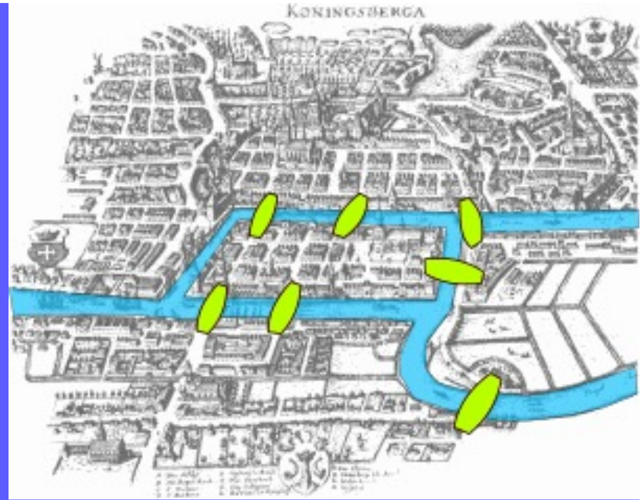
Algorithms

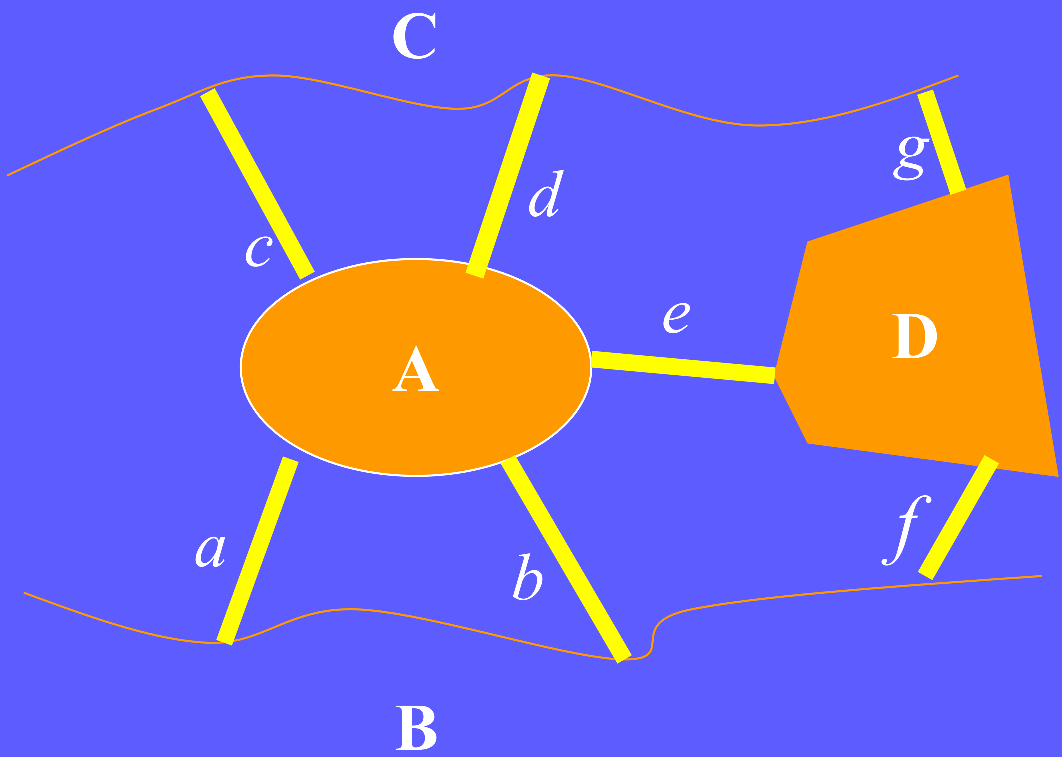
Graph-1

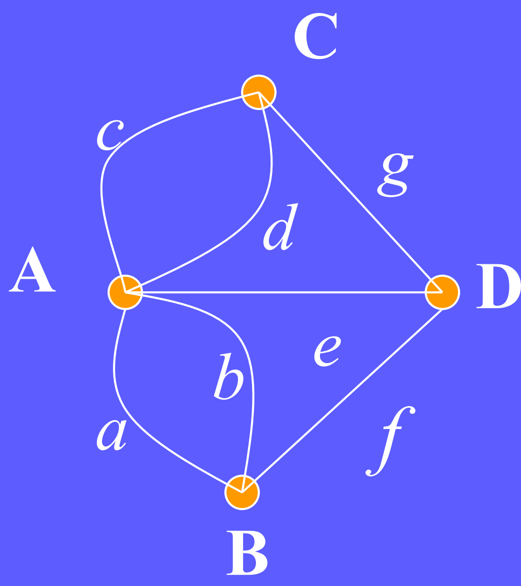
Introduction

Graph Theory



- **1736, Euler's Koenigsberg bridge problem**



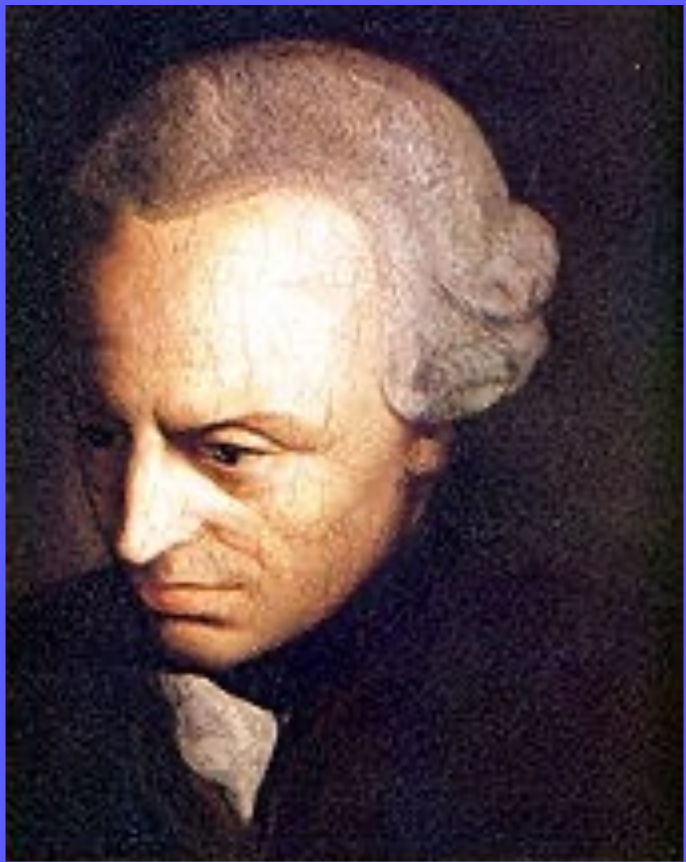


Leonhard Euler

Read Euler, read Euler, he is the master of us all

He ceased to calculate and to live

Immanuel Kant

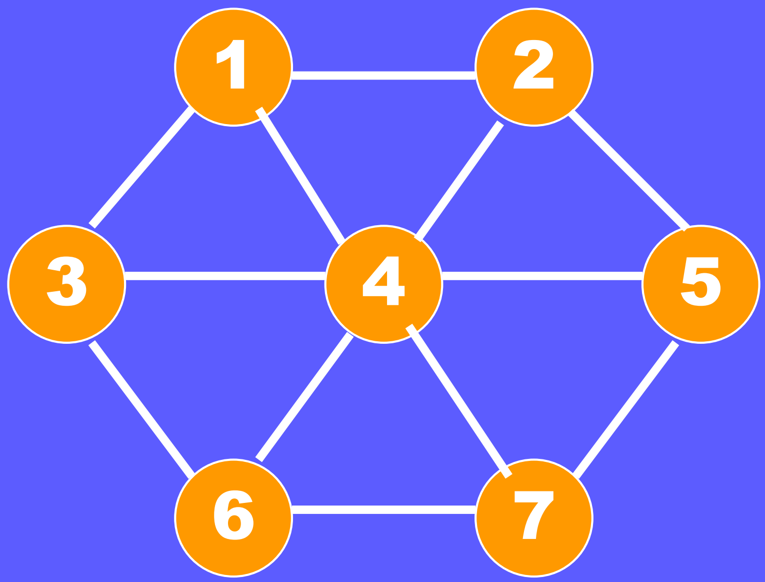


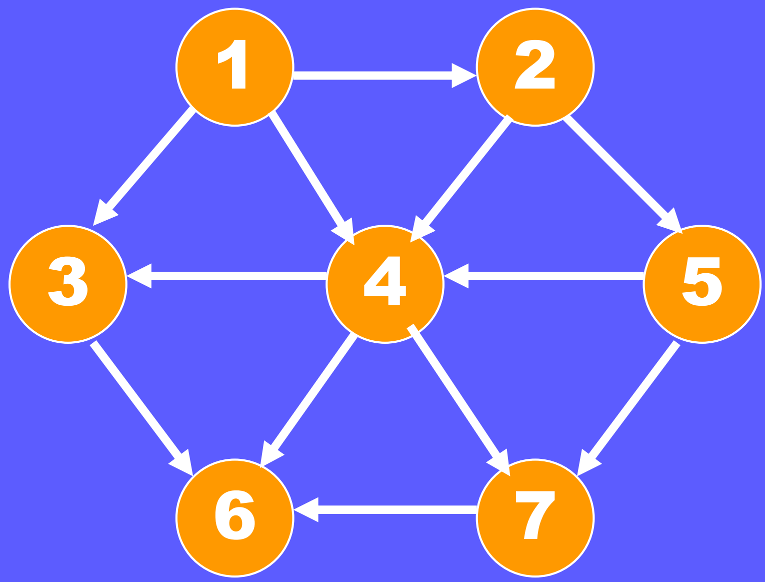
Two things fill the mind with ever new and increasing admiration and awe, the more often and steadily we reflect upon them: The starry heavens above me and the moral law within me.

Terminology of Graph

- **Graph: $G=(V, E)$, V: a set of vertices, E: a set of edges**
- **Edge (arc): a pair (v,w), where $v, w \in V$**
- **Adjacent: w is adjacent to v if $(v, w) \in E$**
- **Directed graph (Digraph):**
  graph if pairs are ordered (directed edge)
- **Undirected graph: if $(v,w) \in E$, $(v,w)=(w,v)$**

Undirected vs. Directed

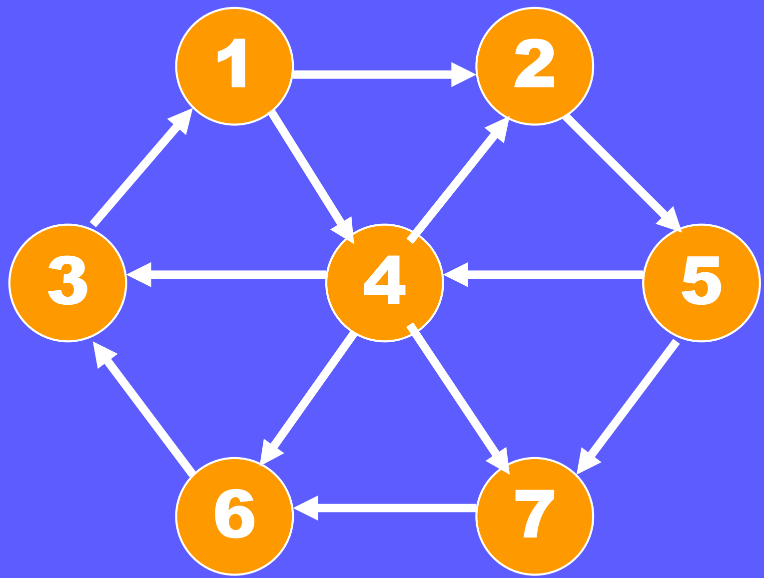




