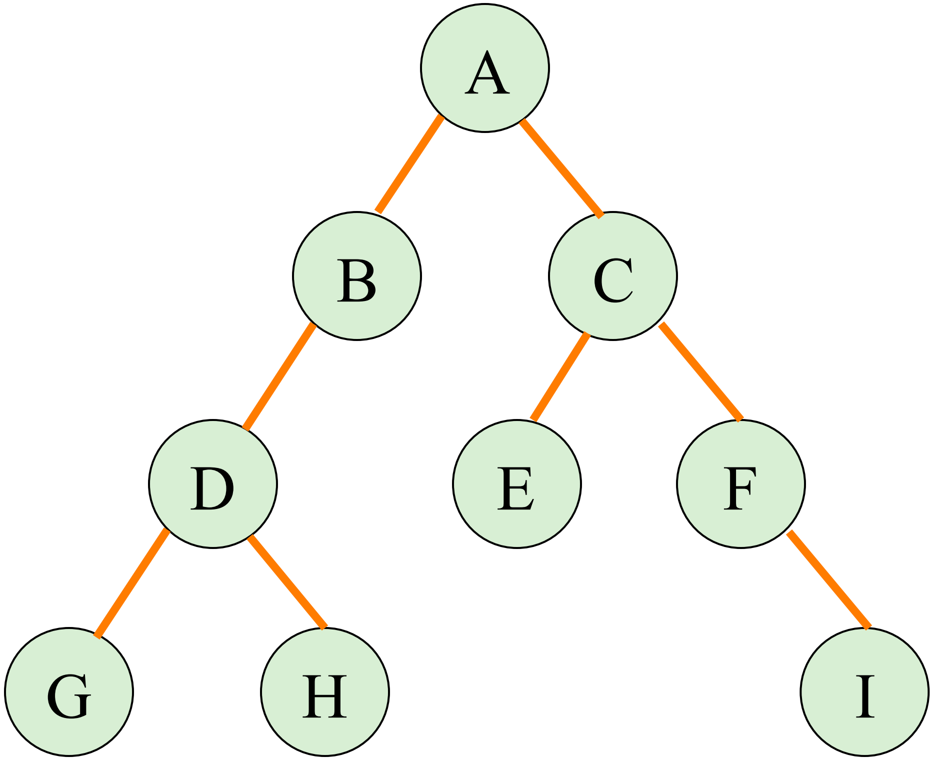
Graph Traversals

Tree traversal reminder



Pre order

A B D G H C E F I

In order

G D H B A E C F I

Post order

G H D B E I F C A

Level order

A B C D E F G H I

Connected Components

- The connected component of a node s is the largest set of nodes reachable from s. A generic algorithm for creating connected component(s):

```
R = {s}
while ∃edge(u,v) : u ∈ R ∧ v ∉ R
  add v to R
```

