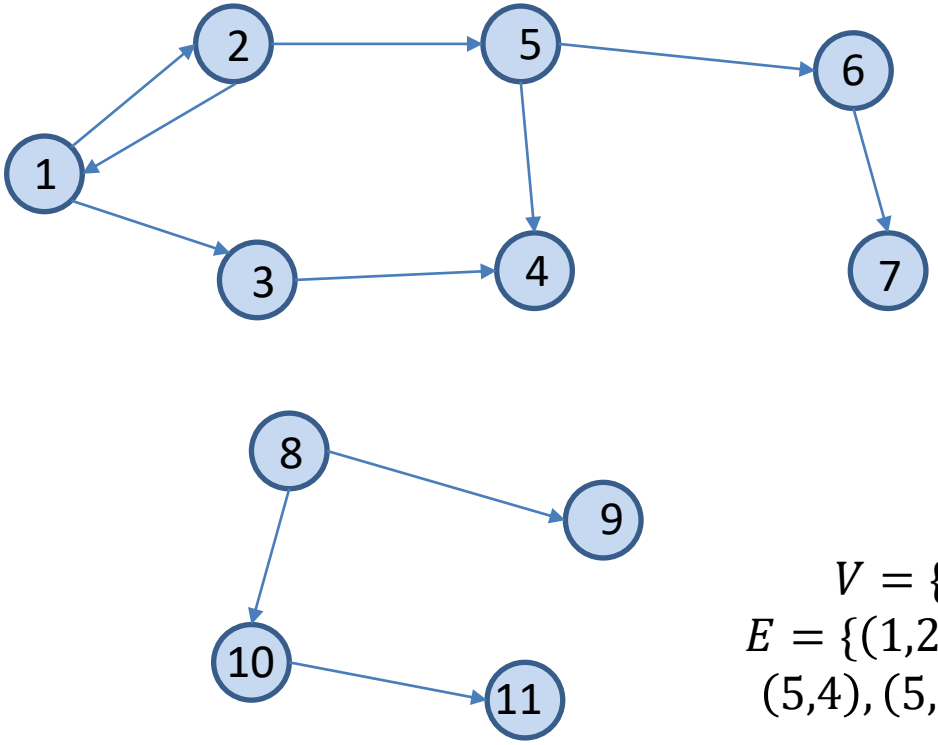
- A **graph** $G = (V, E)$ consists of a set of **vertices** V and a set of **edges** E, where the edges in E are formed from pairs of **distinct** vertices in V.
- If the edges have directions then we have a **directed graph (κατευθυνόμενο γράφο)** or **digraph**. In this case edges are ordered pairs of vertices e.g., (u, v) and are called **directed**. If (u, v) is a directed edge then u is called its **origin** and v is called its **destination**.
- If the edges do not have directions then we have an **undirected graph (μη-κατευθυνόμενος γράφο)**. In this case edges are unordered pairs of vertices e.g., $\{v, u\}$ and are called **undirected**.
- For simplicity, we will use the directed pair notation noting that in the undirected case (u, v) is the same as (v, u).

Example of a Directed Graph



$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$
$$E = \{(1,2), (1,3), (2,5), (3,4),$$
$$(5,4), (5,6), (6,7), (8,9), (8,10), (10,11)\}$$

One More Example of a Directed Graph

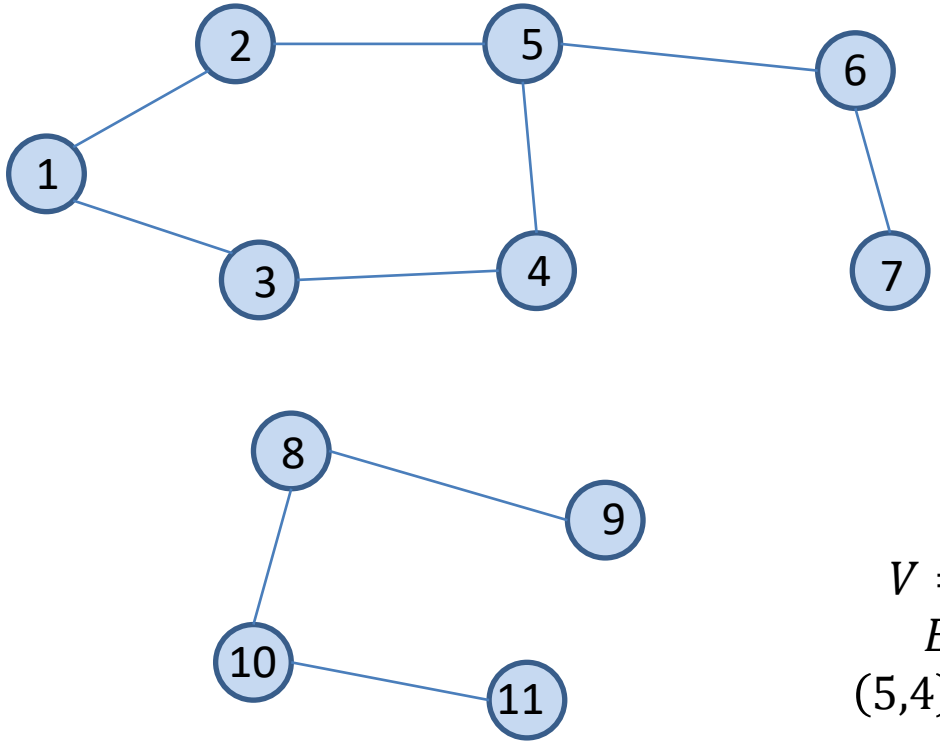


$$G = (V, E)$$

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

$$E = \{(1,2),\ (2,1), (1,3), (2,5), (3,4), (5,4), (5,6), (6,7), (8,9), (8,10), (10,11)\}$$

Example of an Undirected Graph



$$G = (V, E)$$
$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$
$$E = \{(1,2), (1,3), (2,5), (3,4), (5,4), (5,6), (6,7), (8,9), (8,10), (10,11)\}$$

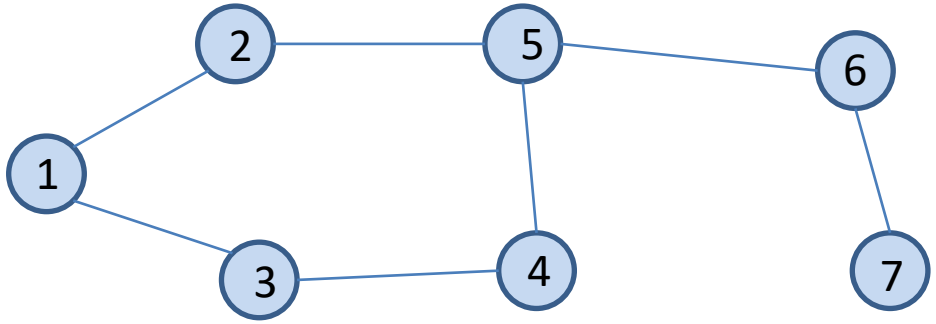
Note

- In the following slides, when we say simply graph in a definition, the definition will apply to both directed and undirected graphs.

Definitions (cont'd)

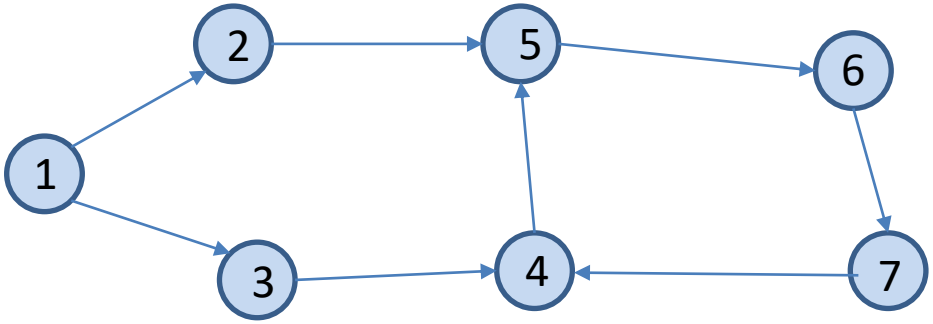
- Two different vertices v_i and v_j in a graph $G = (V, E)$ are said to be **adjacent (γειτονικές)** if there exists an edge $e \epsilon E$ such that $e = (v_i, v_j)$.
- When the graph is undirected, the adjacency relation is symmetric.
- In an undirected graph, an edge is said to be **incident (προσπίπτουσα)** on a vertex if the vertex is one of the edge's endpoints.
- If (u, v) is an edge in a directed graph $G = (V, E)$, we say that (u, v) is **incident from** or **leaves** vertex u and is **incident to** or **enters** vertex v.

Examples – Undirected Graph



Vertices 1 and 2 are adjacent. Edge $(1,2)$ is incident on vertices 1 and 2.

Examples – Directed Graph

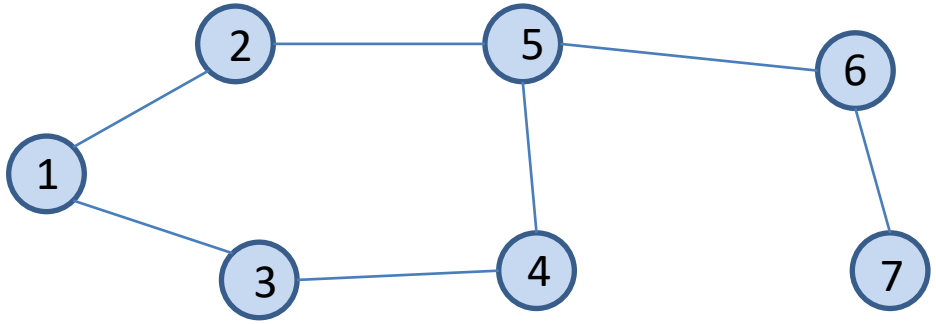


Vertices 1 and 2 are adjacent. Edge $(1, 2)$ is incident from (or leaves) vertex 1 and is incident on (or enters) vertex 2.

Definitions (cont'd)

- A **path (μονοπάτι)** p in a graph $G = (V, E)$ is a sequence of vertices of V of the form $p = v_1 v_2 \dots v_n, (n \geq 2)$ in which each vertex v_i is adjacent to the next one v_{i+1} (for $1 \leq i \leq n - 1$).
- In the above case, we say that p **contains** the vertices $v_1, v_2, \dots, v_n$ and the edges (v_i, v_{i+1}) for $1 \leq i \leq n - 1$.
- The **length** of a path is the number of edges in it.
- A path is **simple** if each vertex in the path is distinct.
- A **subpath** of a path is a contiguous subsequence of its vertices.
- In a directed graph, we often use the term **directed path** for obvious reasons.

Examples – Undirected Graph

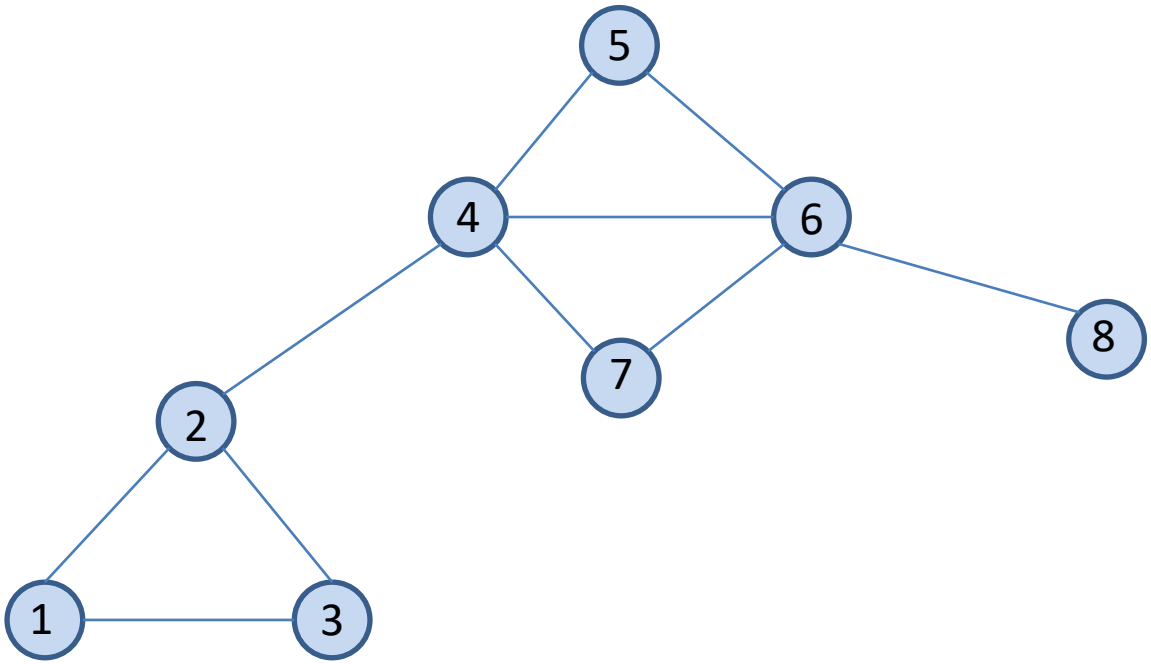


The sequence of vertices $1, 2, 5, 6$ is a simple path of length 3.
The path $1, 2, 5, 6, 5$ is not simple.
The path $2, 5, 6$ is a subpath of the path $1, 2, 5, 6, 7$.

Definitions (cont'd)

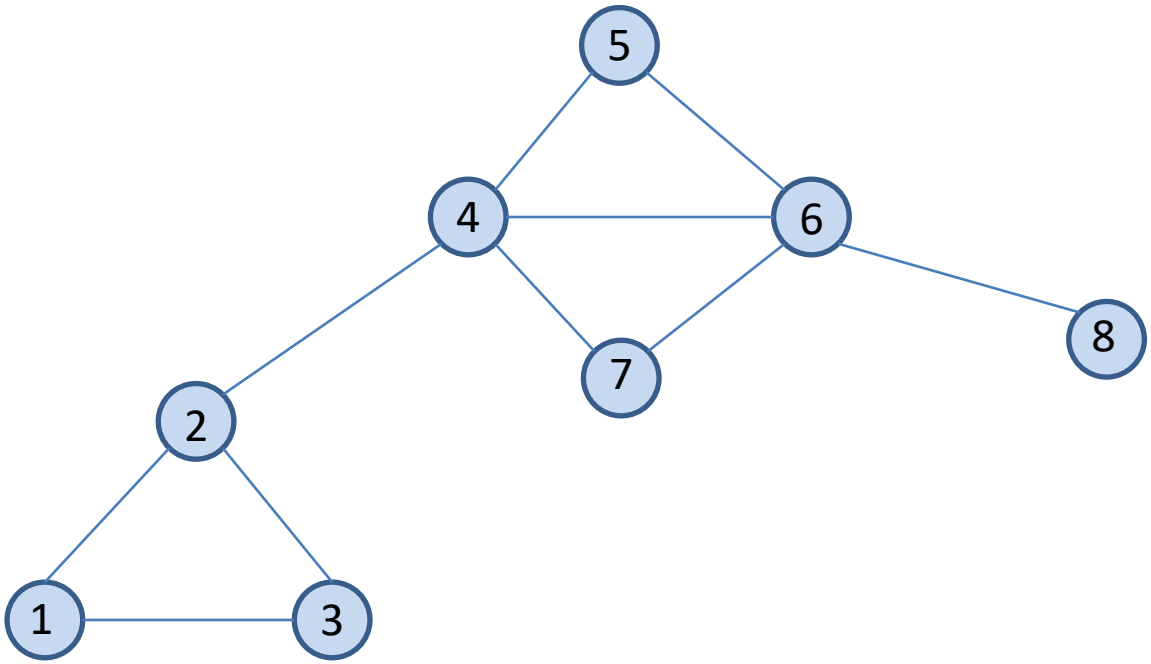
- A **cycle** is a path $p = v_1 v_2 \ldots v_n$ of length greater than one that begins and ends at the same vertex (i.e., $v_1 = v_n$).
- A **simple cycle** is a path that travels through three or more **distinct** vertices and connects them into a loop.
- Formally, if p is a path of the form $p = v_1 v_2 \ldots v_n$, then p is a **simple cycle** if and only if $n > 3, v_1 = v_n$ and $v_i \neq v_j$ for distinct i and j in the range $1 \leq i, j \leq n - 1$.
- Simple cycles **do not repeat edges**.

Examples – Undirected Graph



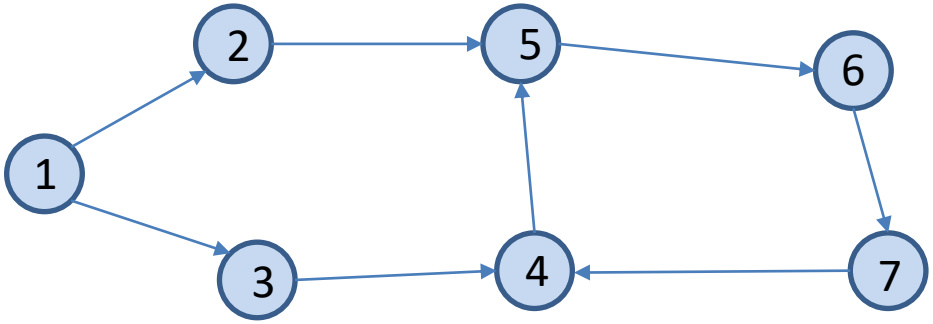
Four simple cycles: (1,2,3,1) (4,5,6,7,4) (4,5,6,4) (4,6,7,4)

Example (cont'd)



One non-simple cycle: (4,5,6,4,7,6,4)

Examples – Directed Graph

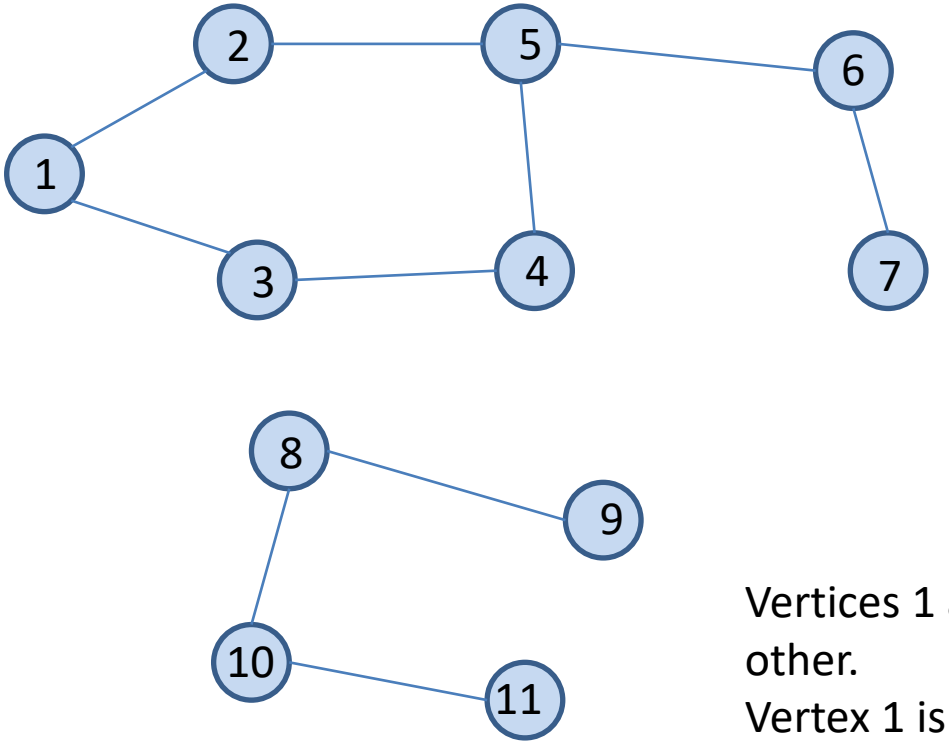


The sequence of vertices $1, 2, 5, 6$ is a simple (directed) path of length 3.
The sequence $2, 5, 4$ is not a path because there is no directed edge $(5, 4)$.
The path $5, 6, 7, 4, 5$ is a simple cycle.

Reachability

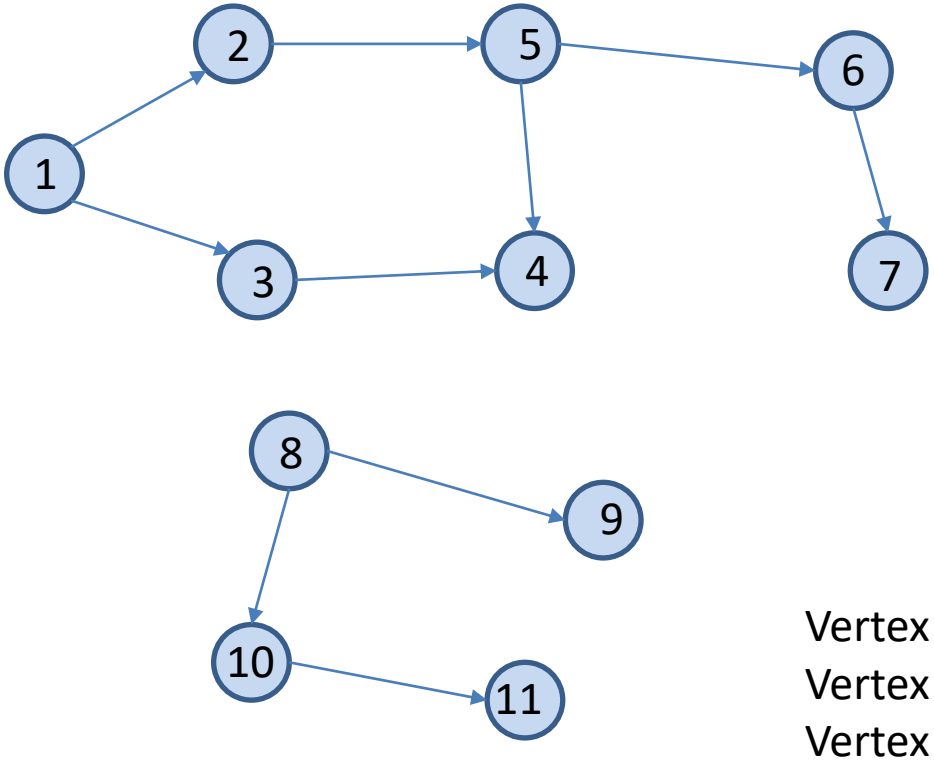
- Let G be a graph. If there is a path p from v to u in G then we say that u is **reachable (προσβάσιμη)** from v via p.

Examples - Undirected Graph



Vertices 1 and 6 are reachable from each other.
Vertex 1 is not reachable from vertex 8 (or 9, 10, 11).

Examples – Directed Graph

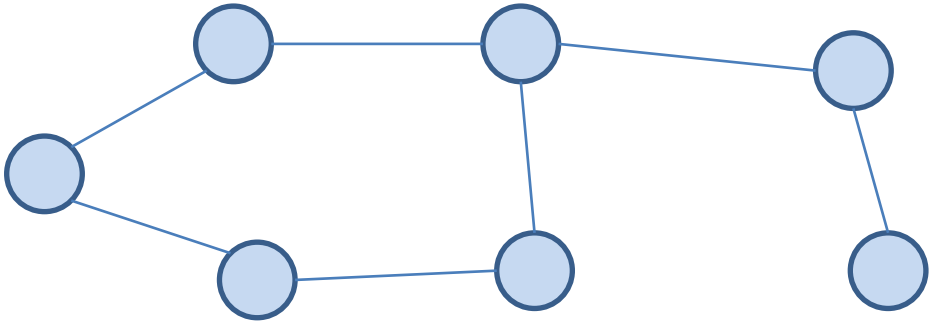


Vertex 6 is reachable from vertex 1.
Vertex 1 is not reachable from vertex 6.
Vertex 1 is not reachable from vertex 8 (or 9, 10, 11).

Connected Undirected Graphs

- An undirected graph G is **connected (συνεκτικός)** if every vertex is reachable from every other vertex in the graph (i.e., if there is a path from every vertex to every other vertex).

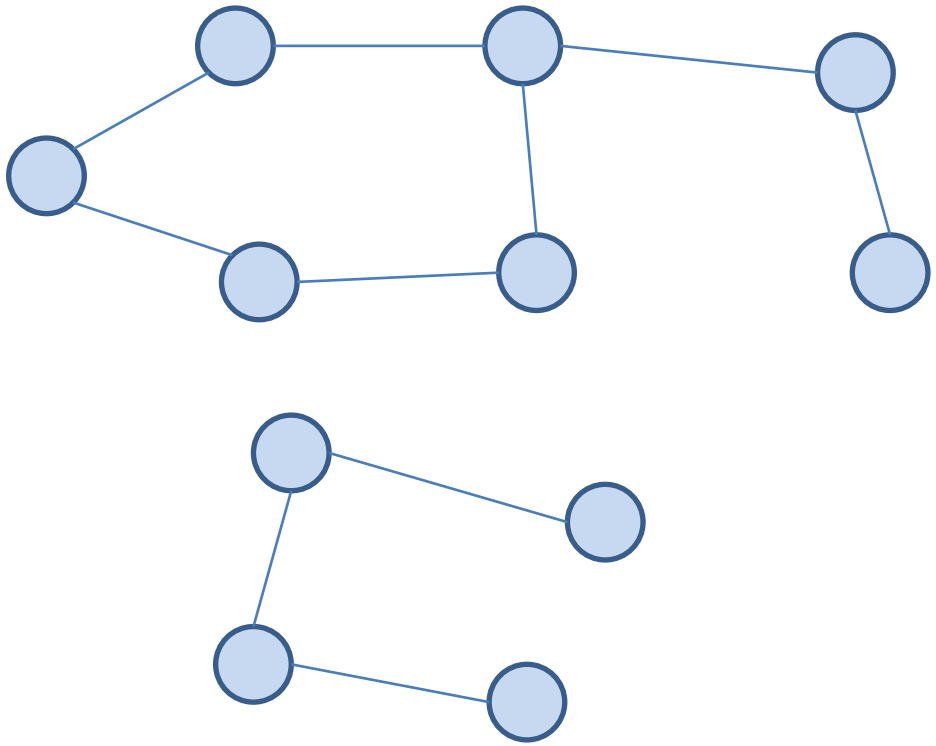
Example of Connected Undirected Graph



Connected Components of Undirected Graphs

- In the undirected graph $G = (V, E)$, a **connected component (συνεκτική συνιστώσα)** is a maximal subset S of the vertices V that are all reachable from each other.
- By maximal we mean that there is no bigger subset T of vertices in V such that T properly contains S and such that T has the same reachability property.
- An undirected graph G **can always be separated** into connected components $S_1, S_2, \ldots, S_n$ such that $S_i \cap S_j = \emptyset$ whenever $i \neq j$.
- The connected components of an undirected graph are the **equivalence classes** of the vertices under the "is reachable from" relation.
- An undirected graph is **connected** if it has exactly one connected component.

Example of Undirected Graph and its Separation into Two Connected Components



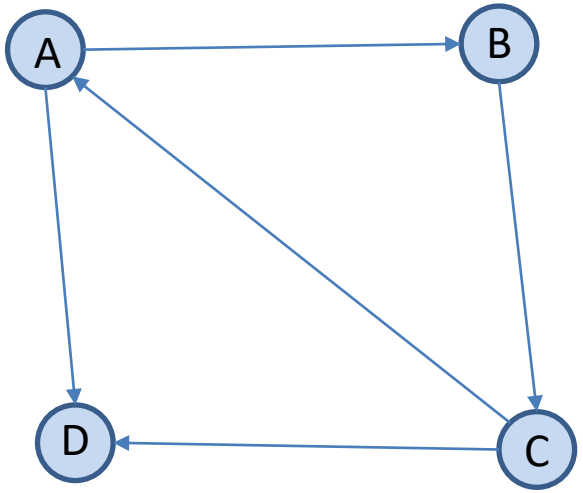
Strongly Connected Directed Graphs

- A directed graph is **strongly connected (ισχυρά συνεκτικός)** if every two vertices are reachable from each other (i.e., there is a path from the first to the second and vice versa).