Terminology of Graph (Cont.)

- **Path: a sequence of vertices $w_1, w_2, w_3, \ldots, w_N$ where $(w_i, w_{i+1}) \in E, \forall\ 1 \le i \le N$.**
- **Length of a path: number of edges on the path.**
- **Loop: an edge (v, v) from vertex to itself**
- **Simple path: a path where all vertices are distinct except the first and last.**
- **Cycle in a directed graph: a path such that $w_1 = w_N$**
- **Acyclic graph (DAG): a directed graph which has no cycles.**

Simple Path



$1 \to 4 \to 3 \to 1$: simple path

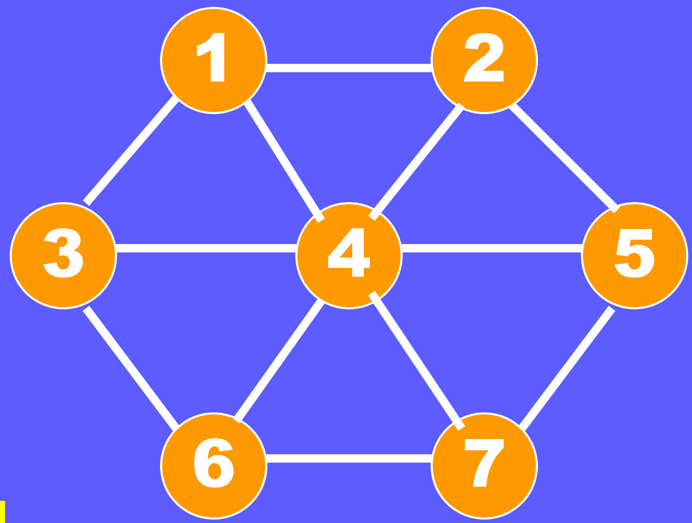
$1 \to 4 \to 2 \to 5 \to 4 \to 3 \to 1$: Non-simple path

Terminology of Graph (Cont.)

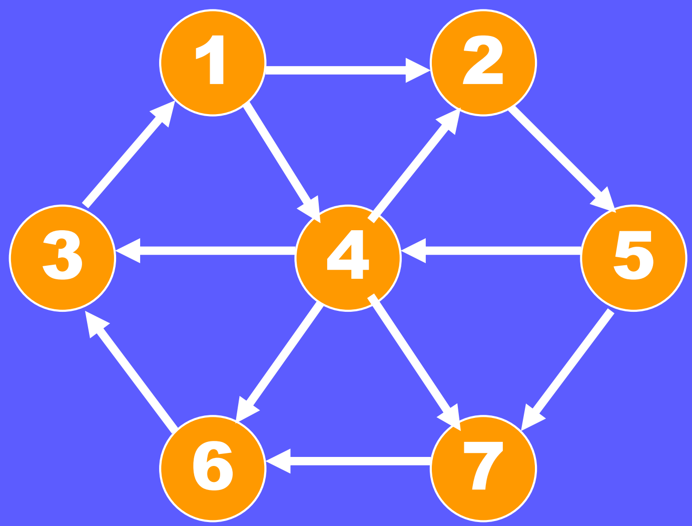
- **Connected: an undirected graph if there is a path from every vertex to every vertex.**
- **Strongly connected: a directed graph if there is a path from every vertex to every vertex.**
- **Weakly connected: a directed graph which is not strongly connected, but the underlying graph is connected.**
- **Complete graph: a graph in which there is an edge between every pair of vertices.**