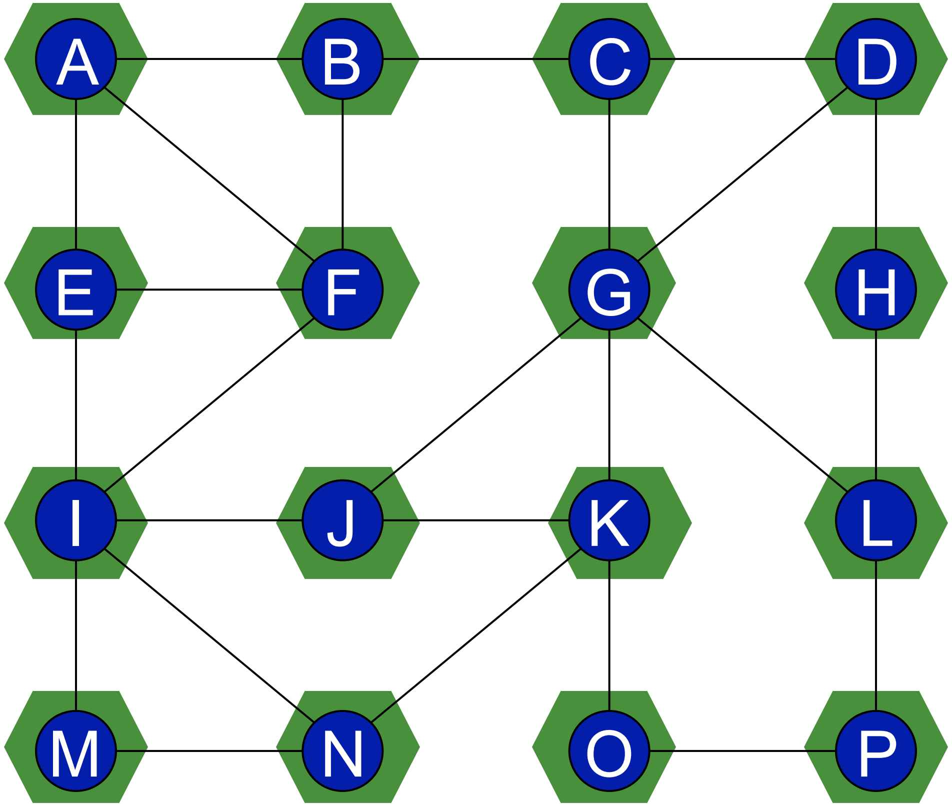
- Upon termination, R is the connected component containing s.
  - Breadth First Search (BFS): explore in order of distance from s.
  - Depth First Search (DFS): explores edges from the most recently discovered node; backtracks when reaching a dead-end.

Graph Traversals – Depth First Search

- Depth First Search starting at u

```
DFS(u):
  mark u as visited and add u to R
  for each edge (u,v) :
    if v is not marked visited :
      DFS(v)
```

Depth First Search



Question

- What determines the order in which DFS visits nodes?

- The order in which a node picks its outgoing edges

Depth first search algorithm

```
dfs(in v:Vertex)
  mark v as visited
  for (each unvisited vertex u adjacent to v)
    dfs(u)
```

- Need to track visited nodes
- Order of visiting nodes is not completely specified
  - if nodes have priority, then the order may become deterministic

    ```
    for (each unvisited vertex u adjacent to v in priority order)
    ```
- DFS applies to both directed and undirected graphs
- Which graph implementation is suitable?

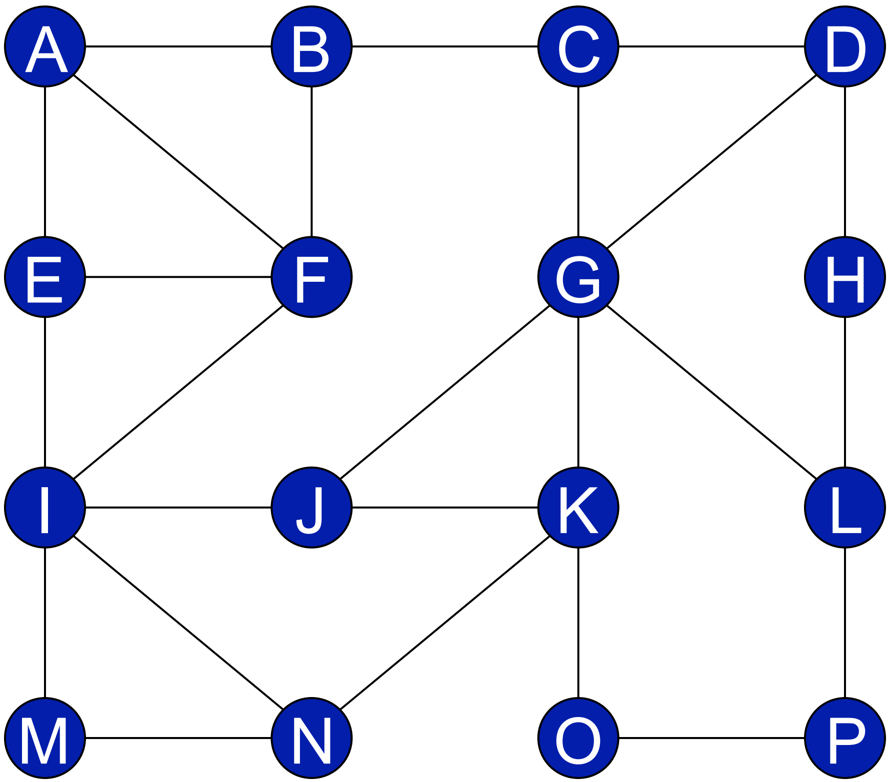
Iterative DFS: explicit Stack

```
dfs(in v:Vertex)
    s – stack for keeping track of active vertices
    s.push(v)
   mark v as visited
   while (!s.isEmpty())  {
        if (no unvisited vertices adjacent to the vertex on top of the stack) {
            s.pop()  //backtrack
        else {
            select unvisited vertex u adjacent to vertex on top of  the stack
            s.push(u)
            mark u as visited
        }
   }
```