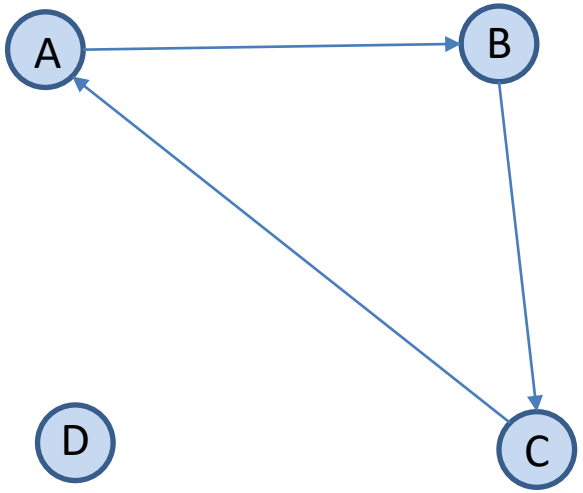
Strongly Connected Components of a Directed Graph

- A **strongly connected component (ισχυρά συνεκτική συνιστώσα)** of a directed graph is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex.
- More formally, let $G = (V, E)$ be a directed graph. We can partition V into equivalence classes $V_i, 1 \leq i \leq r$, such that vertices v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let $E_i, 1 \leq i \leq r$, be the set of edges with endpoints in V_i. The graphs $G_i = (V_i, E_i)$ are called the **strongly connected components** or just **strong components (ισχυρές συνιστώσες)** of G.
- A directed graph with only one strong component is **strongly connected**.

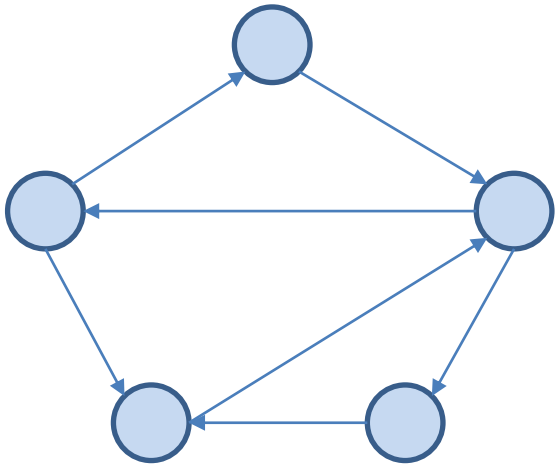
Example Directed Graph



The Strong Components of the Digraph



Another Example Directed Digraph

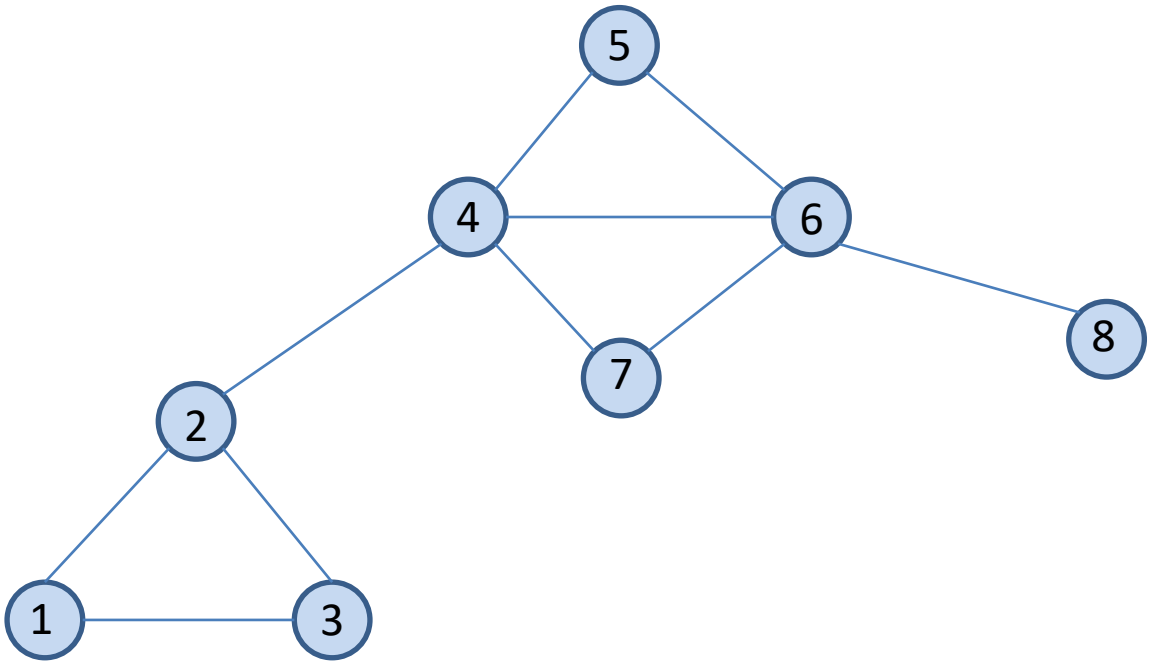


- This graph consists of a **single** strong component.

Degree in Undirected Graphs

- In an undirected graph G, the **degree** (**βαθμός**) of vertex x is the number of edges e in which x is one of the endpoints of e.
- The degree of a vertex x is denoted by $\deg(x)$.

Example



The degree of vertex 1 is 2. The degree of vertex 4 is 4. The degree of vertex 8 is 1.

Proposition

- If G is an undirected graph with m edges, then

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

- Proof?

Proof

- An edge (u, v) is counted twice in the summation above; once by its endpoint u and one by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges.

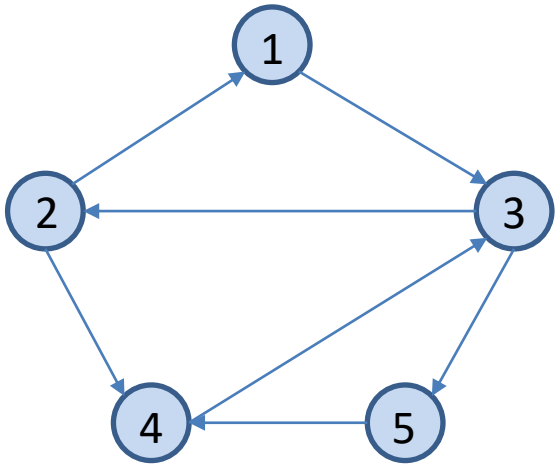
Predecessors and Successors in Directed Graphs

- If x is a vertex in a **directed** graph $G = (V, E)$ then the set of **predecessors (προηγούμενων)** of x denoted by $Pred(x)$ is the set of all vertices $y \in V$ such that $(y, x) \in E$.
- Similarly the set of **successors (επόμενων)** of x denoted by $Succ(x)$ is the set of all vertices $y \in V$ such that $(x, y) \in E$.

In-Degree and Out-Degree in Directed Graphs

- The **in-degree** of a vertex x is the number of predecessors of x.
- The **out-degree** of a vertex x is the number of successors of x.
- We can also define the in-degree and the out-degree by referring to the **incoming** and **outgoing** edges of a vertex.
- The in-degree and out-degree of a vertex x are denoted by $indeg(x)$ and $outdeg(x)$ respectively.

Example



The in-degree of vertex 4 is 2. The out-degree of vertex 4 is 1.

Proposition

- If G is a directed graph with m edges, then

$$\sum_{v\ in\ G} inde\, g(v) = \sum_{v\ in\ G} outdeg(v) = m.$$

- Proof?

Proof

- In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin vertex u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the out-degrees.

Proposition

- Let G be a graph with n vertices and m edges. If G is undirected, then $m \leq \frac{n(n-1)}{2}$, and if G is directed, then $m \leq n(n-1)$.
- Proof?

Proof

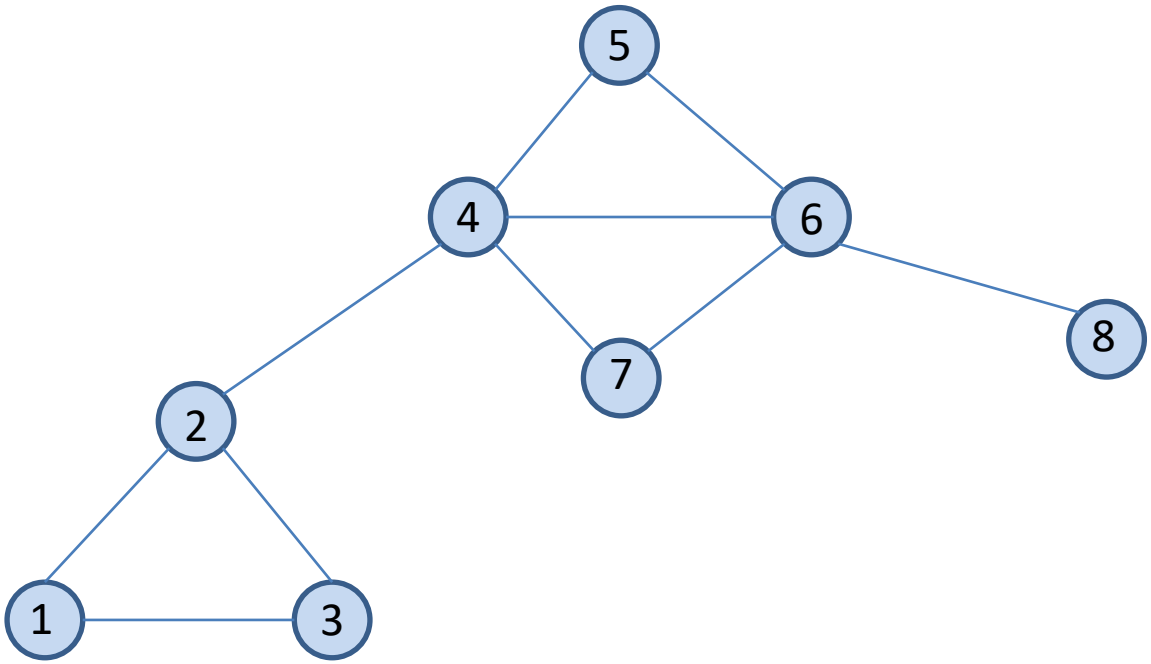
- If G is undirected then the maximum degree of a vertex is $n-1$. Therefore, from the previous proposition about the sum of the degrees, we have $2m \leq n(n-1)$.
- If G is directed then the maximum in-degree of a vertex is $n-1$. Therefore, from the previous proposition about the sum of the in-degrees, we have $m \leq n(n-1)$.

More definitions

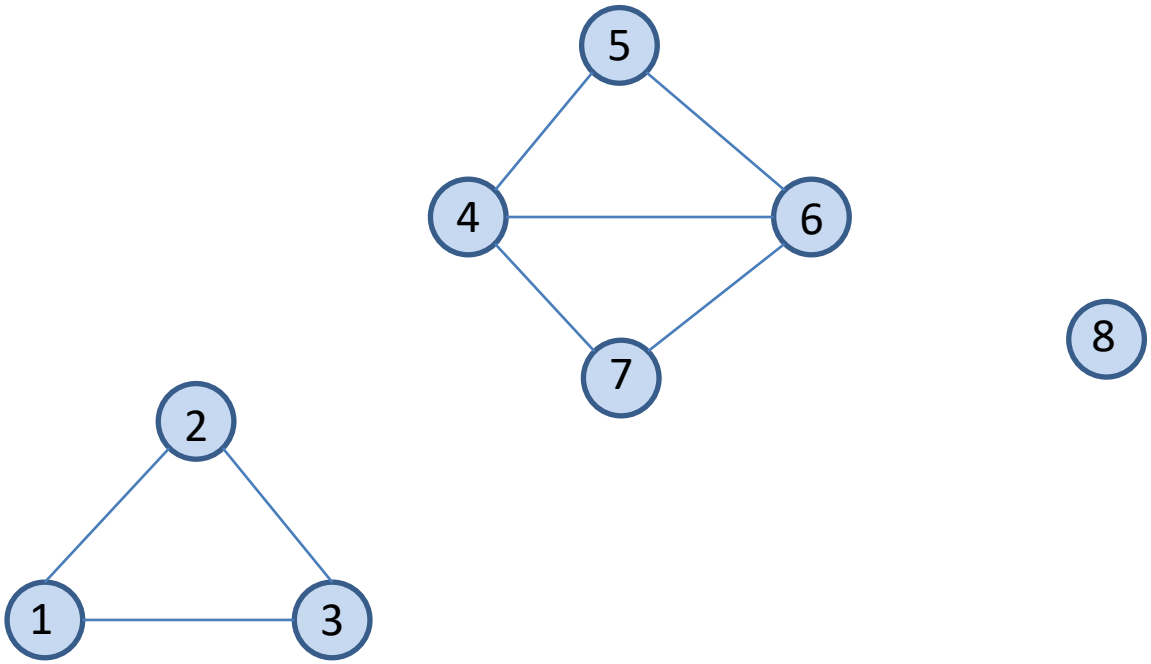
- A **subgraph (υπογράφος)** of a graph G is a graph H whose set of vertices and set of edges are subsets of the set of vertices and the set of edges of G respectively.

- A **spanning subgraph (υπογράφος επικάλυψης)** of G is a subgraph of G that contains all the vertices of G.

Example Undirected Graph G

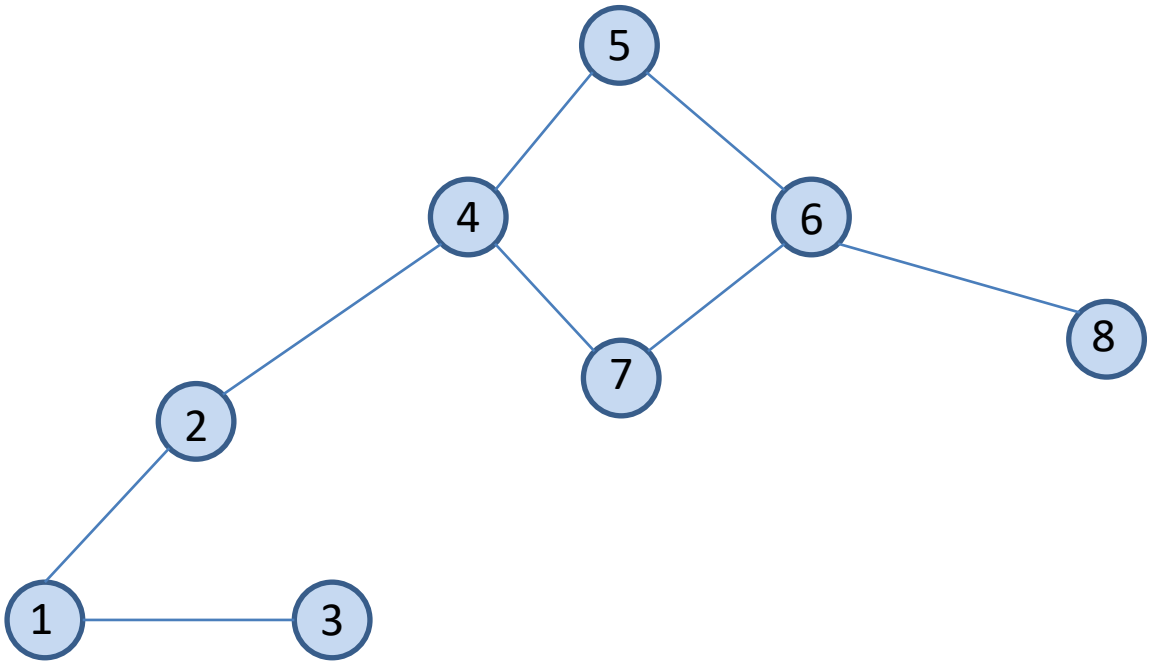


Example (cont'd)



The above are three subgraphs of the previous graph G.

Example (cont'd)

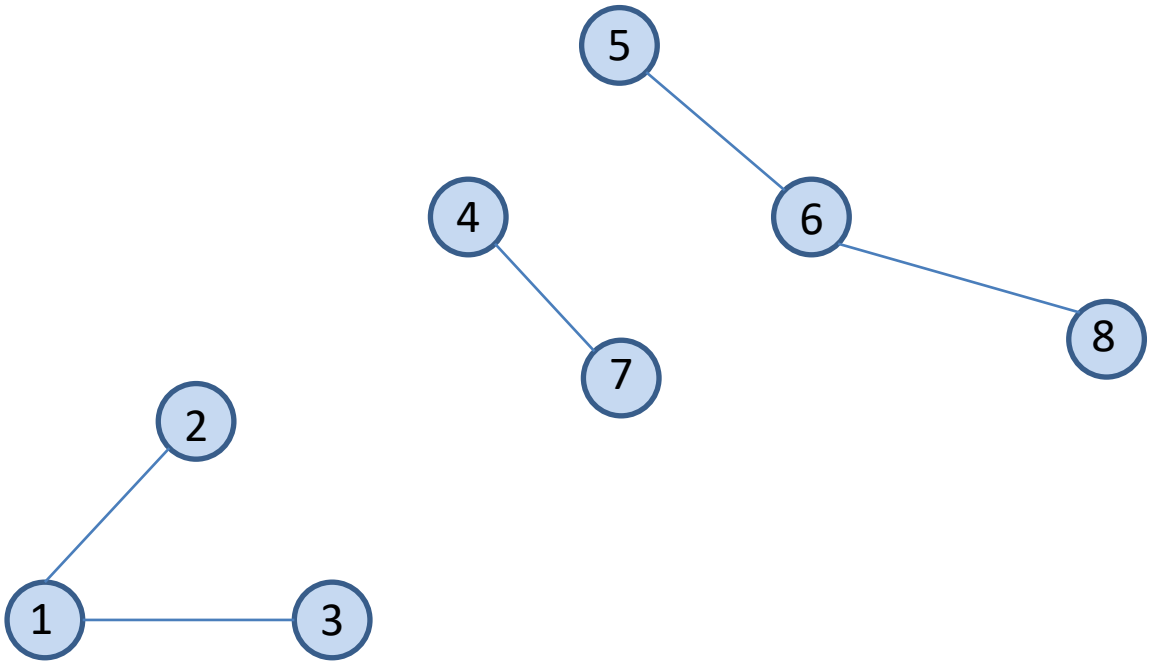


The above graph is a spanning subgraph of the previous graph G.

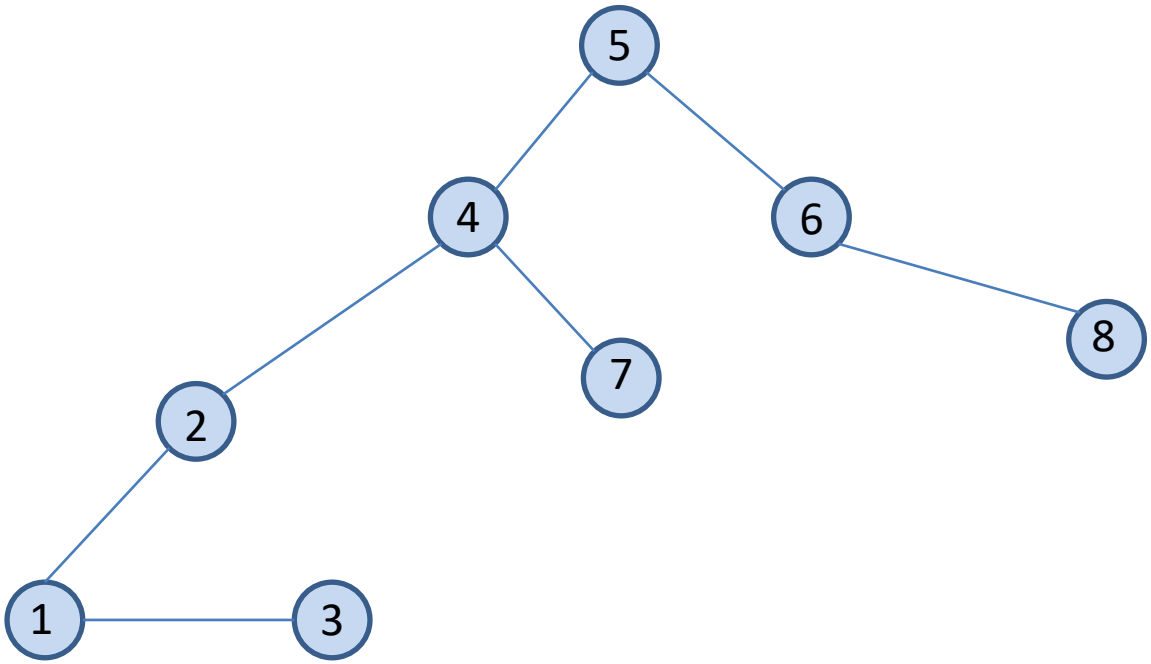
More Definitions for Undirected Graphs

- A **forest (δάσος)** is an undirected graph without cycles.
- A **free tree (ελεύθερο δένδρο)** is a connected forest i.e., a connected, undirected graph without cycles.
- The trees that we studied in earlier lectures are **rooted trees (δένδρα με ρίζα)** and they are different than free trees.
- A **spanning tree (δένδρο επικάλυψης ή επικαλύπτον δένδρο)** of an undirected graph is a spanning subgraph that is a free tree.
- **A spanning forest (δάσος επικάλυψης ή επικαλύπτον δάσος)** is an undirected graph which is the union of spanning trees, one for each connected component of the graph.

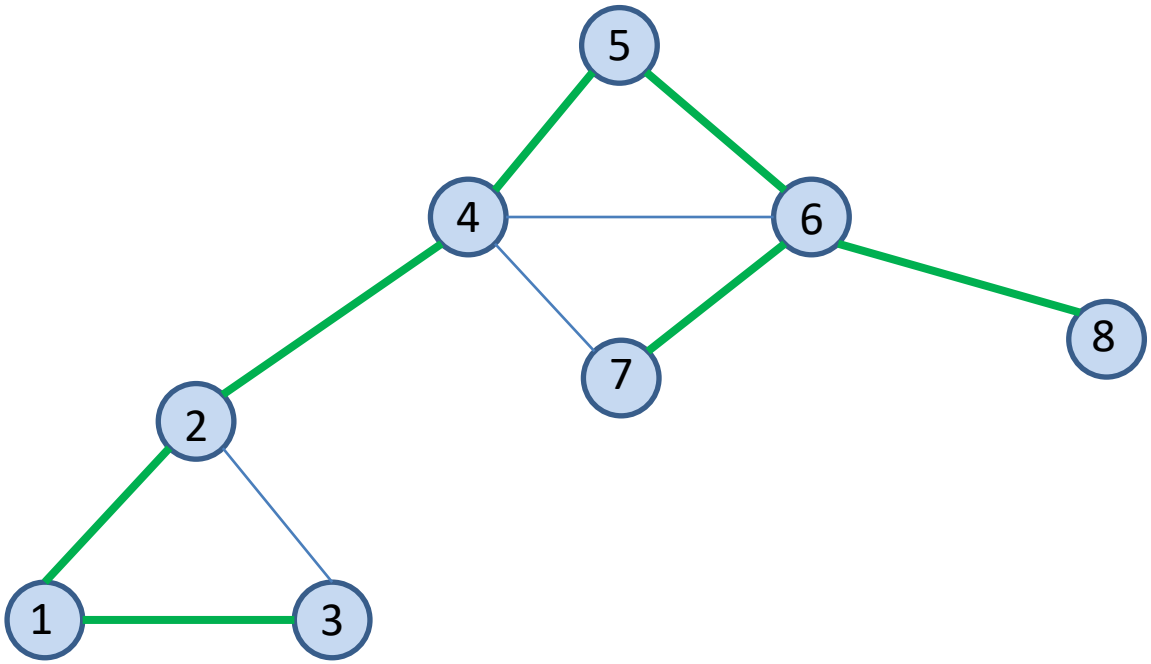
Example Forest



Example Free Tree

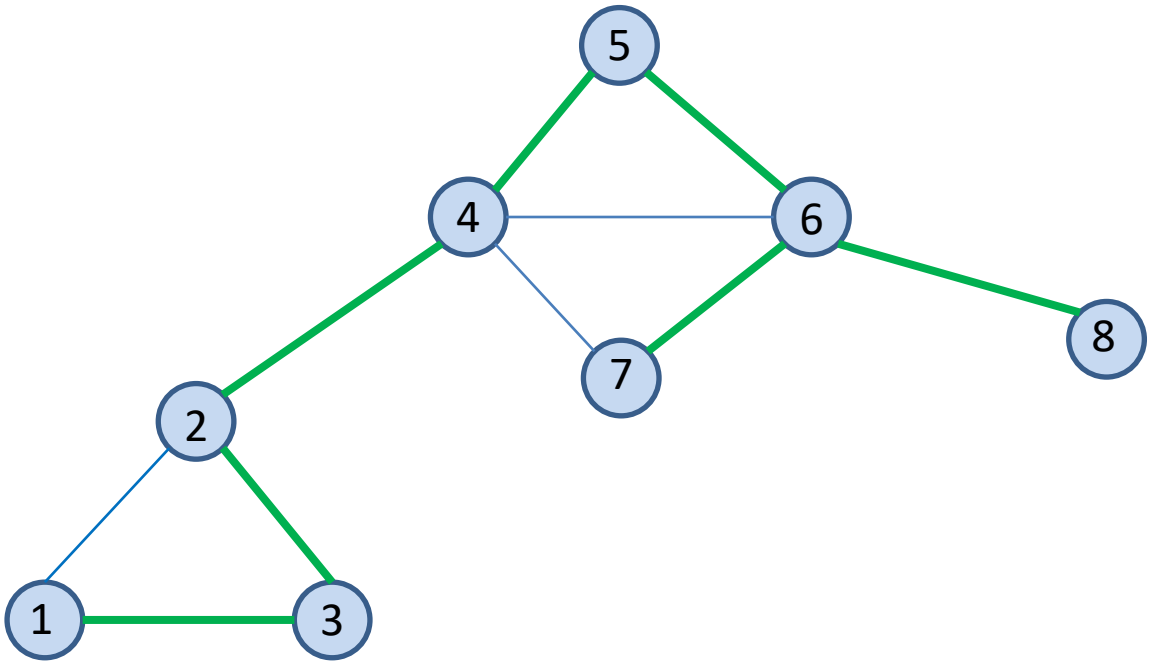


Example



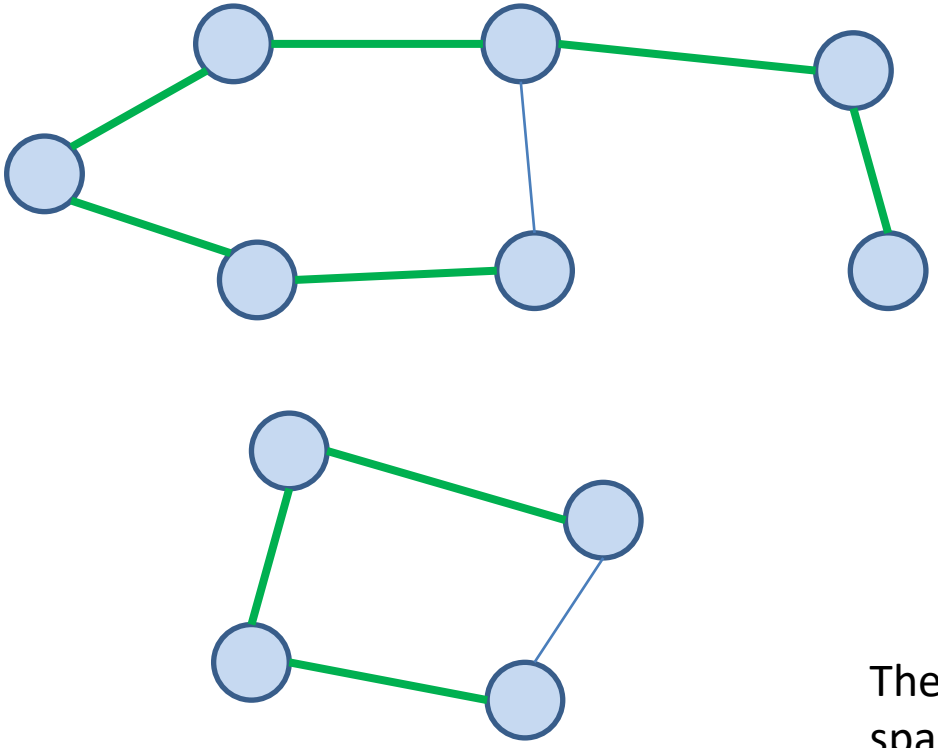
The thick green lines define a spanning tree of the graph.

