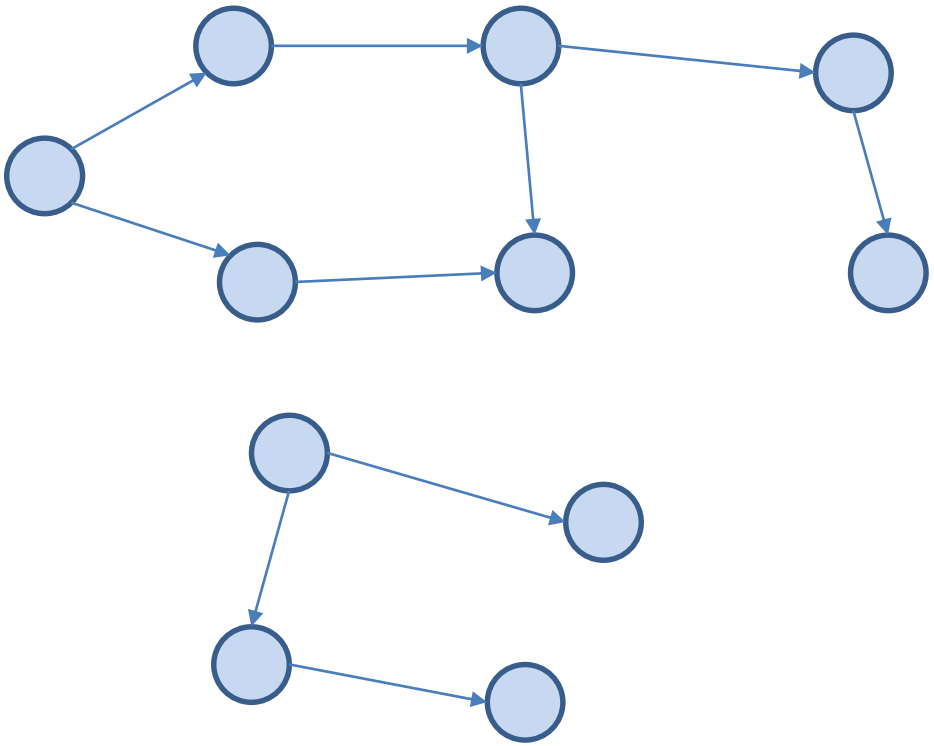
Graphs (Γράφοι)

Manolis Koubarakis

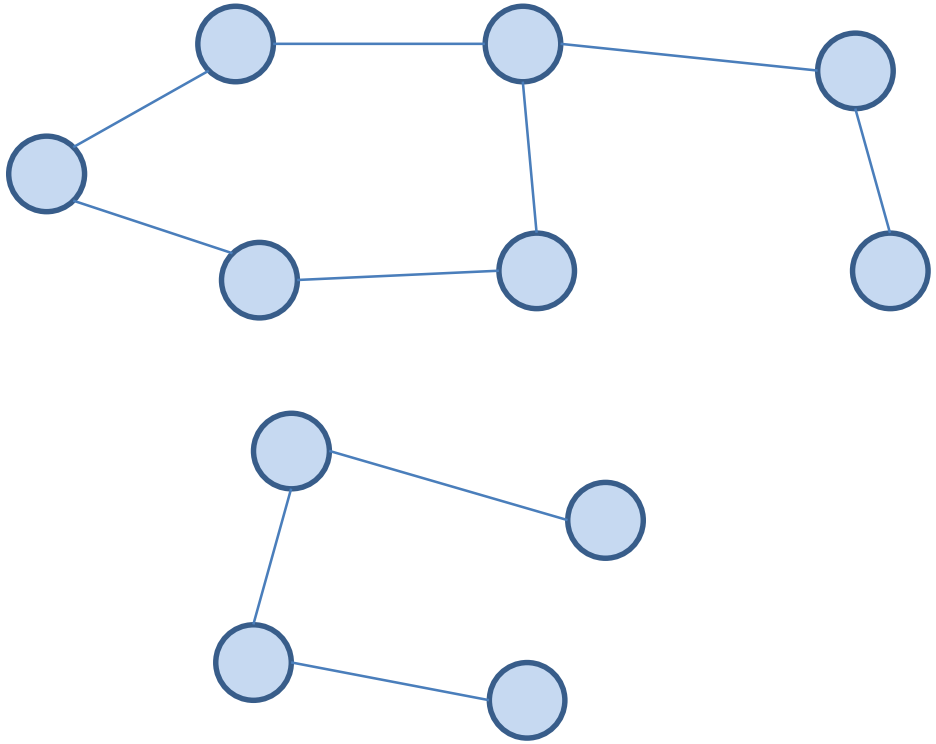
Graphs

- **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.

Example of Graphs (Directed)



Example of Graphs (Undirected)

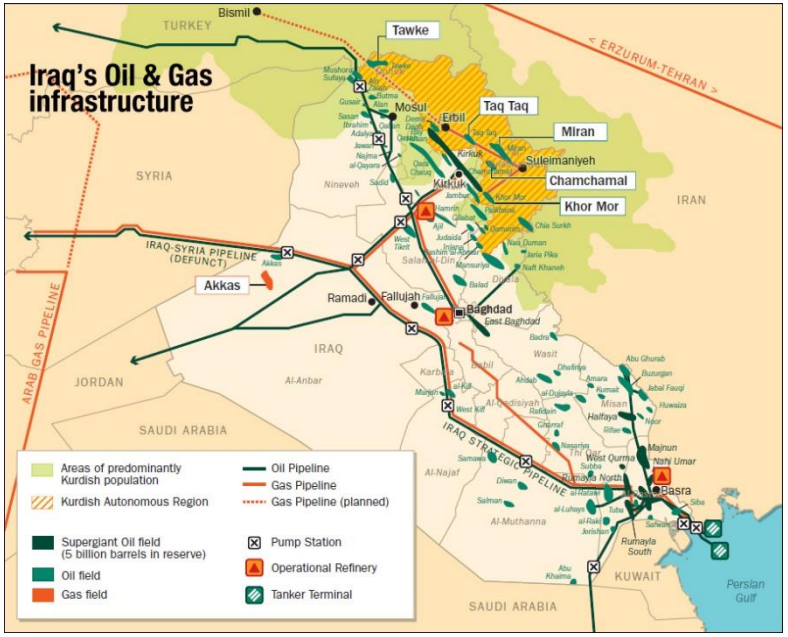


Examples of Graphs

- 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?

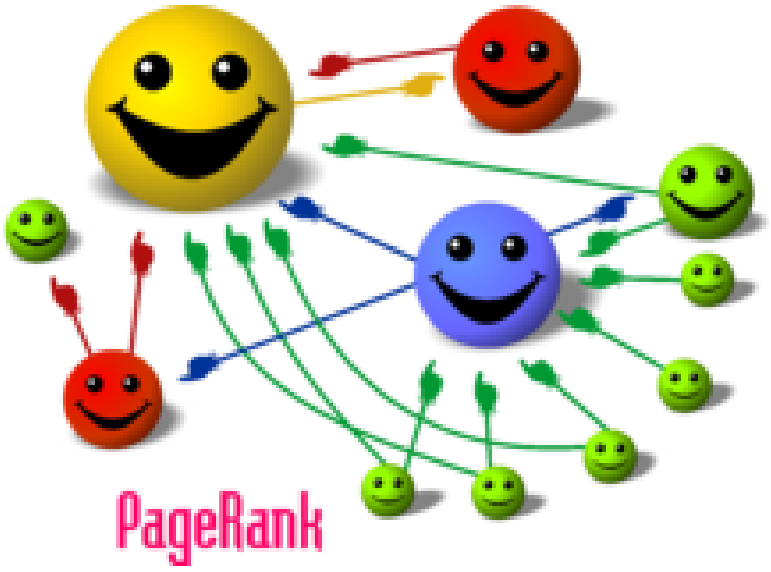


Examples (cont'd)

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

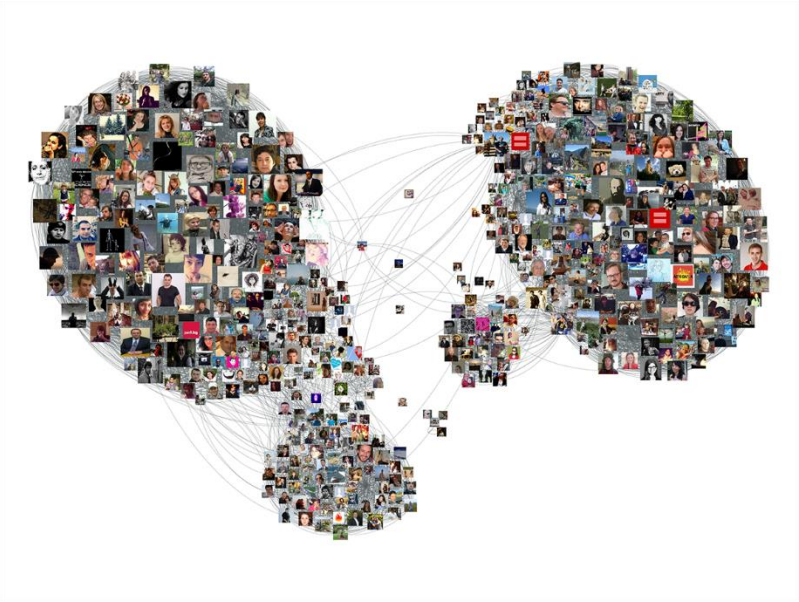
Examples (cont'd)

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



Examples (cont'd)

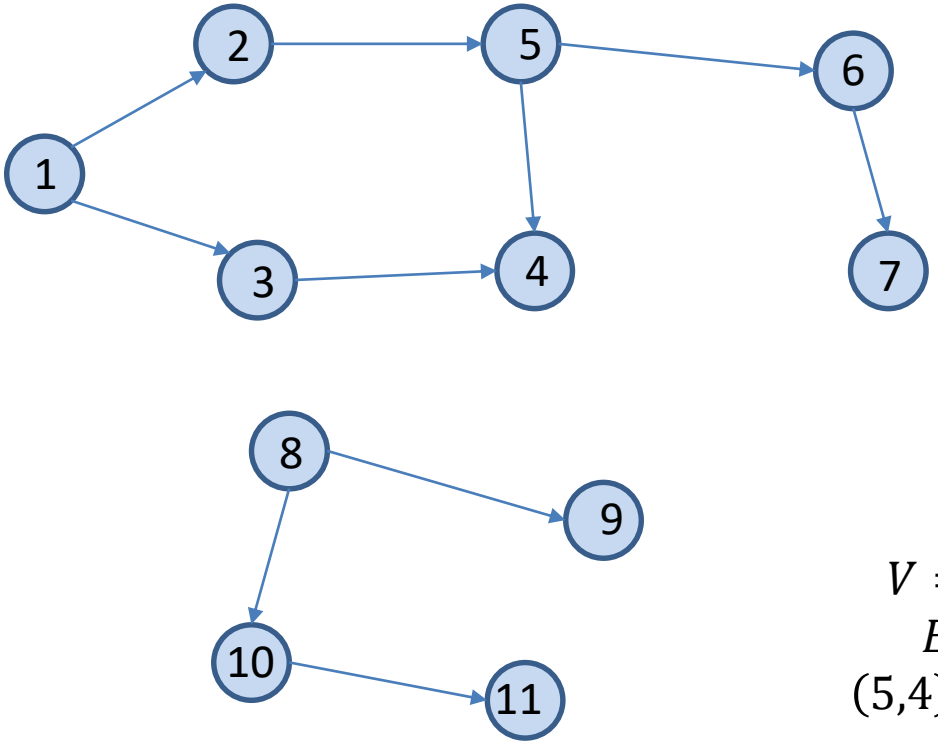
- The Facebook social network
- **Interesting problem**: Are John and Mary connected? What interesting clusters exist?



Formal Definitions

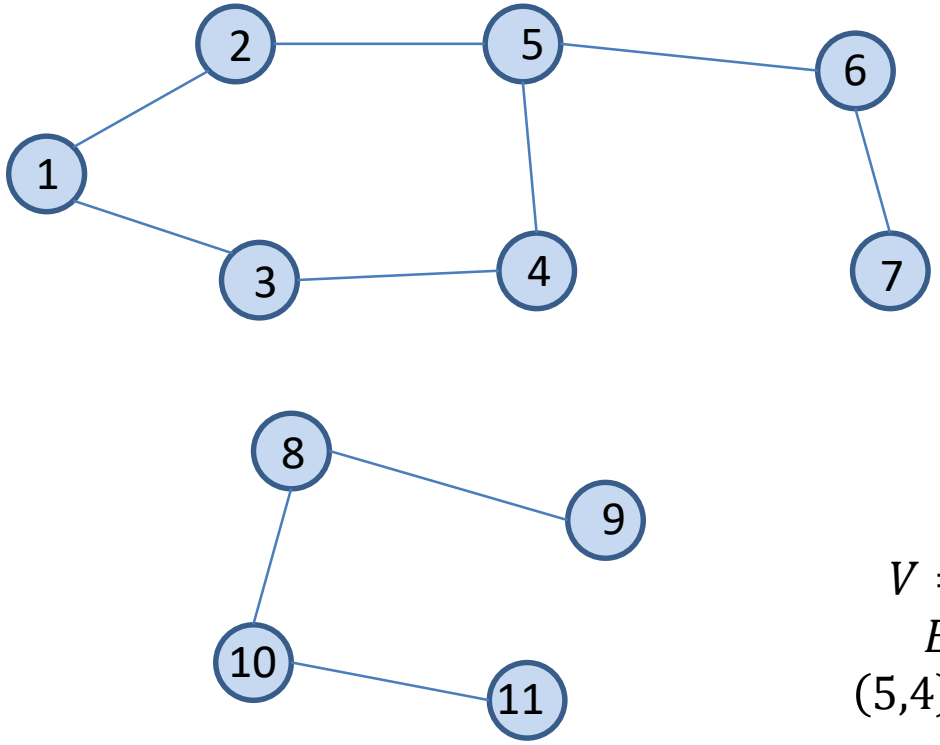
- A **graph** $G = (V, E)$ consists of a set of **vertices** V and a set of **edges** E, where the edges in E are formed from pairs of **distinct** vertices in V.
- If the edges have directions then we have a **directed graph (κατευθυνόμενο γράφο)** or **digraph**. In this case edges are ordered pairs 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).
- When we say simply graph, we will mean an undirected graph.