Connected Graph

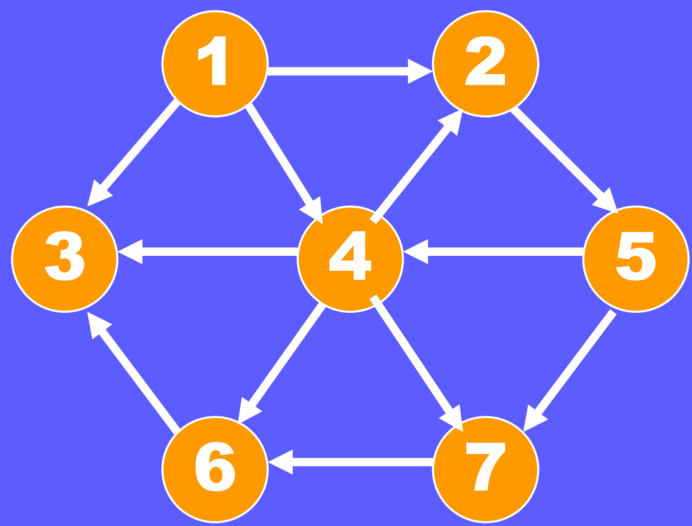
connected



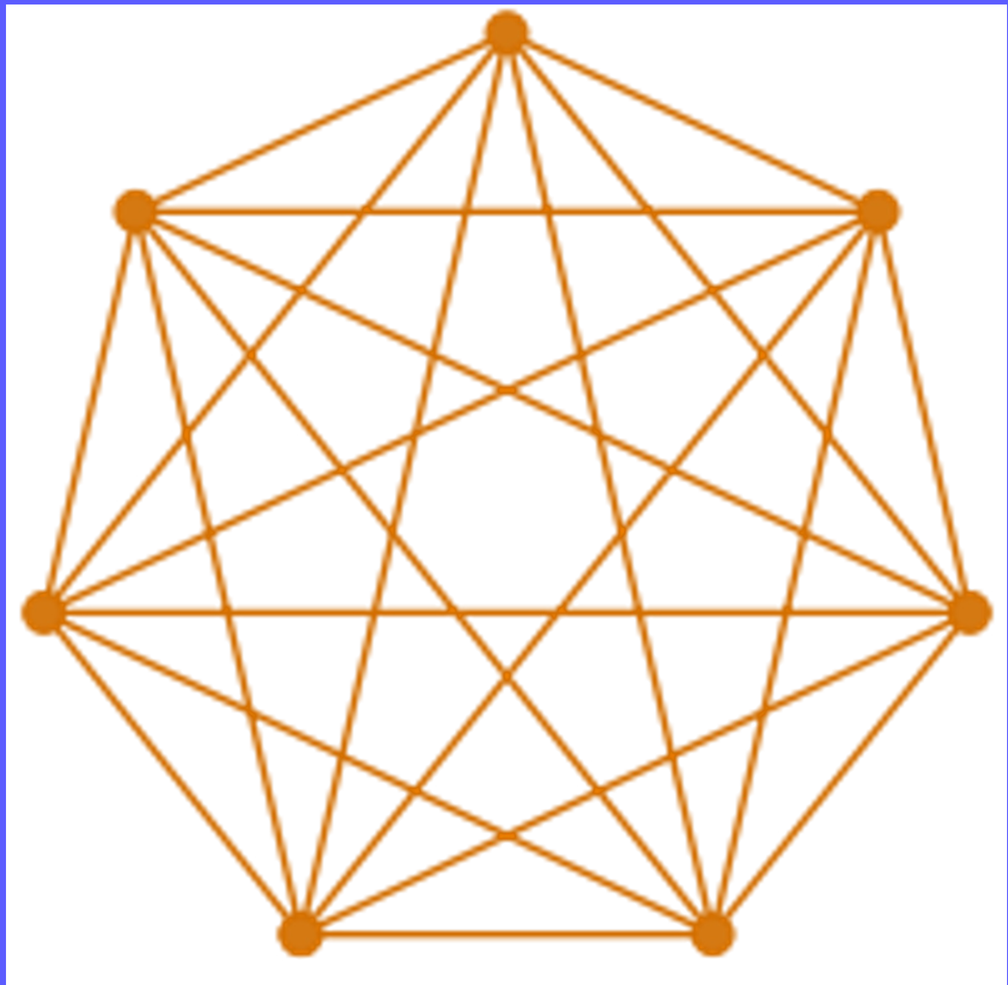
strongly connected



weakly connected



Complete Graph

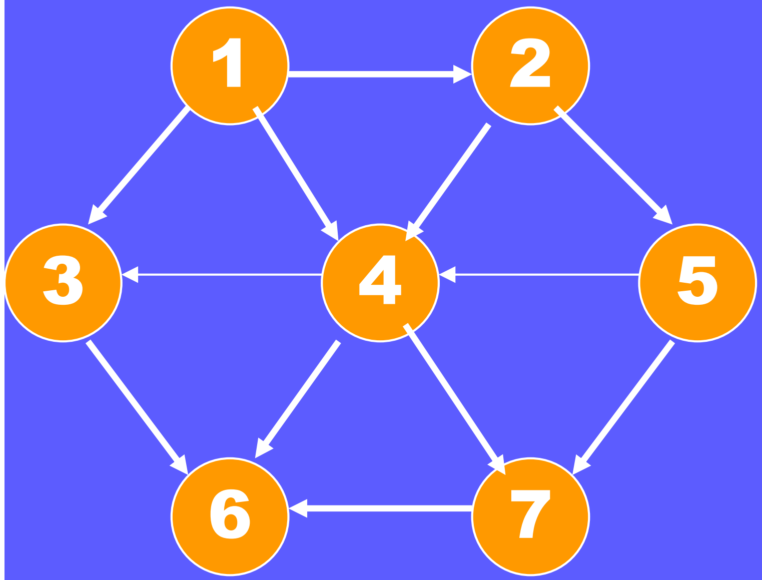


(ref: https://en.wikipedia.org/wiki/Complete_graph)

Representation of Graphs

- **Data structures for representation of graphs**
  - **adjacency matrix representation**
  - **adjacency list representation**

