Graphs

CS2: Data Structures and Algorithms
Colorado State University

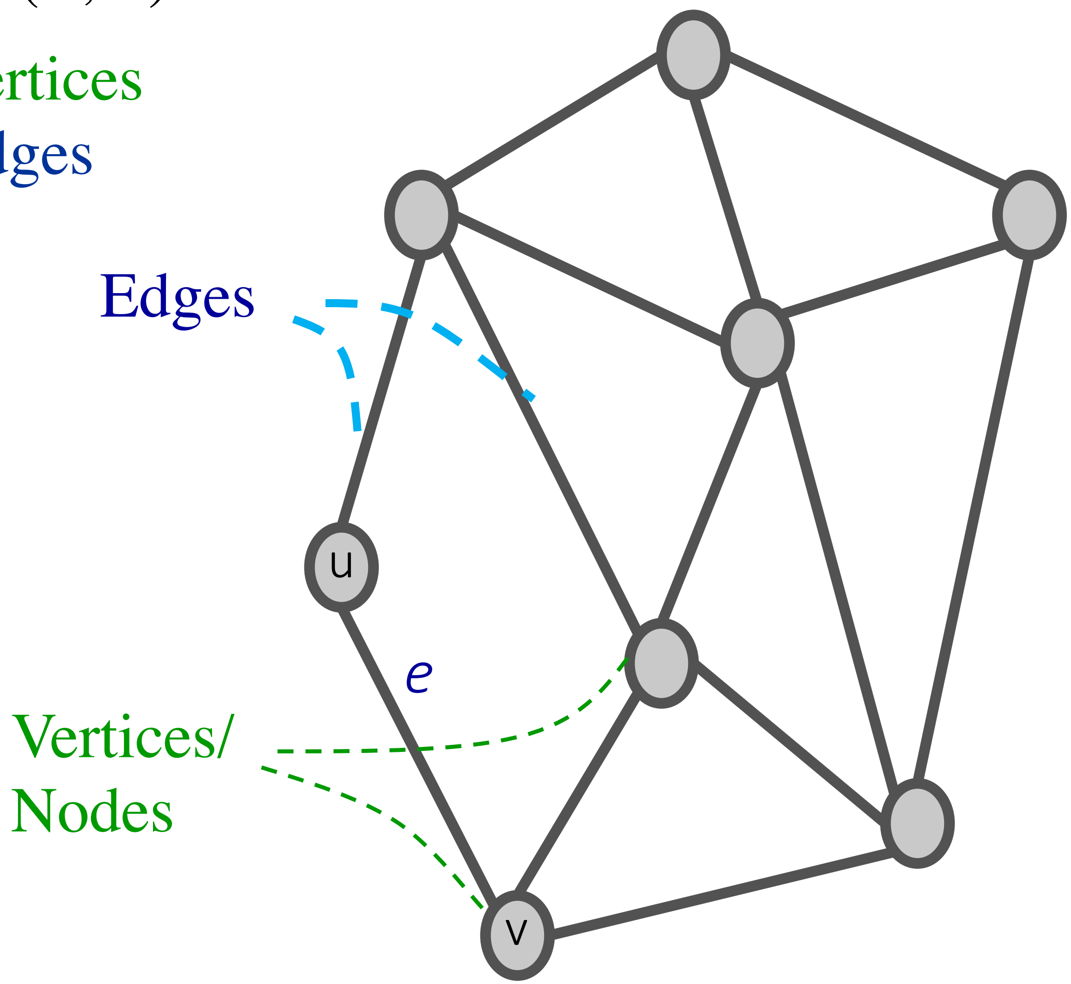
Modified slides by Wim Bohm, Sudipto Ghosh and Russ Wakefield

Graph terminology

G=(V, E)

Vertices

Edges



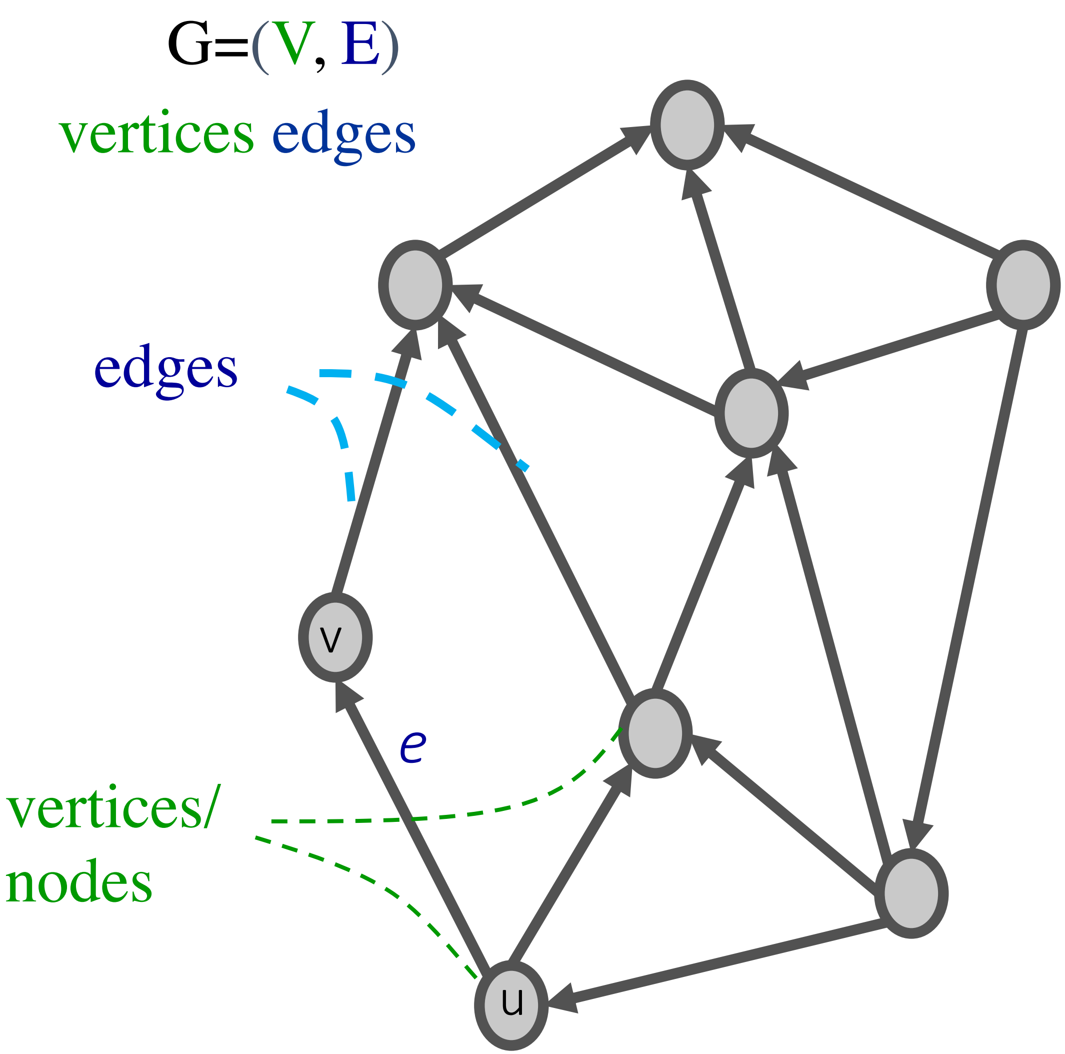
An edge is incident on the two vertices it connects.