Example of a Directed Graph



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

Example of an Undirected Graph



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

More Definitions

- 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 $e = (v_i, v_j)$.
- An edge is said to be **incident (προσπίπτουσα)** on a vertex if the vertex is one of the edge's endpoints.
- A **path (μονοπάτι)** p in a graph $G = (V, E)$ is a sequence of vertices of V of the form $p = v_1 v_2 \ldots v_n, (n \geq 2)$ in which each vertex v_i is adjacent to the next one v_{i+1} (for $1 \leq i \leq n - 1$).
- 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 **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., $v_1 = v_n$).

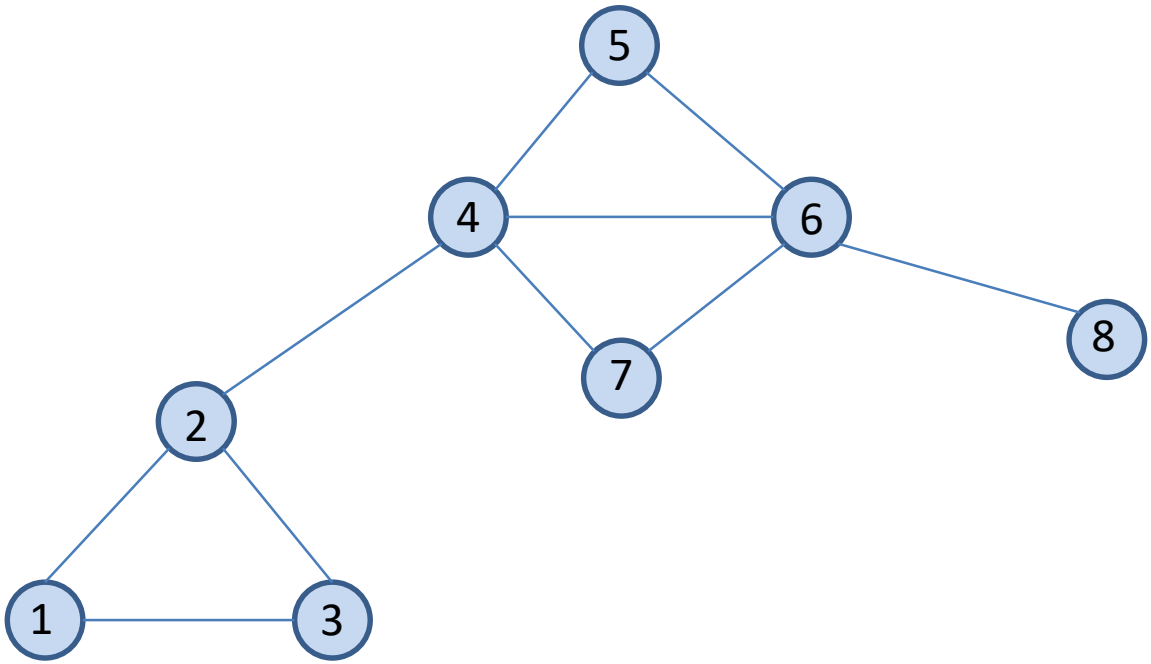
Definitions (cont'd)

- A **directed path** is a path such that all edges are directed and are traversed along their direction.
- A **directed cycle** is similarly defined.

Definitions (cont'd)

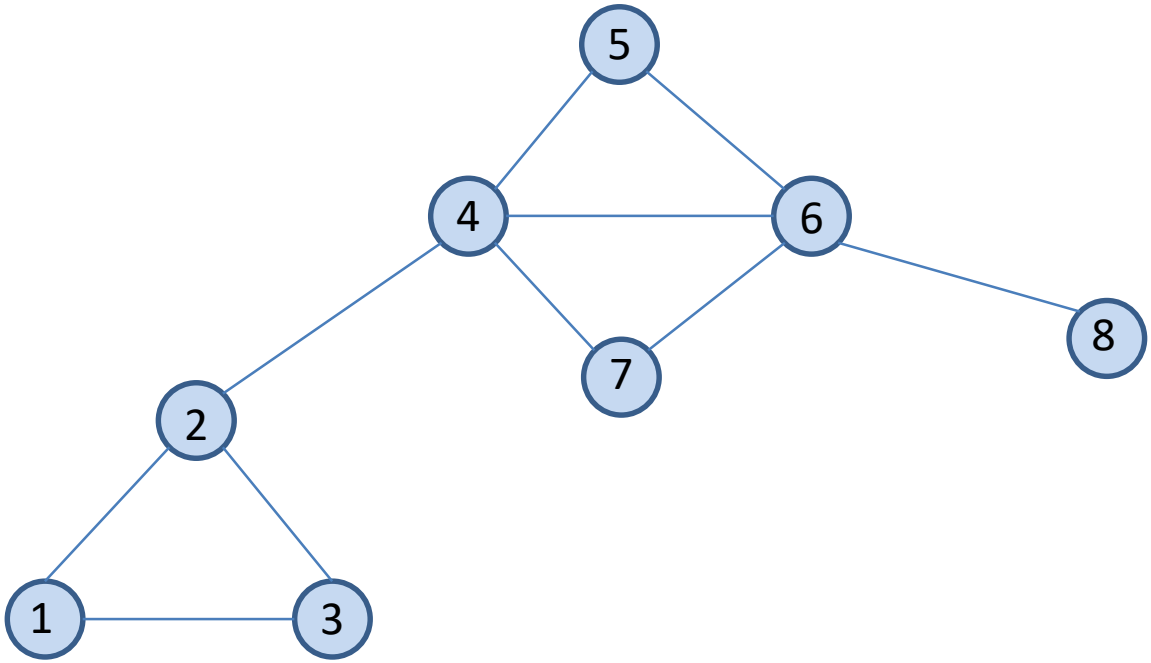
- A **simple cycle** is a path that travels through three or more **distinct** vertices and connects them into a loop.
- Formally, if p is a path of the form $p = v_1 v_2 \ldots v_n$, then p is a **simple cycle** if and only if $n > 3, v_1 = v_n$ and $v_i \neq v_j$ for distinct i and j in the range $1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

Example



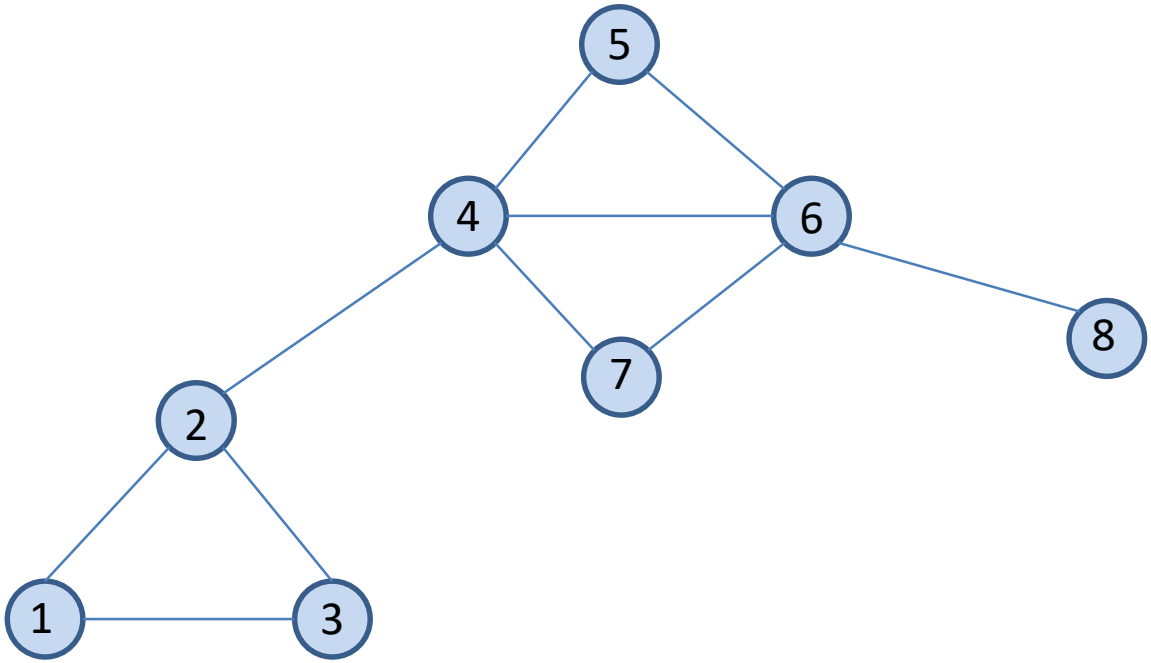
Four simple cycles: (1,2,3,1) (4,5,6,7,4) (4,5,6,4) (4,6,7,4)

Example (cont'd)



Two non-simple cycles: (1,2,1) (4,5,6,4,7,6,4)

Example (cont'd)



A path that is not a cycle: (1,2,4,6,8)

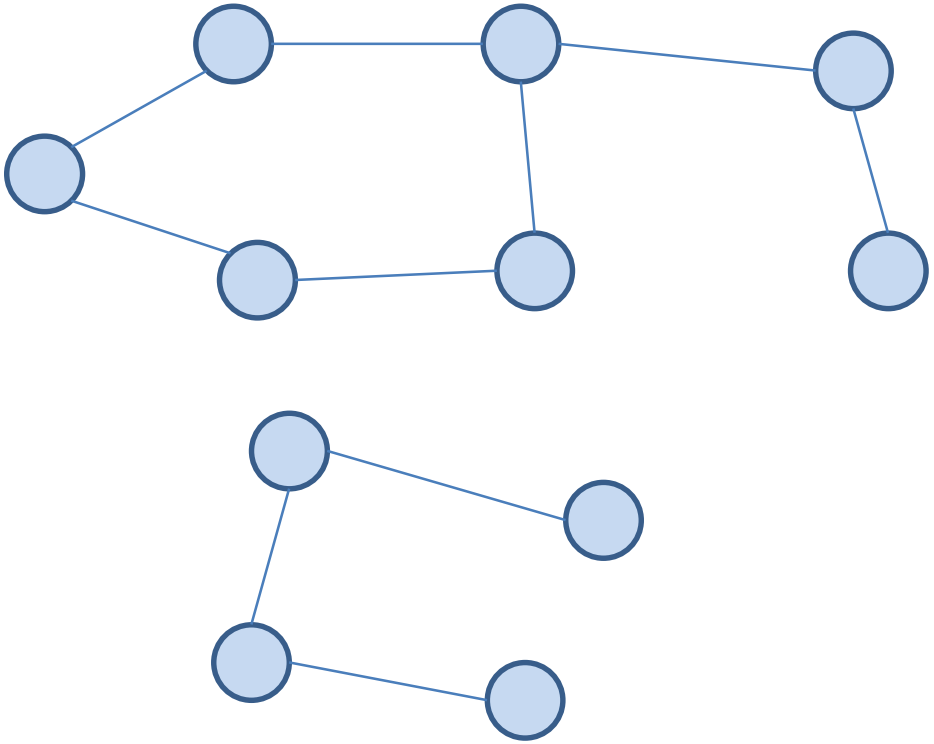
Connectivity and Components

- Two vertices in a graph $G = (V, E)$ are said to be **connected (συνδεδεμένες)** if there is a path from the first to the second in G.
- Formally, if $x \in V$ and $y \in V$, where $x \neq y$, then x and y are **connected** if there exists a path $p = v_1 v_2 \ldots v_n$ in G such that $x = v_1$ and $y = v_n$.

Connectivity and Components (cont'd)

- In the graph $G = (V, E)$, a **connected component (συνεκτική συνιστώσα)** is a subset S of the vertices V that are all connected to one another.
- A connected component S of G is a **maximal connected component (μέγιστη συνεκτική συνιστώσα)** provided there is no bigger subset T of vertices in V such that T properly contains S and such that T itself is a connected component of G.
- An undirected graph G can always be separated into maximal connected components $S_1, S_2, \ldots, S_n$ such that $S_i \cap S_j = \emptyset$ whenever $i \neq j$.