Adjacency Matrix Representation

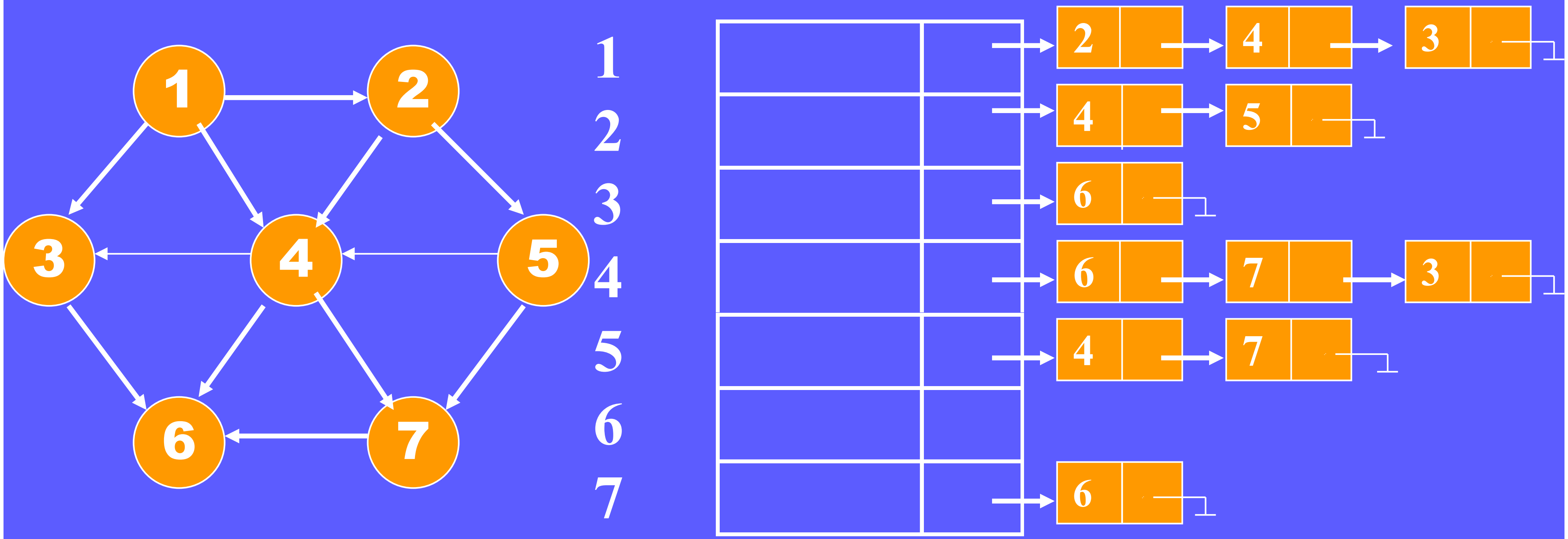


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

■Space: $\Theta(|V|^2)$, good for dense, not for sparse

* Undirected graph: symmetric matrix

Adjacency List Representation



- Space: O(|V|+|E|) good for sparse

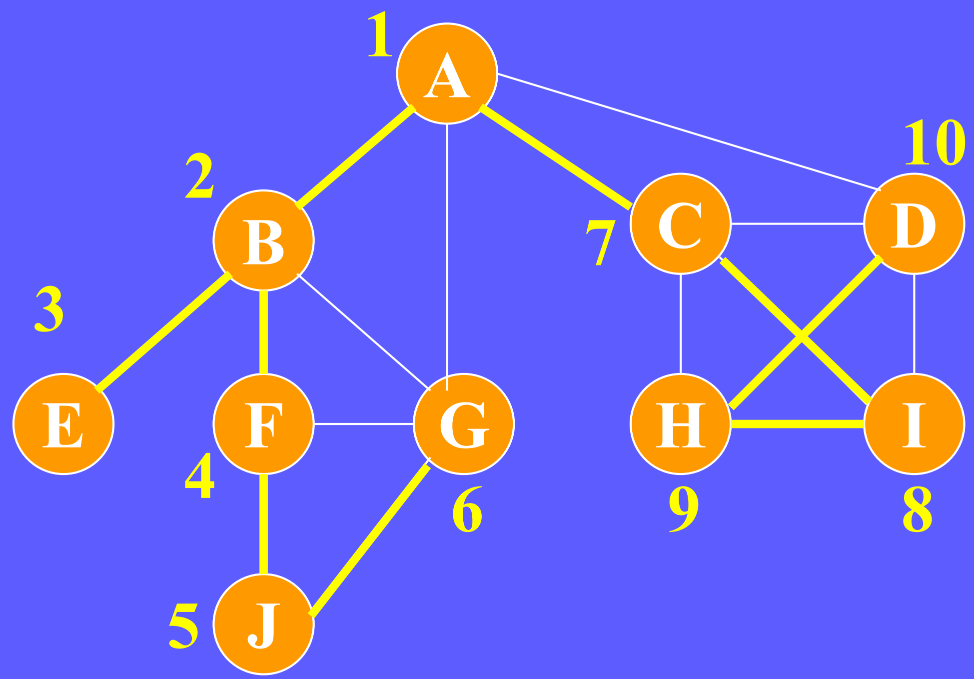
Graph Traversal

(pp. 189~199)

Graph Traversal

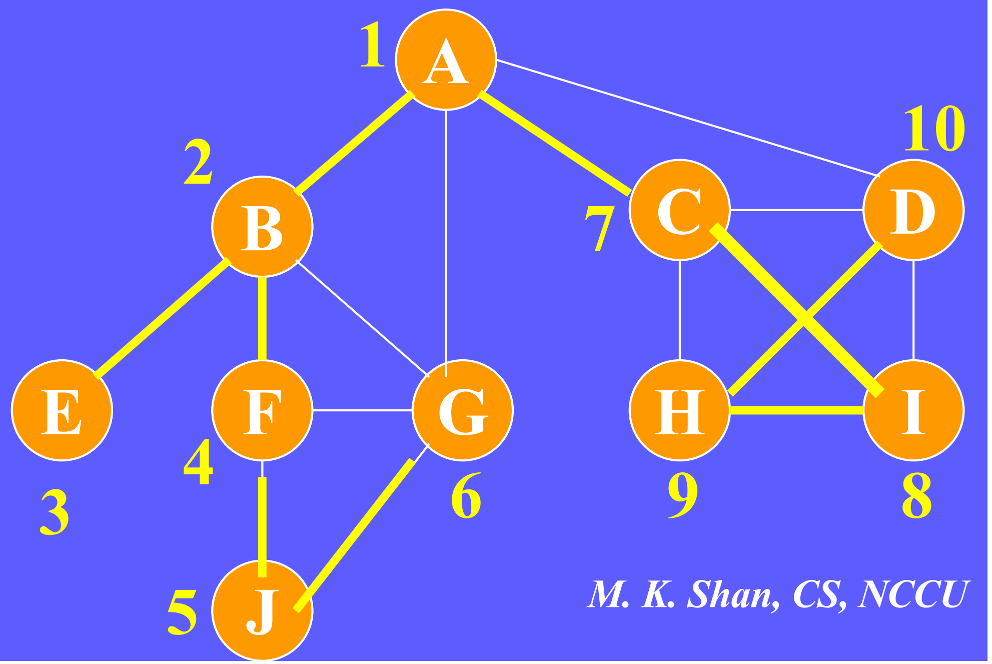
- Traverse: visiting the vertices in graph
- Traversal algorithms
  - Depth-First Search (DFS): 先深後廣
  - Breadth-First Search (BFS): 先廣後深

Depth-First Search



Depth-First Search

```
Algorithm DFS(G,v);
Begin
    mark v;
    for all edges (v,w) do
        if w is unmarked then DFS(w)
End
```



Lemma 7.1

- **If G is connected**

  Then

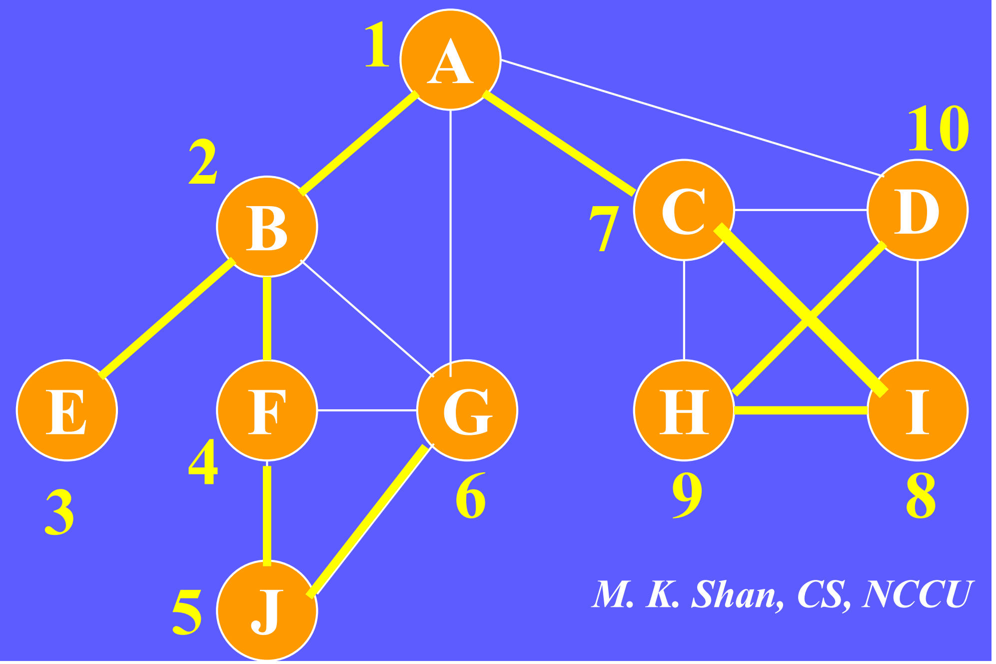
  (1) all its vertices will be marked by algorithm DFS

  (2) all its edges will be looked at least once during the execution of algorithm DFS

Generalized Depth-First Search

Algorithm DFS(G,v);
Begin
    mark v;
    <u>prework</u>(v)
    for all edges (v,w) do
        if w is unmarked then DFS(w)
        <u>postwork</u>(v,w)
End