Example (cont'd)



The thick green lines define another spanning tree of the graph.

Example

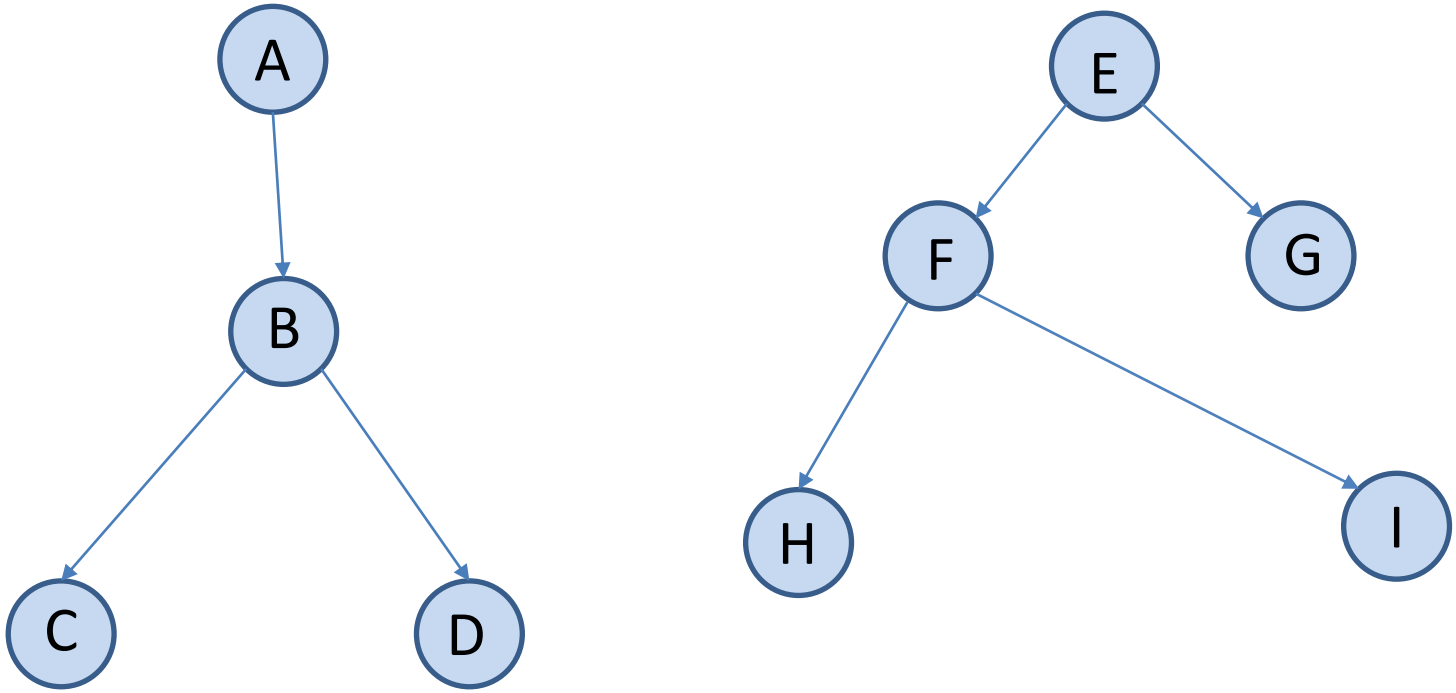


The thick green lines define a spanning forest which consists of two spanning trees.

Directed Graphs

- We can give similar definitions for forest and tree for the case of directed graphs but now our trees will be directed, rooted trees in which all edges point away from the root (technically such a tree is called an **arborescence** or **directed rooted tree** if you don't like words that come from French).
- The terms spanning tree and spanning forest are used **only** for undirected graphs.

Example

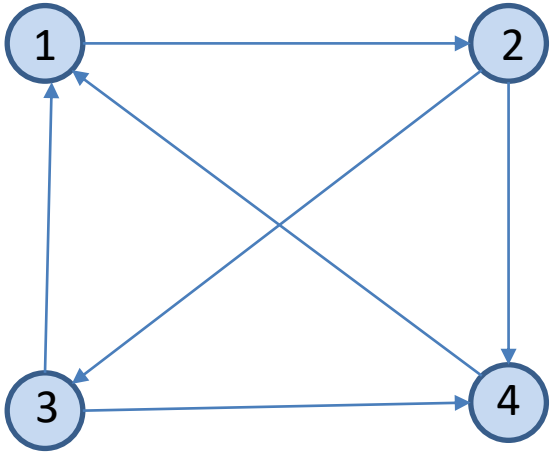


- A forest consisting of two directed rooted trees.

Graph Representations: Adjacency Matrices

- Let $G = (V, E)$ be a graph. Suppose we number the vertices in V as $v_1, v_2, \ldots, v_n$.
- The **adjacency matrix (πίνακας γειτνίασης)** T corresponding to G is an $n \times n$ matrix such that $T[i, j] = 1$ if there is an edge $(v_i, v_j) \in E$, and $T[i, j] = 0$ if there is no such edge in E.

Example



A directed graph G

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	1
4	1	0	0	0

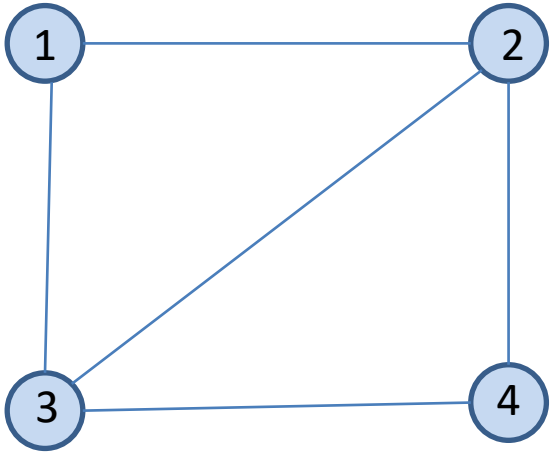
The adjacency matrix for graph G

Adjacency Matrices

- The adjacency matrix of an **undirected graph** G is a **symmetric matrix** i.e., $T[i, j] = T[j, i]$ for all i and j in the range $1 \leq i, j \leq n$.

- The adjacency matrix for a **directed graph** need not be symmetric.

Example

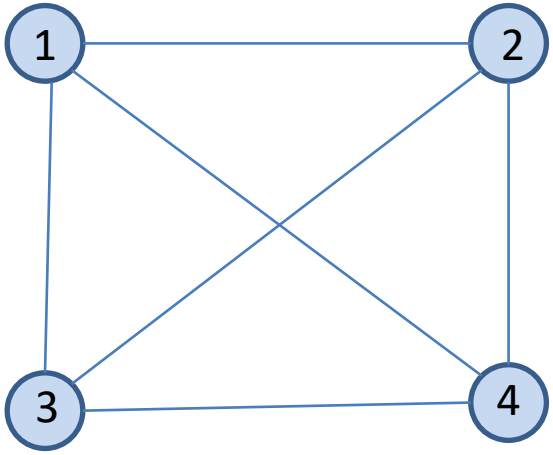


An undirected graph G

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

The adjacency matrix for graph G

Another Example



An undirected graph G

	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

The adjacency matrix for graph G

Adjacency Matrices (cont'd)

- The **diagonal entries** in an adjacency matrix (of a directed or undirected graph) **are zero**, since graphs, as we have defined them, are not permitted to have looping self-referential edges that connect a vertex to itself.

Adjacency Matrices in C

```
#define MAXVERTEX 10

typedef enum {FALSE, TRUE} Boolean
/* FALSE and TRUE will be 0 and 1 respectively */

typedef Boolean
        AdjacencyMatrix[MAXVERTEX][MAXVERTEX]

typedef struct graph {
   int n    /*number of vertices in graph */
   AdjacencyMatrix A;
} Graph;
```

Adjacency Sets

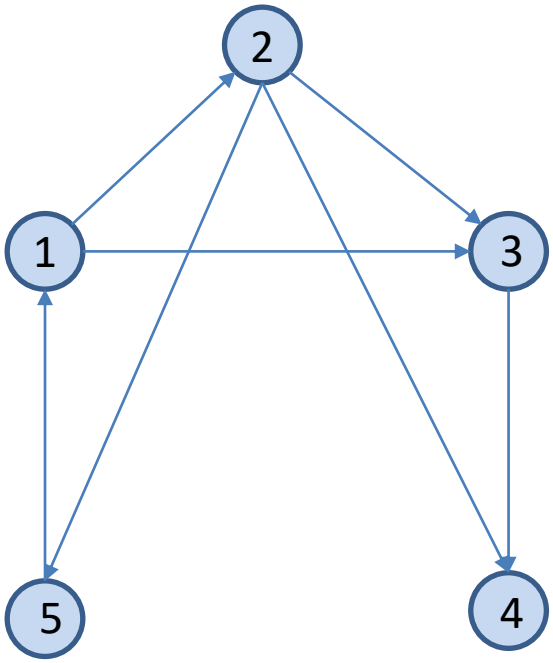
- Another way to define a graph $G = (V, E)$ is to specify **adjacency sets (σύνολα γειτνίασης)** for each vertex in V.
- Let V_x stand for the set of all vertices **adjacent** to x in an undirected graph G or the set of all vertices that are **successors** of x in a directed graph G.
- If we give both the vertex set V and the collection $A = \{V_x | x \in V\}$ of adjacency sets for each vertex in V then we have given enough information to define the graph G.

Graph Representations: Adjacency Lists

- When writing code in C or other languages, we use **adjacency lists (λίστες γειτνίασης)** to represent the adjacency set V_x for each vertex x in the graph.

- Adjacency matrices and adjacency lists are the two **standard representations** of graphs.

Example Directed Graph

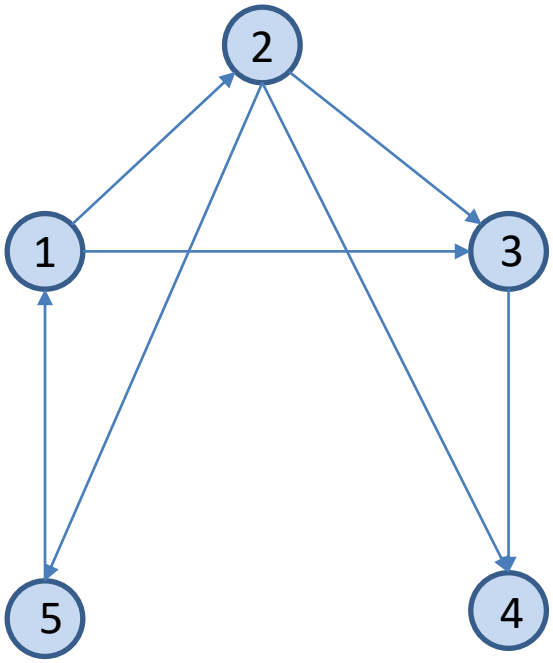


A directed graph G

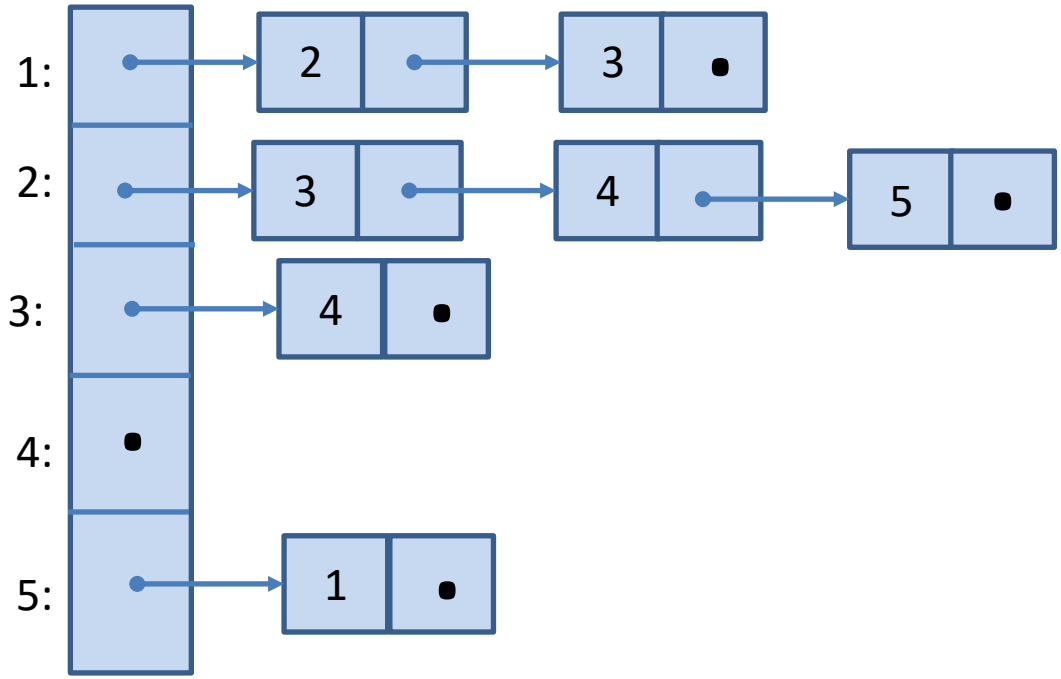
Vertex Number	Out Degree	Adjacency list
1	2	2 3
2	3	3 4 5
3	1	4
4	0	
5	1	1

The **sequential** adjacency lists for graph G. Notice that vertices are listed in their **natural order**.

Example Directed Graph

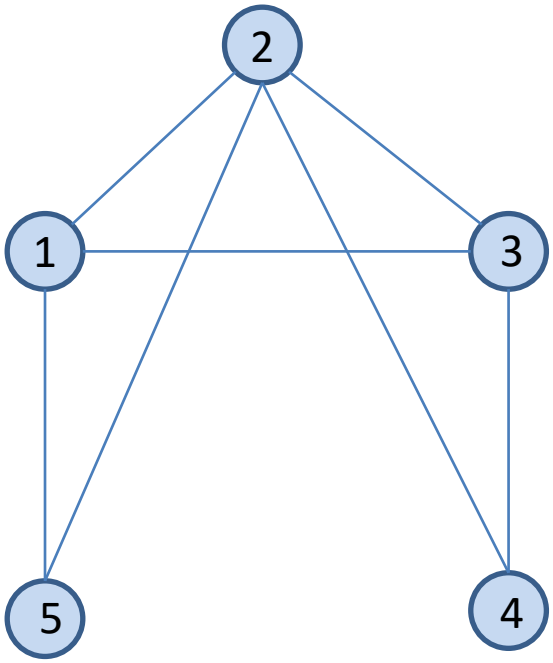


A directed graph G



The **linked** adjacency lists for graph G. Notice that vertices in a list are organized according to their **natural order.**

Example Undirected Graph



An undirected graph G

Vertex Number	Degree	Adjacency list
1	3	2 3 5
2	4	1 3 4 5
3	3	1 2 4
4	2	2 3
5	2	1 2

The sequential adjacency lists for graph G

Example Undirected Graph

- The linked adjacency list representation for the graph of the previous slide is similar to the one for the directed case.

Assumptions

- In the previous slides, the vertices of the graph appear in their **natural order** in the arrays and linked lists used in the adjacency matrix and the adjacency links representations.
- This will be our assumption in all examples from now on.
- In the adjacency lists representation, we also showed the degrees of vertices. This information is not necessary and can be omitted.

Sequential Adjacency Lists in C

```
typedef int AdjacencyList[MAXVERTEX];

typedef struct graph{
   int n; /*number of vertices in graph */
   int degree[MAXVERTEX];
   AdjacencyList A[MAXVERTEX];
} Graph;
```

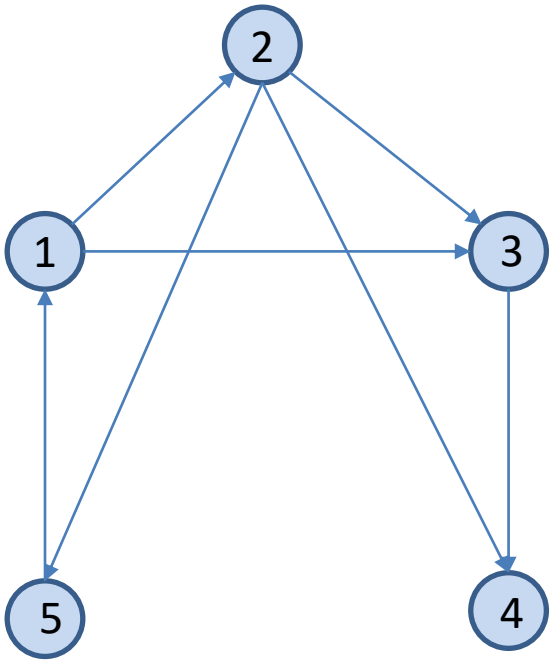
In the above representation, each index `i` of array `A` is a vertex of the graph while each element `A[i]` is an array storing the adjacency list of vertex `i`.

Linked Adjacency Lists in C

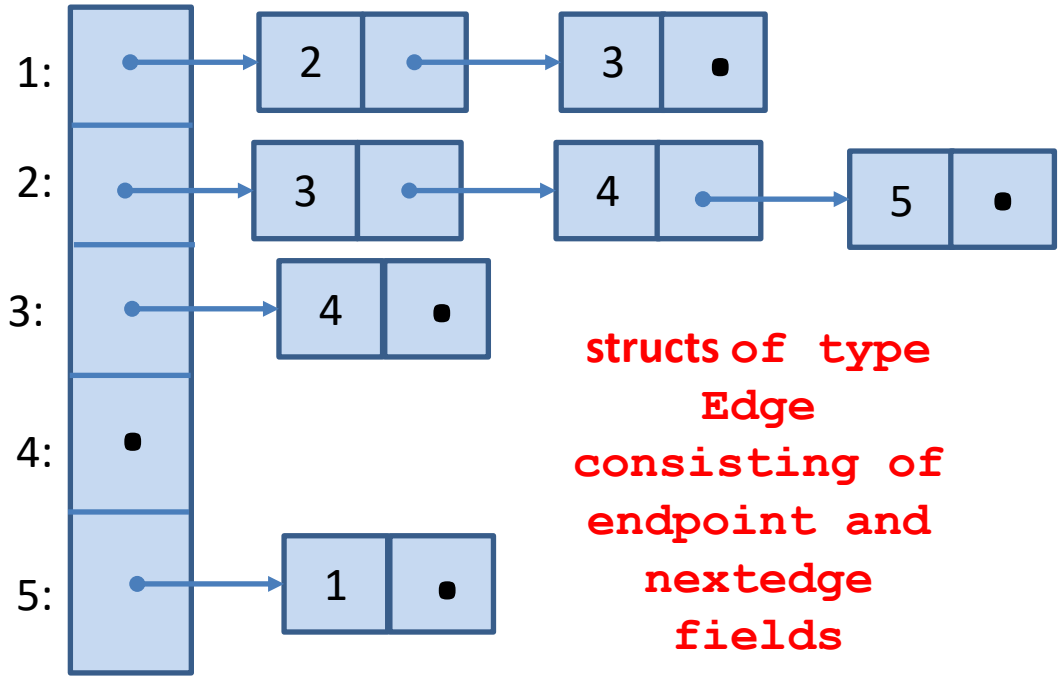
```
typedef int Vertex;

typedef struct edge {
  Vertex endpoint;
  struct edge *nextedge;
} Edge;

typedef struct graph{
   int n; /*number of vertices in graph */
   Edge *firstedge[MAXVERTEX];
} Graph;
```

In the above representation, each index `i` of array `firstedge` is a vertex and each element `firstedge[i]` is a pointer to a linked list that contains the adjacent vertices of vertex `i`. This implementation of adjacency lists is used in the code that we give below.

Example Directed Graph and its Representation



A directed graph G



array of pointers `firstedge`

structs of type Edge consisting of endpoint and nextedge fields

field n of type int

Linked Adjacency Lists in C (cont'd)

- The previous representation used an **array for the vertices and linked lists for the adjacency lists.**
- We can use **linked lists for the vertices** as well, as follows.

Linked Adjacency Lists in C (cont'd)

```
typedef struct vertex Vertex;
typedef struct edge Edge;

struct vertex {
  Edge *firstedge;
  Vertex *nextvertex;
}

struct edge {
  Vertex *endpoint;
  Edge *nextedge;
};

typedef Vertex *Graph;
```

We do **not** use this implementation of adjacency lists in the code that we give below.

Space Complexity

- The space complexity of the **adjacency matrix** representation of a graph with n vertices is $\boldsymbol{O(n^2)}$.
- The space complexity of the **adjacency list** representation of a graph with n vertices and e edges is $\boldsymbol{O(n+e)}$.

Questions

- Which graph representation would you use under the following circumstances?
  - The graph is **sparse** (e.g., it contains 10^6 nodes but only 10^5 edges).
  - The graph is **dense** (e.g., it contains 10^6 nodes and 10^{10} edges).

- The answer will depend on what amount of space we have available and what operations we would like to perform on the graph.

Graph Searching

- To search a graph `G`, we need to visit all vertices of `G` in some systematic order.
- Let us define an enumeration

```
typedef enum {FALSE, TRUE} Boolean;
```

- Each vertex `v` can be a structure with a `Boolean` valued member `v.Visited` which is initially `FALSE` for all vertices of `G`. When we visit `v`, we will set it to `TRUE`.

An Algorithm for Graph Searching

```
void GraphSearch(G,v)
{
     Let G=(V,E) be a graph.
     Let C be an empty container.

     for (each vertex x in V){
        x.Visited=FALSE;
     }

     Put v into C;
     while (C is non-empty){
        Remove a vertex x from container C;
        if (!(x.Visited)){
           Visit(x);
           x.Visited=TRUE;
           for (each vertex w in V_x){
              if (!(w.Visited)) Put w into C;
           }
         }
      }
}
```

Question

- If we only put `w` into `C` if `!(w.Visited)` is true, shouldn't any `x` we remove from `C` automatically satisfy `!(x.Visited)` ?
- Is it possible then that we don't need the test

```
if (!(x.Visited)) ?
```

Answer

- The `if (!(x.Visited))` check inside the loop (after removing `x` from `C`) is **necessary**.
- Here's why: a vertex can be added to the container `C` **multiple times** before it is ever removed and processed. **This happens when a vertex is a neighbor of several other vertices that are processed earlier.**
- The `if (!(x.Visited))` check makes sure that this situation is handled properly.

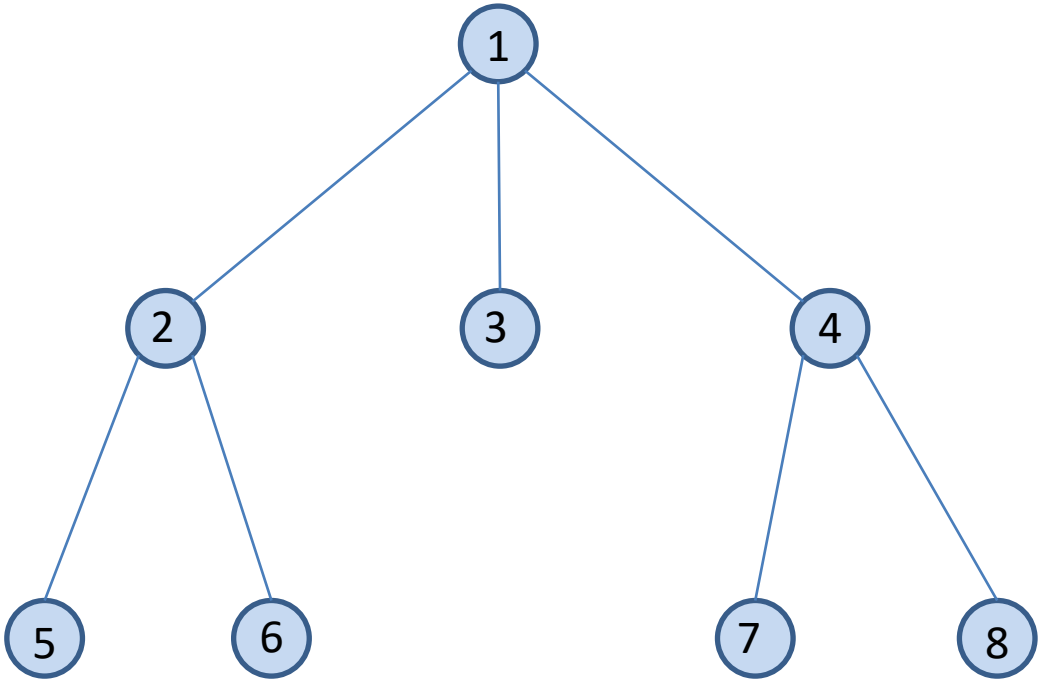
Exercise

- Give a concrete example that illustrates the discussion on the previous slide.

Graph Searching (cont'd)

- Let us now consider algorithm `GraphSearch` in the case that the container `C` is a **stack**.