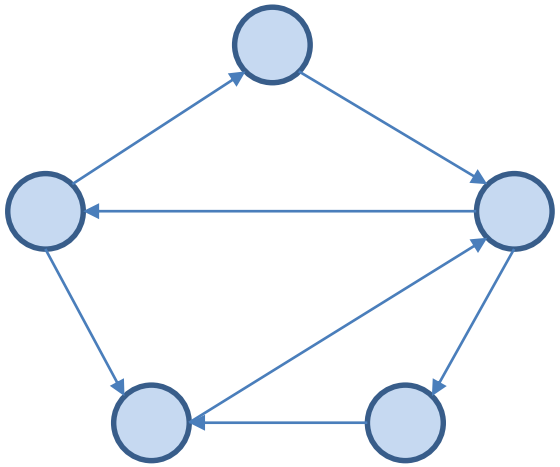
Example of Undirected Graph and its Separation into Two Maximal Connected Components



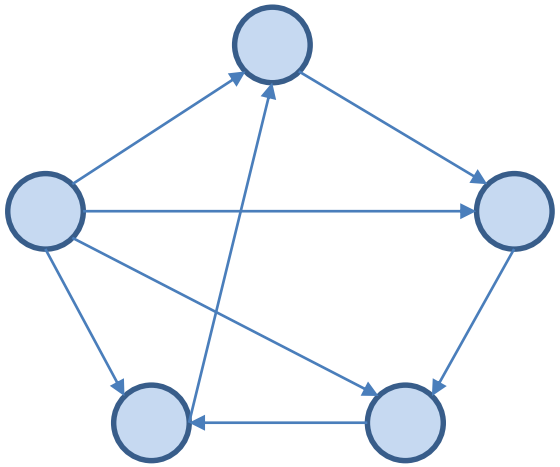
Connectivity and Components in Directed Graphs

- A subset S of vertices in a **directed** graph G is **strongly connected (ισχυρά συνεκτικό)** if for each pair of distinct vertices (v_i, v_j) in S, v_i is connected to v_j and v_j is connected to v_i .
- A subset S of vertices in a **directed** graph G is **weakly connected (ασθενώς συνεκτικό)** if for each pair of distinct vertices (v_i, v_j) in S, v_i is connected to v_j or v_j is connected to v_i .

Example: A Strongly Connected Digraph



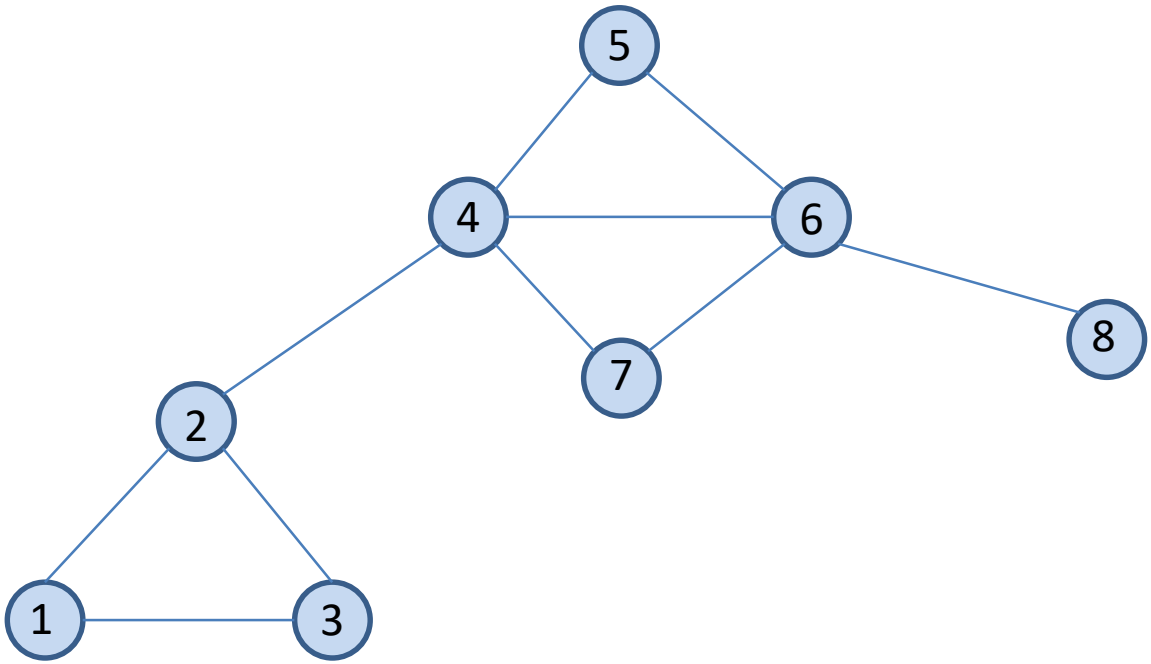
Example: A Weakly Connected Digraph



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

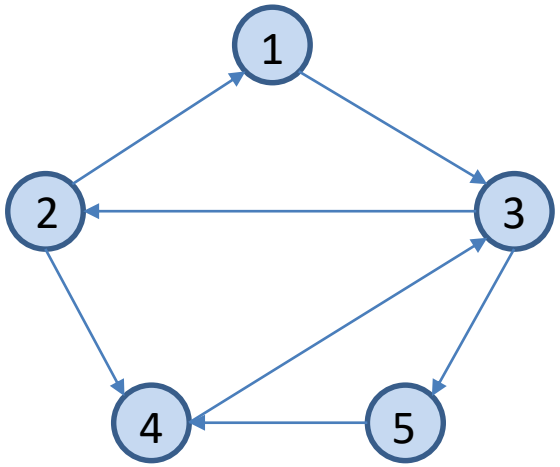
Predecessors and Successors in Directed Graphs

- 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 node 4 is 2. The out-degree of node 4 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.

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?

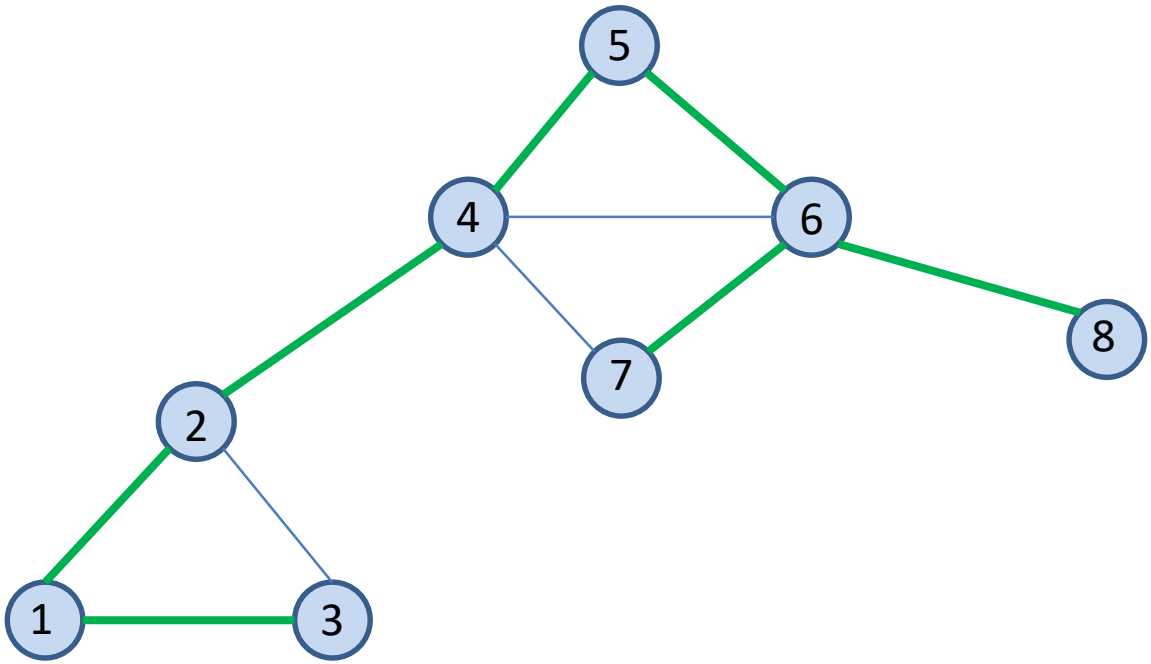
Proof

- 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

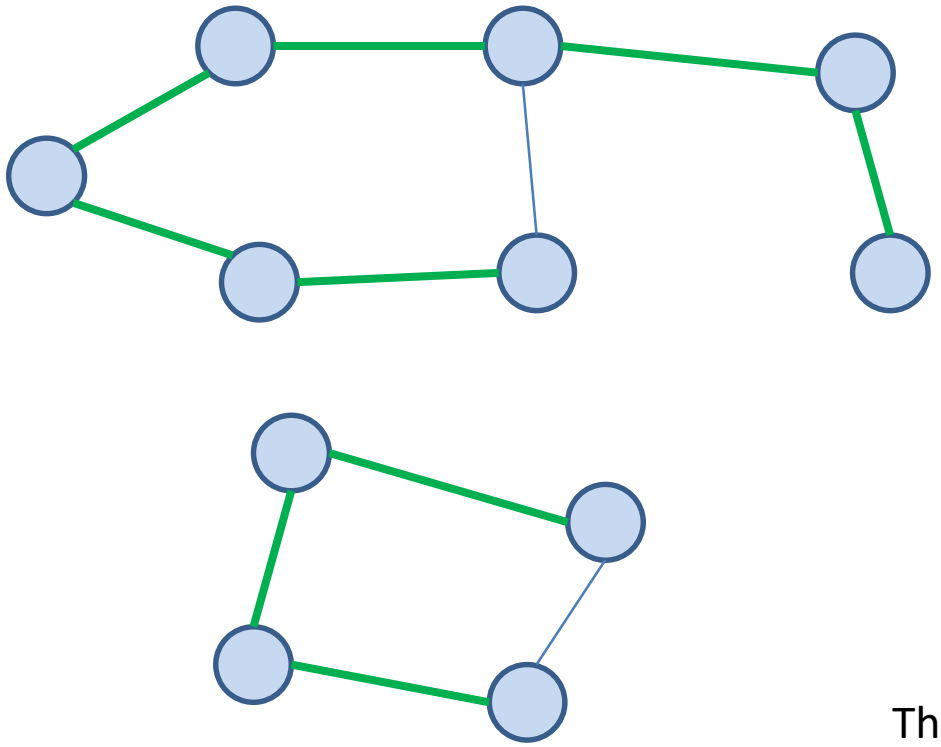
- A **subgraph (υπογράφος)** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G respectively.
- A **spanning subgraph (υπογράφος επικάλυψης)** of G is a subgraph of G that contains all the vertices of G.
- A **forest (δάσος)** is a graph without cycles.
- A **free tree (ελεύθερο δένδρο)** is a connected forest i.e., a connected 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)** of a graph is a spanning subgraph that is a free tree.

Example



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

Example

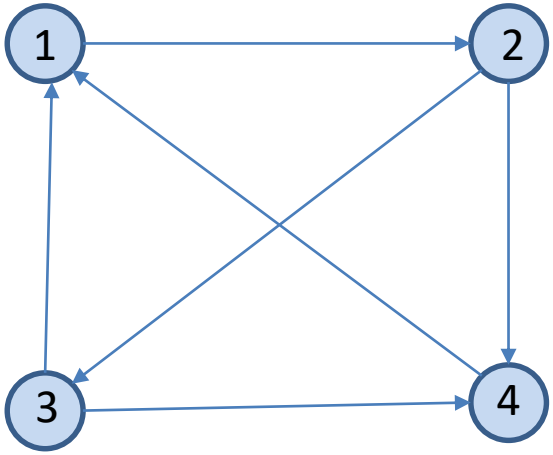


The thick green lines define a forest which consists of two free trees.

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 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

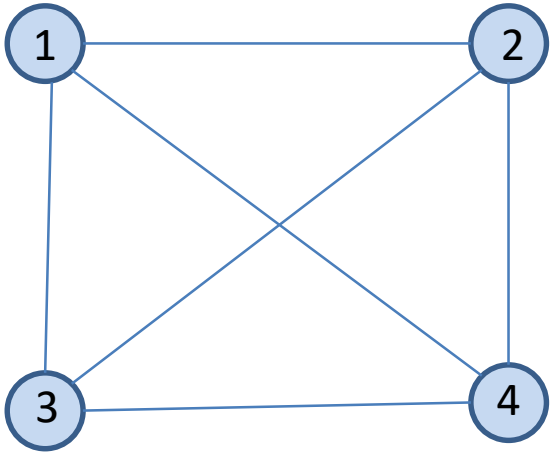
- The adjacency matrix of an **undirected graph** G is a **symmetric matrix** i.e., $T[i, j] = T[j, i]$ for all i and j in the range $1 \leq i, j \leq n$.

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

Adjacency Matrices

- 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.