Two vertices are adjacent (or neighbors) if they are connected by an edge.

The number of neighbors of a vertex is its degree.

In a **weighted graph** the edges have a weight (cost, length,..)

Directed graphs



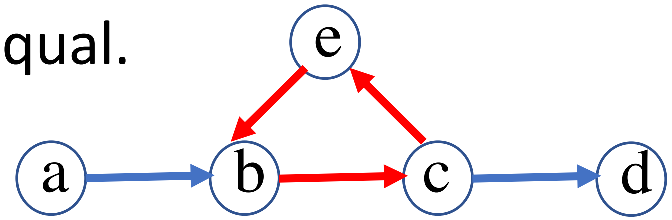
Edge (u, v) goes from vertex u to vertex v.

in-degree of a vertex: the number of edges pointing into it.
out-degree of a vertex: the number of edges pointing out of it.

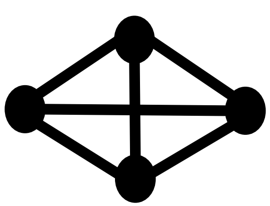
Graph definitions

- **Path**: sequence of nodes $(v_0..v_n)$ such that for all i: (v_i, v_{i+1}) is an edge. So a path is a sequence of edges $((v_0, v_1), (v_1, v_2), \ldots (v_{n-1}, v_n))$
- **Path length**: number of edges in the path, or sum of weights. **Simple path:** all nodes distinct.
- **Cycle**: path with first and last node equal.

e.g., ((b,c) (c,e) (e,b)) in



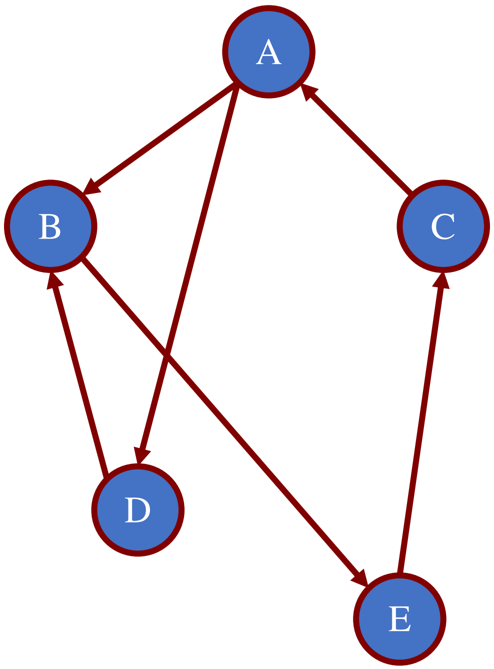
- **Acyclic graph**: graph without cycles. **DAG**: directed acyclic graph.
- In a **complete graph** all nodes in the graph are adjacent, e.g.,



Adjacency matrix of a graph

mapping of vertex labels to array indices

Label	Index
A	0
B	1
C	2
D	3
E	4



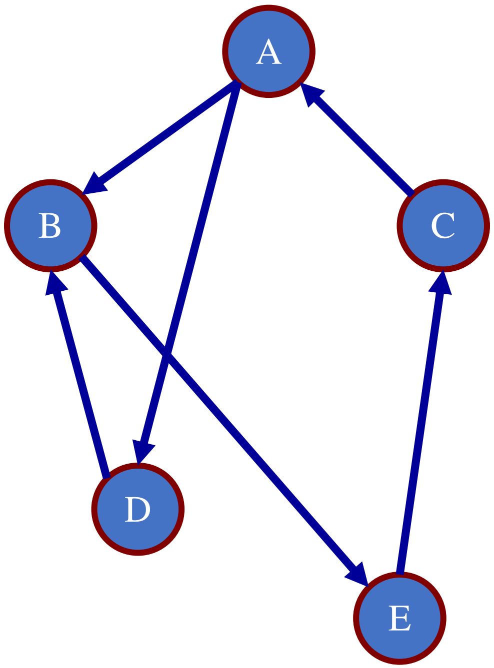
	0	1	2	3	4
0	0	1	0	1	0
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0

For an undirected graph, what would the adjacency matrix look like?