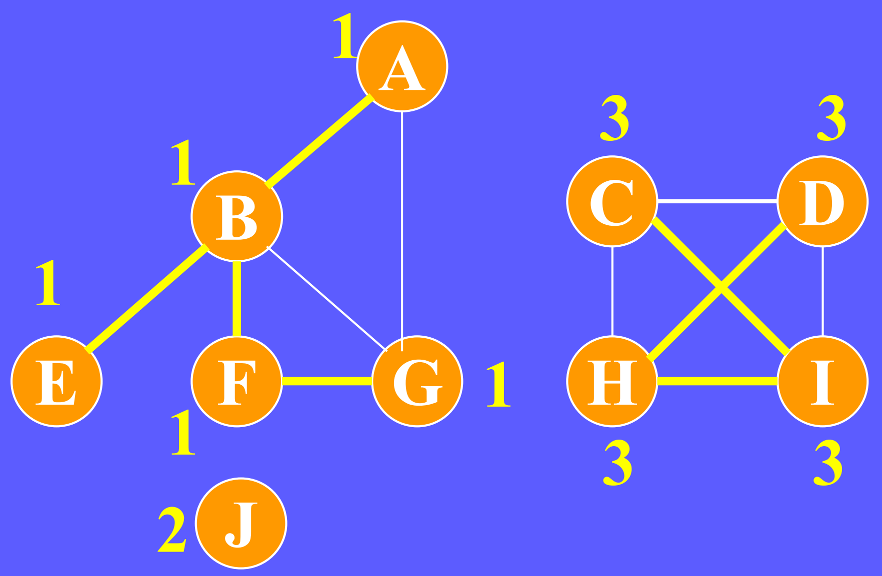
*** prework: mark time**
*** postwork**
    (1) backtrack
    (2) w is a marked vertex



M. K. Shan, CS, NCCU

Finding Connected Components

Algorithm Connected_Components (G)
Input: G(=(V,E)
Output: assignment of component number
Begin
    component_no:=1;
    while there is an unmarked vertex v do
        DFS(G, v); {prework v.component:=component_no}
        component_no:=component_no+1;
End



Algorithm DFS(G,v);
Begin
    mark v;
    v.component:=component_no;
    for all edges (v,w) do
        if w is unmarked then DFS(w)
End

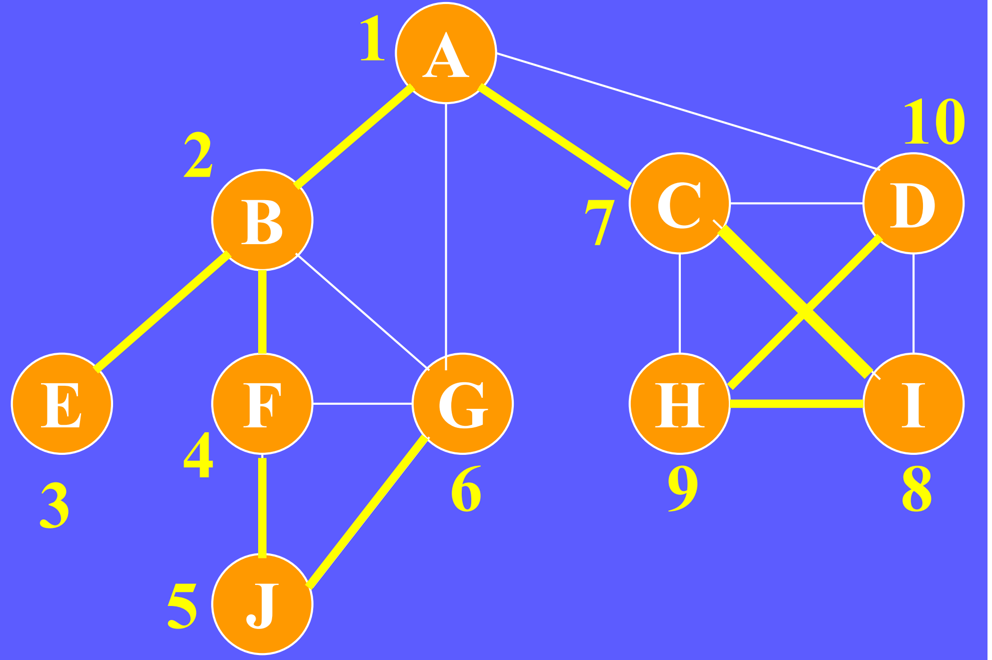
M. K. Shan, CS, NCCU

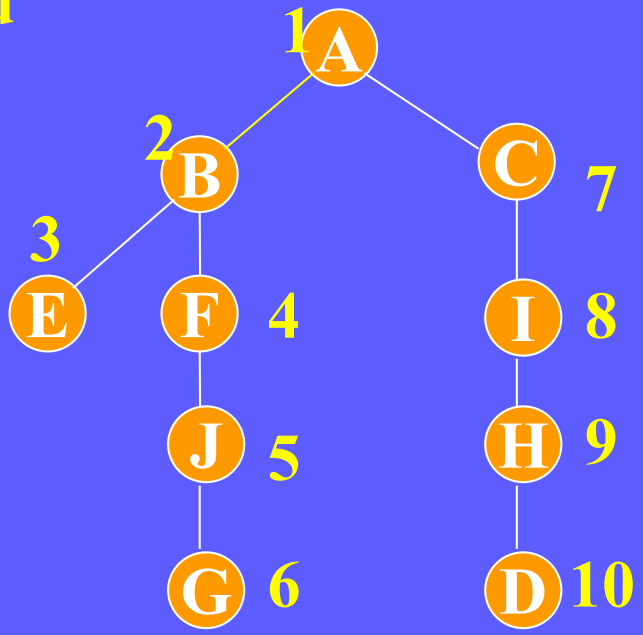
DFS Numbering

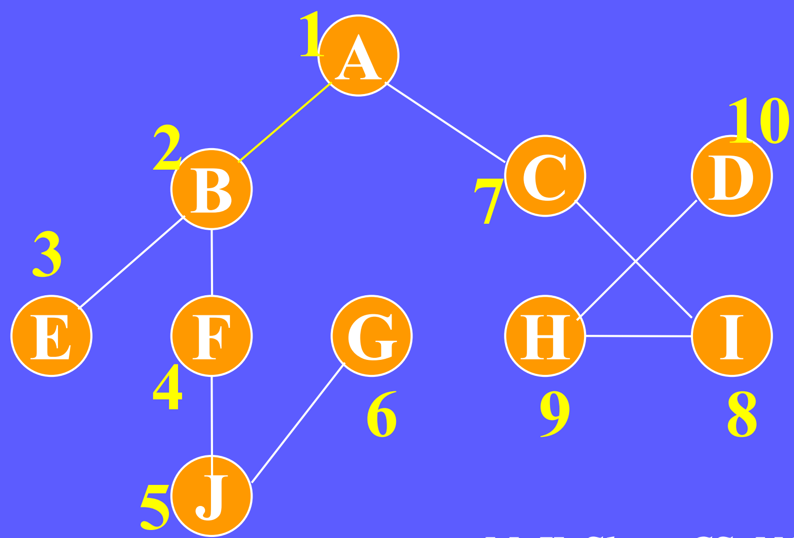
```
Algorithm DFS_Numbering (G, v)
Begin
   Initialize DFS_Number := 1;
   DFS(G, v)
End

Algorithm DFS(G,v);
Begin
    mark v;
    v.DFS := DFS_Number;
    DFS_Number := DFS_Number + 1;
    for all edges (v,w) do
        if w is unmarked then DFS(w)
        postwork(v,w)
End
```

Build DFS Tree

```
Algorithm Build_DFS_Tree(G, v);
Begin
    mark v;
    for all edges (v,w) do
        if w is unmarked then
            add edge (v,w) to T
            Build_DFS_ Tree(G, w)
End
```







Lemma 7.2 (for Undirected DFS Trees)

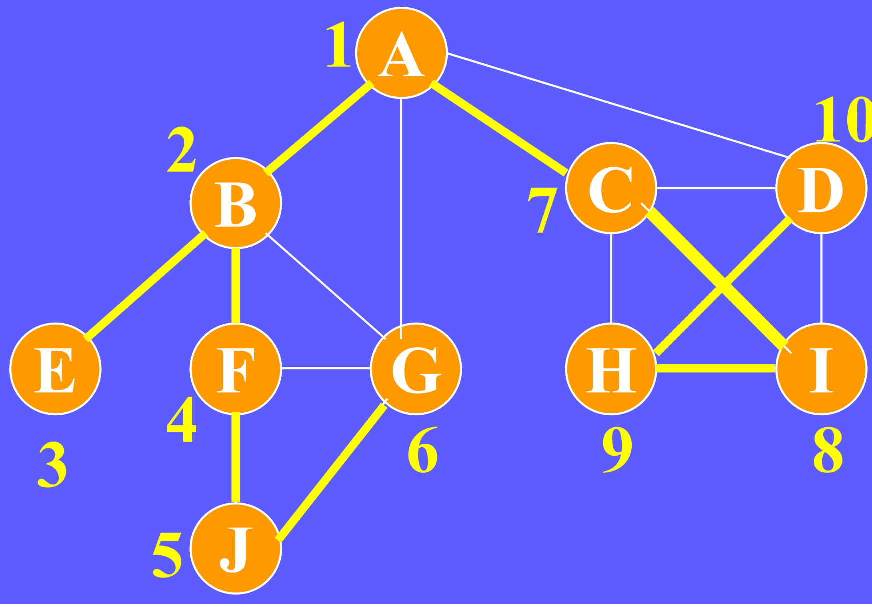
Let G = (V, E) a connected undirected graph