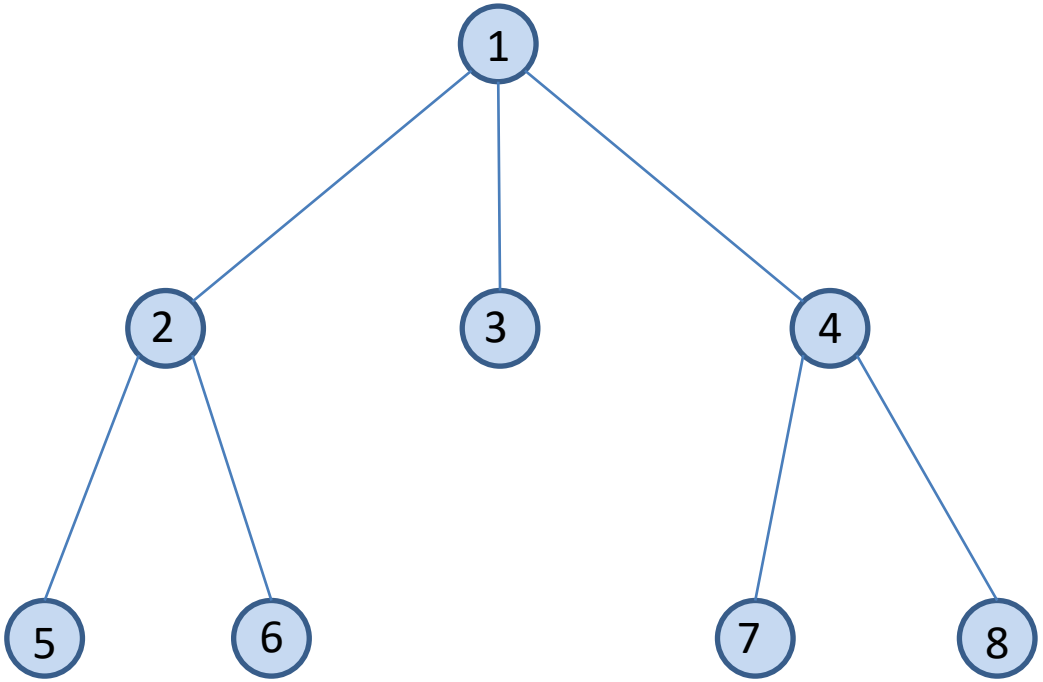
Example Undirected Graph



Representation with Adjacency Lists

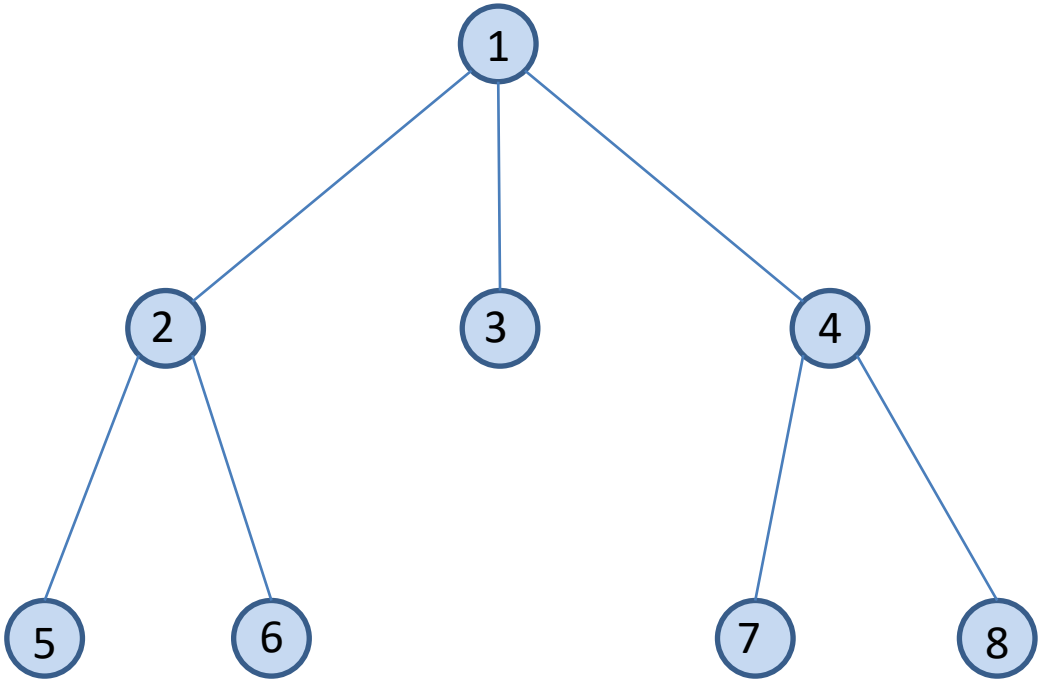
Vertex Number	Adjacency list
1	2 3 4
2	1 5 6
3	1
4	1 7 8
5	2
6	2
7	4
8	4

Example (cont'd)



What is the order in which vertices are visited if the start vertex is 1?

Example (cont'd)



The vertices are visited in the order 1, 4, 8, 7, 3, 2, 6 and 5.

Depth-First Search (DFS)

- When $\mathbb{C}$ is a **stack**, the tree in the previous example is searched in **depth-first order**.
- **Depth-first search (αναζήτηση πρώτα κατά βάθος)** at a vertex always goes down (by visiting unvisited children) before going across (by visiting unvisited brothers and sisters).
- Depth-first search of a graph is analogous to a **pre-order traversal** of an ordered tree.

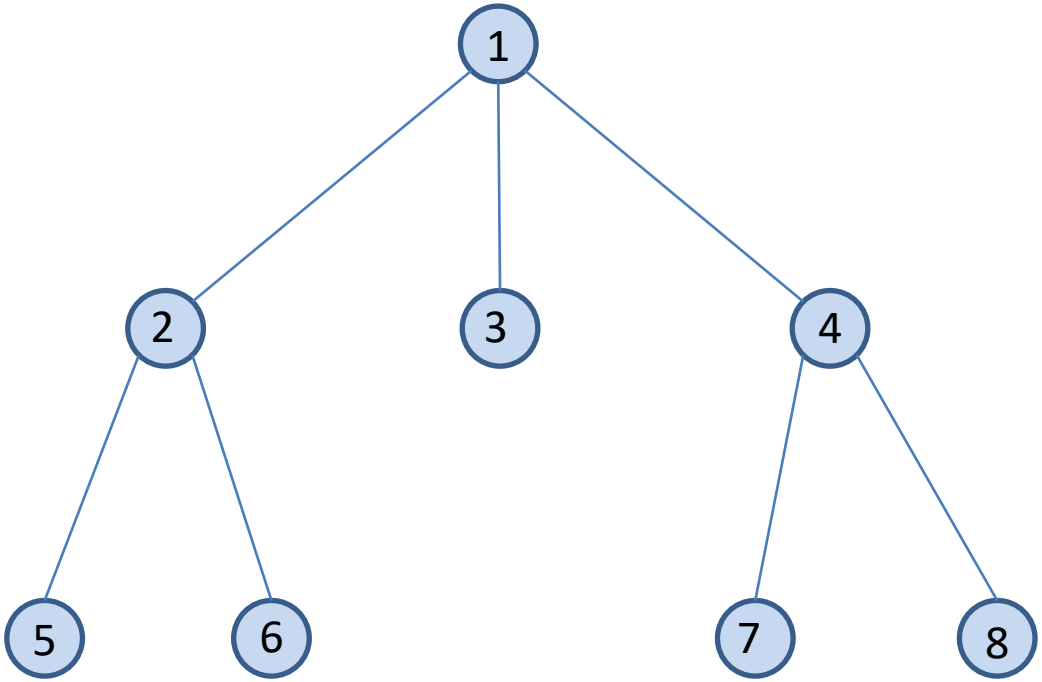
DFS (cont'd)

- The strategy followed by DFS, as its name implies, is to search "deeper" in the graph whenever possible.
- In DFS, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it.
- When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, the one of them is selected as a new source and the search is repeated from that source.
- This entire process is repeated until all vertices are discovered.

Graph Searching (cont'd)

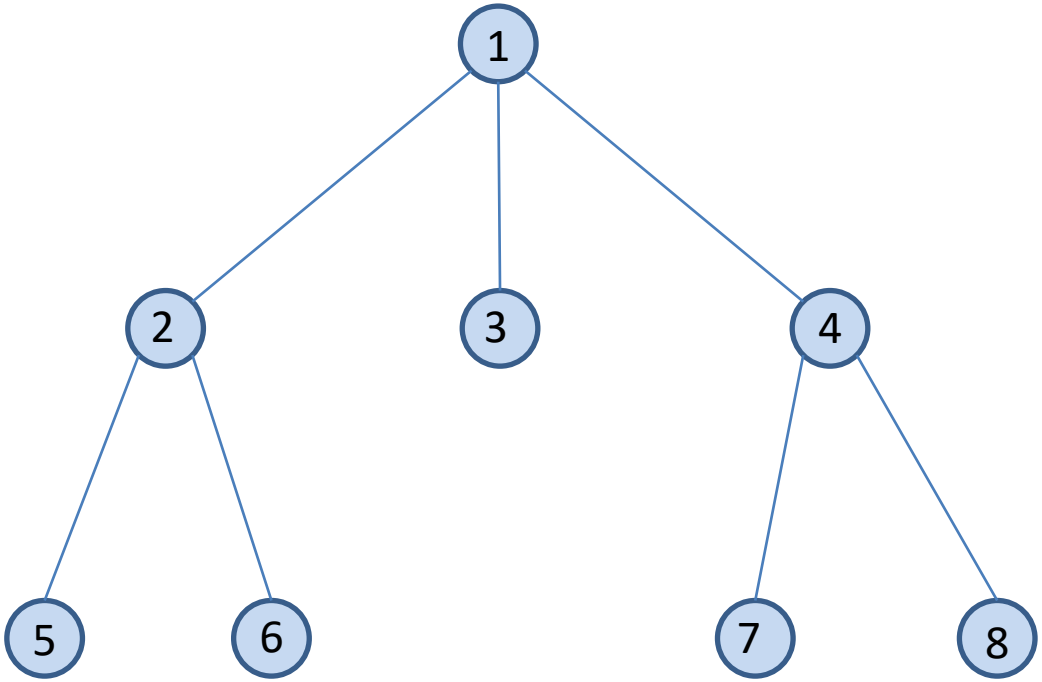
- Let us consider what happens when the container C is a **queue**.

Example Undirected Graph



What is the order in which vertices are visited if the start vertex is 1?

Example (cont'd)



The vertices are visited in the order 1, 2, 3, 4, 5, 6, 7 and 8.

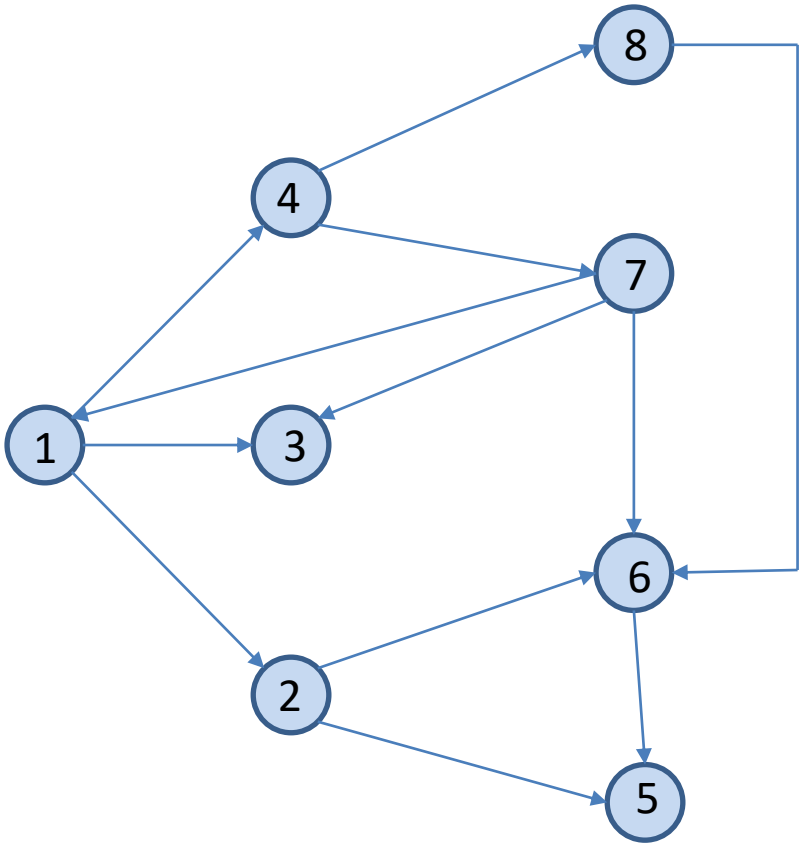
Breadth-First Search (BFS)

- When $\mathbb{C}$ is a **queue**, the tree in the previous example is searched in **breadth-first order**.
- **Breadth-first search (αναζήτηση πρώτα κατά πλάτος)** at a vertex always goes broad before going deep.
- Breadth-first traversal of a graph is analogous to a traversal of an ordered tree that visits the nodes of the tree in **level-order**.
- BFS subdivides the vertices of a graph in **levels**. The starting vertex is at level 0, then we have the vertices adjacent to the starting vertex at level 1, then the vertices adjacent to these vertices at level 2 etc.

BFS (cont'd)

- BFS works as follows.
- We start from the start vertex (level 0) and visit all the vertices that we can reach by following one edge (these are the vertices at level 1).
- Then we visit all vertices that we can reach from the start vertex by following two edges (these are the vertices at level 2).
- We continue with the vertices at level 3 and so forth.

Example Directed Graph

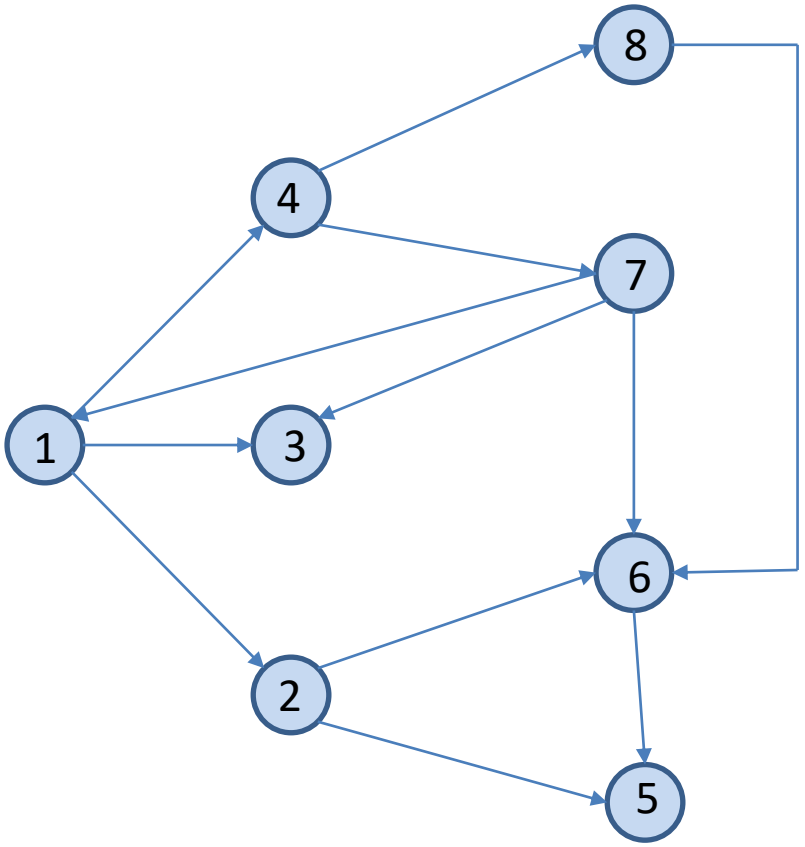


What is the order of visiting vertices for DFS if the start vertex is 1?

Representation with Adjacency Lists

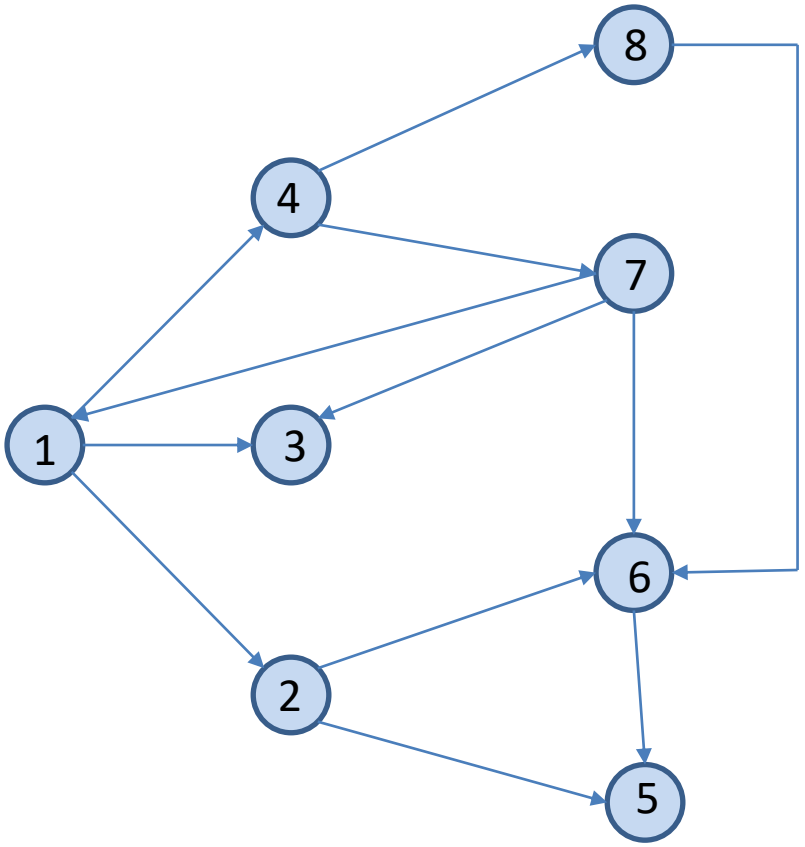
Vertex Number	Adjacency list
1	2 3 4
2	5 6
3	
4	7 8
5	
6	5
7	1 3 6
8	6

Example (cont'd)



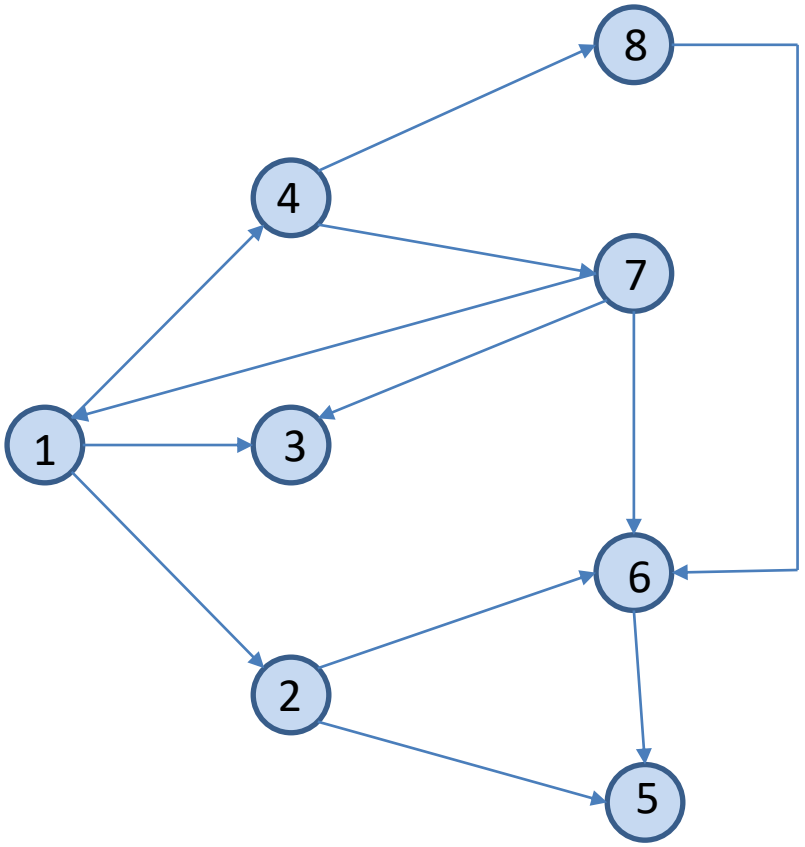
DFS visits the vertices in the order 1, 4, 8, 6, 5, 7, 3 and 2.

Example (cont'd)



What is the order of visit for BFS if the start vertex is 1?

Example (cont'd)



BFS visits the vertices in the order 1, 2, 3, 4, 5, 6, 7 and 8.

Exhaustive Search

- Both the stack version or the queue version of the algorithm `GraphSearch` will visit every vertex in a graph `G` provided that `G` consists of a single strongly connected component.
- If this is not the case, then we can enumerate all the vertices of `G` and run `GraphSearch` starting from each one of them in order to visit all the vertices of `G`.

Exhaustive Search (cont'd)

```
void ExhaustiveGraphSearch(G)
{
  Let G=(V,E) be a graph.

  for (each vertex v in G){
    GraphSearch(G, v)
  }
}
```

Theseus in the Labyrinth

- DFS can be simulated using a **string** and **a can of paint** for painting the vertices i.e., using a version of the algorithm that Theseus might have used in the labyrinth of the Minotaur!
- BFS is analogous to a group of people exploring the graph by fanning out in all directions.

Implementing DFS in C

- We will now show how to implement depth-first search in C **using a recursive function (not a stack as earlier). This is the standard implementation of DFS.**
- We will use the **linked adjacency lists** representation of a graph.
- We will write a function `DepthFirst` which calls the recursive function `Traverse`.

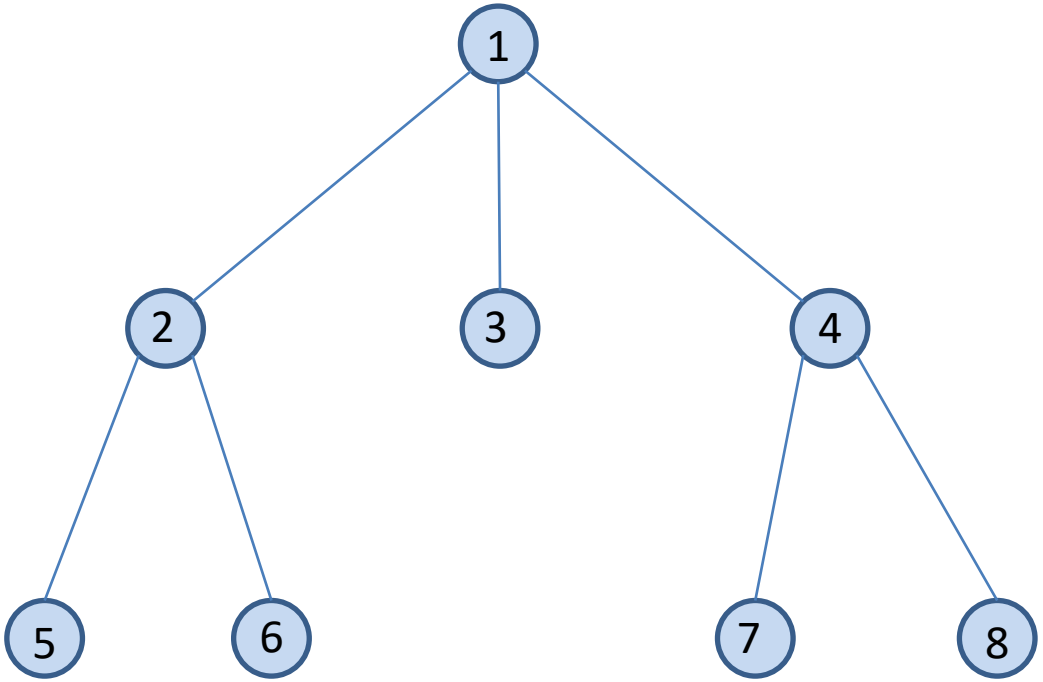
Implementing DFS in C (cont'd)

```
/* global variable visited */
Boolean visited[MAXVERTEX];

/* DepthFirst: depth-first traversal of a graph
   Pre: The graph G has been created.
   Post: The function Visit has been performed at each vertex of G in
   depth-first order
   Uses: Function Traverse produces the recursive depth-first order */

void DepthFirst(Graph G, void (*Visit)(Vertex x))
{

  Vertex v;

  for (v=0; v < G.n; v++)
    visited[v]=FALSE;
  for (v=0; v < G.n; v++)
    if (!visited[v]) Traverse(G, v, Visit);
}
```

Implementing DFS in C (cont'd)

```
/* Traverse: recursive traversal of a graph
   Pre: v is a vertex of graph G
   Post: The depth first traversal, using function Visit, has been
   completed for v and for all vertices adjacent to v.
   Uses: Traverse recursively, Visit */

void Traverse(Graph G, Vertex v, void (*Visit)(Vertex x))
{
  Vertex w;
  Edge *curedge;

  visited[v]=TRUE;
  Visit(v);

  curedge=G.firstedge[v];    /* curedge is a pointer to the first edge (v,_) of V */
  while (curedge){
    w=curedge->endpoint;      /* w is a successor of v and (v,w) is the current edge */
    if (!visited[w]) Traverse(G, w, Visit);
    curedge=curedge->nextedge;  /*curedge is a pointer to the next edge (v,_) of V */
  }
}
```

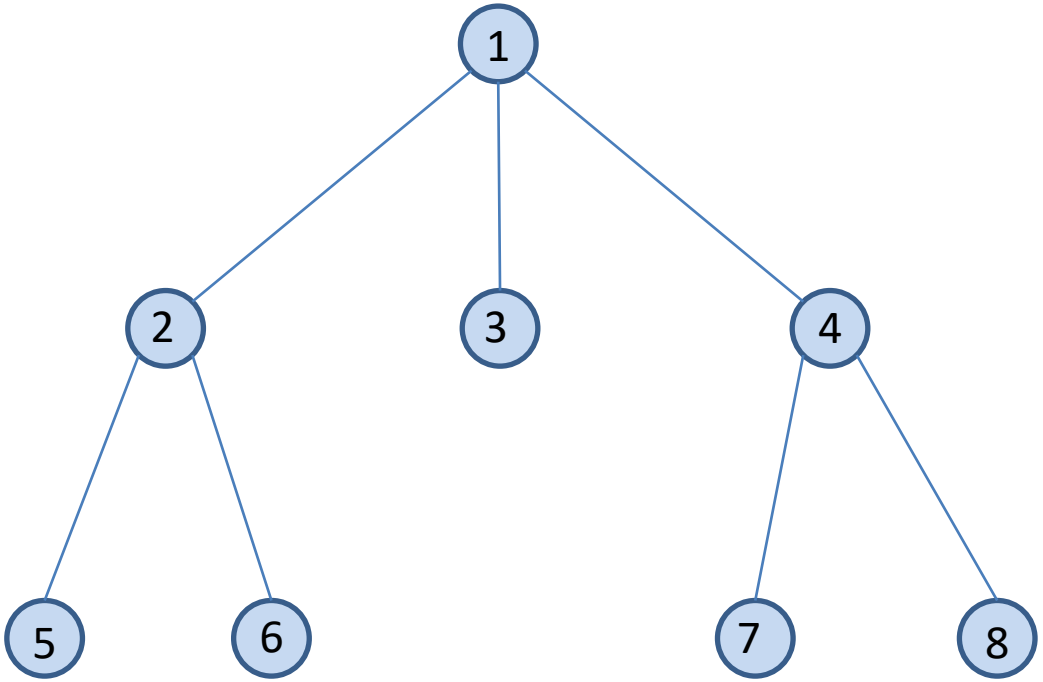
Example Undirected Graph



Representation with Adjacency Lists

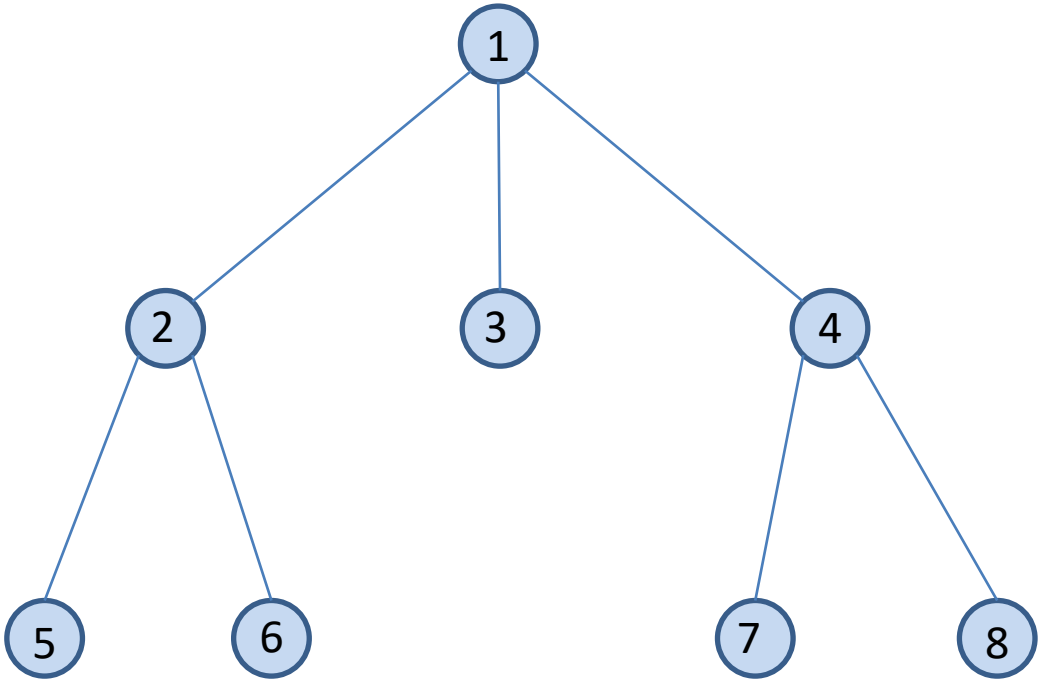
Vertex Number	Adjacency list
1	2 3 4
2	1 5 6
3	1
4	1 7 8
5	2
6	2
7	4
8	4

Example of Recursive DFS



What is the order vertices are visited if the start vertex is 1?