Adjacency matrix: n x n matrix with entries that indicate if an edge between two vertices is present

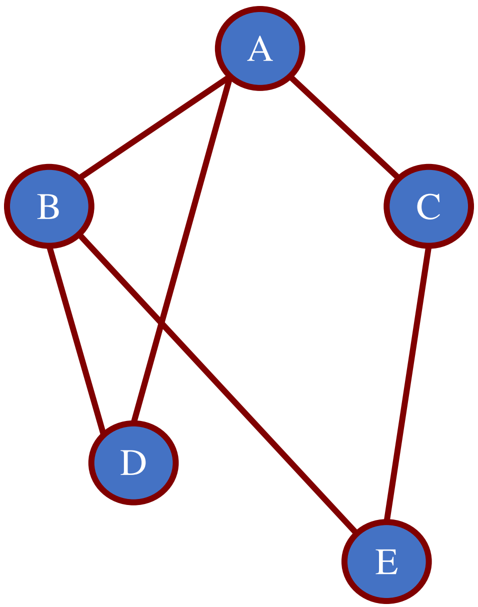
In a weighted graph the entries are the weights

Adjacency list for a directed graph



Index	Label	
0	A	→ B D
1	B	→ E
2	C	→ A
3	D	→ B
4	E	→ C

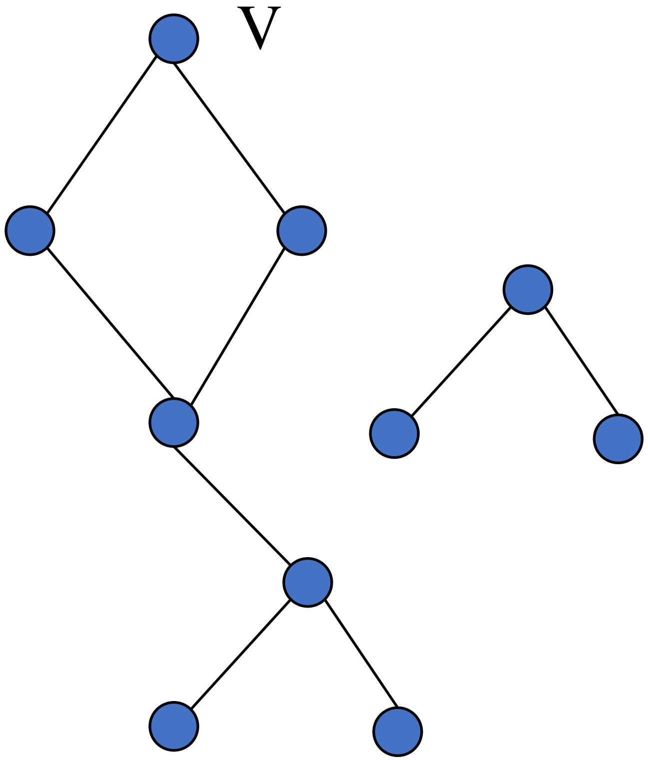
Adjacency list for an undirected graph



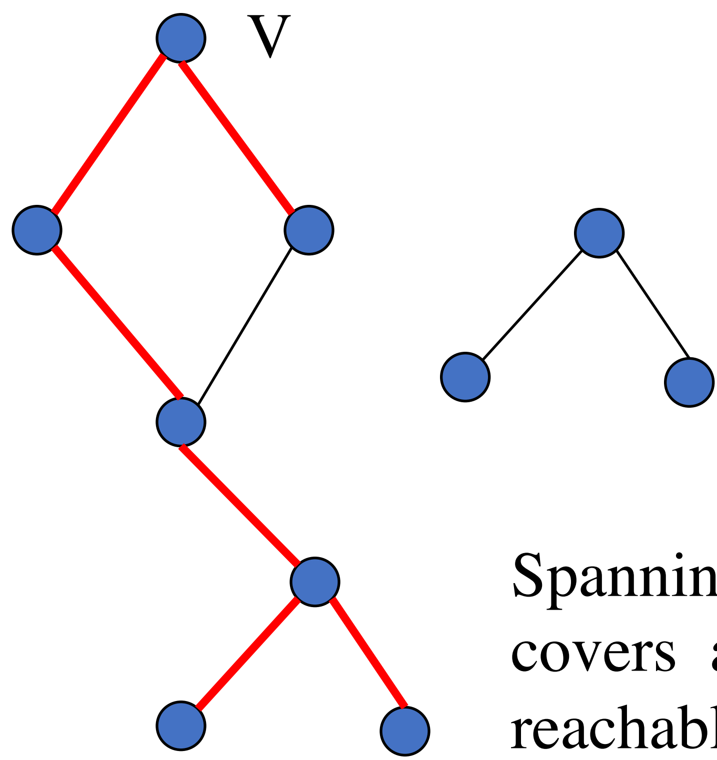
Index	Label			
0	A	B	C	D
1	B	A	D	E
2	C	A	E	
3	D	A	B	
4	E	B	C	

mapping of vertex
labels to lists of edges

Graph



A spanning Tree from V



Spanning tree covers all nodes reachable from V.