T = (V, F) a DFS tree of G

then every edge e ∈ E

either belongs to T (yellow edges)

or connects two vertices of G, one of which is the ancestor of the other in T. (white edges)



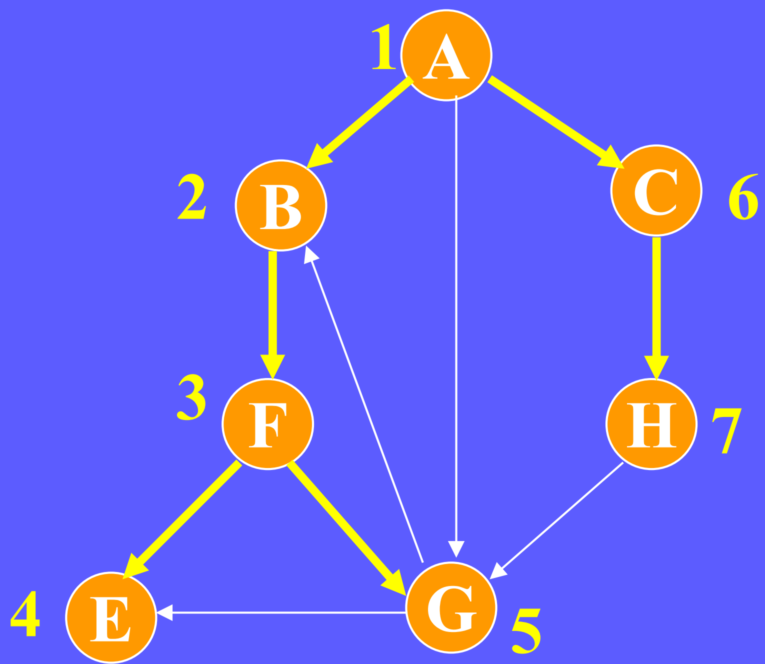
Lemma 7.3 (for directed DFS Trees)

Let G = (V, E) a directed graph

T = (V, F) a DFS tree of G

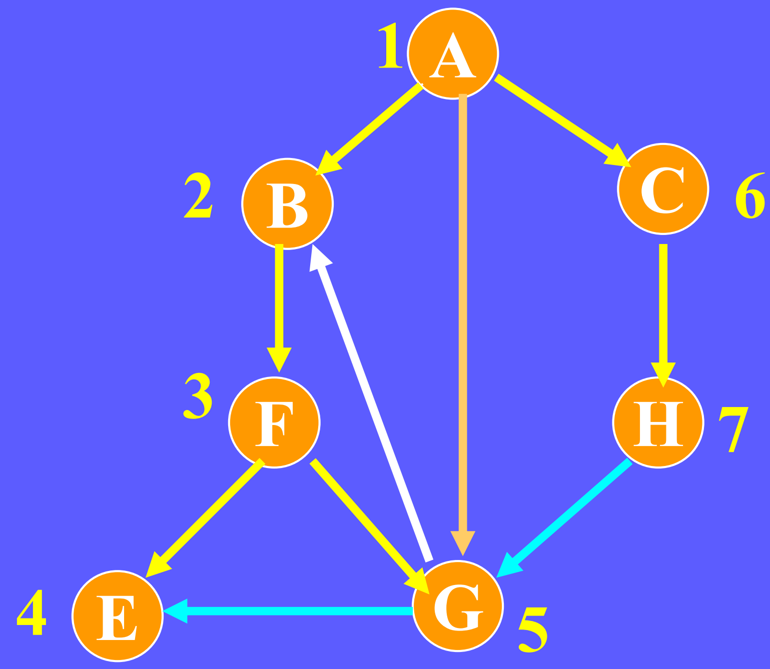
If (v, w) $\in$ E and v.DFS_Number < w.DFS_Number,

then v is the ancestor of w in the tree T



Four Types of Edges

- tree edges
- back edges
- forward edges
- cross edges



* In undirected DFS trees, there exists no cross edges

* In directed DFS trees, cross edge must cross from right to left)