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 in C

```
#define MAXVERTEX 10

typedef enum {FALSE, TRUE} Boolean

typedef Boolean
        AdjacencyMatrix[MAXVERTEX][MAXVERTEX]

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

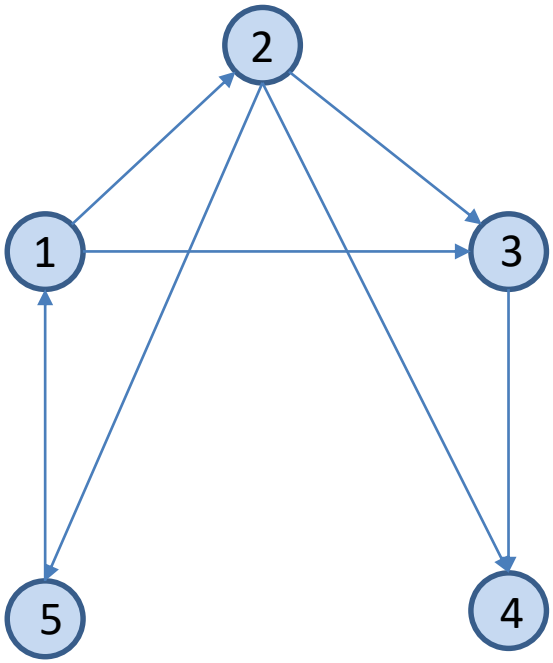
Adjacency Sets

- 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

- Another family of representations for a graph uses **adjacency lists (λίστες γειτνίασης)** to represent the adjacency set V_x for each vertex x in the graph.

Example Directed Graph

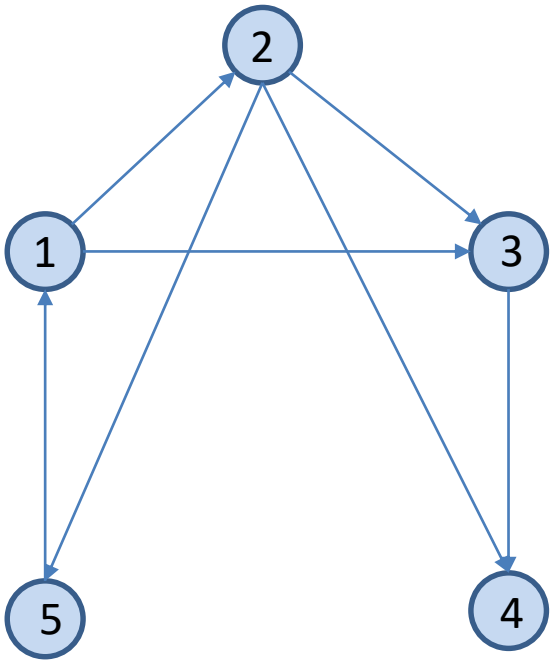


A directed graph G

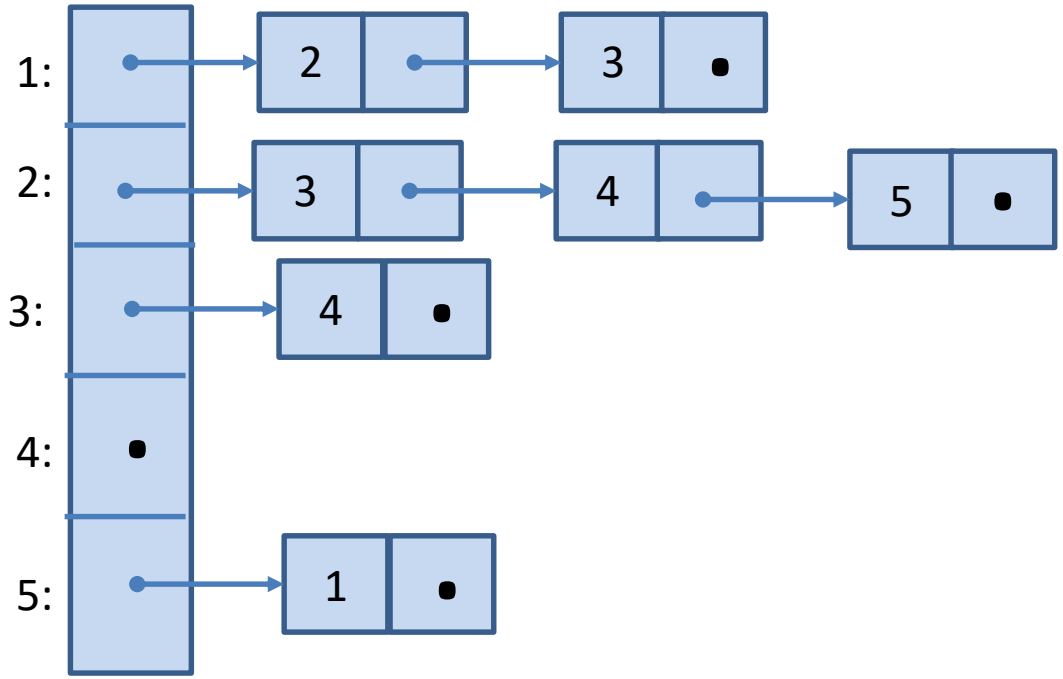
Vertex Number	Out Degree	Adjacency list
1	2	2 3
2	3	3 4 5
3	1	4
4	0	
5	1	1

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

Example Directed Graph

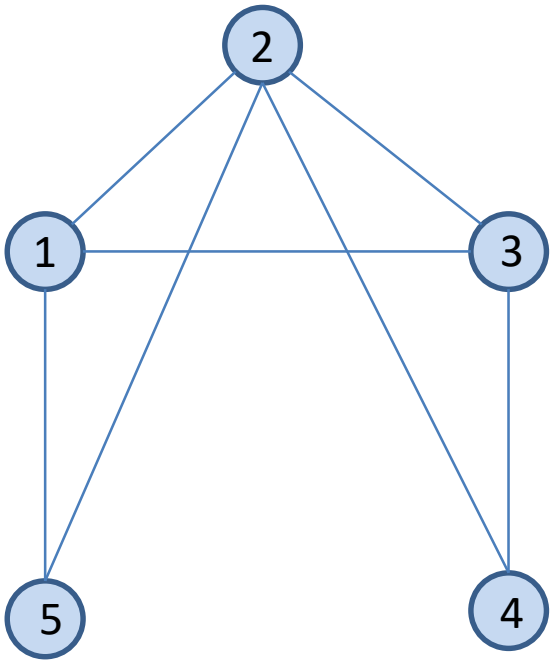


A directed graph G



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 4
5	2	1 2

The sequential adjacency lists for graph G

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;
```

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;
```

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;
```

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`.

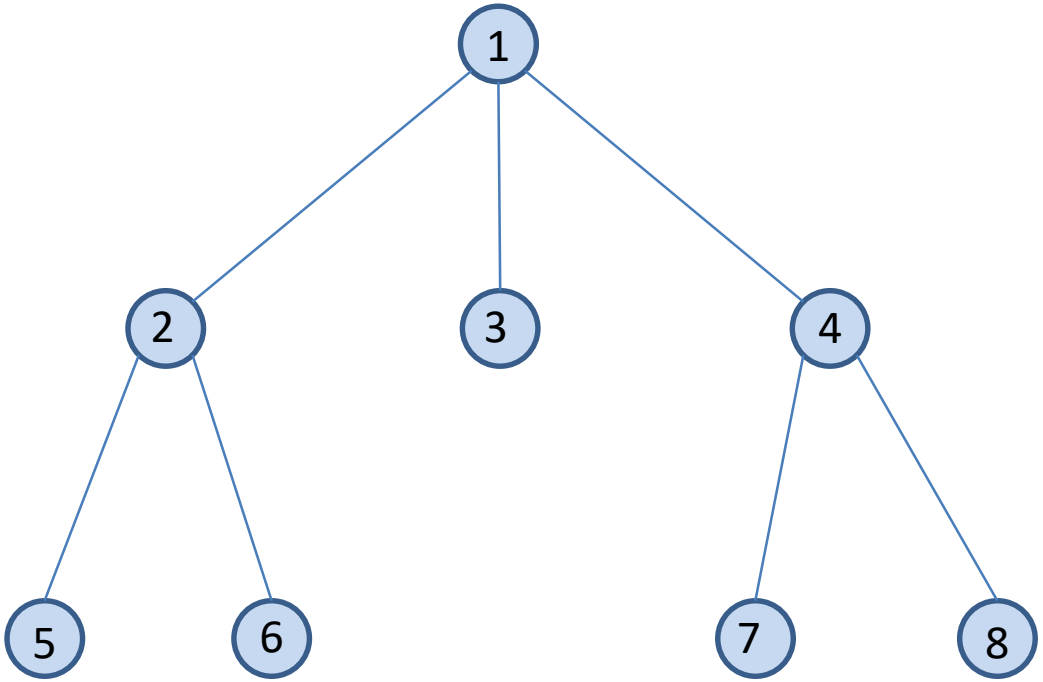
An Algorithm for Graph Searching

```
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;
           }
         }
      }
}
```

Graph Searching (cont'd)

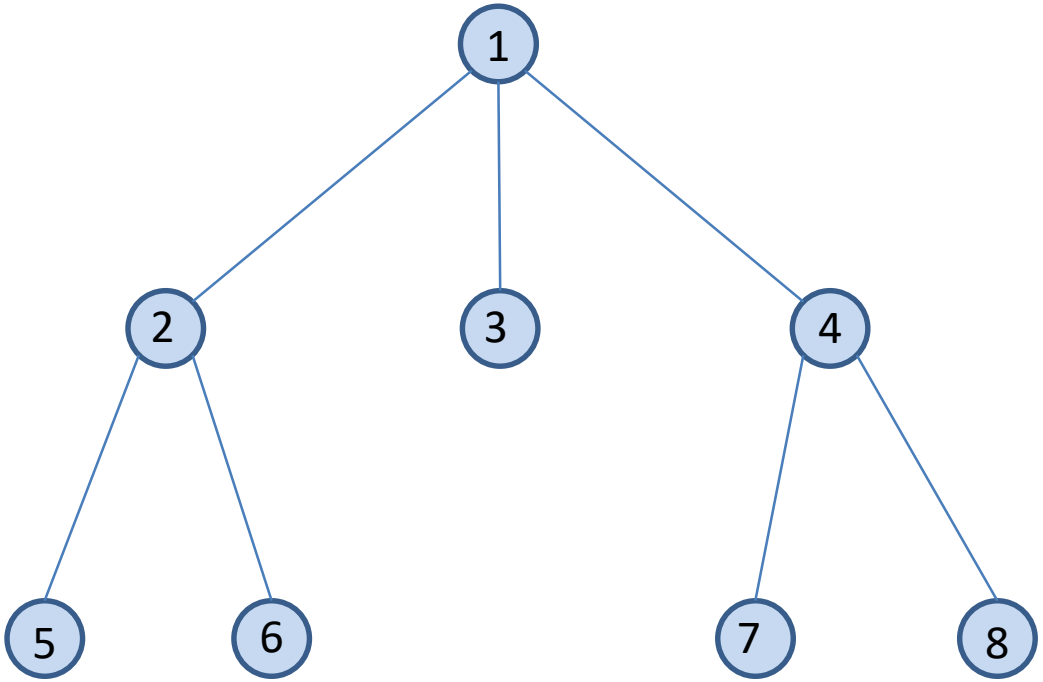
- Let us consider what happens when the container $\mathbb{C}$ is a stack.

Example



What is the order vertices are visited?

Example (cont'd)



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

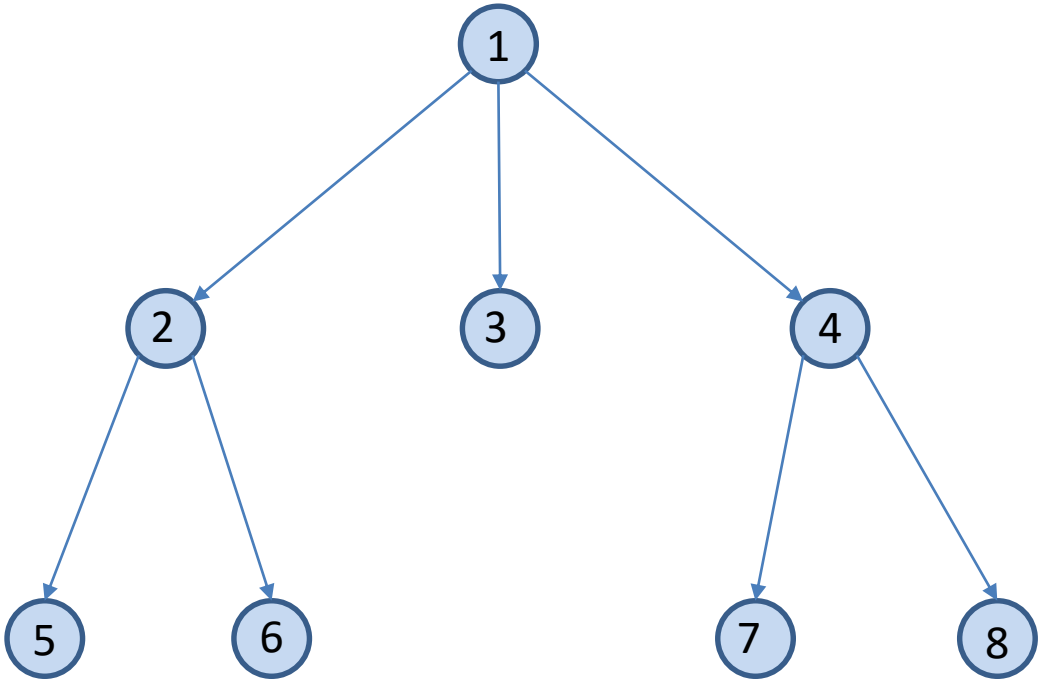
Depth-First Search (DFS)

- 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.

Graph Searching (cont'd)

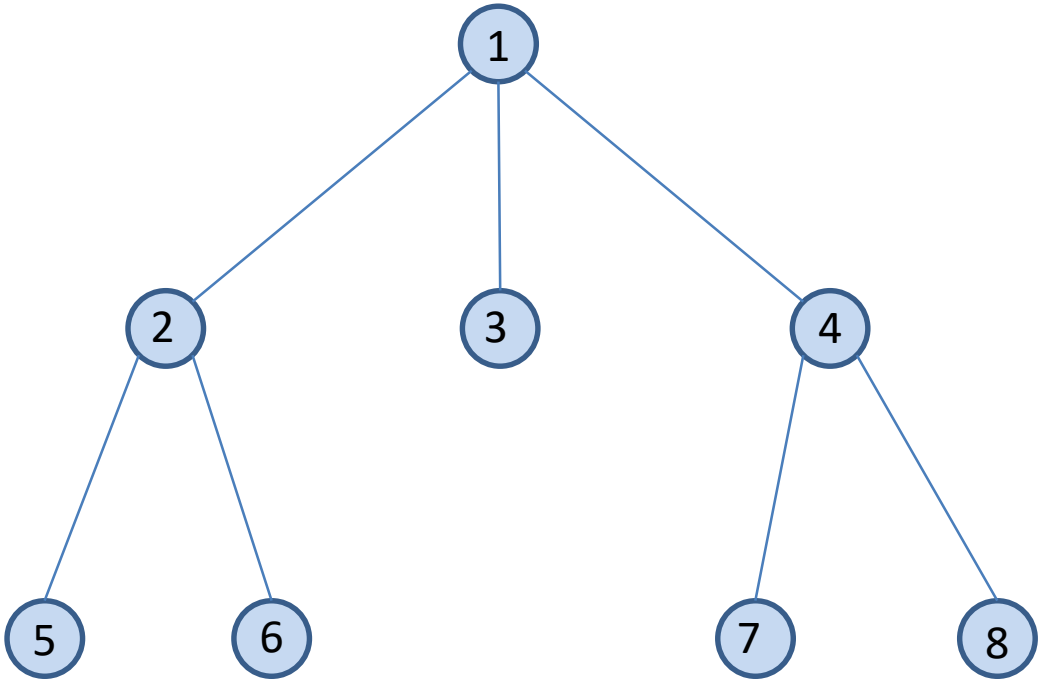
- Let us consider what happens when the container $\mathbb{C}$ is a queue.