Example (cont'd)



The vertices are visited in the order 1, 2, 5, 6, 3, 4, 7 and 8. This is **different** than the order we got when using a stack!

Important

- There is **more than one way to traverse a graph** in a DFS fashion (e.g., starting from a different node gives us different DFS traversals).
- The **order** the vertices of a graph are visited by DFS depends on:
  - The representation of the graph (e.g., adjacency matrix vs. adjacency list, the order of vertices in the two representations).
  - The implementation of the DFS algorithm (e.g., recursive DFS vs. using a stack).

Important (cont'd)

- DFS **traverses all edges and visits all the vertices** reachable from the start vertex (i.e., all the vertices in the connected component containing the start vertex), **regardless of in what order** it examines edges incident on each vertex.

Complexity of DFS

- DFS as implemented above (by the recursive function `Traverse` and using adjacency lists) on a graph with e edges and n vertices has complexity $\boldsymbol{O(n+e)}$.
- To see why, observe that **on no vertex is `Traverse` called more than once**, because as soon as we call `Traverse` with parameter v, we mark v visited and we never call `Traverse` on a vertex that has previously been marked as visited.
- Thus, the total time spent going down the adjacency lists is proportional to the lengths of those lists, that is $O(e)$.
- We also need an initialization step which marks all vertices as unvisited before we run `Traverse`. This step has complexity $O(n)$.
- Thus, the total complexity is $O(n+e)$.

Complexity of DFS (cont'd)

- If DFS is implemented using an adjacency matrix, then its complexity will be $O(n^2)$.
- If the graph is **dense (πυκνός)**, that is, it has close to $O(n^2)$ edges the difference of the two implementations is minor as they would both run in $O(n^2)$ time.
- If the graph is **sparse (αραιός)**, that is, it has close to $O(n)$ edges, then the adjacency matrix approach would be much slower than the adjacency list approach.

Complexity of DFS (cont'd)

- The complexity of function `DepthFirst` is $O(n(n+e))$ because we also have the second `for` statement that makes sure that all connected components of the graph are explored.

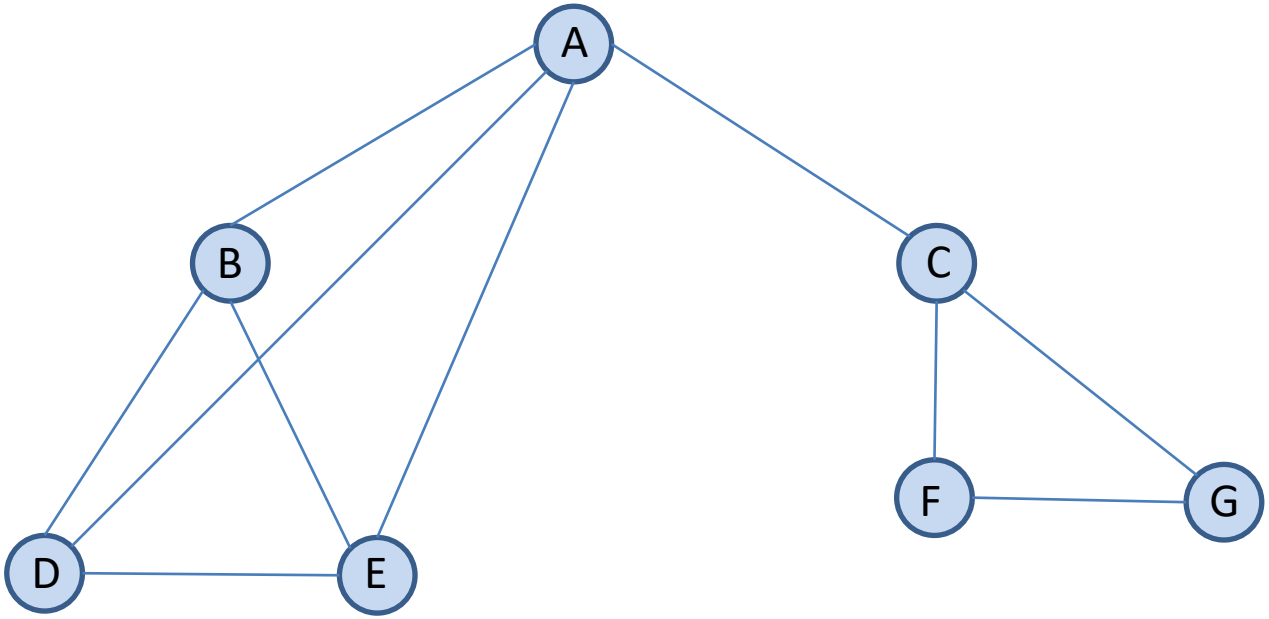
Implementing BFS in C

- Let us now show how to implement breadth-first search in C.
- The algorithm `BreadthFirst` makes use of a queue which can be implemented using any of the implementations we presented in earlier lectures.

Implementing BFS in C (cont'd)

```
/* BreadthFirst: breadth-first traversal of a graph
   Pre: The graph G has been created
   Post: The function visit has been performed at each vertex of G, where the vertices
   are chosen in breadth-first order.
   Uses: Queue functions */

void BreadthFirst(Graph G, void (*Visit)(Vertex))
{
  Queue Q;
  Boolean visited[MAXVERTEX];
  Vertex v, u, w;
  Edge *curedge;

  for (v=0; v < G.n; v++)
    visited[v]=FALSE;

  InitializeQueue(&Q);

  for (u=0; u < G.n; u++)
    if (!visited[u]){
      Insert(u, &Q);
      do {
         Remove(&Q, &v);;
         if (!visited[v]){
           visited[v]=TRUE;
           Visit(v);
         }

         curedge=G.firstedge[v];  /* curedge is a pointer to the first edge (v,_) of V */
         while (curedge){
            w=curedge->endpoint;  /* w is a successor of v and (v,w) is the current edge */
            if (!visited[w]) Insert(w, &Q);
            curedge=curedge->nextedge;  /*curedge is a pointer to the next edge (v,_) of V */
         }
      } while (!Empty(&Q));
    }
}
```

Example

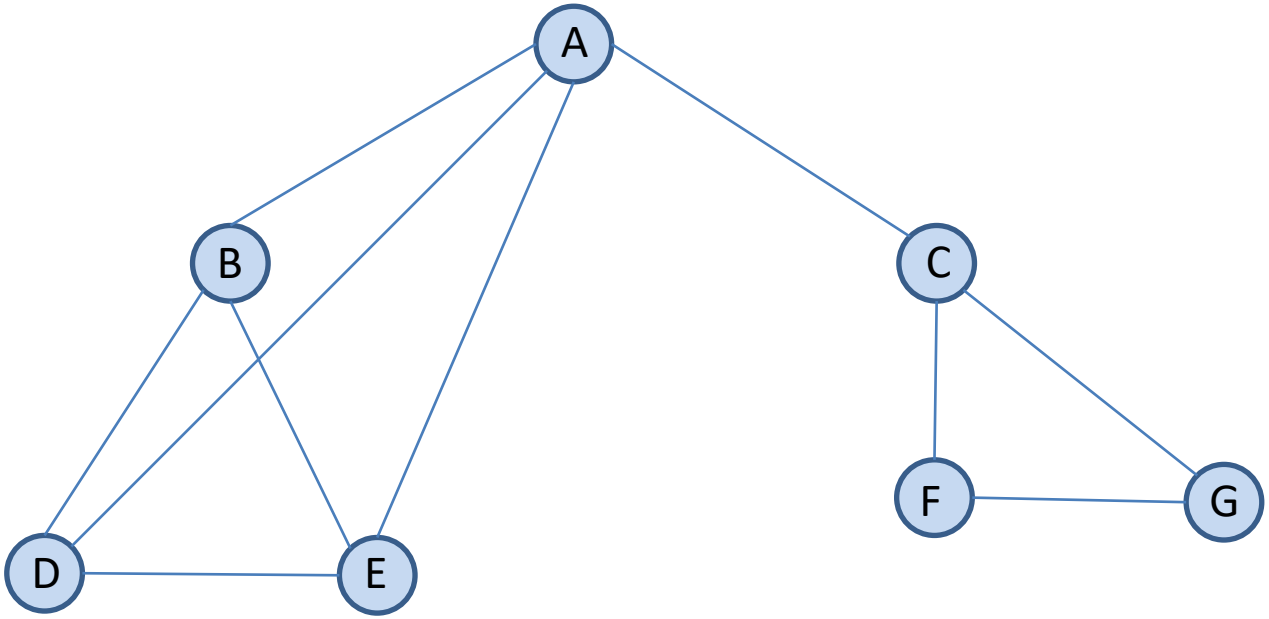


- If we search the above graph G using the function `BreadthFirst` starting from vertex A, what is the order the various vertices will be visited?

Representation of G with Adjacency Lists

Vertex ID	Adjacency list
A	B C D E
B	A D E
C	A F G
D	A B E
E	A B D
F	C G
G	F C

Example (cont'd)



- The vertices will be visited in the following order: A, B, C, D, E, F, G

Example (cont'd)

- Let us study how the algorithm will run.
- At first, A will be inserted in the queue. Then, A will be removed from the queue and marked as visited. Then, its adjacency list will be explored, and vertices B, C, D and E will be found and added to the queue.
- Then, B will be removed from the queue and marked as visited. Then, its adjacency list will be explored, and vertices D and E will be added to the queue (A will not be added since it has been marked as visited).
- But wait! Why are D and E added to the queue although they are there already?

Algorithm `BreadthFirst` Revisited

- What we have just demonstrated is **an inefficiency of `BreadthFirst`.**
- We can fix this inefficiency by checking whether a vertex is already in the queue, so that we do not add it for a second time.
- Another way to fix the problem is the following: instead of marking a vertex as being visited when we take it **off** the queue, we do so when we put it **on** the queue.

Complexity of BFS

- Let us now consider the previous algorithm `BreadthFirst` but we remove the `for` statement that makes sure that all connected components of the graph are explored (so we do a BFS from a given start vertex).
- BFS, implemented as above (with adjacency lists and not allowing a vertex appearing more than once in the queue), has the same complexity as DFS i.e., $\boldsymbol{O(n+e)}$.

DFS of an Undirected Graph

- During a depth-first traversal of an undirected graph, certain edges, when traversed, lead to unvisited vertices. The edges leading to new vertices are called **tree edges (ακμές δένδρου)** or **discovery edges (ακμές ανακάλυψης).**

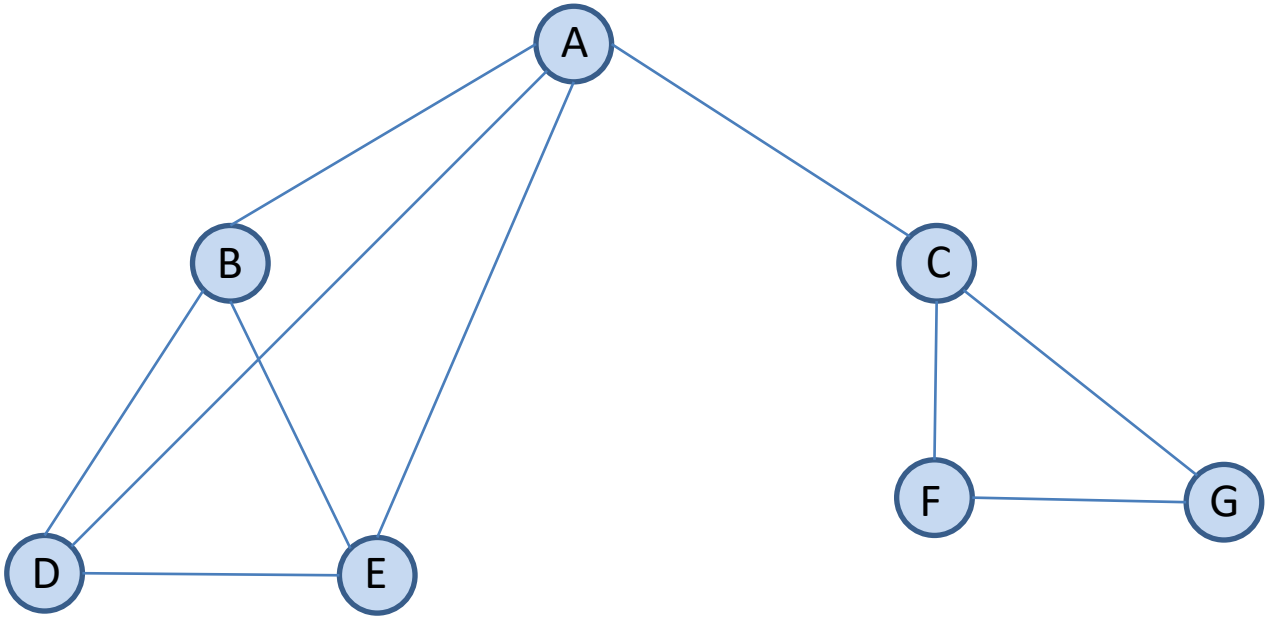
- Tree edges form a **DFS spanning forest (δάσος επικάλυψης για την πρώτα κατά βάθος αναζήτηση)** for the given graph. This forest has one tree for each connected component of the graph and one tree node for each graph vertex.

- There are also edges that do not belong to the spanning forest. These are called **back edges (ακμές οπισθοχώρησης)** and go from a vertex to another vertex we have already visited (i.e., one of its ancestors in the spanning forest).

- Tree edges are those edges (v, w) such that `Traverse` with parameter v directly calls `Traverse` with parameter w or vice versa.

- Back edges are those edges (v, w) such that `Traverse` with parameter v inspects vertex w but does not call `Traverse` because w has already been visited.

Example

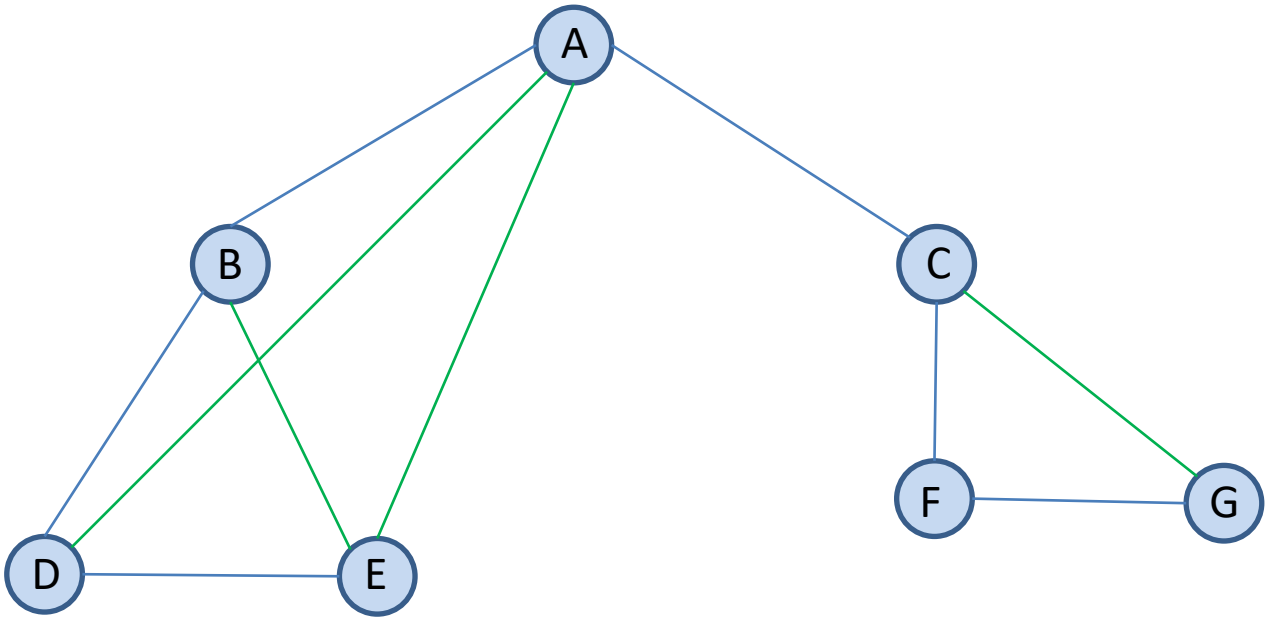


- Let us search the above graph G using the function `DepthFirst` presented earlier starting from vertex A.

Representation of G with Adjacency Lists

Vertex ID	Adjacency list
A	B C D E
B	A D E
C	A F G
D	A B E
E	A B D
F	C G
G	F C

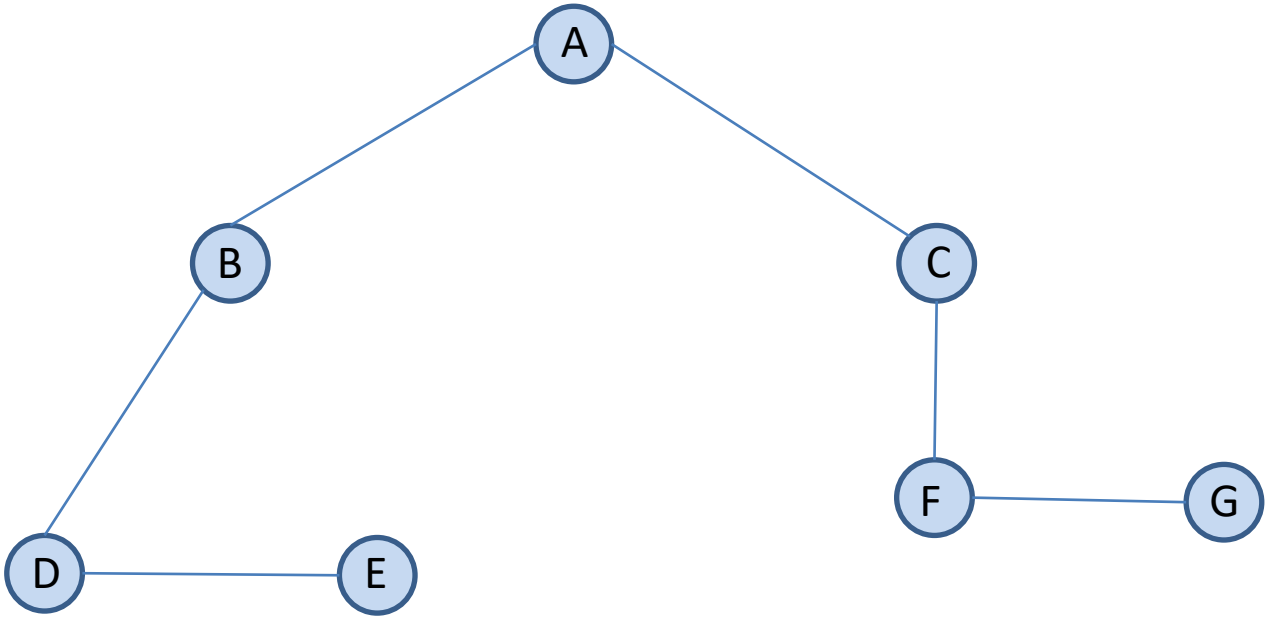
Example (cont'd)



Tree edges

Back edges

Example (cont'd)



- The above is the depth-first spanning forest for the graph G we saw previously. The forest consists of one tree only.

Important

- The edge types are properties of the **dynamics of the search, rather than only the graph**.
- Different depth-first spanning forests of the same graph can differ remarkably in character.
- Can you give examples for the above facts?

Question

- How can we modify function `Traverse` so that it outputs the edges of the depth-first spanning tree (tree edges) and the back edges?

Proposition

- Let G be an undirected graph with n vertices and e edges which is represented with adjacency lists. A DFS traversal of G can be executed in $O(n + e)$ time and can be used to solve the following problems in $O(n + e)$ time:
  1. Checking if G is connected
  2. Computing a spanning tree of G if G is connected.
  3. Computing a path between two given vertices of G, if such a path exists.
  4. Computing a cycle of G or discovering that G does not have cycles.

- The proof of the proposition is left as an exercise.

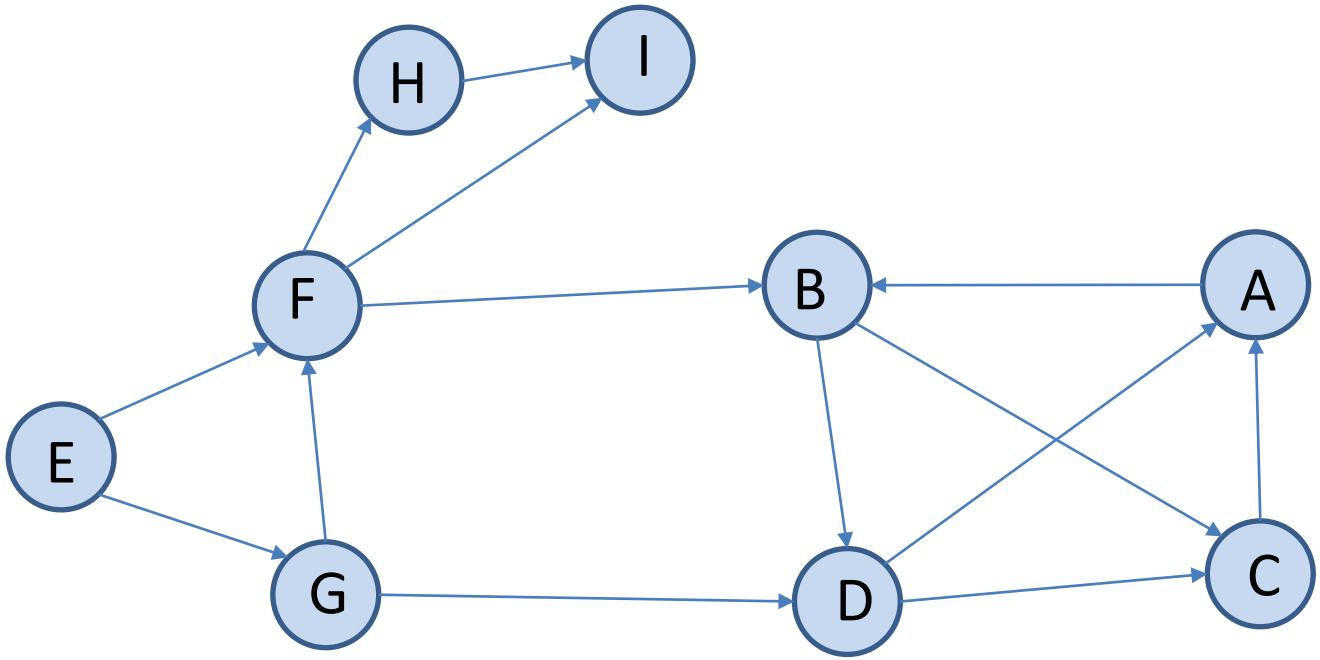
DFS Traversal of a Directed Graph

- During a depth-first traversal of a directed graph, certain edges, when traversed, lead to unvisited vertices. The edges leading to new vertices are called **tree edges (ακμές δένδρου)** and they form a **DFS forest (δάσος πρώτα κατά βάθος αναζήτησης)** for the given digraph.

- This forest has one **DFS tree** for each strong connected component of the graph and one tree node for each graph vertex.

- There are also edges that do not belong to the DFS forest and can be classified as follows:
  - **Back edges (ακμές οπισθοχώρησης)** that go from a vertex to one of its ancestors in a DFS tree.
  - **Forward edges (ακμές προώθησης)** that go from a vertex to one of its descendants in a DFS tree.
  - **Cross edges (εγκάρσιες ακμές).** These are all the remaining edges. They can go between vertices in the same DFS tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different DFS trees.

Example

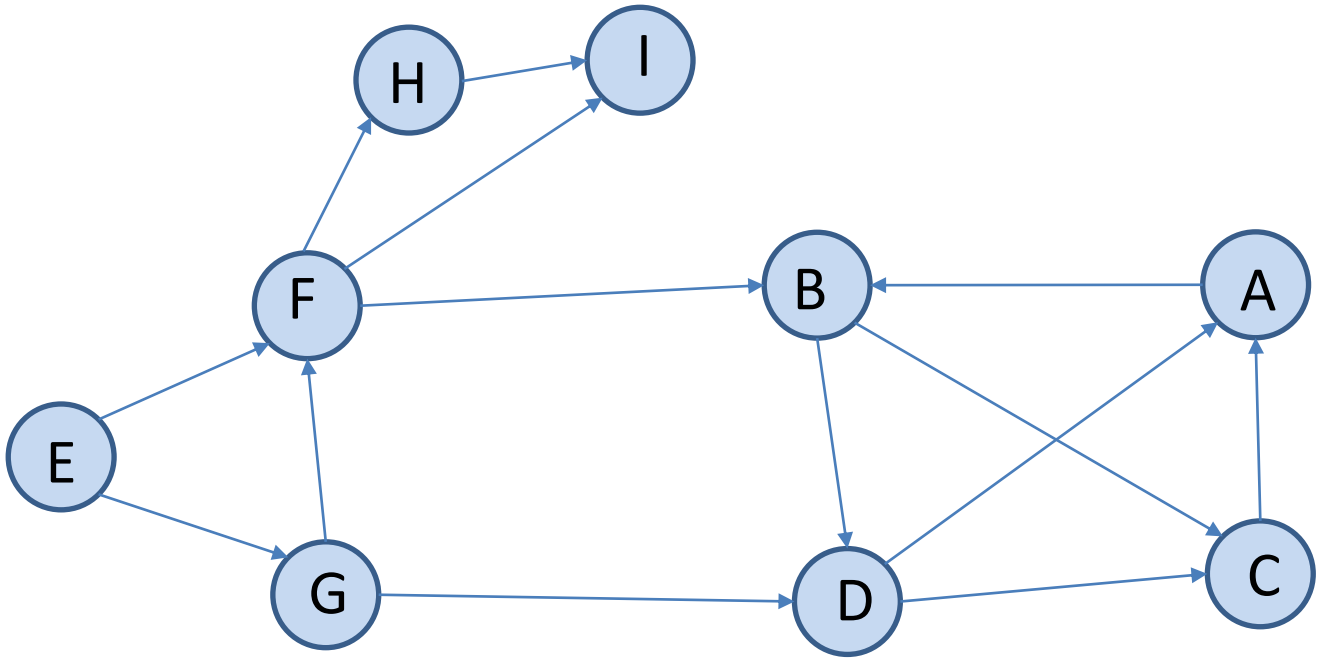


- Let us search the above graph G using the function `DepthFirst` presented earlier, starting from vertex A (more precisely: we are using the exhaustive graph search version of `DepthFirst` since G has more than one strong components).