Not unique

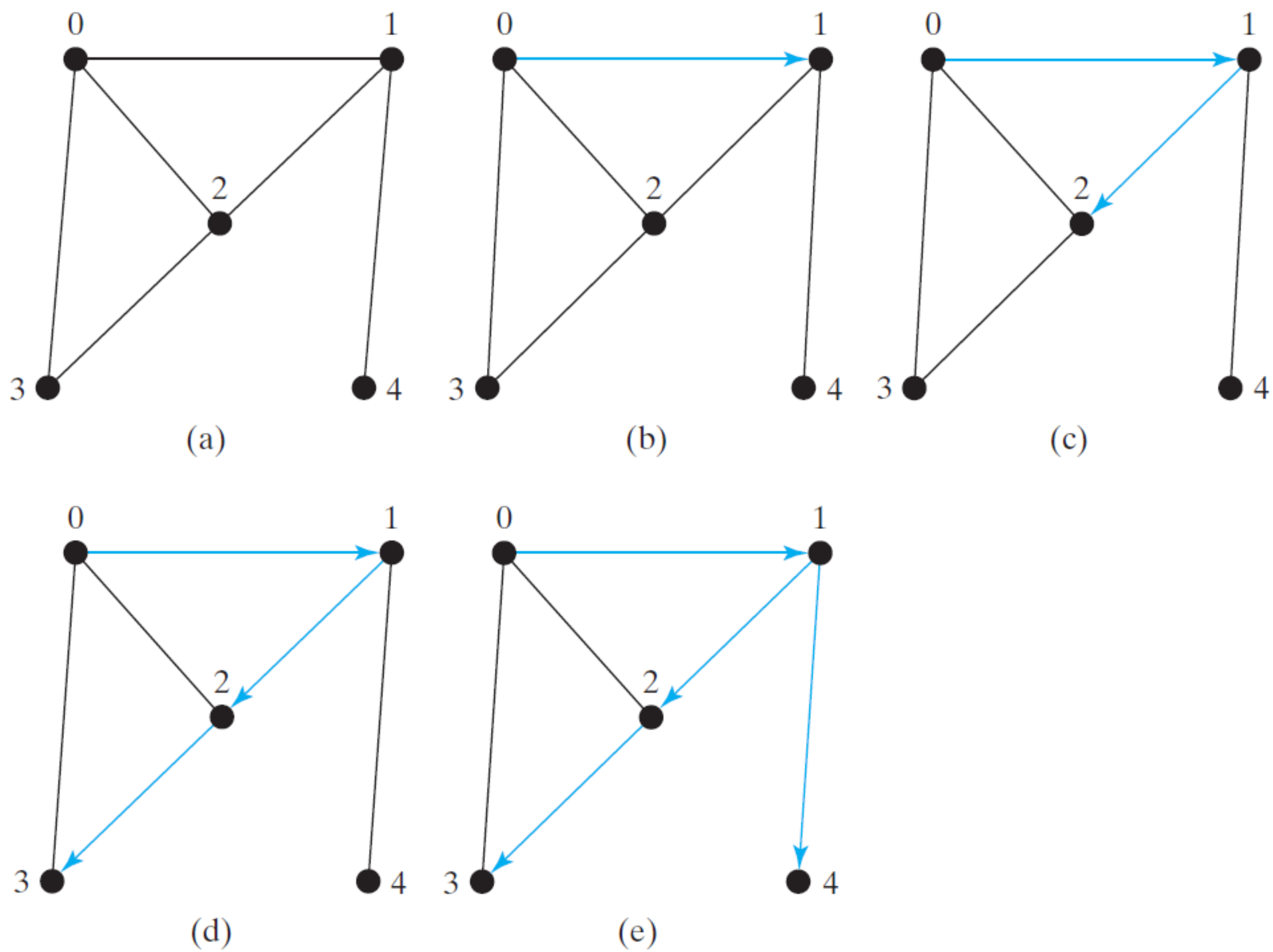
Depth-First Search

Depth-first search of a graph is like depth first search of a tree, but here we need to make sure we don't visit nodes more than once. We do this by marking nodes visited or not-visited (initially: not-visited).

```
// Input: G = (V, E) and a starting vertex v
// Output: a DFS spanning tree rooted at v
Tree dfs(vertex v) {
 visit v;
 set v visited
 for each neighbor w of v
  if (w has not been visited) {
   set v as the parent for w;
   dfs(w);
  }
}
```

Depth-first search builds a spanning tree of all of the reachable nodes from the starting vertex v, using marking of the visited nodes.

Depth-First Search Example



Applications of the DFS

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected.

- Finding a path between the root and another vertex.

- Finding connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

- Detecting whether there is a cycle in the graph, and finding a cycle in the graph.

Breadth-First Search

The breadth-first traversal of a graph is like the breadth-first traversal of a tree.

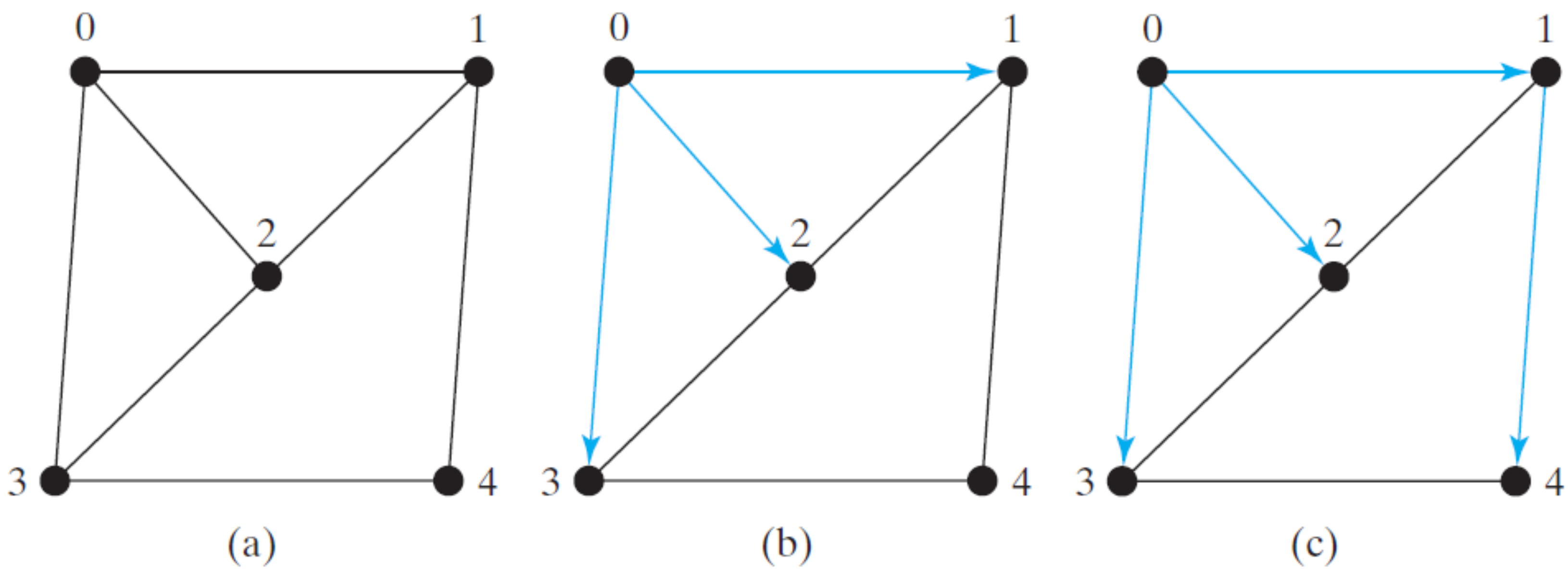
Breadth-first search, just as Depth-first search, results in a spanning tree.