Example



What is the order vertices are visited?

Example (cont'd)

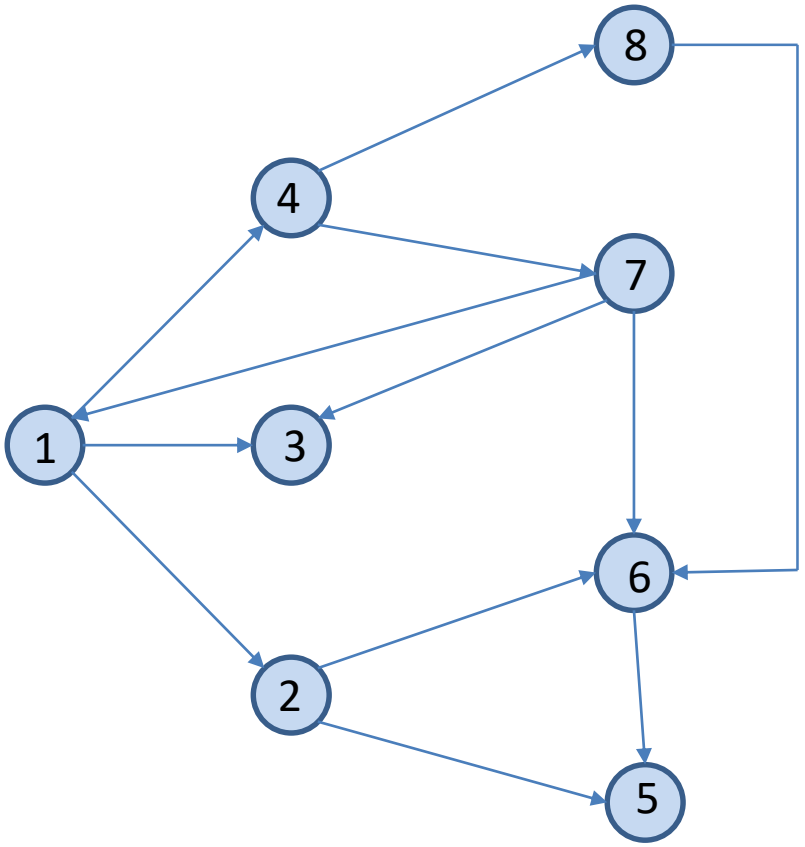


The vertices are visited in the order 1, 2, 3, 4, 5, 6, 7 and 8.

Breadth-First Search (BFS)

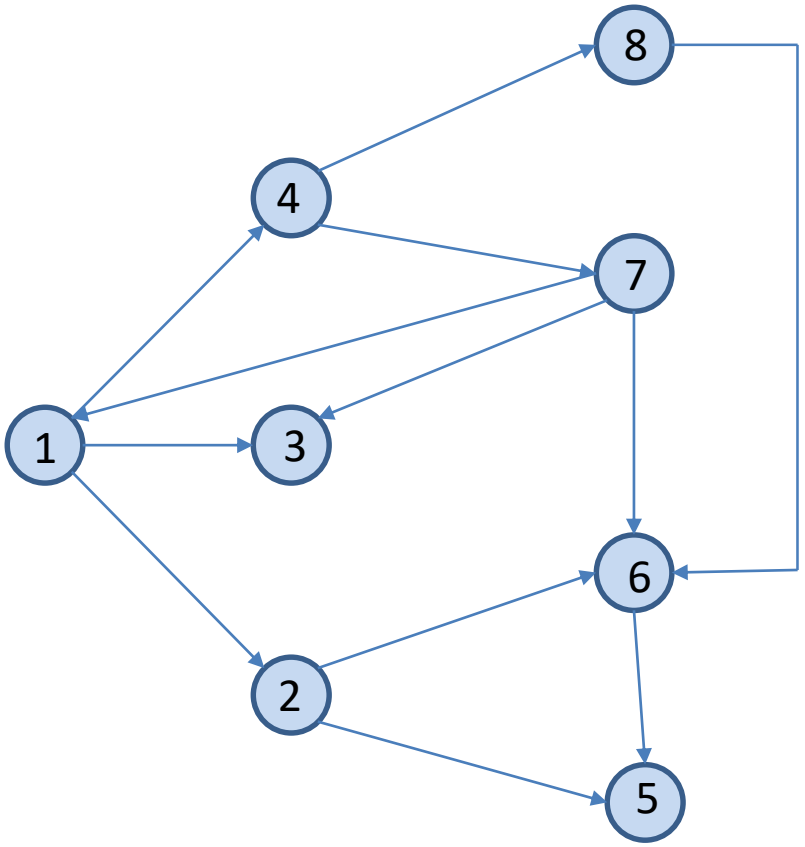
- When $\mathbb{C}$ is a **queue**, the tree in the previous example is searched in **breadth-first order**.
- **Breadth-first search (αναζήτηση πρώτα κατά πλάτος)** at a vertex always goes broad before going deep.
- Breadth-first traversal of a graph is analogous to a traversal of an ordered tree that 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.

Example



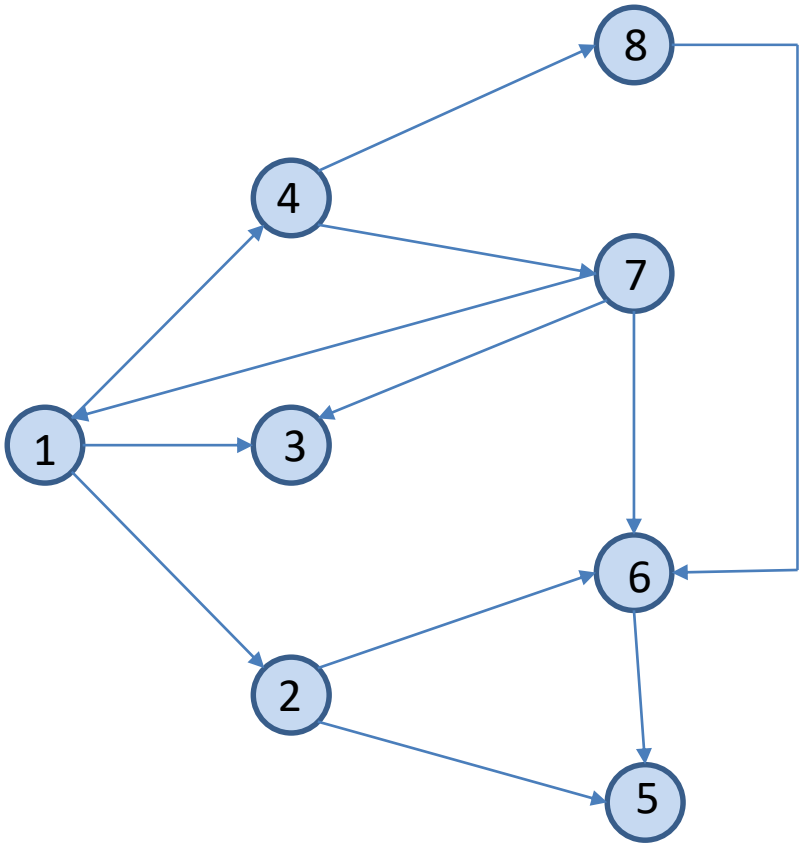
What is the order of visiting vertices for DFS?

Example (cont'd)



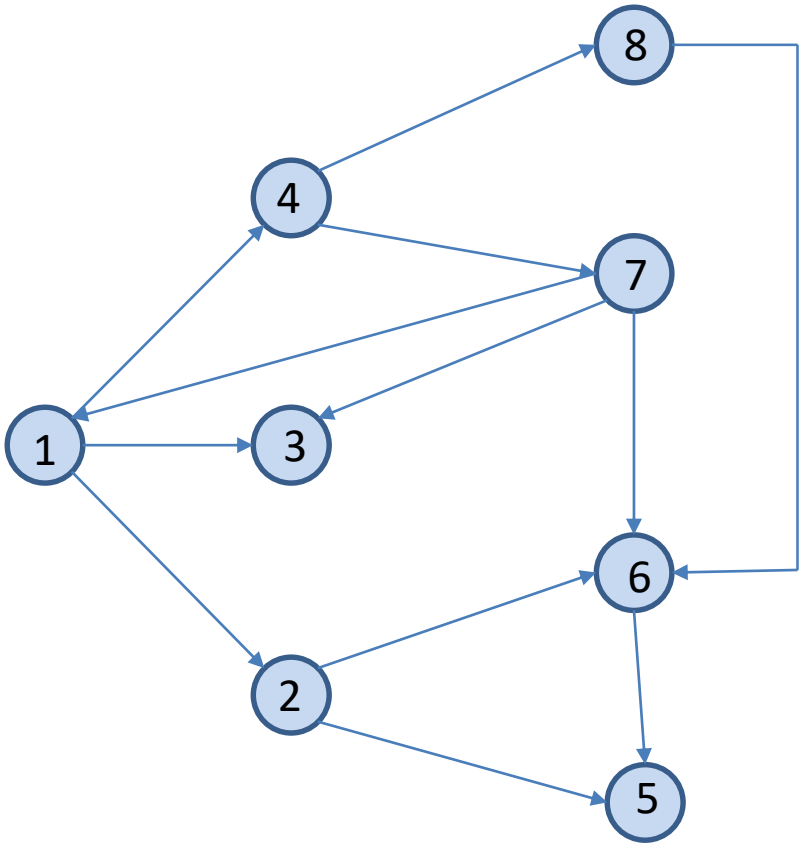
Depth-first search visits the vertices in the order 1, 4, 8, 6, 5, 7, 3 and 2

Example (cont'd)



What is the order of visit for BFS?

Example (cont'd)



Breadth-first search visits the vertices in the order 1, 2, 3, 4, 5, 6, 7 and 8.

Exhaustive Search

- Either 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 and BFS 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!

Implementing DFS in C

- We will now show how to implement depth-first search in C.
- We will use the **linked adjacency lists** representation of a graph.
- We will write a function `DepthFirst` which calls the recursive function `Traverse`.

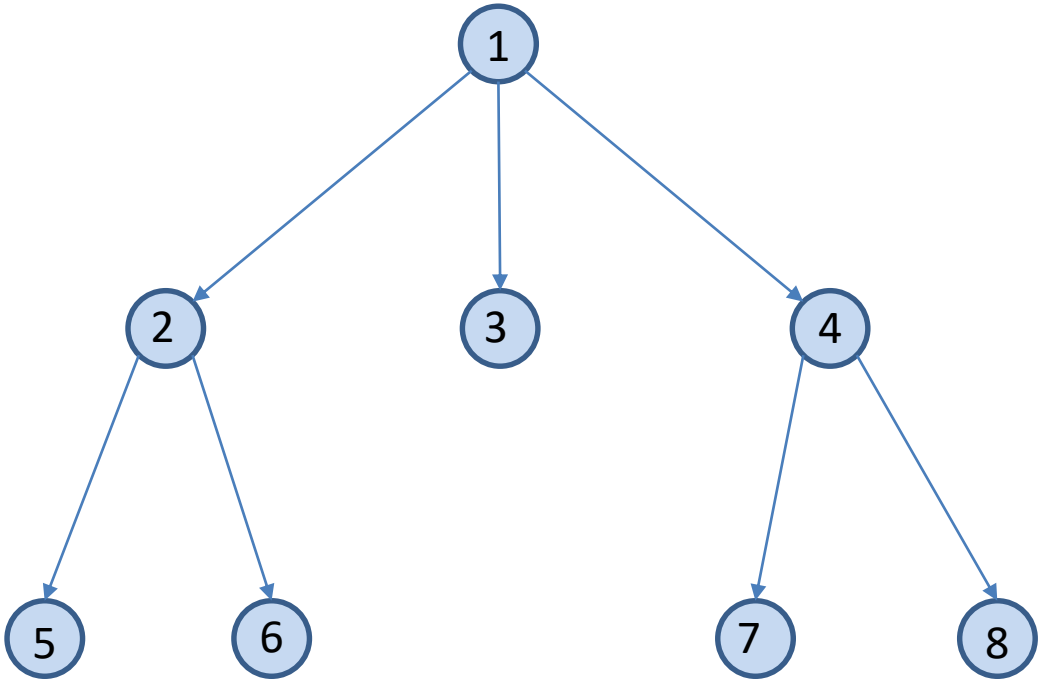
Implementing DFS in C (cont'd)

```
/* 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)

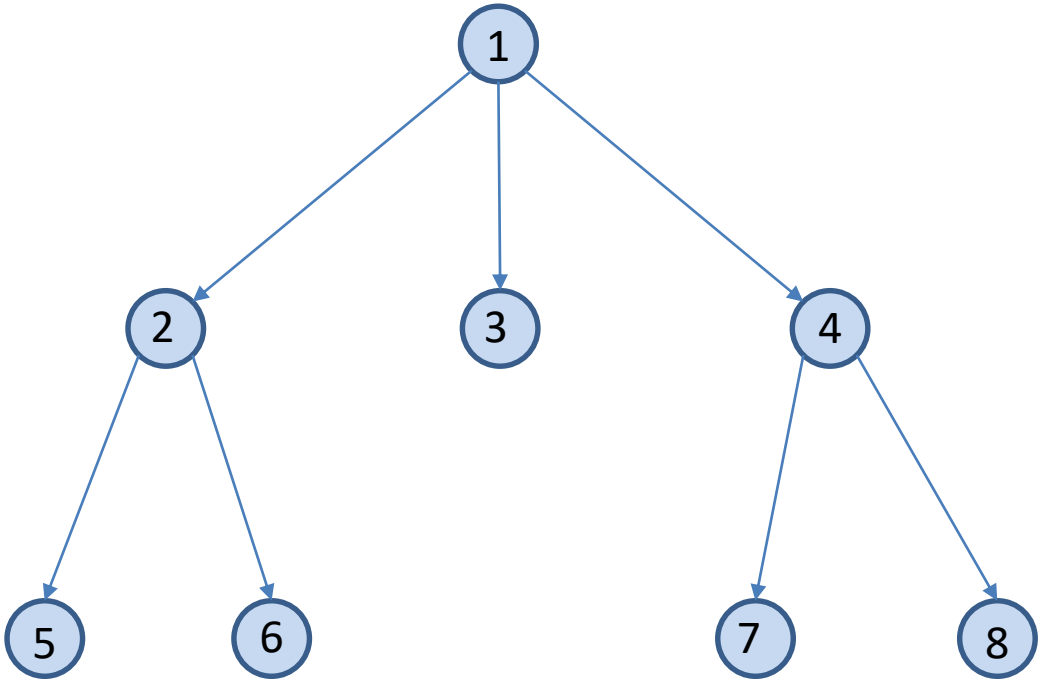
```
/* Traverse: recursive traversal of a graph
   Pre: v is a vertex of graph G
   Post: The depth first traversal, using function Visit, has been
   completed for v and for all vertices adjacent to v.
   Uses: Traverse recursively, Visit */