Representation with Adjacency Lists

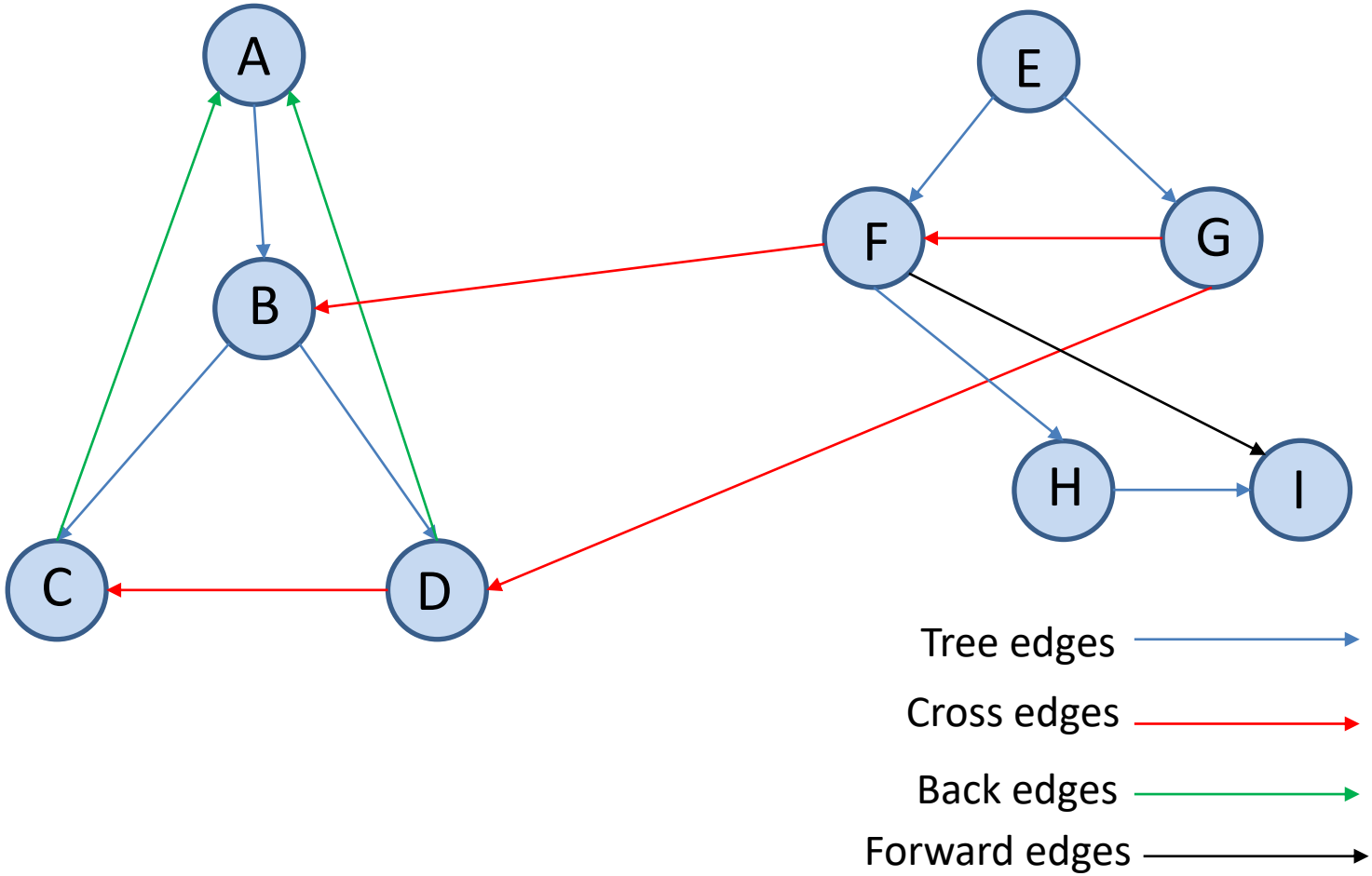
Vertex ID	Adjacency list
A	B C
B	C D
C	A
D	A C
E	F G
F	H I
G	F
H	I
I	

Example (cont'd)

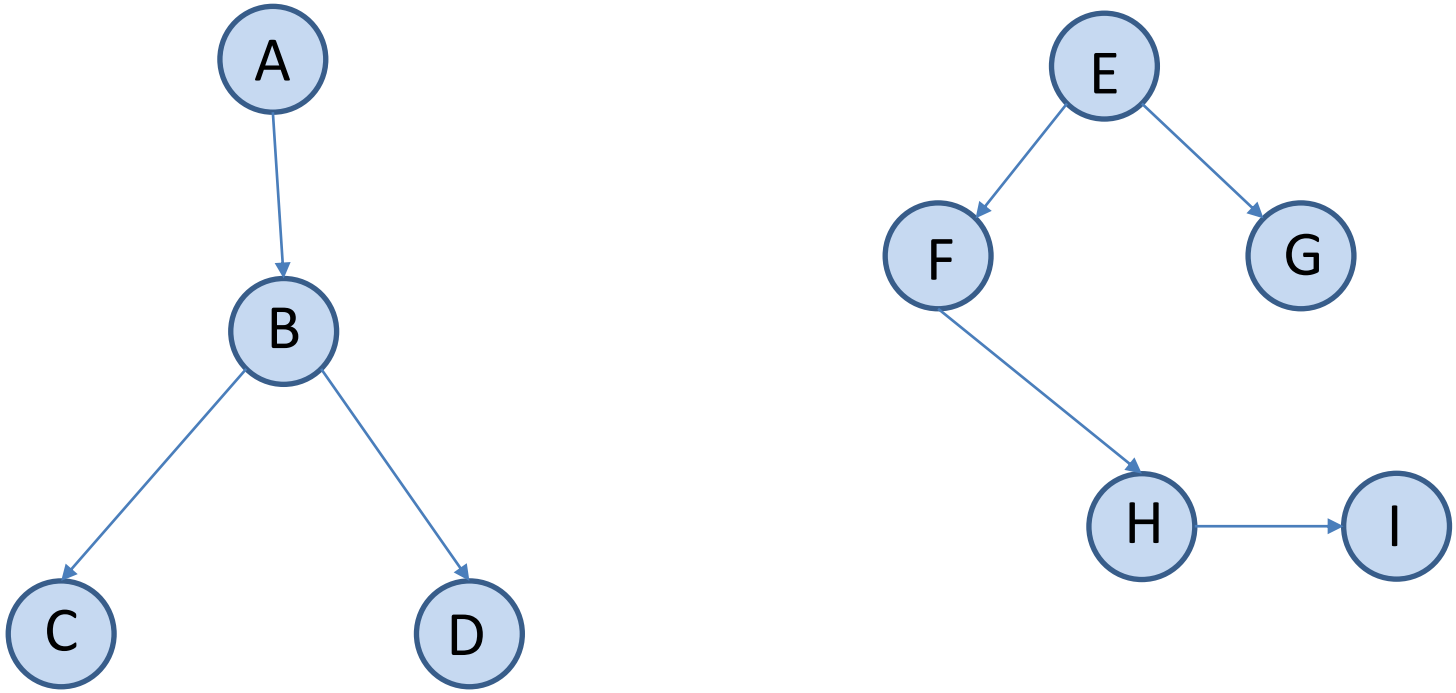


- The vertices will be visited in the order A, B, C, D, E, F, H, I, G.

Example (cont'd)



Example (cont'd)



- The above is the **DFS forest** of graph G for the depth-first traversal we saw previously.

Important

- As we also saw for undirected graphs, the edge types are properties of the **dynamics of the search, rather than only the graph**.
- Different DFS forests of the same graph can differ remarkably in character.
- Even the number of trees in the DFS forest depends upon the start vertex.
- Can you give examples for the above facts?

Classification of Edges

- How do we distinguish among the four types of edges?
- **Tree edges** are easy to find since they lead to an unvisited vertex during DFS.
- The other three types of edges can be distinguished by keeping track of the preorder and postorder numbering of nodes and using an interesting proposition that we present below.

Preorder Numbering of Vertices

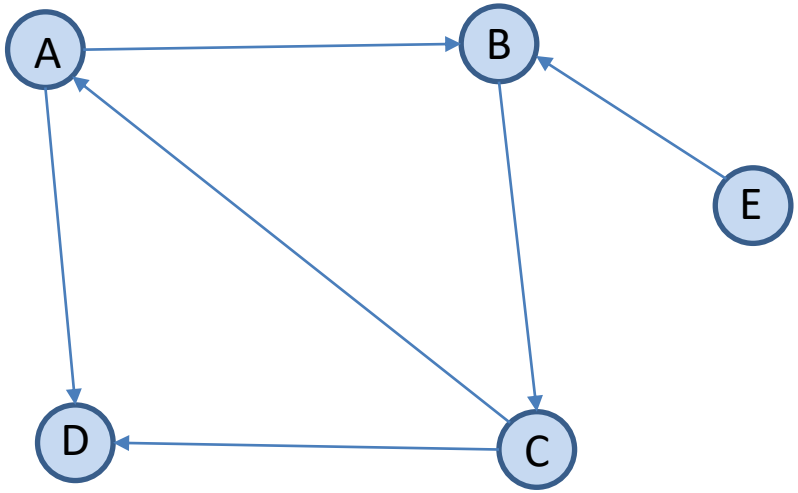
- We can **number the vertices** of a digraph in the order in which we first mark them as visited during a DFS.
- For this, we can use a counter `count` that is initially set to zero, a vertex-indexed array `pre` with all its elements set to `-1` initially, and add the code

```
pre[v]=count++;
```

in the function `Traverse`, immediately after the statement marking a vertex as visited. **Exercise**: modify `Traverse` as needed.

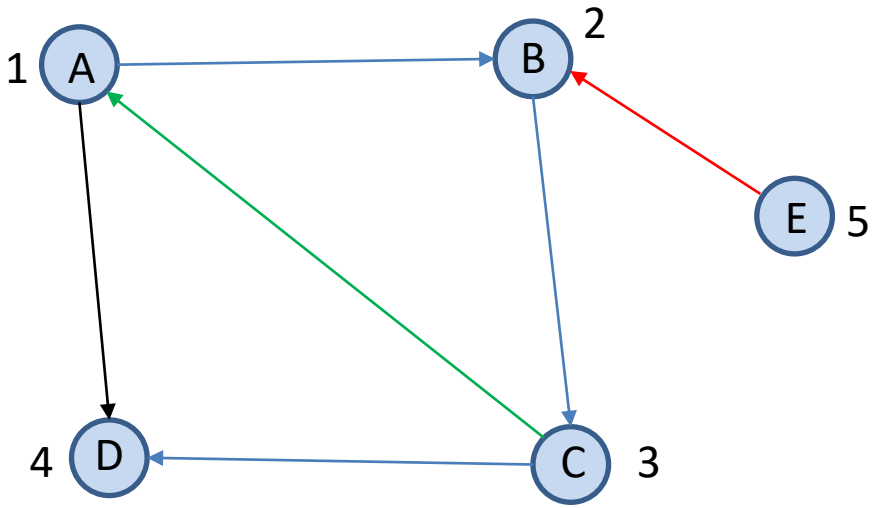
- We will call this the **preorder numbering (προδιατεταγμένη αρίθμηση)** of a digraph.

Example Directed Graph



- Consider performing a DFS starting from vertex A.

Preorder Numbering



Tree edges

Forward edges

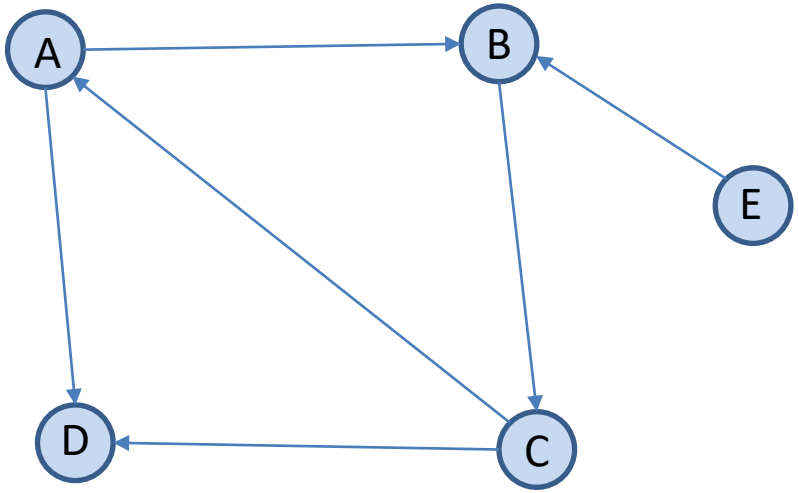
Back edges

Cross edges

Postorder Numbering of Vertices

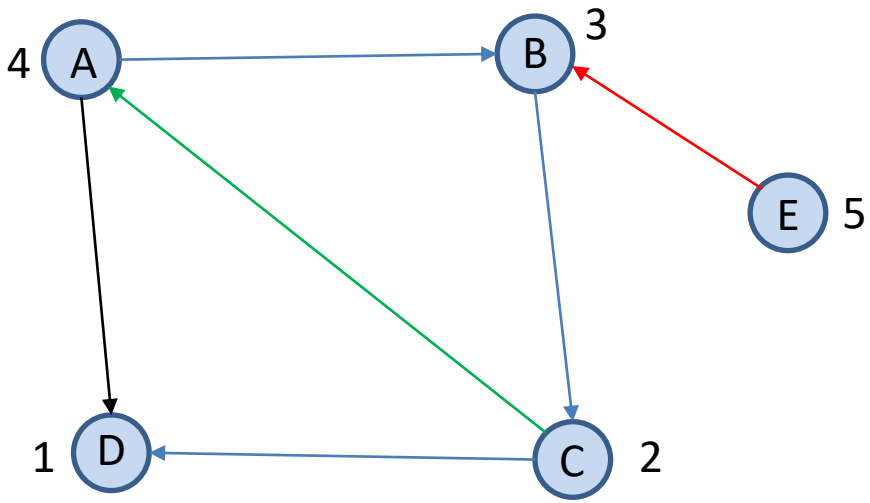
- We can also have a **postorder numbering (μεταδιατεταγμένη αρίθμηση)** of the vertices of a digraph.
- This is the order that we **finish** processing them (just before returning from the recursive function `Traverse`).
- Postorder numbering can be implemented in a similar way as the preorder one, using a vertex-indexed array `post`, a counter and introducing appropriate code in the function `Traverse`.

Example Directed Graph



- Consider performing a DFS starting from vertex A.

Postorder Numbering



Tree edges

Forward edges

Back edges

Cross edges

Proposition

- In a DFS forest corresponding to a digraph, an edge to a visited vertex is a **back edge** if it leads to a vertex with a higher postorder number; otherwise, it is a **cross edge** if it leads to a vertex with a lower preorder number and a **forward edge** if it leads to a vertex with a higher preorder number.
- Proof?

Proof

- These facts follow from the definitions of arrays `pre` and `post` and how they are updated by function `Traverse`.
- A vertex's ancestors in a DFS tree have lower preorder numbers and higher postorder numbers.
- A vertex's descendants in a DFS tree have higher preorder numbers and lower postorder numbers.
- Both numbers are lower in previously visited vertices in other DFS trees and both numbers are higher in yet-to-be-visited vertices in other DFS trees.
- An edge (v, u) so that v is not an ancestor or a descendant of u in the same DFS tree is such that `pre[u]<pre[v]`.

Proposition

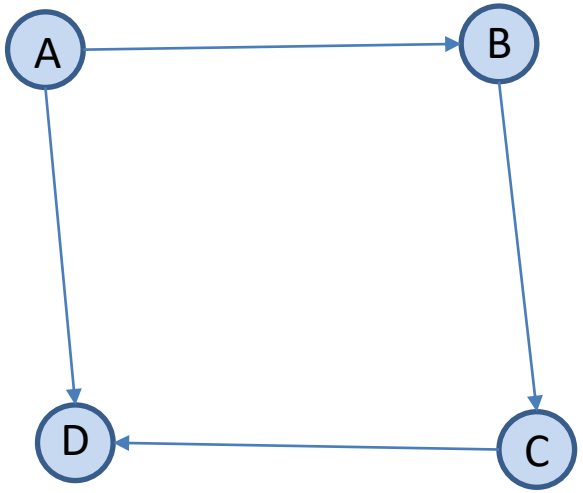
- Let G be a directed graph. A DFS traversal of G starting from a vertex s visits all vertices of G that are reachable from s. In addition, the DFS tree contains directed paths from s to each vertex that is reachable from s.

- The proof of the proposition is left as an exercise.

Transitive Closure

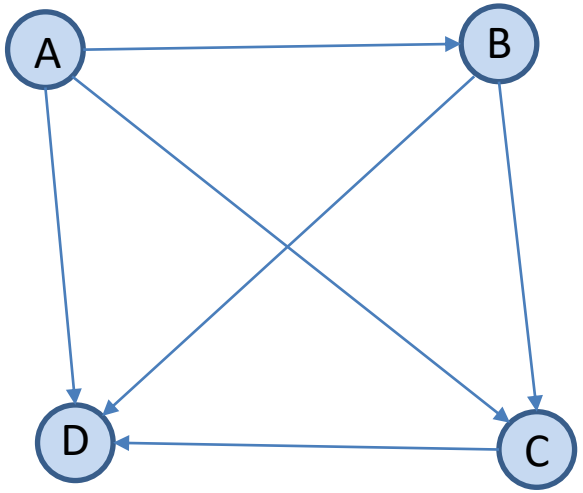
- The **transitive closure (μεταβατική κλειστότητα)** of a directed graph G is a directed graph G^* such that the vertices of $G^*$are the same as the vertices of G, and G^* has an edge (u, v) whenever G has a directed path from u to v.

Example Directed Graph G



- What is the transitive closure of G?

The Transitive Closure G^*



Proposition

- Let G be a directed graph with n vertices and e edges which is represented with adjacency lists. The following problems can be solved by an algorithm which traverses G n times using DFS, runs in time $O(n(n+e))$, and uses $O(n)$ additional storage.
  1. For every vertex v of G, computing the subgraph which is reachable from v.
  2. Checking whether G is strongly connected.
  3. Computing the transitive closure G^* of G.

- The proof of the proposition is left as an exercise.

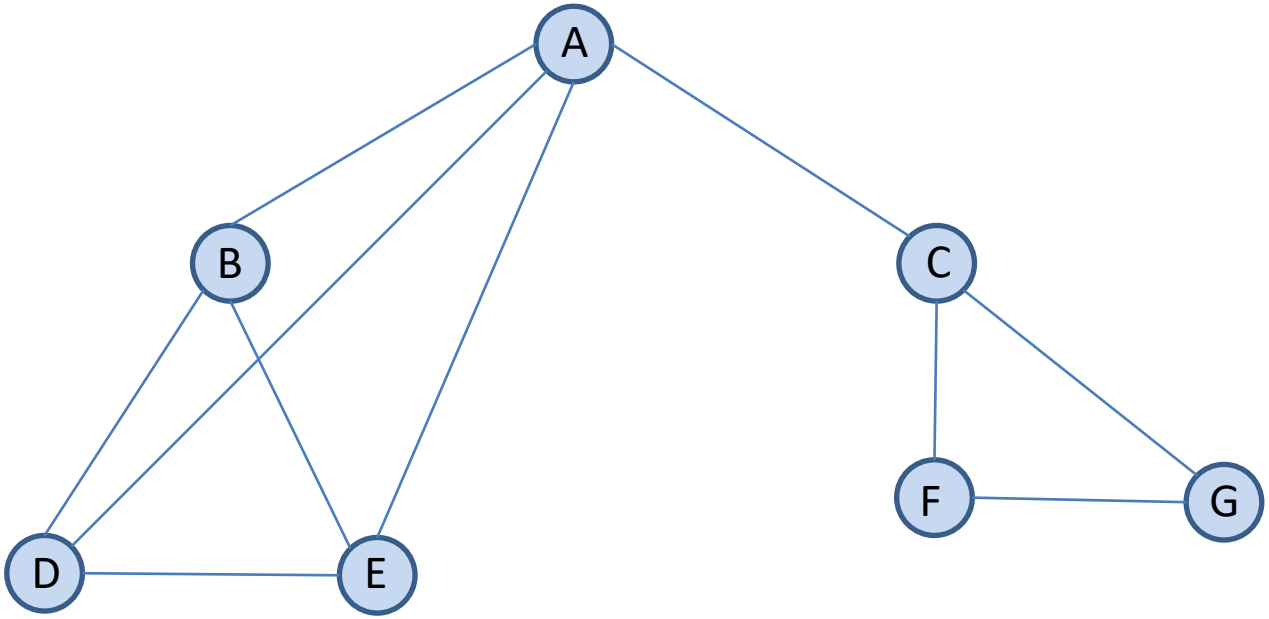
Checking for Strong Connectivity

- The following is a simple algorithm for checking whether a given directed graph G is strongly connected.
- We start doing a DFS of the directed graph G starting from an arbitrary vertex s.
- If there is any vertex of G which is not reachable from s, then G is not strongly connected.
- If the DFS visits all vertices of G, we compute the **reverse** of G, a new directed graph G_r by reversing the direction of each edge in G.
- Then we do a second DFS in G_r starting from s.
- If this second DFS visits all vertices of G, then the graph is strongly connected, because every vertex reachable from s can also reach s using DFS.

BFS of an Undirected Graph

- We can build a spanning forest when we perform a breadth-first search as well. We call this the **BFS spanning forest** of the graph.
- This forest has one **BFS tree** for each connected component of the graph and one tree node for each graph vertex.
- The breadth-first spanning forest is built by tree edges. An edge (v, w) is a **tree edge** if vertex w is first visited from vertex v in the inner while loop of function `BreadthFirst`.
- Every edge that is not a tree edge is a **cross edge**, that is, it connects two vertices neither of which is an ancestor of the other in the BFS spanning forest.

Example

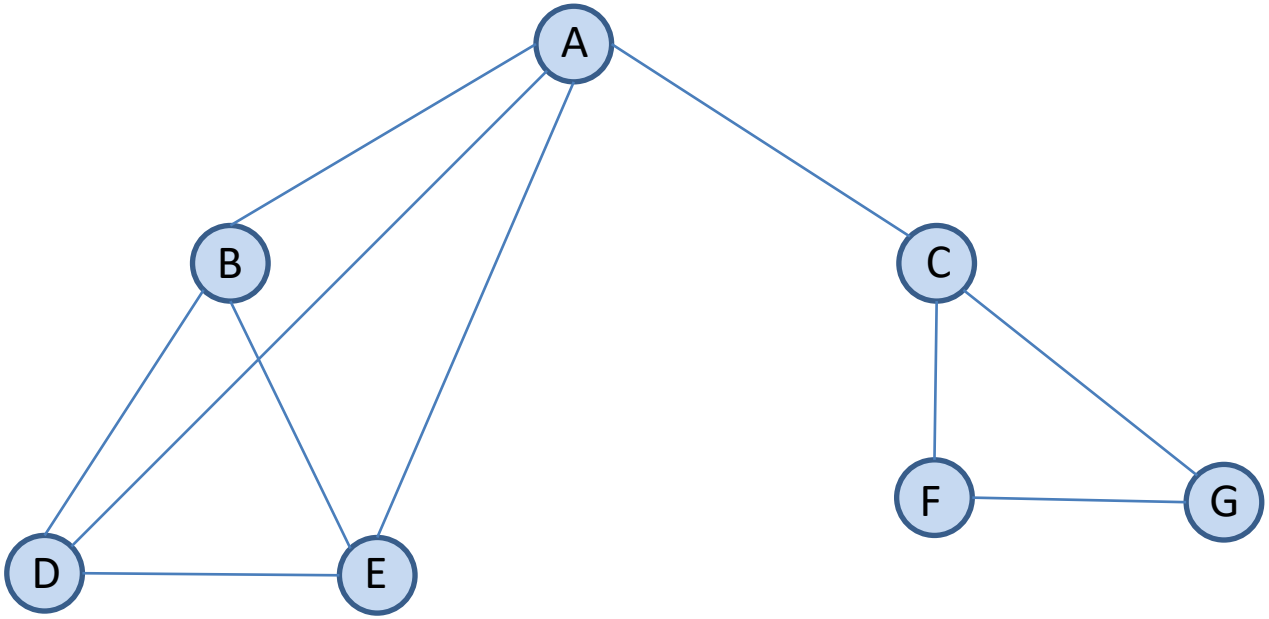


Let us execute the algorithm `Breadth-First` presented earlier on the above graph G starting from vertex A. **In what order will the algorithm visit the vertices of G and why?**

Representation of G with Adjacency Lists

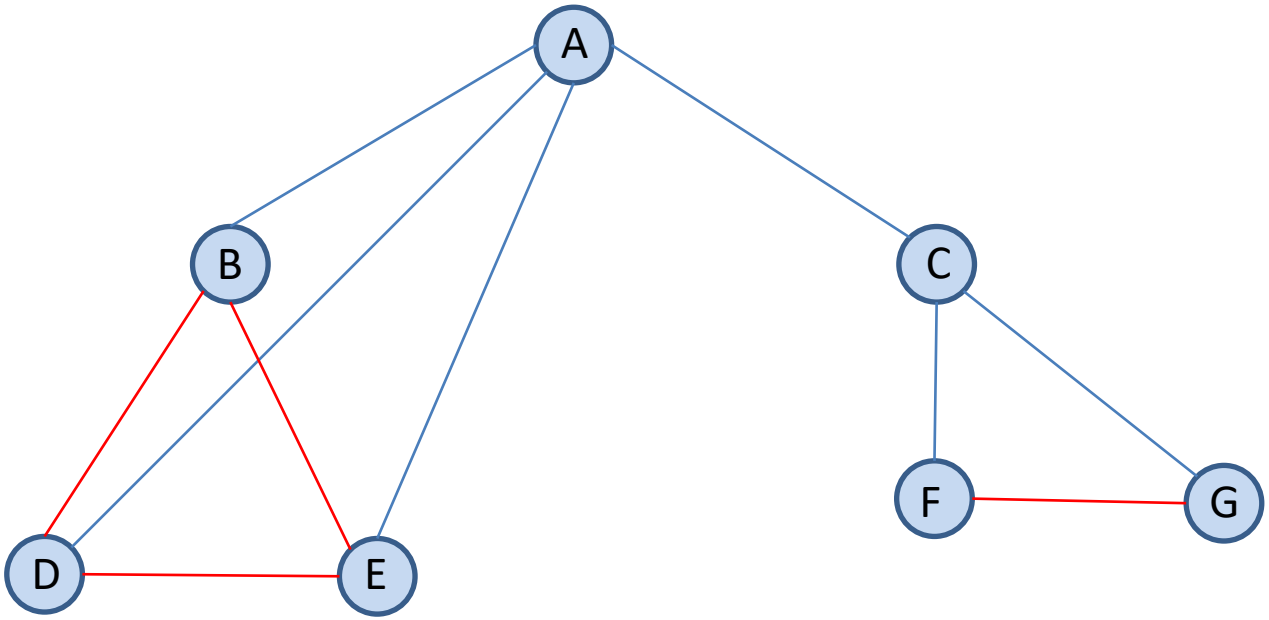
Vertex ID	Adjacency list
A	B C D E
B	A D E
C	A F G
D	A B E
E	A B D
F	C G
G	F C

Example (cont'd)



The vertices will be visited in the order A, B, C, D, E, F, G.