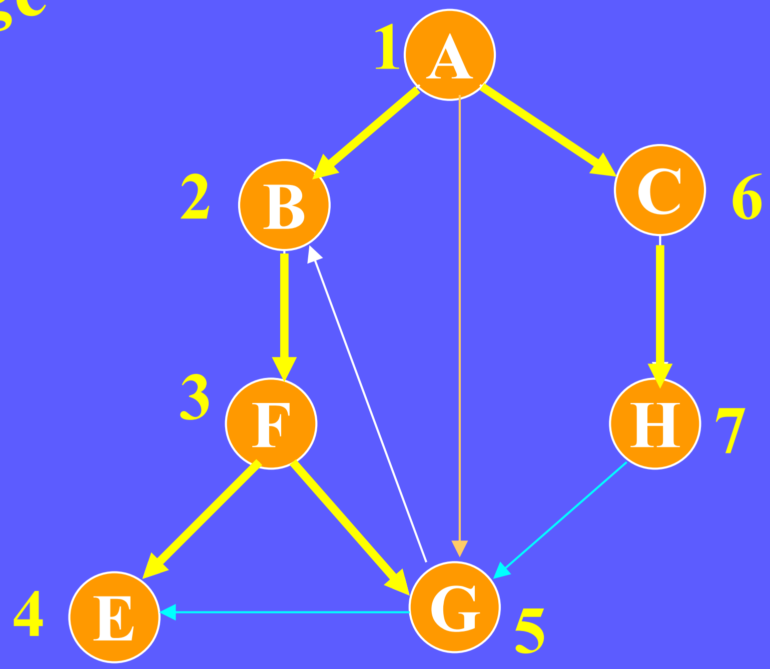
Lemma 7.4

Let G = (V, E) be a directed graph

T be a DFS tree of G

Then G contains a directed cycle

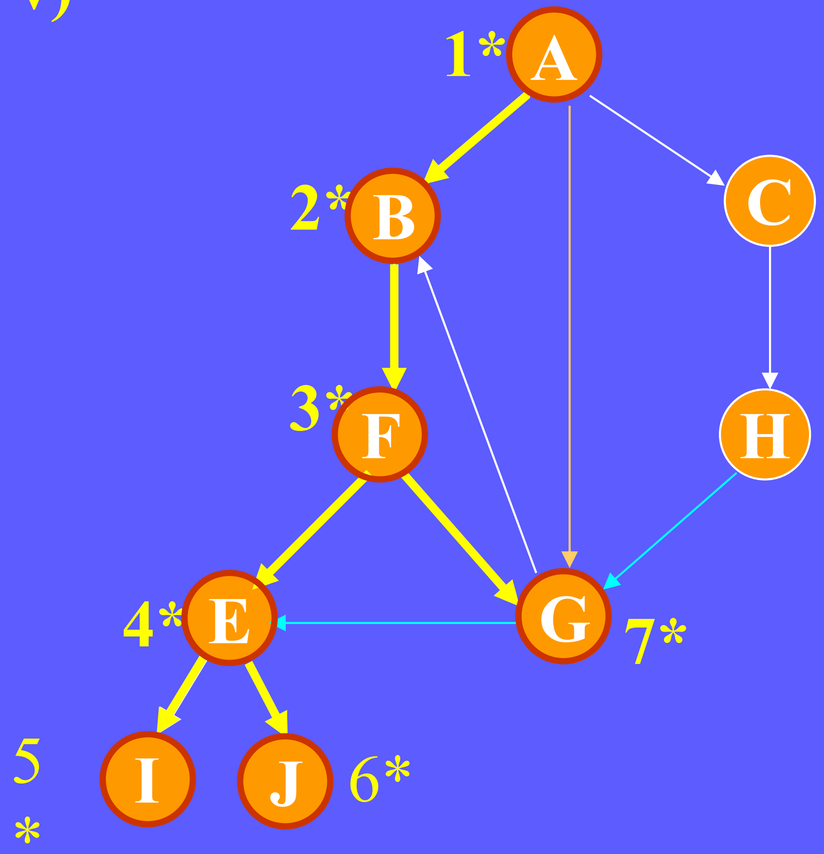
iff G contains a back edge



Find_a_Cycle

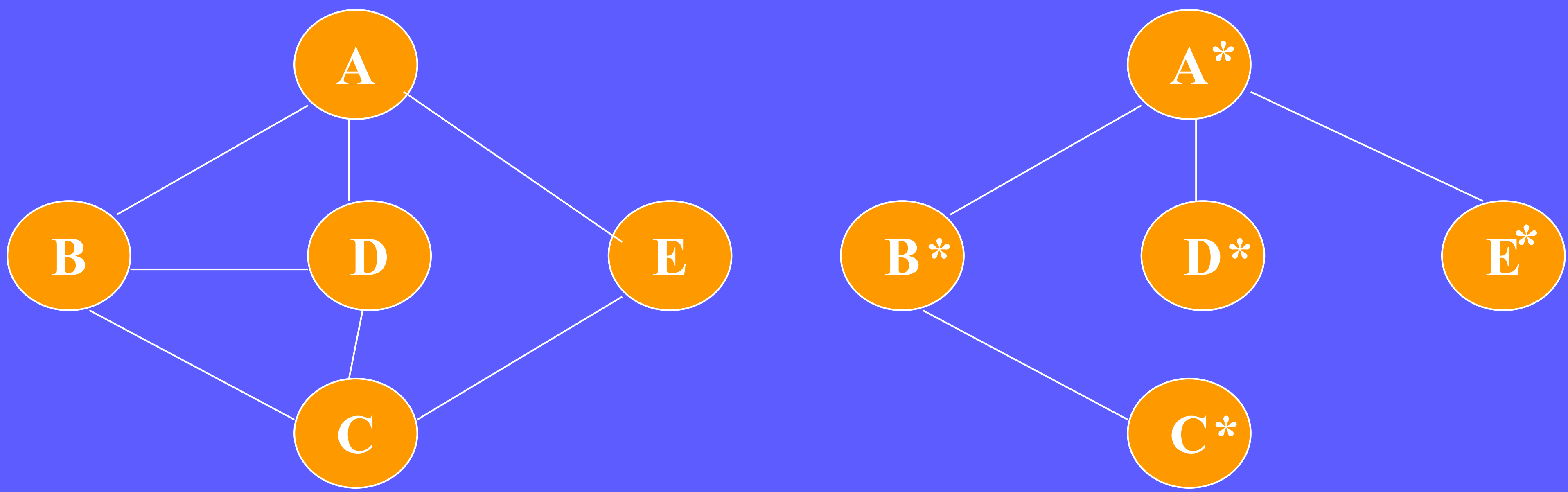
Algorithm Find_a_Cycle(G, v)
Begin
    for each vertex v
        v.on_the_path:=false
    DFS(G, v)
End

Algorithm DFS(G,v);
Begin
    mark v;
    v.on_the_path:=true;
    for all edges (v,w) do
        if w is unmarked then DFS(w)
        if w.on_the_path then Find_a_Cycle:=true; halt;
    v.on_the_path:=false;
End

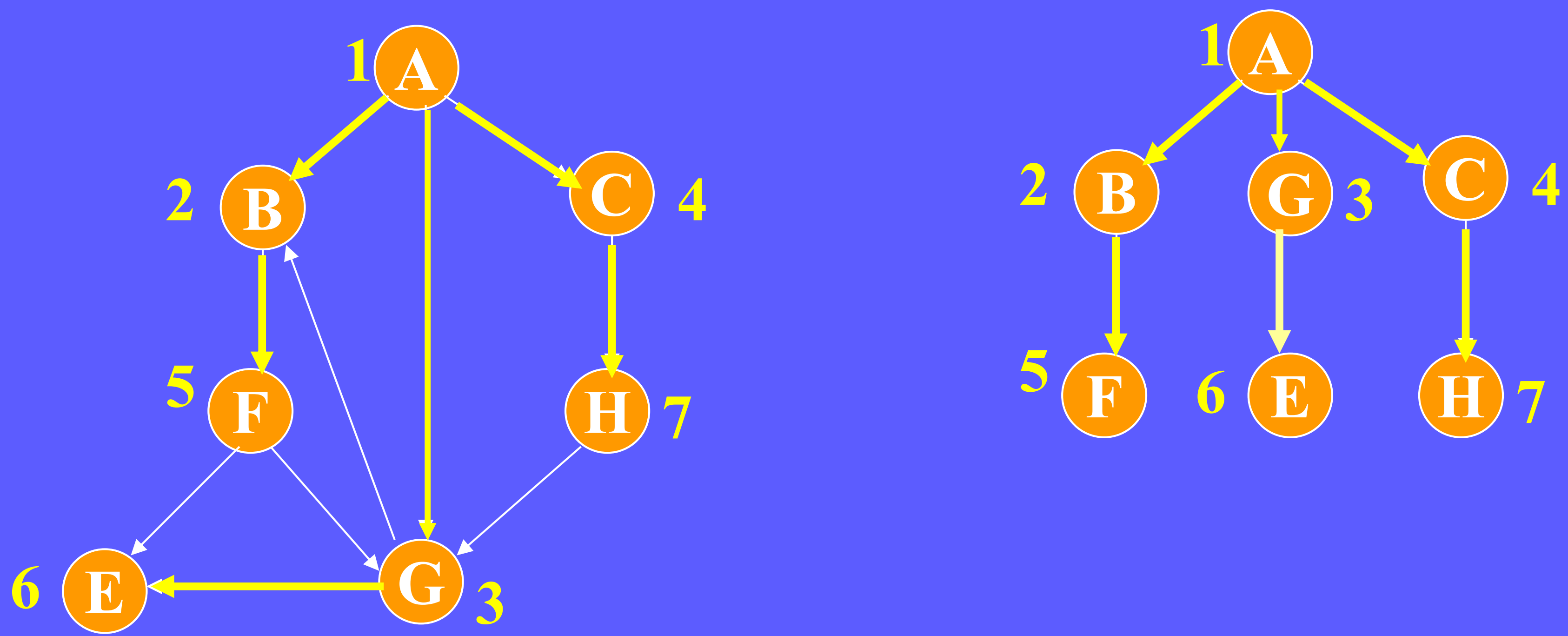


Breadth First Search

- Breadth First search (BFS): level order tree traversal
- BFS algorithm: using queue



Breadth-First Search



Algorithm of BFS

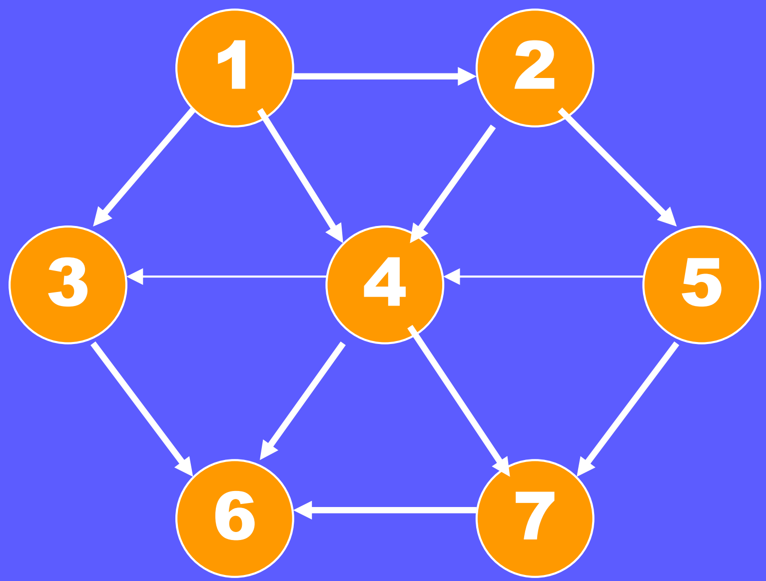
```
Algorithm BFS
Begin
    mark v;
    put v in queue;
    while queue is not empty do
        remove the first vertex w from queue;
        prework on w;
        for all edges (w,x) such that x is unmarked do
            mark x;
            add (w,x) to the tree T;
            put x in queue;
End
```