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 of Recursive DFS



What is the order vertices are visited?

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!

Complexity of DFS

- DFS as implemented above (with 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 $O(e)$.
- The initialization steps in `DepthFirst` have 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.

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 an earlier lecture.

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, w;
  Edge *curedge;

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

  InitializeQueue(&Q);

  for (v=0; v < G.n; v++)
    if (!visited[v]){
      Insert(v, &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));
    }
}
```

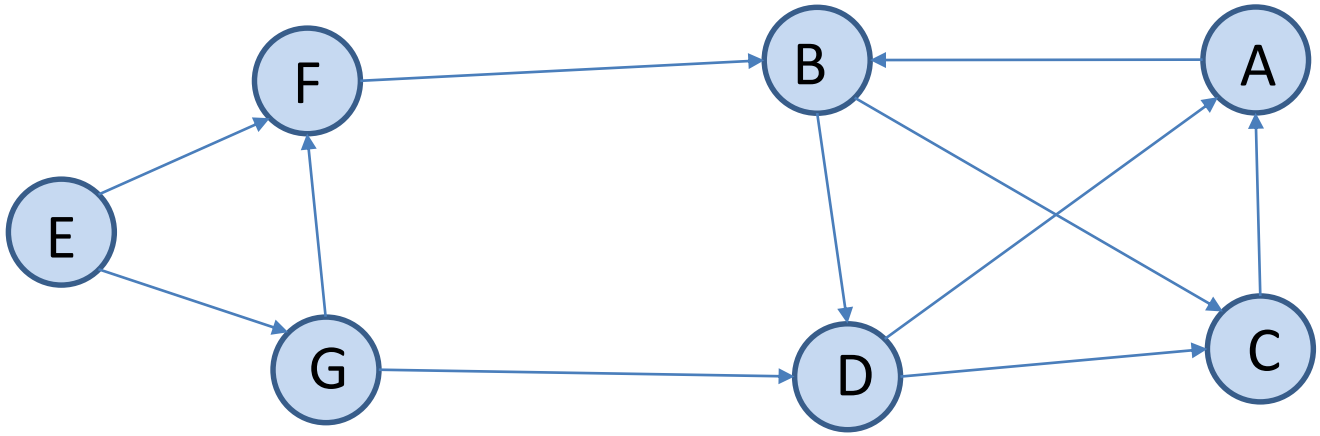
Complexity of BFS

- BFS as implemented above (with adjacency lists) has the same complexity as DFS i.e., $\boldsymbol{O(n+e)}$.

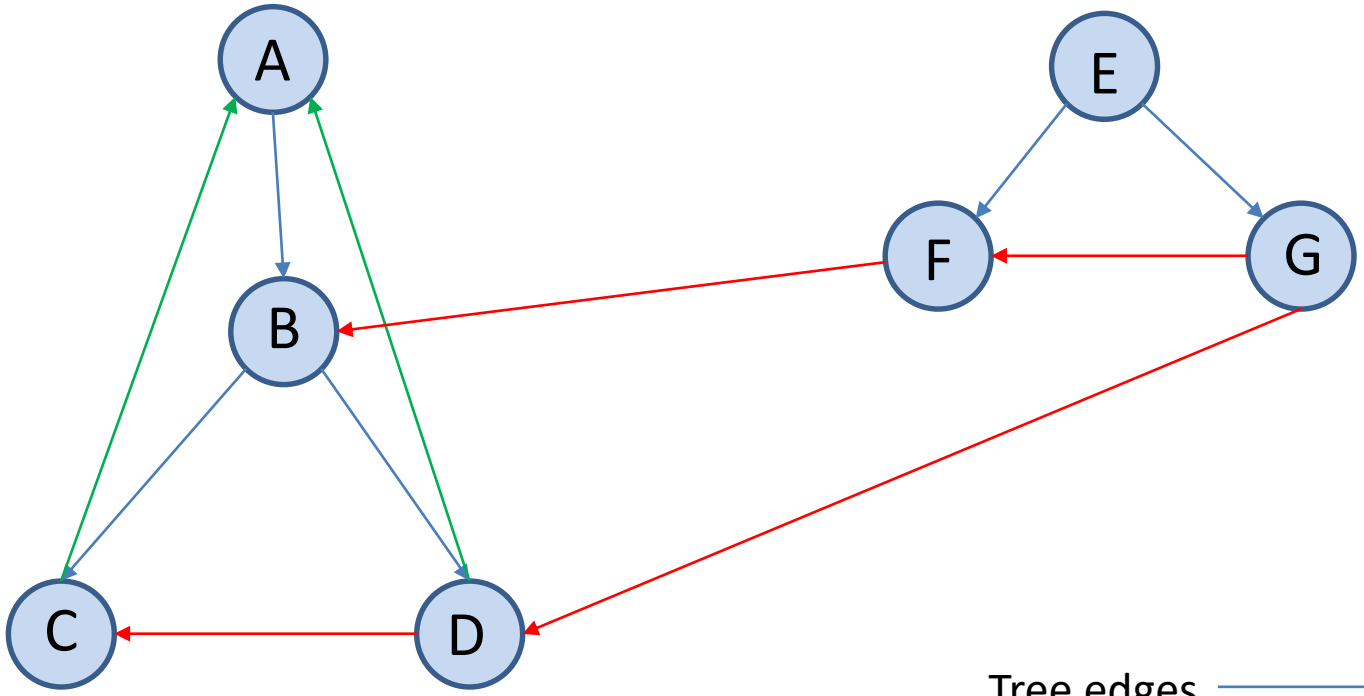
DFS Traversal of a Directed Graph

- 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 **depth-first spanning forest (πρώτα κατά βάθος δάσος επικάλυψης)** for the given digraph.
- There are also edges called **back edges (ακμές οπισθοχώρησης)** that go from a vertex to one of its ancestors in the spanning forest.
- There also edges that do not belong to the spanning forest and go from a vertex to a proper descendant. These are called **forward edges (ακμές προώθησης).**
- Finally, there are edges that go from a vertex to another vertex that is neither an ancestor nor a descendant that are called **cross edges (εγκάρσιες ακμές).**

Example



Example (cont'd)



Tree edges

Cross edges

Back edges

There are no forward edges

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.
- Let us **number the vertices** of the digraph in the order in which we first mark them as visited during a depth first search. For this we can use an array `dfnumber` and add the code

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

in the function `Traverse`, immediately after the statement marking a vertex as visited.

- Let us call this the **depth-first numbering (πρώτα κατά βάθος αρίθμηση)** of a digraph.

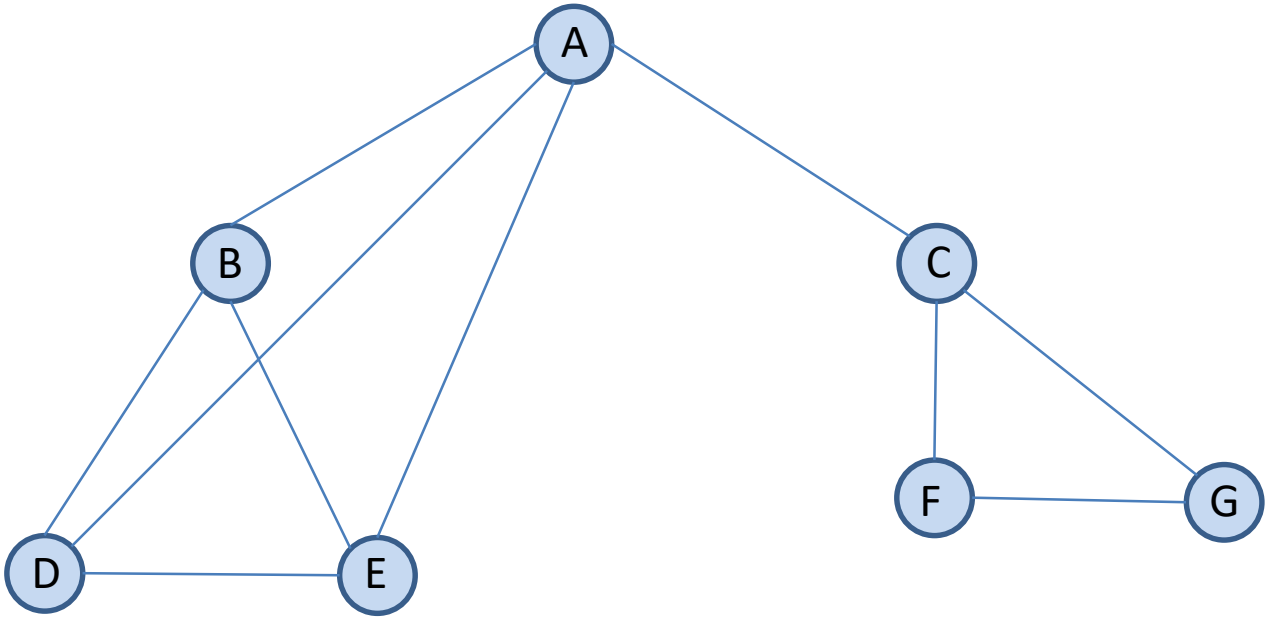
Classification of Edges (cont'd)

- All descendants of a vertex v are assigned depth-first search numbers greater than or equal to the number assigned to v. In fact, w is a **descendant** of v if and only if $dfnumber(v) \leq dfnumber(w) \leq dfnumber(v) + number\ of\ descendants\ of\ v$.
- Thus, we have the following:
  - **Forward edges** go from low-numbered vertices to high-numbered vertices.
  - **Back edges** go from high-numbered vertices to low-numbered vertices.