With breadth-first traversal of a tree, the nodes are visited level by level, using a queue. First the root is visited, then all the neighbors of the root, then the neighbors of the neighbors of the root from left to right, and so on.

Breadth-First Search

```
// Input: G = (V, E) and a starting vertex v, Output: a BFS tree rooted at v
bfs(vertex v) {
  create an empty queue for storing vertices to be visited;
  add v into the queue;
  mark v visited;
  while the queue is not empty {
    dequeue a vertex, say u, from the queue
    for each neighbor w of u
      if w has not been visited {
        add w into the queue;
        set u as the parent for w;
        mark w visited;
      }
  }
}
```

Breadth-First Search Example



Queue: 0 isVisited[0] = true

Queue: 1 2 3 isVisited[1] = true, isVisited[2] = true, isVisited[3] = true

Queue: 2 3 4 isVisited[4] = true

Applications of the BFS

- ❖ Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.

- ❖ Detecting whether there is a path between the root and another vertex.

- ❖ Finding a shortest path between two vertices. The path between the root and any node in the BFS tree is the shortest path between the root and the node.

- ❖ Finding connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

Precedence Graphs

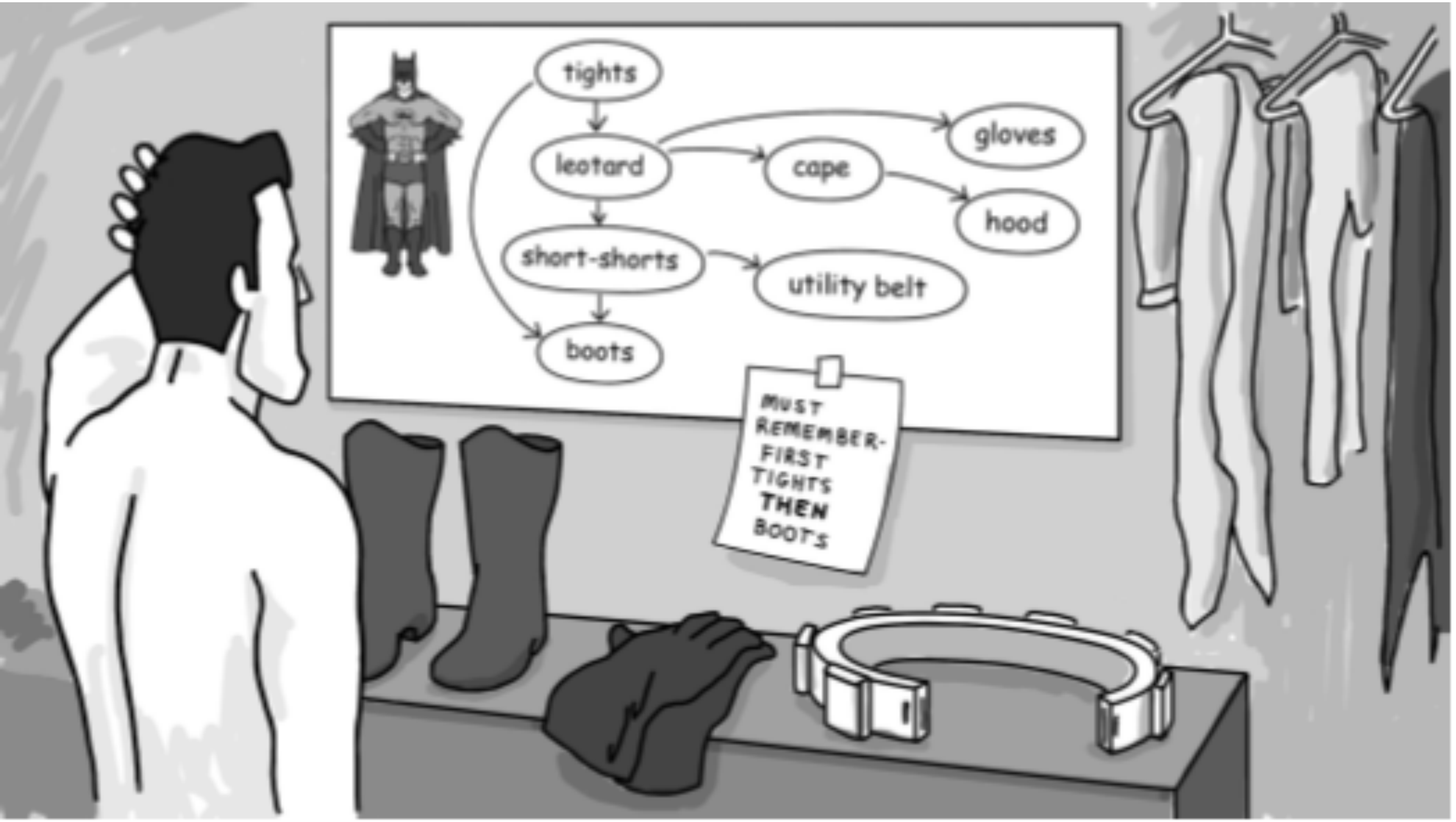
- In a precedence graph, an edge from *x* to *y* indicates *x* should come before *y*, e.g.:
  - prerequisites for a set of courses
  - dependences between programs
  - set of tasks, e.g. building a car or a computer