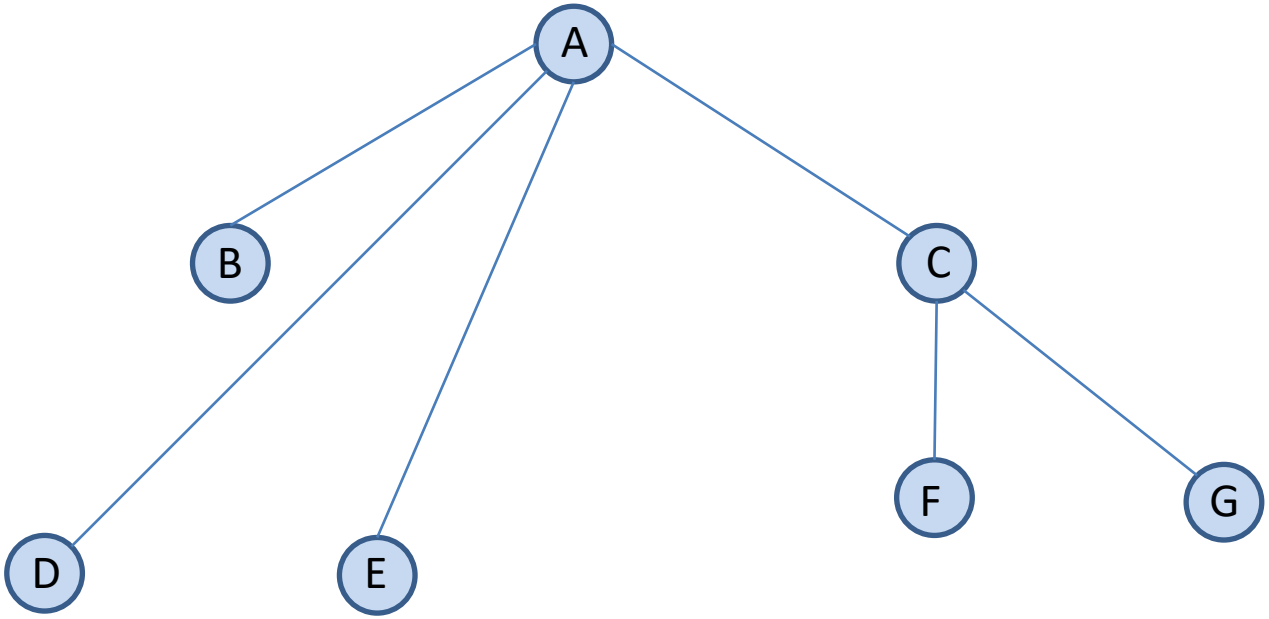
Example (cont'd)



Tree edges

Cross edges

Example (cont'd)



- The above is the BFS spanning forest for the breadth-first traversal of the graph G given previously.

Important

- The edge types are properties of the **dynamics of the BFS, rather than only the graph**.
- Different BFS spanning forests of the same graph can differ remarkably in character.
- Can you give examples for the above facts?

Proposition

- During BFS, vertices enter and leave the queue in order of their distance from the start vertex (where the **distance** of vertex v to vertex u is the length of a shortest path from v to u).
- Proof?

Proof

- A stronger property holds. The queue always consists of zero or more vertices of distance k from the start, followed by zero or more vertices of distance $k + 1$ from the start, for some integer k. This stronger property is easy to prove by induction.

Proposition

- Let G be an undirected graph with n vertices and e edges. A BFS traversal of G requires $O(n + e)$ time. There are also algorithms running in $O(n + e)$ time that are based on BFS and solve the following problems:
  1. Checking whether G is connected.
  2. Computing a spanning tree of G, if G is connected.
  3. Computing the connected components of G.
  4. Given a start vertex s of G, computing, for every other vertex v of G, a path with minimum number of edges between s and v, or reporting that there is no such path.
  5. Computing a cycle of G or reporting that G has no cycles.

Corresponding Proposition for DFS Revisited

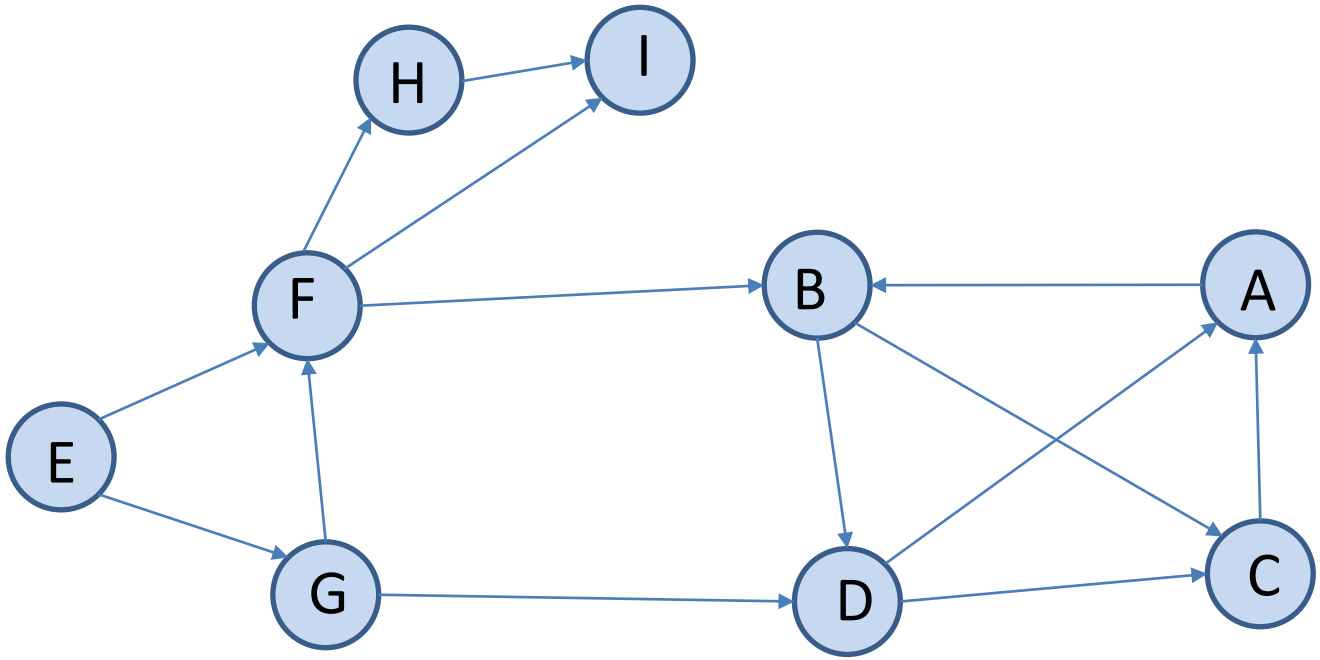
- Let G be an undirected graph with n vertices and e edges which is represented with adjacency lists. A DFS traversal of G can be executed in $O(n + e)$ time and can be used to solve the following problems in $O(n + e)$ time:
  1. Checking if G is connected
  2. Computing a spanning tree of G if G is connected.
  3. Computing a path between two given vertices of G, if such a path exists.
  4. Computing a cycle of G or discovering that G does not have cycles.

- Compare the above proposition with the one on the previous slide. **What are the interesting properties of graphs that can be checked by both DFS and BFS and what are the properties that only one of them can check?**

BFS of Directed Graphs

- BFS can also work on directed graphs.
- The algorithm visits vertices level by level and partitions the edges into two sets: **tree edges** and **non-tree edges**.
- Tree edges define a **BFS forest.**
- Non-tree edges are of two kinds: **back edges** and **cross edges.**
- Back edges connect a vertex to one of its ancestors in the BFS forest. Cross edges connect a vertex to another vertex that is neither its ancestor nor its descendant in the forest.
- There are no forward edges as in the DFS case (why?).

Example

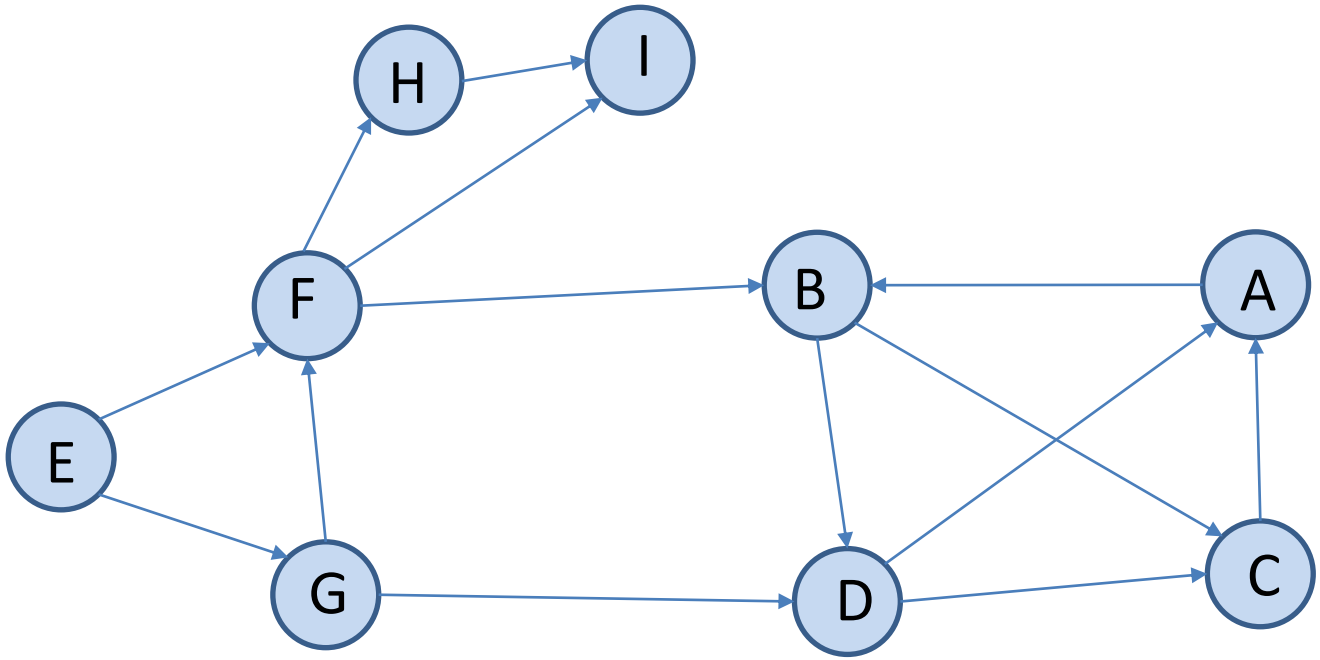


- Let us search this graph G using the function `BreadthFirst` presented earlier starting from vertex A.

Representation with Adjacency Lists

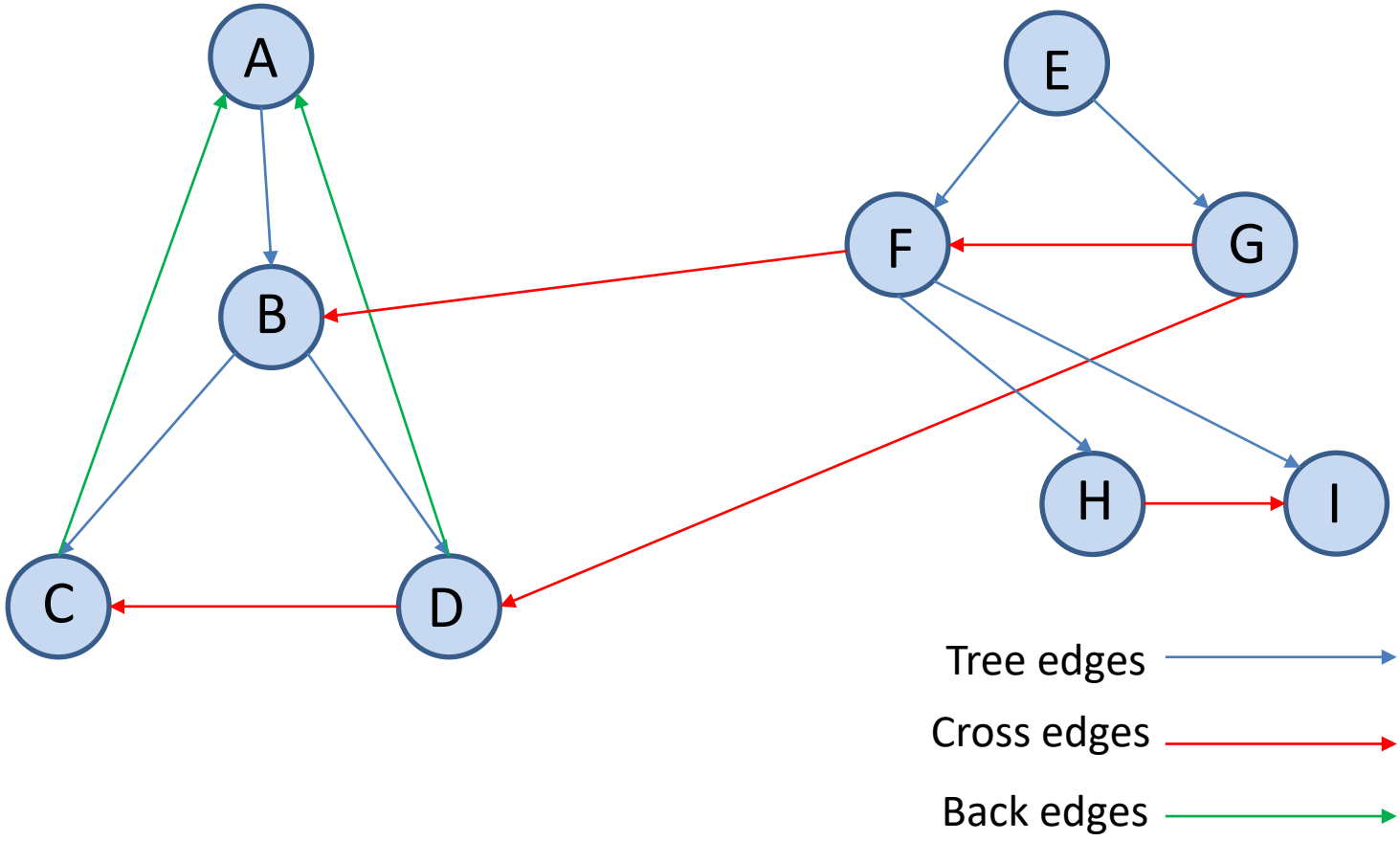
Vertex ID	Adjacency list
A	B C
B	C D
C	A
D	A C
E	F G
F	H I
G	F
H	I
I	

Example (cont'd)

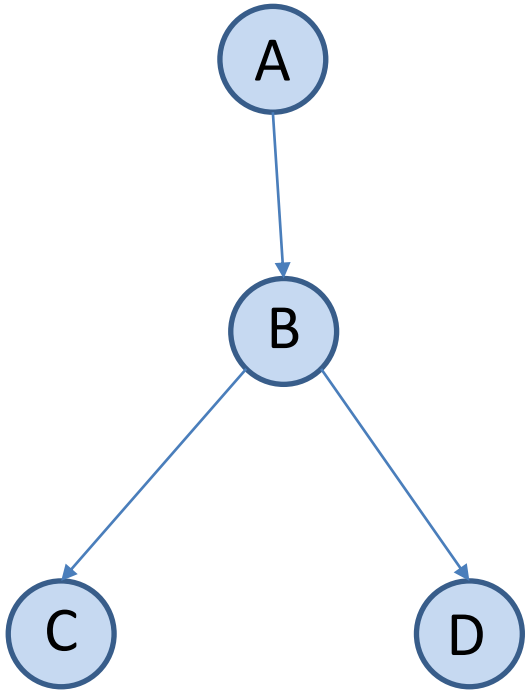


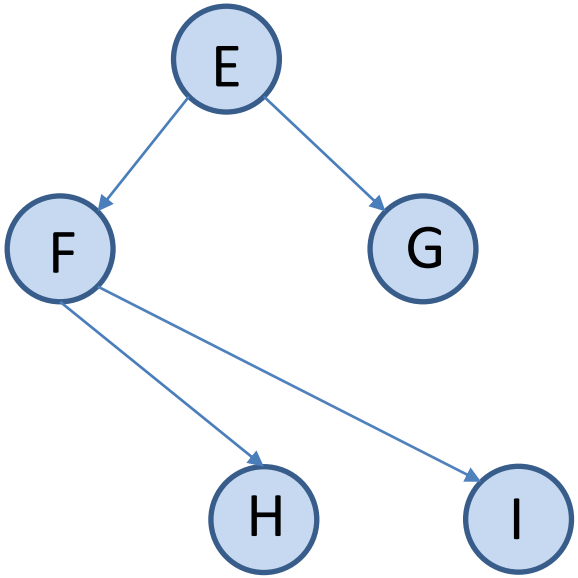
- The vertices will be visited in the order A, B, C, D, E, F, G, H, I.

Example (cont'd)



Example (cont'd)





- The above is the BFS forest of graph G for the BFS traversal we saw previously.

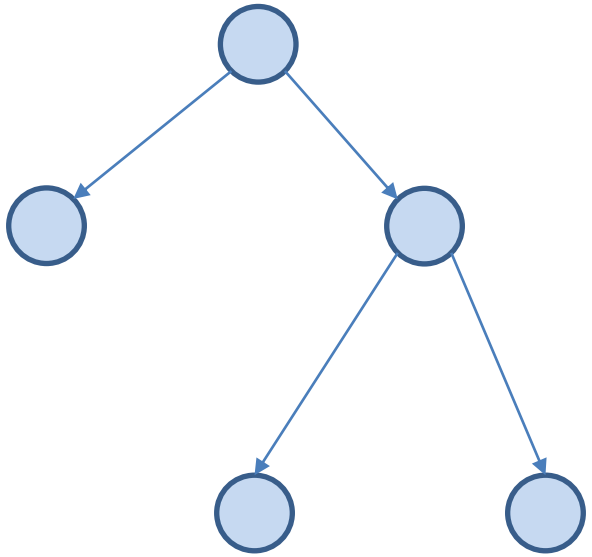
Question

- How do we modify the code of algorithm `Breadth-First` so that we output the edges of various kinds for undirected or directed graphs?

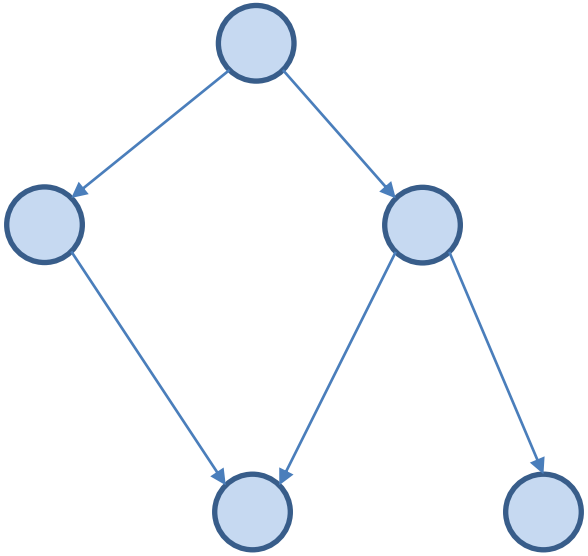
Directed Acyclic Graphs

- Let G be a directed graph with no cycles. Such a graph is called **acyclic**. We abbreviate the term directed acyclic graph to **dag**.
- Dags are more general than trees but less general than arbitrary directed graphs.

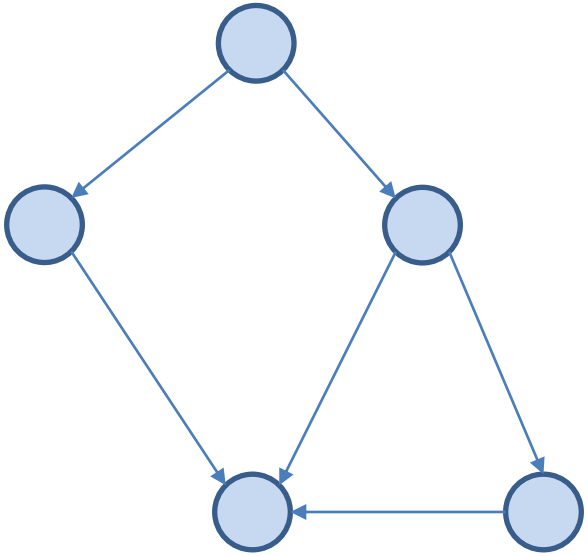
Example Tree



Example Dag



Another Example Dag



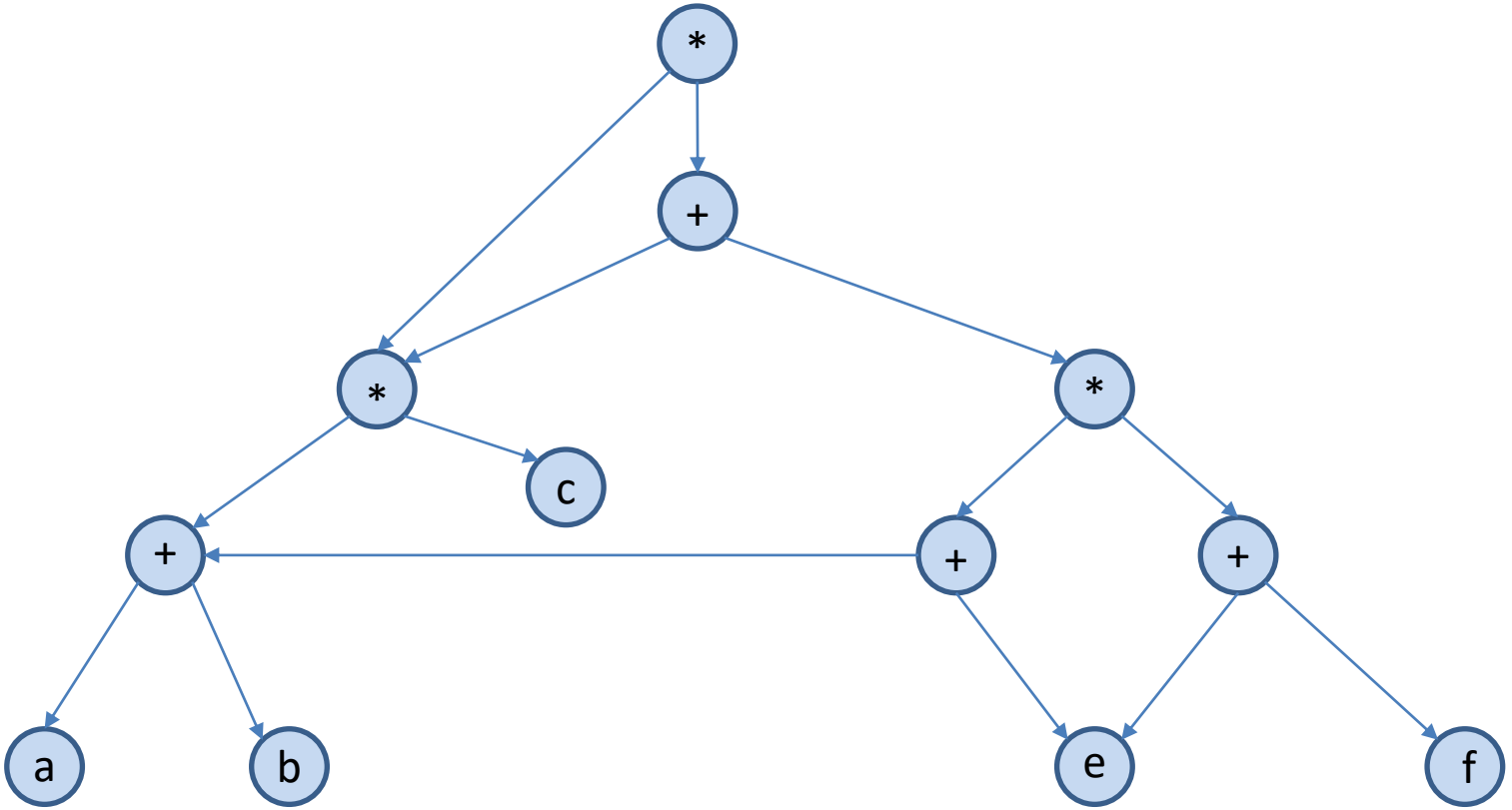
Applications of Dags

- Dags are useful in compilers for representing the syntactic structure of **arithmetic expressions** with common subexpressions.