  - **Cross edges** go from high-numbered vertices to low-numbered vertices.

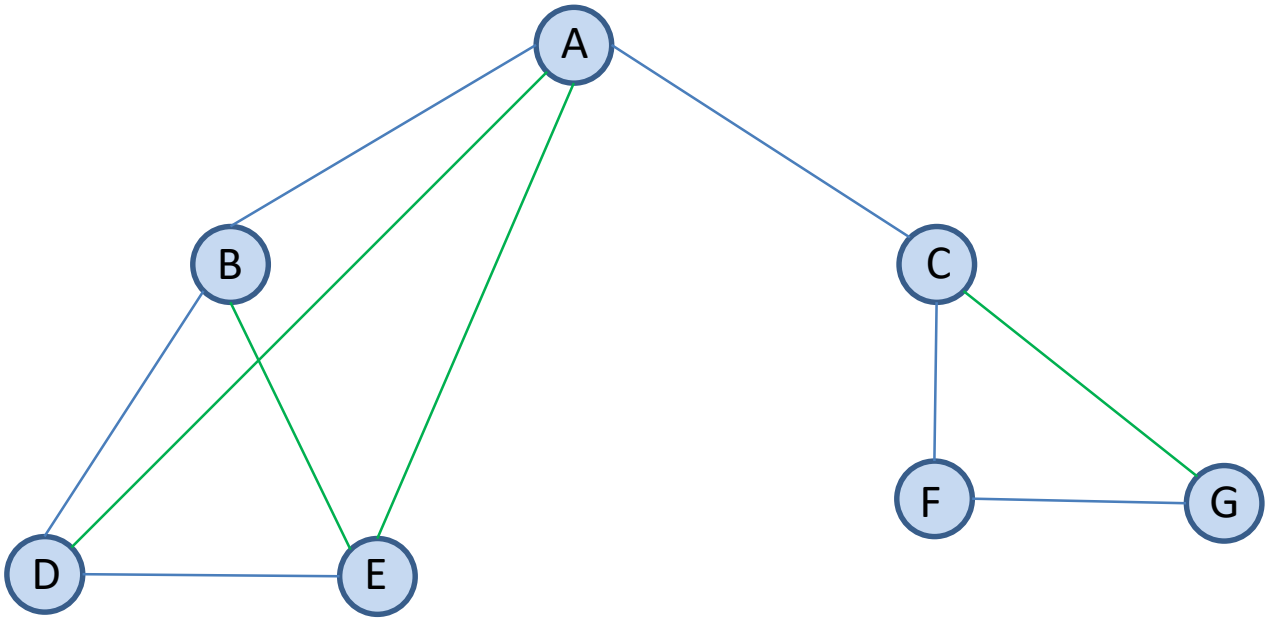
DFS of an Undirected Graph

- During a depth-first search of an undirected graph G, all edges become either **tree edges** or **back edges**.
- 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



Example (cont'd)



Tree edges

Back edges

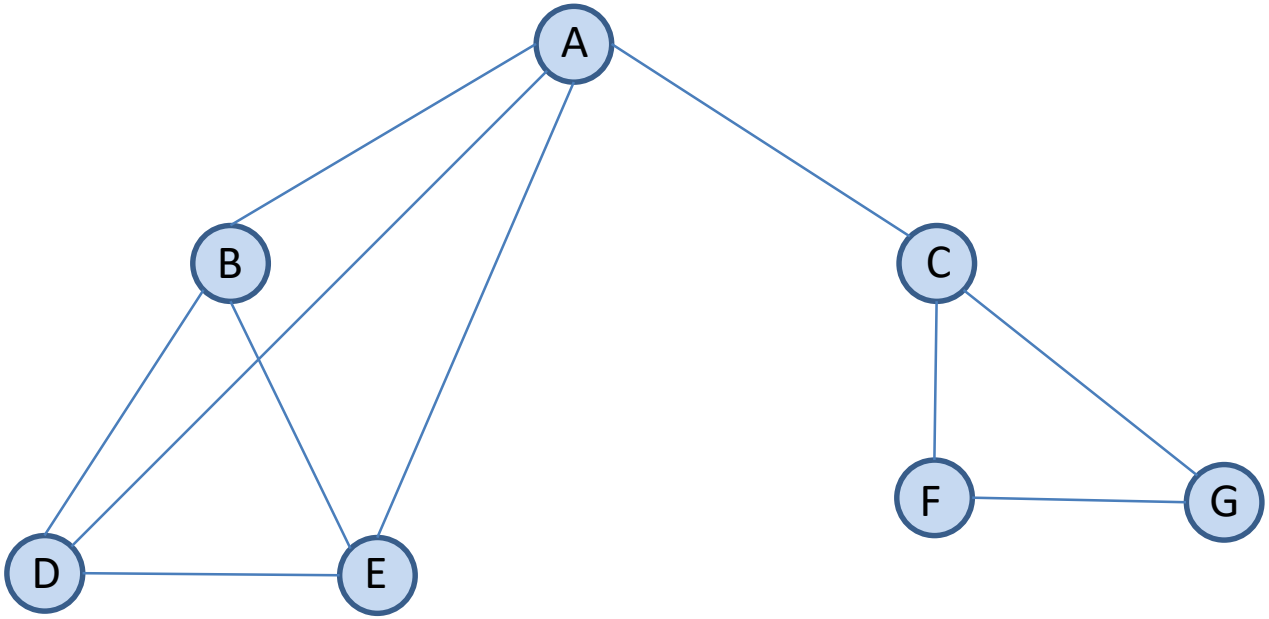
DFS of an Undirected Graph

- During a DFS of an undirected graph G, tree edges form a **depth-first spanning forest** of G.

BFS of an Undirected Graph

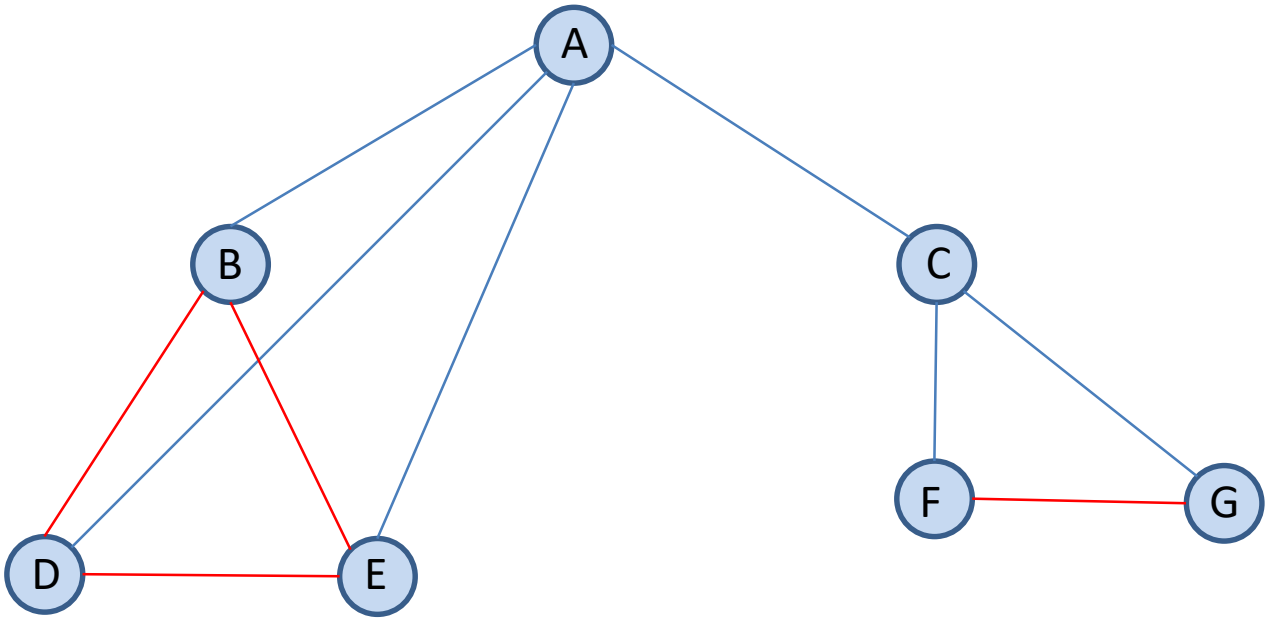
- We can build a **spanning forest** when we perform a breadth-first search as well. We call this the **breadth-first spanning forest** of the graph.
- We consider edge (v, w) to be a **tree edge** if vertex w is first visited from vertex v in the inner while loop of function `BreadthFirst`.
- Every 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.

Example



Let us execute BFS with start node A.

Example (cont'd)



Tree edges

Cross edges

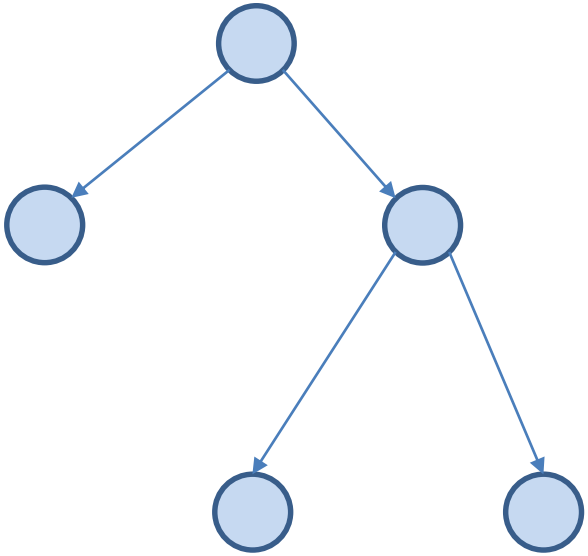
BFS of Directed Graphs

- 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 **breadth-first spanning forest.**
- Non-tree edges are of two kinds: **back edges** and **cross edges.**
- Back edges connect a vertex to one of its ancestors. Cross edges connect a vertex to another vertex that is neither its ancestor nor its descendant.
- There are no forward edges.

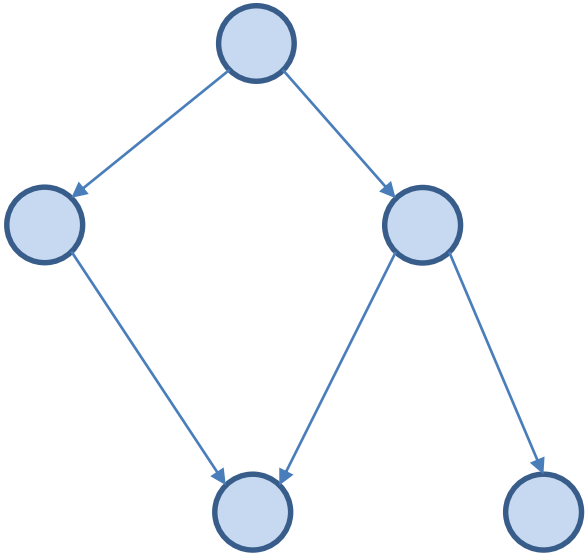
Directed Acyclic Graphs

- 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.

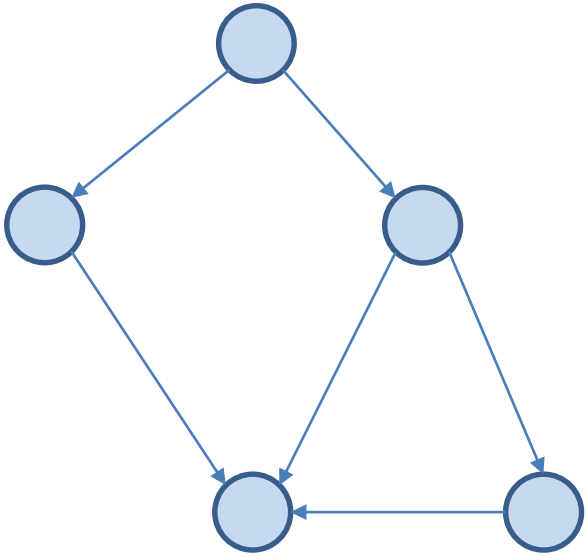
Example Tree



Example Dag



Example Dag



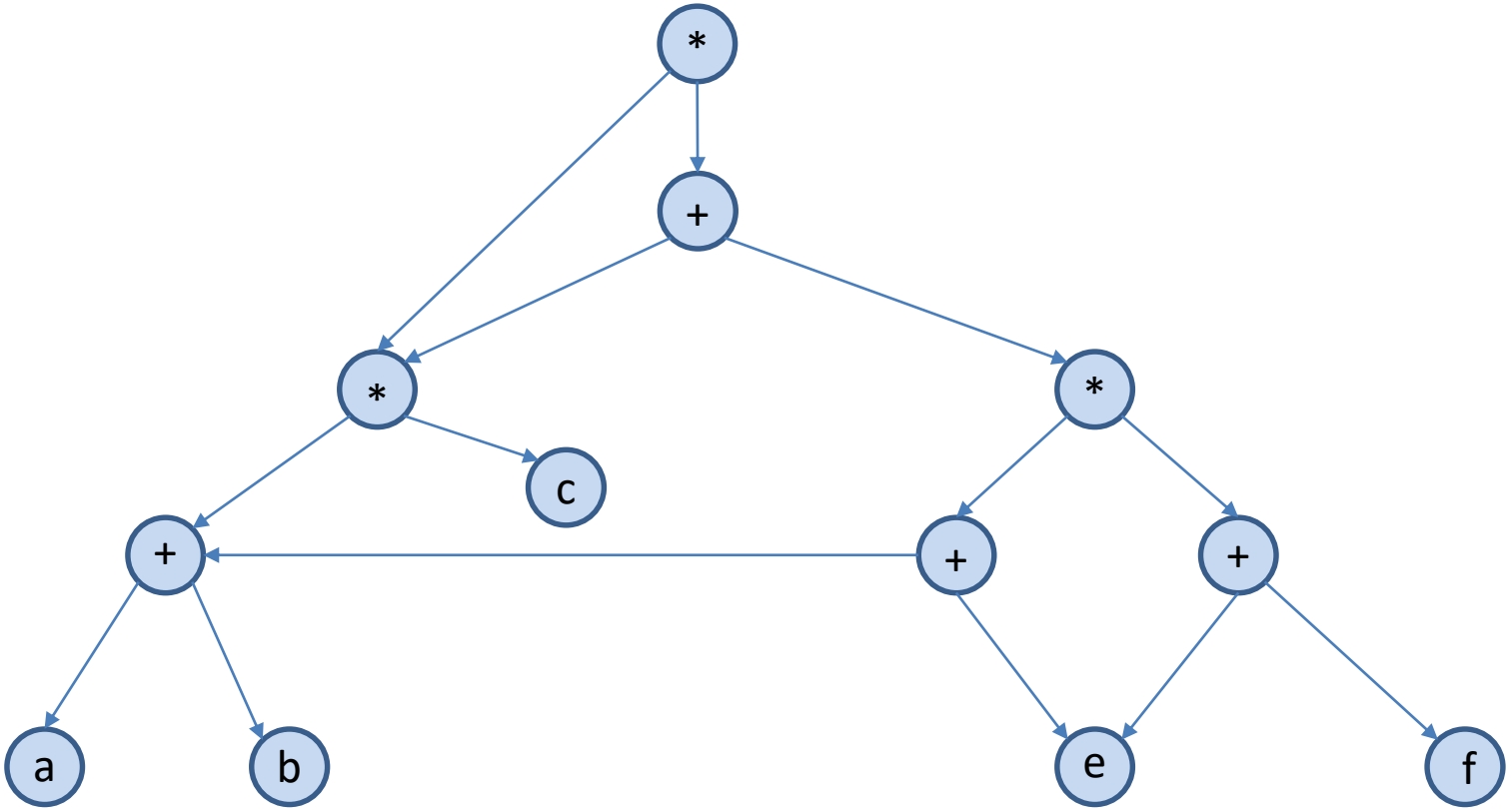
Applications of Dags

- Dags are useful in 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



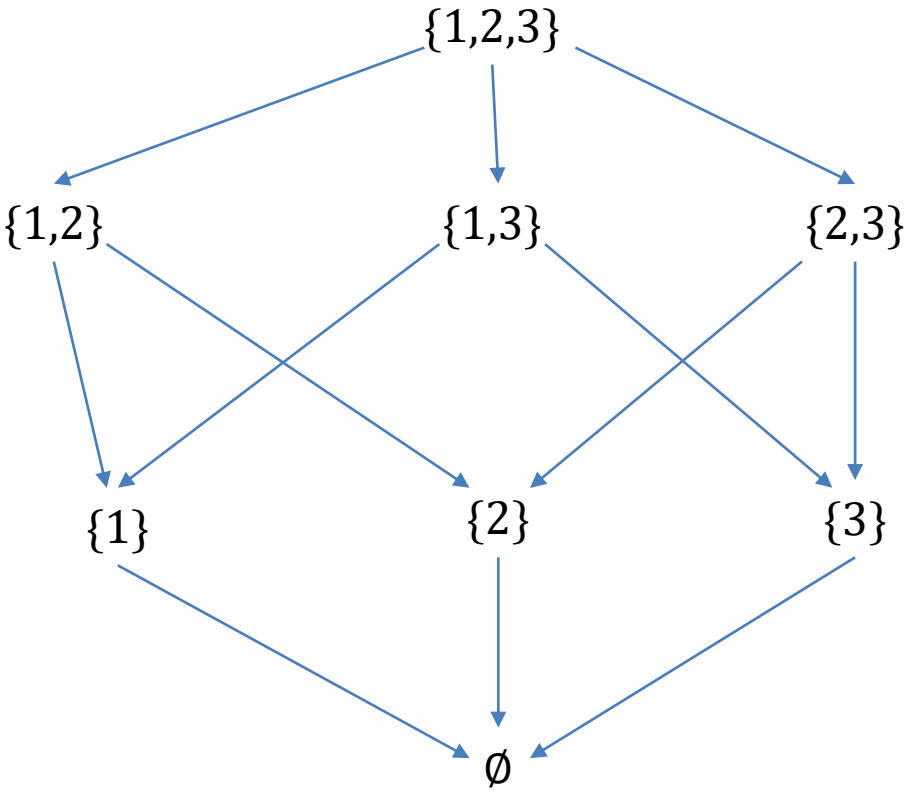
Applications of Dags

- 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 S.

The Dag of the Example



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 DFS 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. Thus, (u, v) cannot be a cross edge. Since the DFS number of u is greater than the DFS number of v, (u, v) cannot be a tree edge or a forward edge. Hence, (u, v) is a back edge.

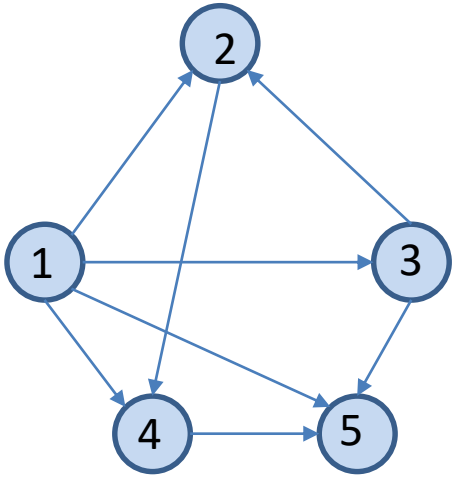
Topological Ordering of a DAG

- 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.

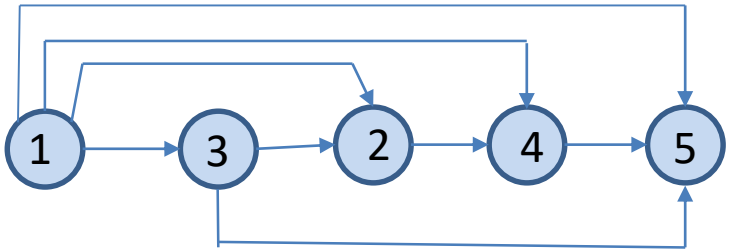
Example

- 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

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 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 */

  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);
    }
  }
}
```