- A precedence graph is a DAG: directed acyclic graph

- Precedence graphs are also called "dependence graphs"

x precedes y → y depends on x

Graphs Describing Precedence



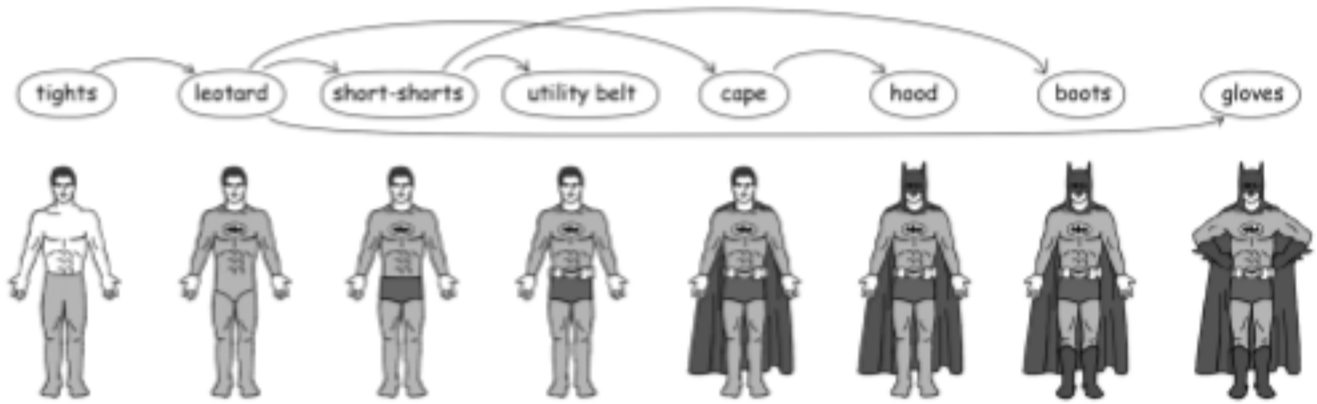
Batman images are from the book "Introduction to bioinformatics algorithms"

Graphs Describing Precedence

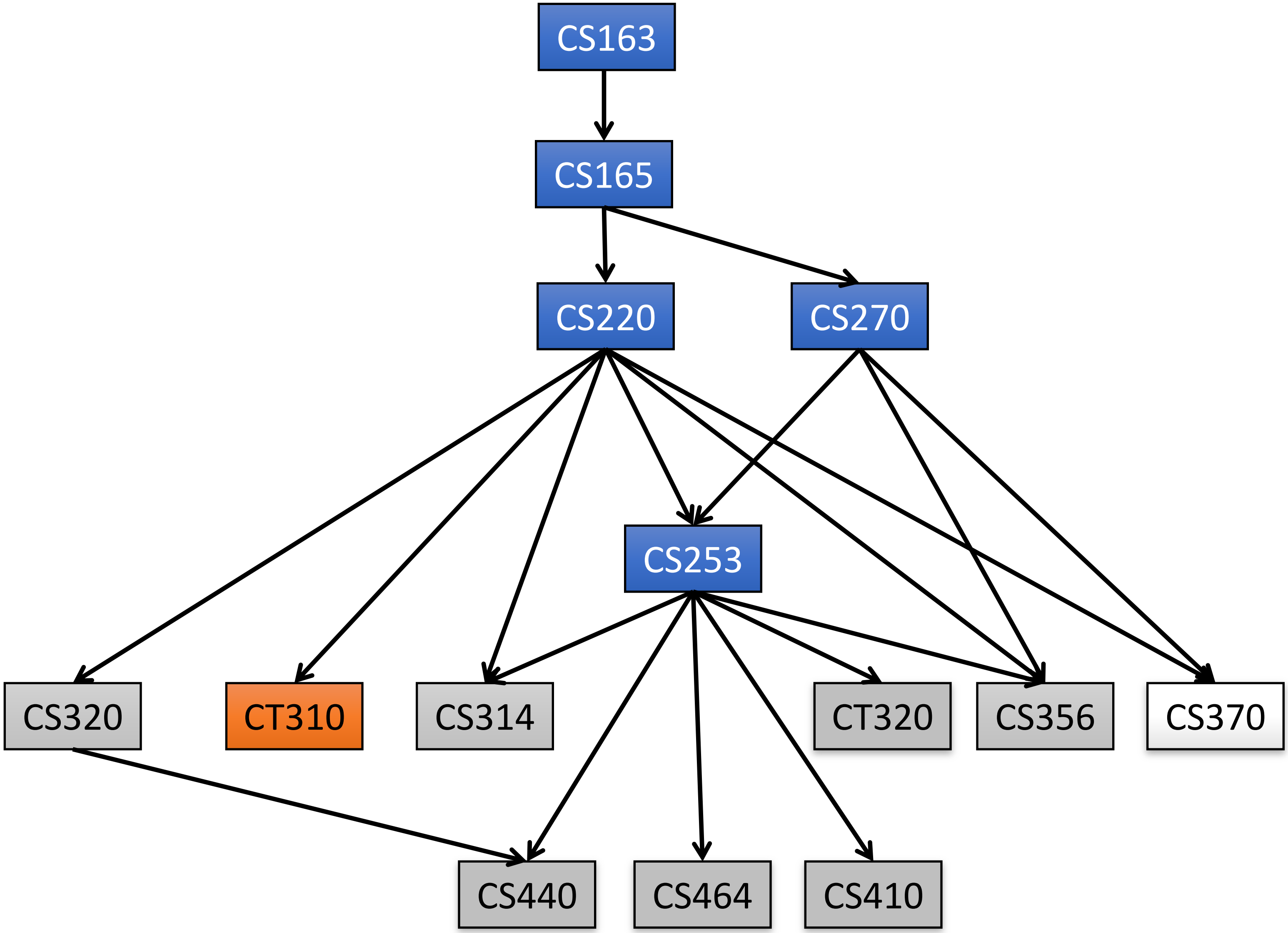
✦We want an ordering of the vertices of the graph that respects the precedence relation