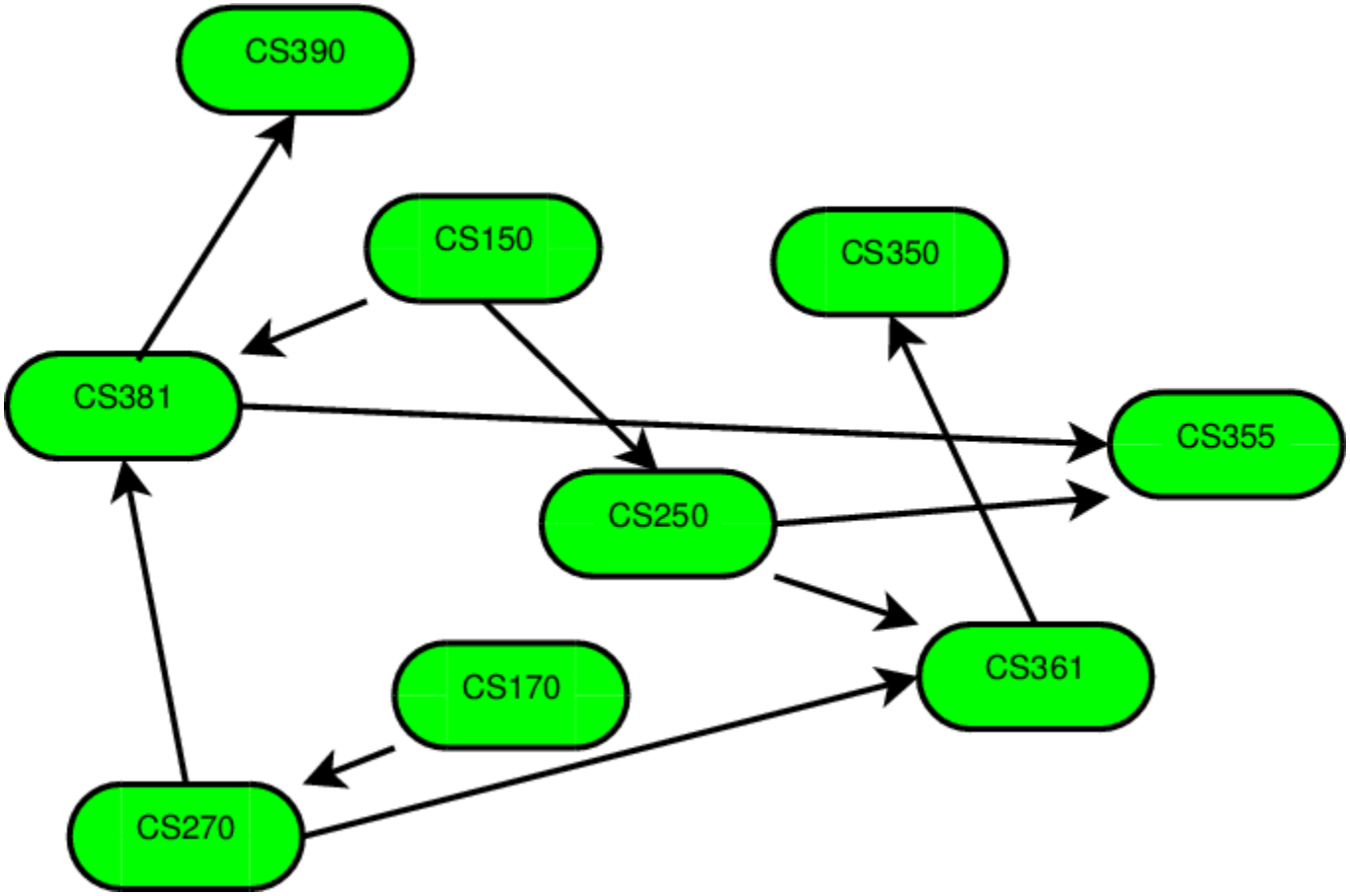
- **Example**: Consider the following arithmetic expression

$$\Big((a+b)*c+\big((a+b)+e\big)*(e+f)\Big) * ((a+b)*c)$$

The Dag for the Example



Prerequisites in a Program of Study



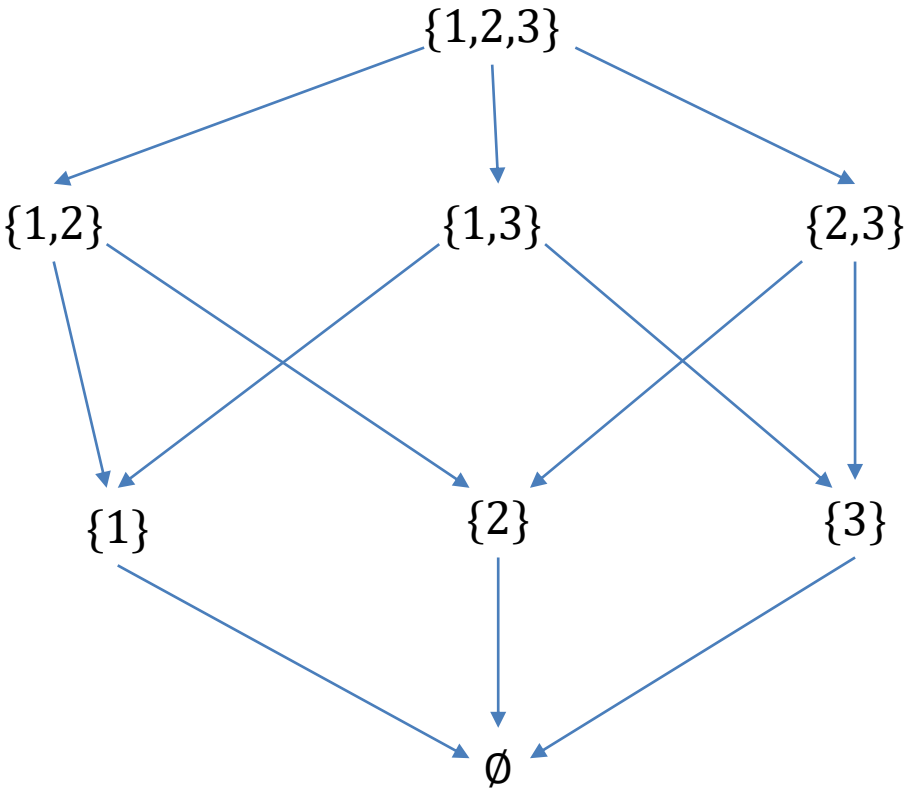
Applications of Dags (cont'd)

- Dags are also useful for representing **partial orders.**
- A **partial order** R on a set S is a binary relation such that
  - For all a in S, $a\ R\ a$ is false (irreflexivity)
  - For all a, b, c in S, if $a\ R\ b$ and $b\ R\ c$ then $a\ R\ c$ (transitivity)
- Two natural examples of partial orders are the "less than" relation ($<$) on integers, and the relation of proper containment ($\subset$) on sets.

Example

- Let $S = \{1, 2, 3\}$ and let $P(S)$ be the power set of S, that is, the set of all subsets of S. The relation $\subset$ is a partial order on $P(S)$.

The Dag of the Example



Class Hierarchies

Test for Acyclicity

- Suppose we are given a digraph G and we wish to **determine whether G is acyclic**.
- DFS can be used to answer this question.
- If a **back edge** is encountered during a DFS then clearly the graph has a cycle.
- Conversely, if the graph has a cycle then a back edge will be encountered in any DFS of the graph. Proof?

Proof

- Suppose G is cyclic. If we do a DFS of G, there will be one vertex v having the lowest preorder number of any vertex on a cycle.
- Consider an edge (u, v) on some cycle containing v. Since u is on the cycle, u must be a descendant of v in the depth-first spanning forest (it cannot be an ancestor; if u was, it would have a higher preorder number than v). Thus, (u, v) cannot be a cross edge.
- Since the preorder number of u is greater than the preorder number of v, (u, v) cannot be a tree edge or a forward edge. Hence, (u, v) is a back edge.

Important

- In undirected graphs, any edge to a previously visited vertex indicates a cycle.
- In directed graphs, this is true only for back edges.

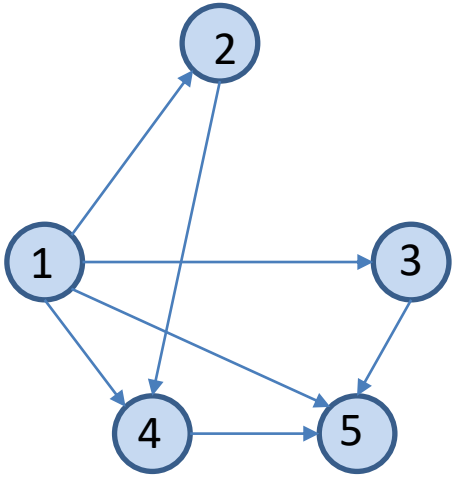
Topological Ordering of a DAG

- A **topological ordering (τοπολογική ταξινόμηση)** of the vertices of a dag G is a sequential list L of the vertices of G (a linear ordering) such that if there is a directed path from vertex A to vertex B in G, then A comes before B in the list L.

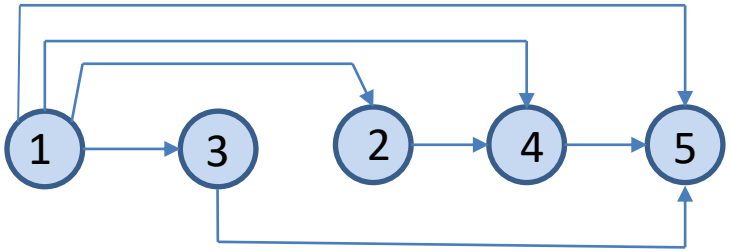
Example

- G might be a graph in which the vertices represent university courses to take and in which an edge is directed from the vertex for course A to the vertex for course B if course A is a **prerequisite** of B.
- Then a topological ordering of the vertices of G gives us a possible way to organize one's studies.

Example

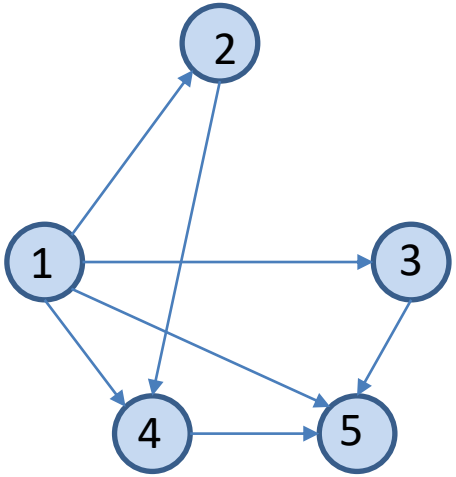


A DAG

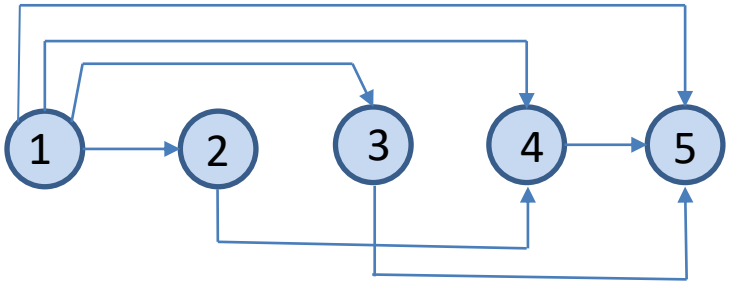


A topological ordering is:
1, 3, 2, 4, 5

Example (cont'd)

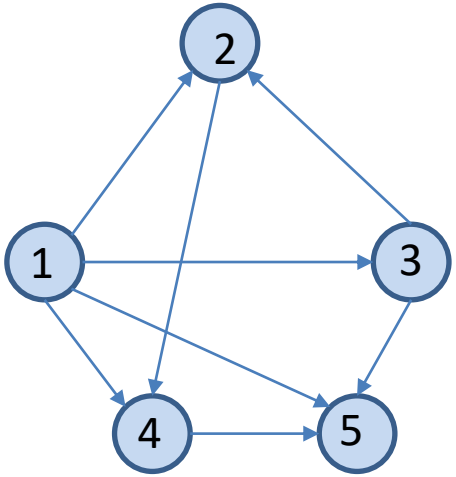


A DAG

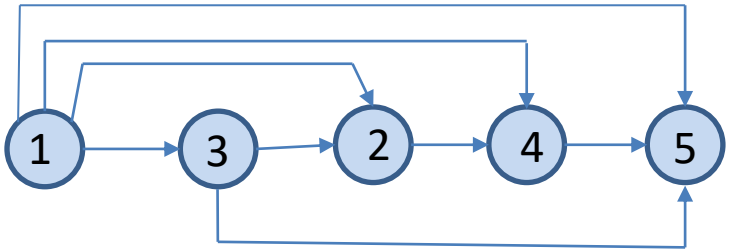


Another topological ordering is: 1, 2, 3, 4, 5

Another Example



A DAG



A topological ordering is:
1, 3, 2, 4, 5

Important

- In general, there can be more than one topological orderings for a given dag.
- **Question**: How many topological orderings do the previous two dags have?

Computing a Topological Ordering

- We will compute **a list of vertices** `L` that contains the vertices of `G` in topological order.
- We will use an array `D` such that `D[v]` gives the number of predecessors `p` of vertex `v` in graph `G` such that `p` is not in `L`.
- We will use a queue `Q` of vertices from where we will take vertices to process (from the front of the queue).
- The vertices of `G` in `Q` will be processed in breadth-first order.
- Initially `Q` will contain all the vertices of `G` with no predecessors.
- When we find a vertex `w` of `G` such that `D[w]==0`, we see that `w` has all its predecessors in list `L`, so we add `w` to the rear of queue `Q` so that it can be processed.

Algorithm for Topological Ordering

```
void BreadthTopSort(Graph G, List *L)
{
   Let G=(V,E) be the input graph.
   Let L be a list of vertices.
   Let Q be a queue of vertices.
   Let D[V] be an array of vertices indexed by vertices
   in V.

    /* Compute the in-degrees D[x] of the vertices x
       in G */
    for (each vertex x in V) D[x]=0;
    for (each vertex x in V){
        for (each successor w in Succ(x)) D[w]++;
    }
```

Algorithm for Topological Ordering (cont'd)

```
/* Initialize the queue Q to contain all
vertices having zero in-degrees */

Initialize(&Q);
for (each vertex x in V){
    if (D[x]==0) Insert(x, &Q);
}
```

Algorithm for Topological Ordering (cont'd)

```
    /* Initialize the list L to be the empty list */
    InitializeList(&L);

    /* Process vertices in the queue Q until the queue becomes
       empty */
    while (!Empty(&Q)){
       Remove(&Q,x);
       AddToList(x,&L);
       for (each successor w in Succ(x)){
          D[w]--;
          if (D[w]==0) Insert(w, &Q);
        }
     }
    /* The list L now contains the vertices of G in
       topological order */
}
```

Implementing Topological Sort in C

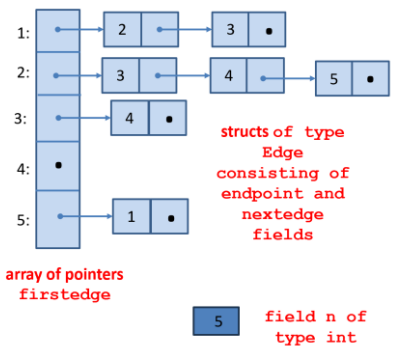
- We first need to define a new type for an array that will be used to store the vertices of a graph in topological order:

```
typedef Vertex Toporder[MAXVERTEX];
```

- We will also use the functions for the ADT queue that we have defined in a previous lecture.

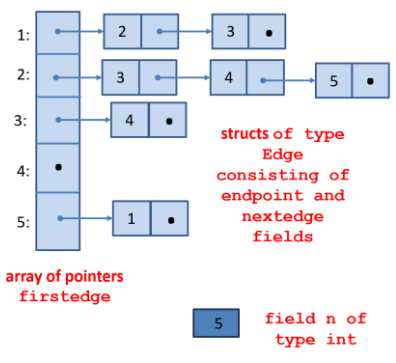
Topological Sort in C (cont'd)

```
/* BreadthTopSort: generates breadth-first topological ordering
   Pre: G is a directed graph with no cycles implemented with a contiguous list of vertices
   and linked adjacency lists.
   Post: The function makes a breadth-first traversal of G and generates the resulting
   topological order in T
   Uses: Queue functions */

void BreadthTopSort(Graph G, Toporder T)
{

  int predecessorcount[MAXVERTEX];     /* number of predecessors of each vertex */
                                       /* (the array D of the previous algorithm) */

  Queue Q;
  Vertex v, succ;
  Edge *curedge;
  int place;

  /* initialize all the predecessor counts to 0  */
  for (v=0; v < G.n; v++)
    predecessorcount[v]=0;

  /* increase the predecessor count for each vertex that is a successor of another one */
  for (v=0; v < G.n; v++)
    for (curedge=G.firstedge[v]; curedge; curedge=curedge->nextedge)
       predecessorcount[curedge->endpoint]++;
```



Topological Sort in C (cont'd)

```
 /* initialize a queue */
  InitializeQueue(&Q);

  /* place all vertices with no predecessors into the queue */
  for (v=0; v < G.n; v++)
    if (predecessorcount[v]==0)
      Insert(v, &Q);

  /* start the breadth-first traversal */
  place=-1;
  while (!Empty(&Q)) {
    /* visit v by placing it into the topological order */
    Remove(&Q, &v);
    place++;
    T[place]=v;

    /* traverse the list of successors of v */
    for (curedge=G.firstedge[v]; curedge; curedge=curedge->nextedge){
       /* reduce the predecessor count for each successor */
       succ=curedge->endpoint;
       predecessorcount[succ]--;
       if (predecessorcount[succ]==0)
         /* succ has no further predecessors, so it is ready to process */
         Insert(succ, &Q);
    }
  }
}
```



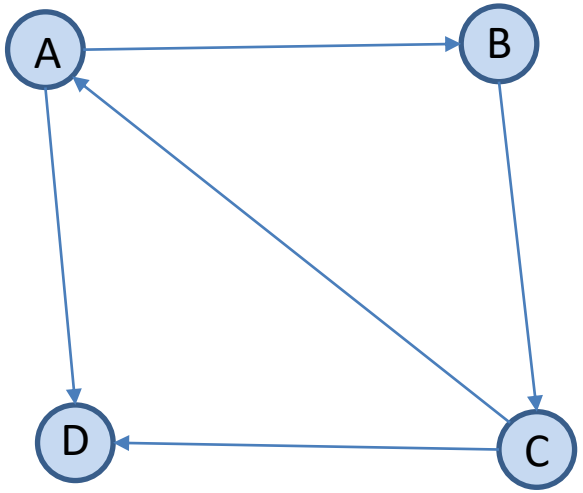
Complexity of Topological Sort

- We can see that the complexity of topological sort is $\boldsymbol{O(n+e)}$.

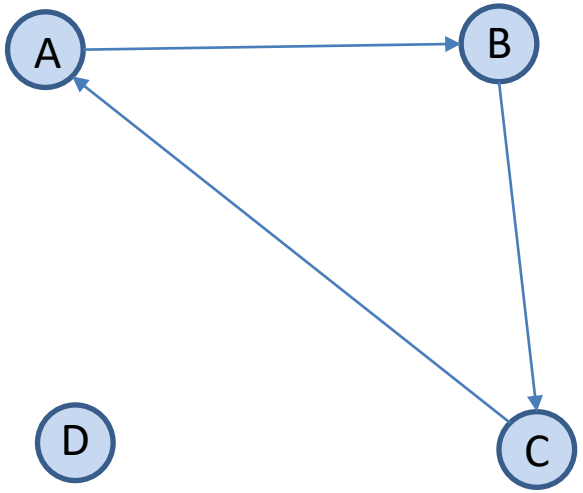
Strongly Connected Components of a Directed Graph (Reminder)

- A **strongly connected component or strong component (ισχυρά συνεκτική συνιστώσα ή ισχυρή συνιστώσα)** of a directed graph is a maximal set of vertices in which there is a path from any one vertex in the set to any other vertex.
- More formally, let $G = (V, E)$ be a directed graph. We can partition V into equivalence classes $V_i, 1 \leq i \leq r$, such that vertices v and w are equivalent if and only if there is a path from v to w and a path from w to v. Let $E_i, 1 \leq i \leq r$, be the set of edges with endpoints in V_i. The graphs $G_i = (V_i, E_i)$ are called the **strongly connected components** or just **strong components (ισχυρές συνιστώσες)** of G.
- A directed graph with only one strong component is strongly connected (ισχυρά συνεκτικός).

Example Directed Graph



The Strong Components of the Digraph

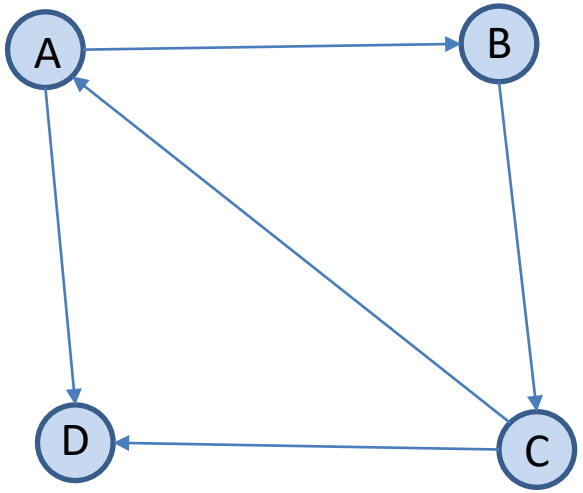


Strong Components (cont'd)

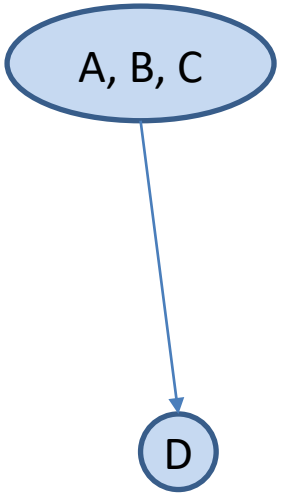
- Every vertex of a directed graph G is in some strong component, but certain edges may not be in any component.
- Such edges, called **cross-component** edges, go from one vertex in one component to a vertex in another.
- We can represent the interconnections among components by constructing a **reduced graph (ελαττωμένο γράφο)** for G.

- The vertices of the reduced graph correspond to the strong components of G.
- There is an edge from vertex C to vertex C' of the reduced graph if there is an edge in G from some vertex in the component C to some vertex in the component C'.
- The reduced graph is always a dag.

Example Directed Graph G



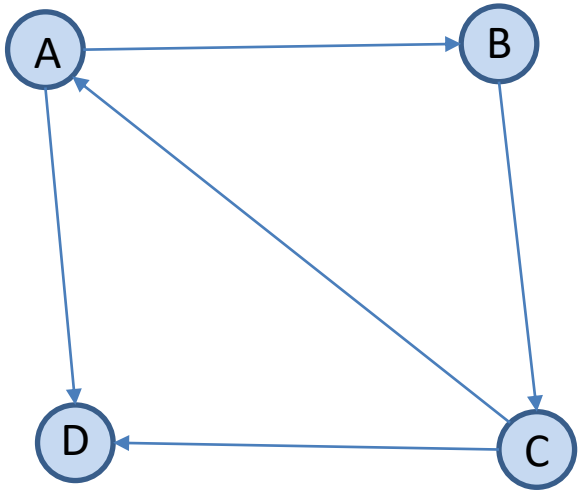
Example Reduced Graph for G



Kosaraju's Algorithm for Computing Strong Components

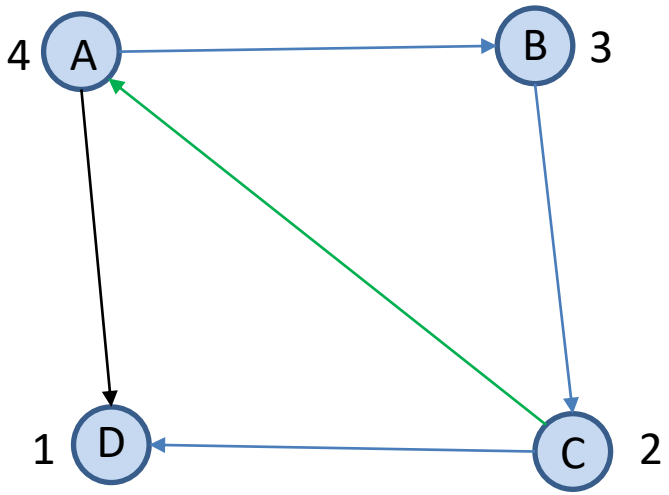
- We can use DFS to compute the strong components of a given directed graph G as follows:
  1. Perform a DFS of G and number the vertices in order of completion of the recursive calls (postorder numbering).
  2. Construct the **reverse** of G, a new directed graph G_r by reversing the direction of each edge in G.
  3. Perform a DFS of G_r, starting the search from the highest numbered vertex according to the postorder numbering assigned in Step 1. If the DFS does not reach all vertices, start the next DFS from the highest-numbered remaining vertex.
  4. Each tree in the resulting DFS forest of G_r gives us a strongly connected component of G.

Example Directed Graph G



- We first perform a DFS starting from vertex A.

After Step 1



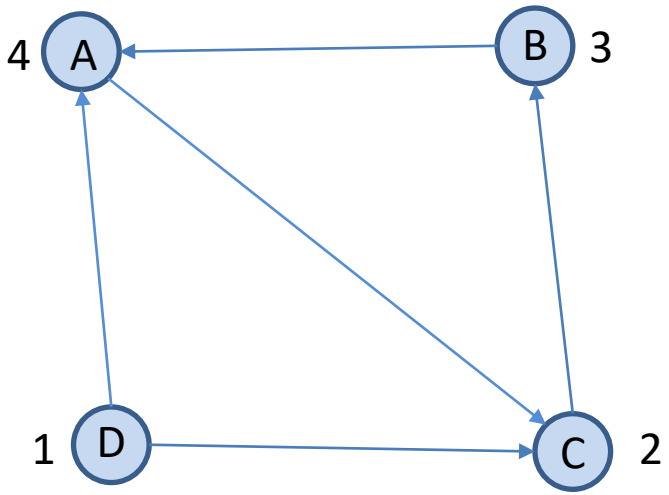
Tree edges

Forward edges

Back edges

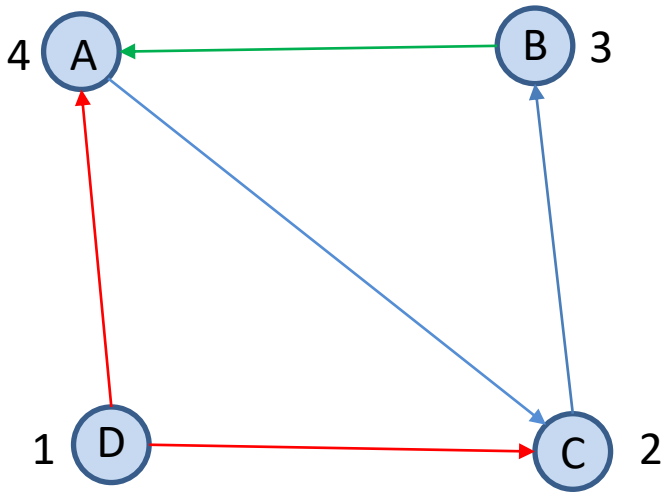
- The numbers show the post-order numbering.

The Reverse Directed Graph G_r



- Then we perform a DFS of G_r starting from the highest-numbered vertex A. The search will visit the vertices in the order A, C, B and D.

Characterization of Edges of G_r

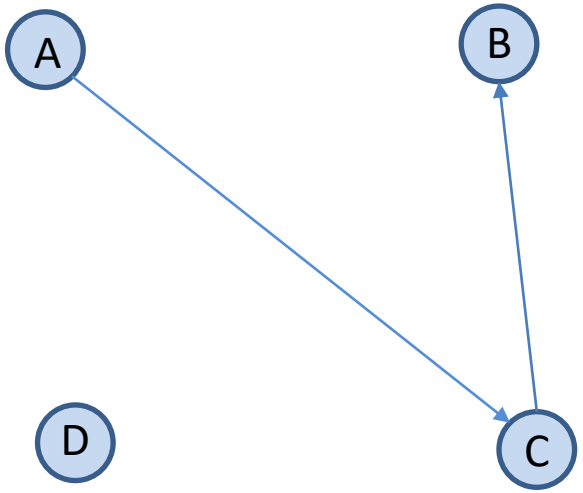


Tree edges

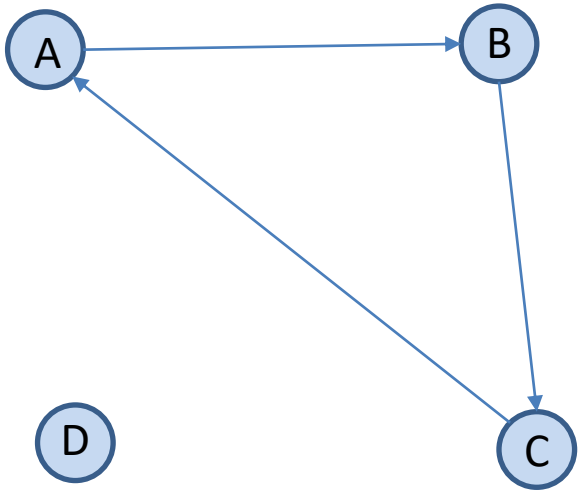
Cross edges

Back edges

DFS Forest for G_r



The Strong Components of G



Complexity of Algorithm for Computing Strong Components

- The complexity of the algorithm we presented for computing strong components is again $\boldsymbol{O(n+e)}$.
- This can be seen easily seen because the complexity for every step of the algorithm is $\boldsymbol{O(n+e)}$.

Readings

- The material in the present slides comes (often verbatim) from the following sources:
  - T. A. Standish. *Data Structures , Algorithms and Software Principles in C*.
    - Chapter 10
  - A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.*
    - Chapters 6 and 7
  - M. T. Goodrich, R. Tamassia and M. H. Goldwasser. *Data Structures and Algorithms in Java. 6th* edition. John Wiley and Sons, 2014.
    - Chapter 14
  - M. T. Goodrich, R. Tamassia. *Δομές Δεδομένων και Αλγόριθμοι σε Java*. 5η έκδοση. Εκδόσεις Δίαυλος.
    - Chapter 13
  - R. Sedgewick. *Algorithms in C. 3rd edition. Part 5. Graph Algorithms.*
    - Chapters 18 and 19
  - T. H Cormen, C. E. Leiserson and R.L. Rivest. *Introduction to Algorithms*.
    - Chapters 5.4 and 23.