Topological Sorting

(pp. 199~201)

Topological Sorting

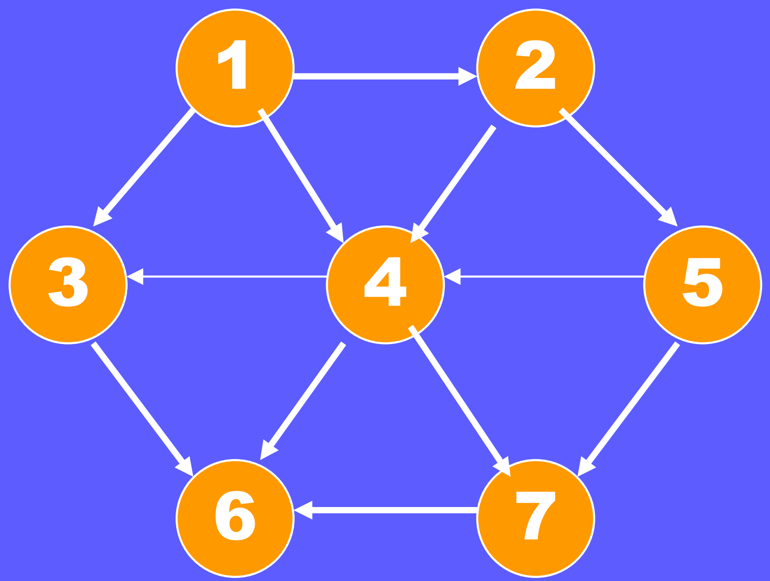
- **Topological sorting:**
  ordering of vertices in a DAG such that
  if there is a path from v_i to v_j,
  then v_j appears after v_i in the ordering.



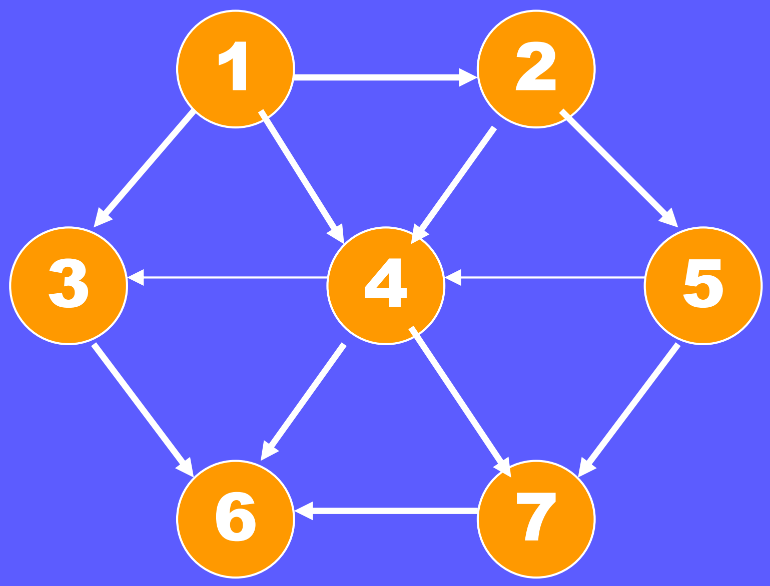
1, 2, 5, 4, 3, 7, 6

Topological Sorting

- if there is a path from v_i to v_j, then v_j appears after v_i in the ordering.
- Prerequisite of Courses
  - 2: {1}, 3: {1, 4}, 4:{1, 2, 5}, 5:{2}, 6:{3, 4, 7}, 7:{4, 5}
  - Ordering of course taking



1, 2, 5, 4, 3, 7, 6

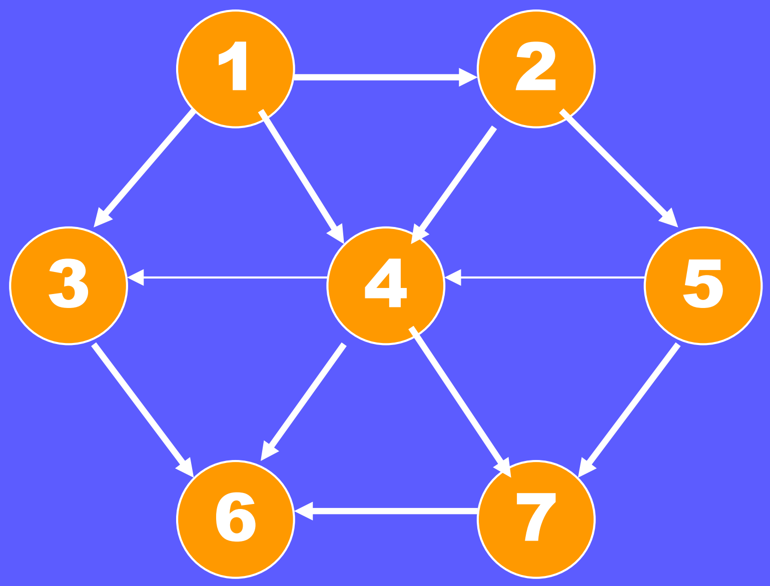


1, 2, 5, 4, 3, 7, 6

	Deg.		Deg.		Deg.		Deg.		Deg.		Deg.		Deg.	
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
2	1--		0	2	0	2	0	2	0	2	0	2	0	2
3	2--		1		1		1--		0	5	0	5	0	5
4	3--		2--		1--		0	4	0	4	0	4	0	4
5	1		1--		0	3	0	3	0	3	0	3	0	3
6	3		3		3		3--		2--		1--		0	7
7	2		2		2--		1--		0		0	6	0	6

Algorithm for Topological Sorting

```
Void Topsort(Graph G)     /* O(|V|^2)  */
{
  int Counter;
  vertex V,W;
  for (Counter=0; Counter < NumVertex; Counter++)
  {
       V=FindNewVertexOfDegreeZero();
       TopNum[V]=Counter;
       For each W adjacent to V
          Indegree[W]--;
  };
};
```



1, 2, 5, 4, 3, 7, 6

	Deg.	
1	0	1
2	1--	
3	2--	
4	3--	
5	1	
6	3	
7	2	

2

	Deg.	
1	0	1
2	0	2
3	1	
4	2--	
5	1--	
6	3	
7	2	

5

	Deg.	
1	0	1
2	0	2
3	1	
4	1--	
5	0	3
6	3	
7	2--	

4

	Deg.	
1	0	1
2	0	2
3	1--	
4	0	4
5	0	3
6	3--	
7	1--	

3
7

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	2--	
7	0	

3
7

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	1--	
7	0	6

6

	Deg.	
1	0	1
2	0	2
3	0	5
4	0	4
5	0	3
6	0	7
7	0	6

Improved Algorithm for Topological Sorting

```
void Topsort(Graph G);    /*   O(|E|+|V|)   */
{   queue Q;
    int Counter=0;
    vertex V,W;
    Q=CreateQueue(NumVertex);    MakeEmpty(Q);
    for each vertex V
        if  (Indegree[V] == 0)
            Enqueue(V,Q);
    While (!IsEmpty(Q)) {
        V=Dequeue(Q);
        TopNum[V] = ++Counter;
        for each W adjacent to V
            if (--Indegree[W] == 0)
                Enqueue(W,Q);       }
    if (Counter != NumVertex)
        Error("Cycle!");
    DisposeQueue(Q);
}
```