- Example: An ordering of CS courses

✦The graph must not contain cycles. **WHY?**



CS Courses Required for CS and ACT Majors

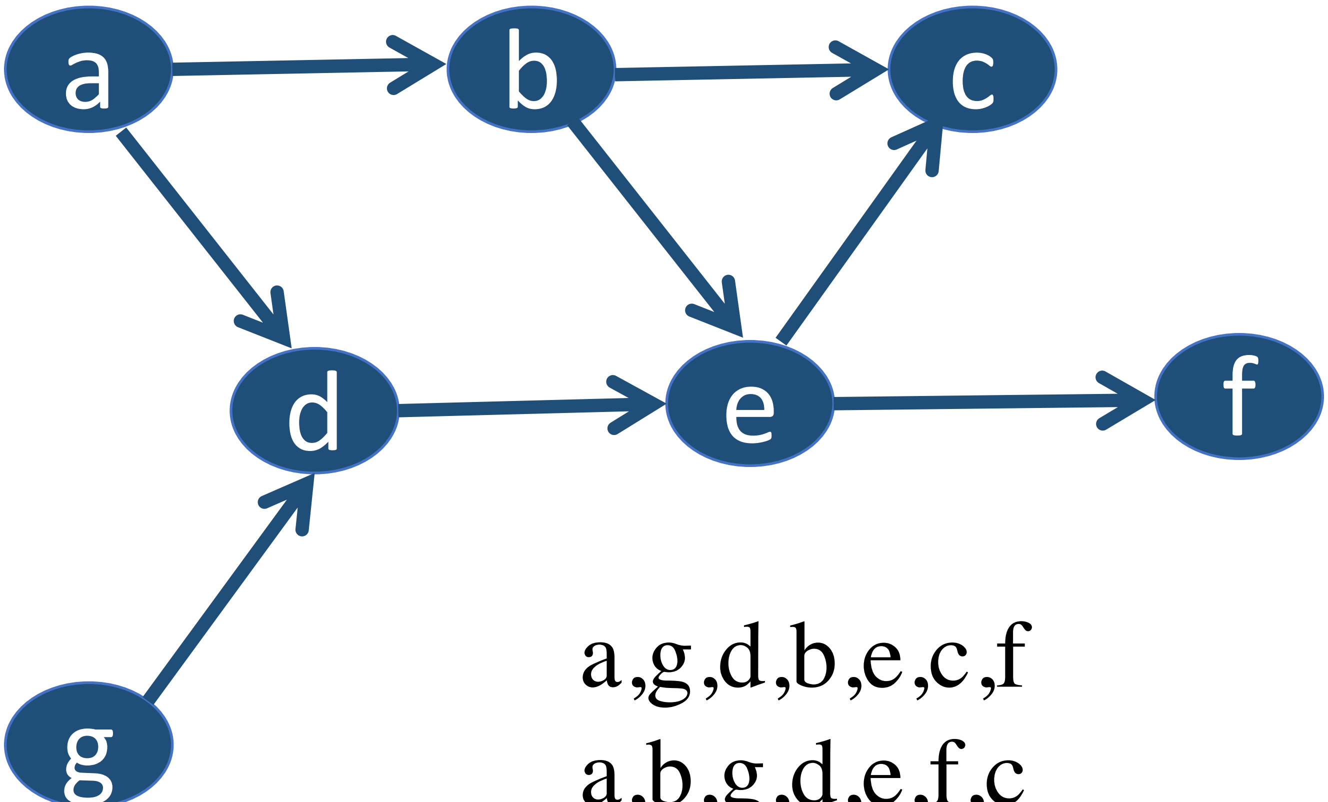


Topological Sorting of DAGs

- ✦DAG: Directed Acyclic Graph
- ✦Topological sort: listing of nodes such that if *(a,b)* is an edge, *a* appears before *b* in the list

Question: Is a topological sort unique?