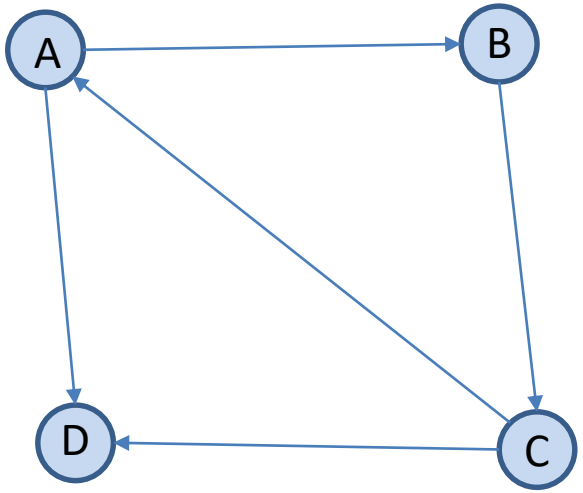
Complexity of Topological Sort

- The complexity of topological sort is again $\boldsymbol{O(n+e)}$.

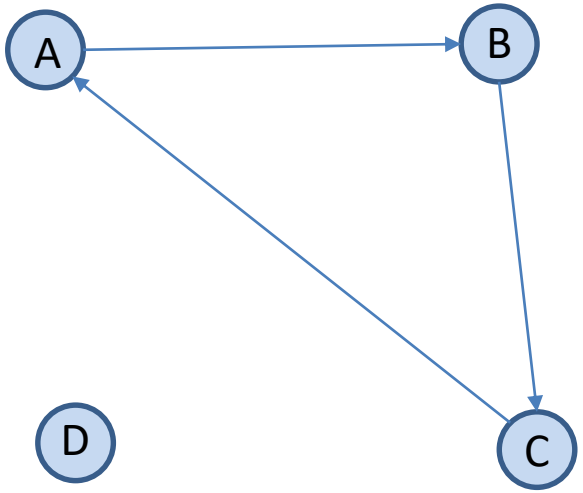
Strongly Connected Components

- 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 said to be **strongly connected (ισχυρά συνδεδεμένος).**

Example Directed Graph



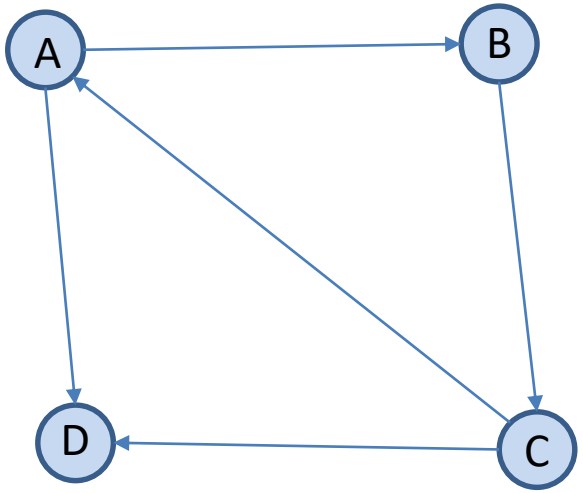
The Strong Components of the Digraph



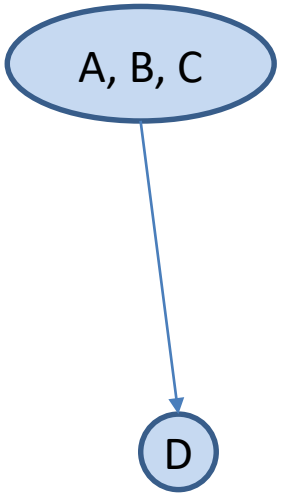
Strong Components (cont'd)

- 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. 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



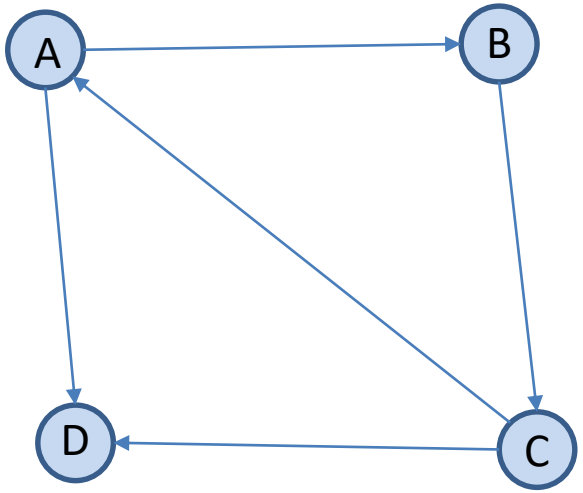
Example Reduced Graph for G



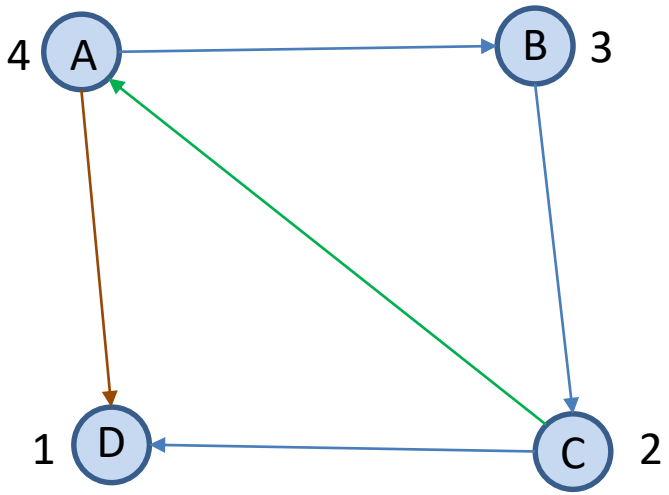
Algorithm for Computing Strong Components

- 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.
  2. Construct 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 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 spanning forest is a strongly connected component of G.

Example Directed Graph G



After Step 1

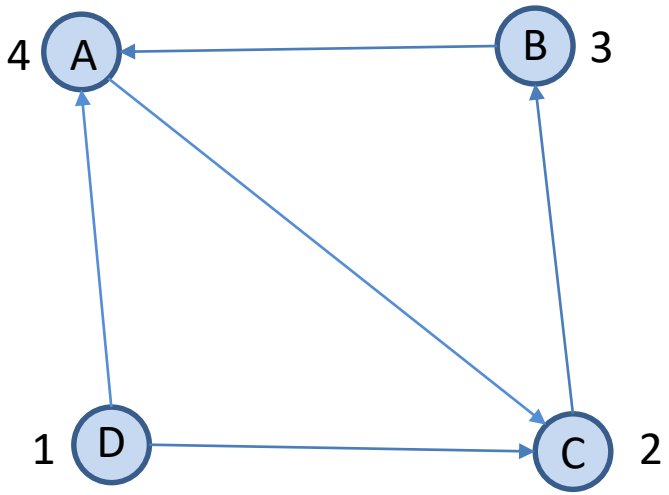


Tree edges

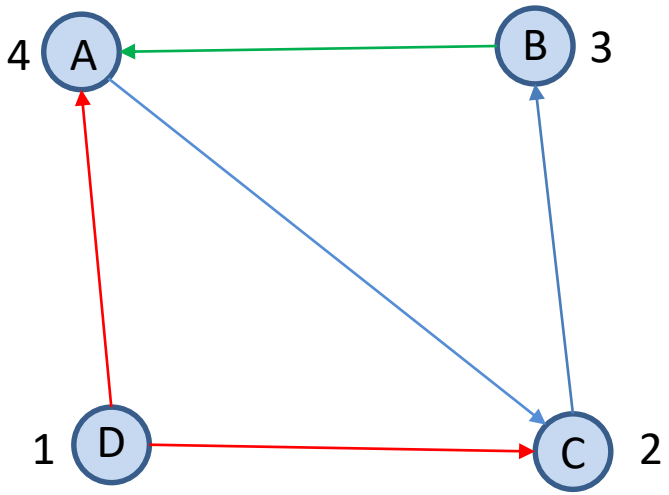
Forward edges

Back edges

The DAG G_r



Depth-first Spanning Forest for G_r

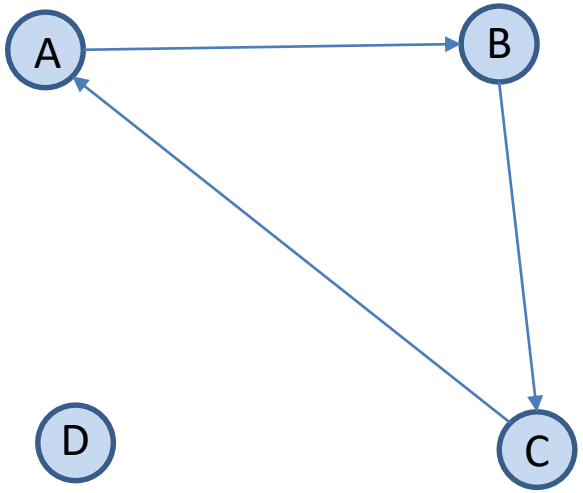


Tree edges

Cross edges

Back edges

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 $\boldsymbol{O(n+e)}$.

Readings

- T. A. Standish. *Data Structures , Algorithms and Software Principles in C*.
  - Chapter 10
- R. Kruse and C.L. Tondo and B. Leung. Data Structures and Program Design in C. 2nd edition.
  - Chapter 11
- A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.*
  - Chapters 6 and 7
- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++.* 2nd edition.
  - Chapter 13