A directed graph without cycles



a,g,d,b,e,c,f

a,b,g,d,e,f,c

Topological Sort Algorithm

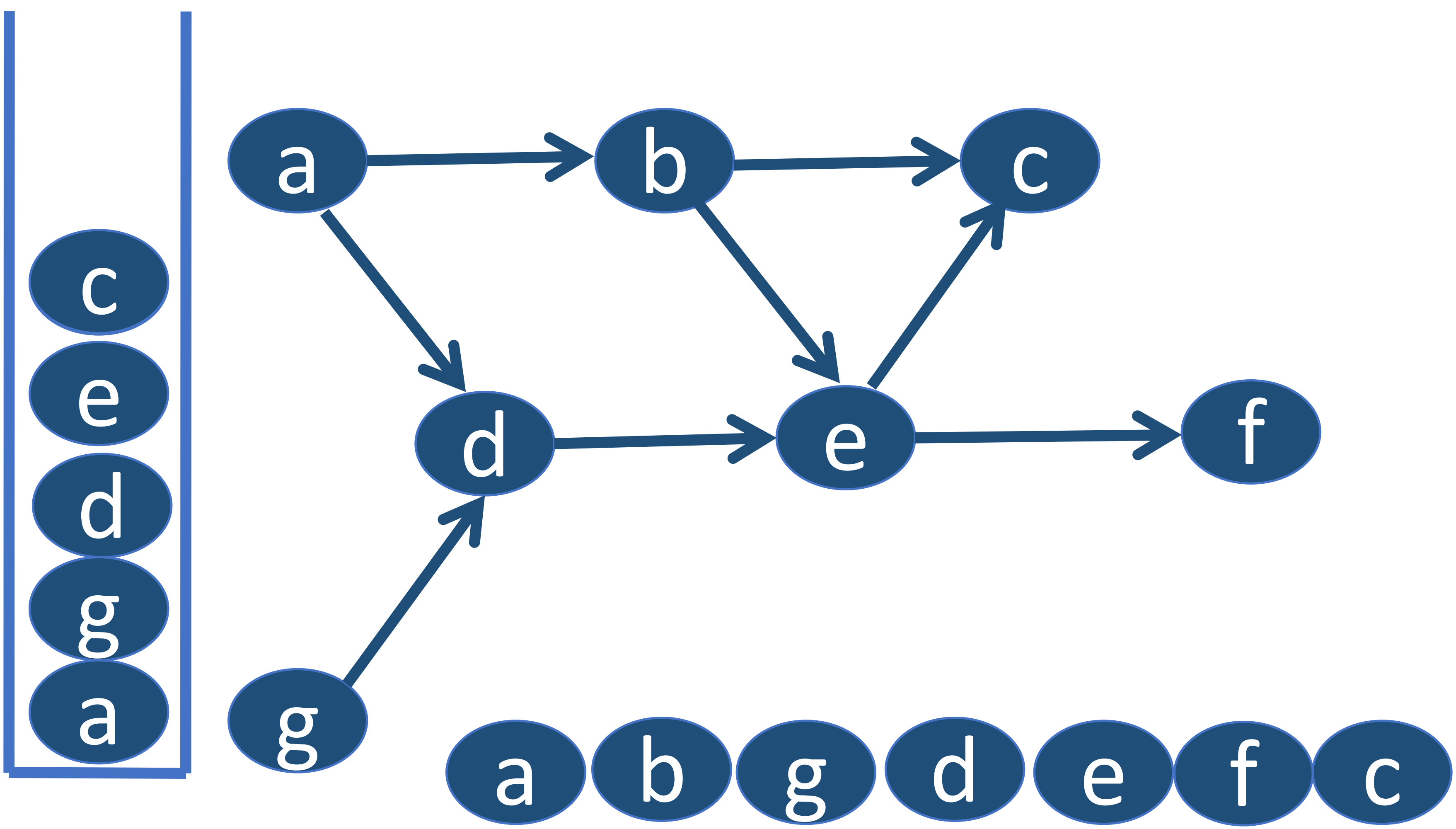
- ✦Modification of DFS: Traverse tree using DFS starting from all nodes that have no predecessor.

- ✦Add a node to the list when ready to backtrack.

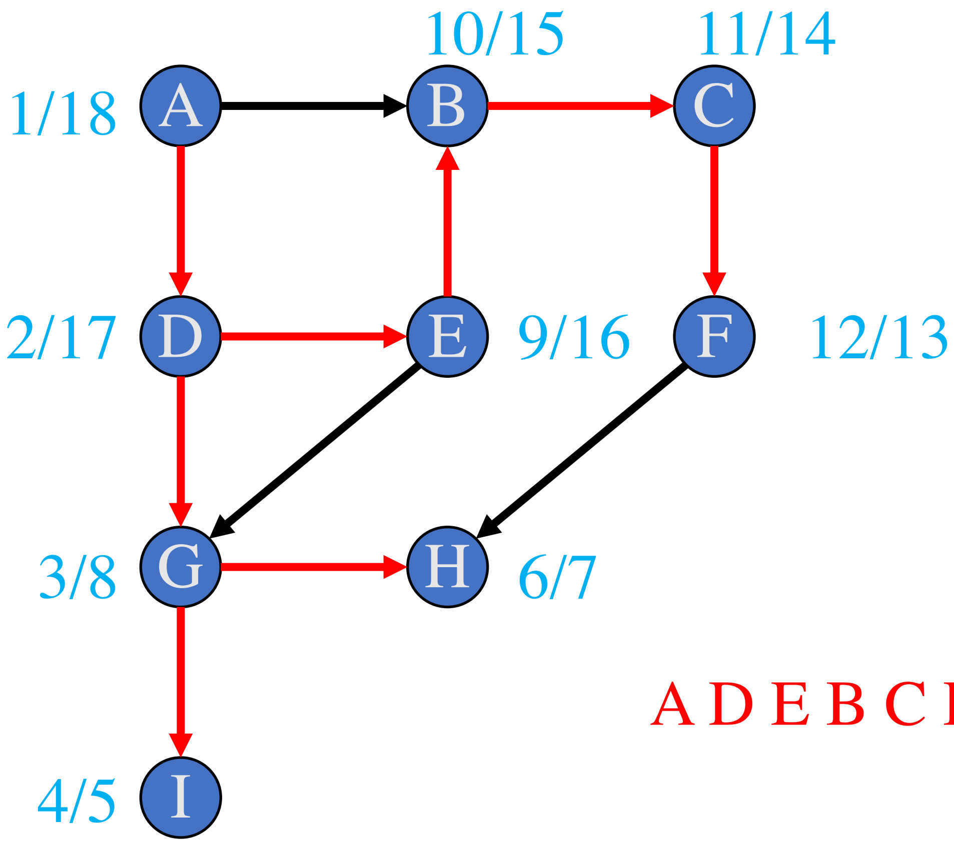
Topological Sort Algorithm

```
List topoSort(Graph theGraph)
  // use stack stck and list lst,  push all roots
  for (all vertices v in the graph theGraph)
   if (v has no predecessors)
      stck.push(v)
      Mark v as visited
   // DFS
  while (!stck.isEmpty())
   if (all neigbors of the vertex on top of stck have been visited)
      v = stck.pop()
      lst.add(0, v)
   else
      Select an unvisited neighbor u of v on top of the stack
      stck.push(u)
      Mark u as visited
      Set v as parent of u
  return lst
```

Example



Topological sorting solution



A D E B C F G H I

Red edges represent spanning tree