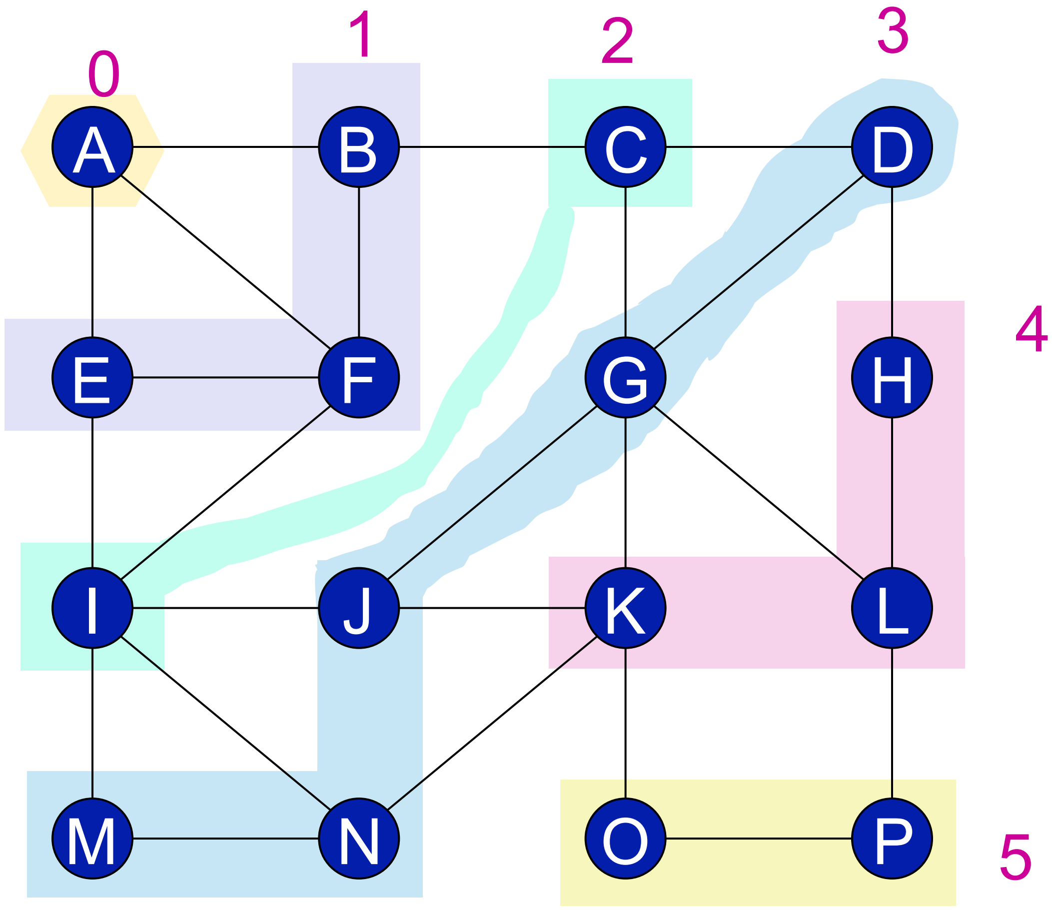
Breadth First Search (BFS)

- Is like level order in trees
- Which is not a BFS traversal starting from A?

A. A, B, C, D, …
B. A, B, F, E, …
C. A, E, F, B, …
D. A, B, E, F, …



Breadth First Search



BFS

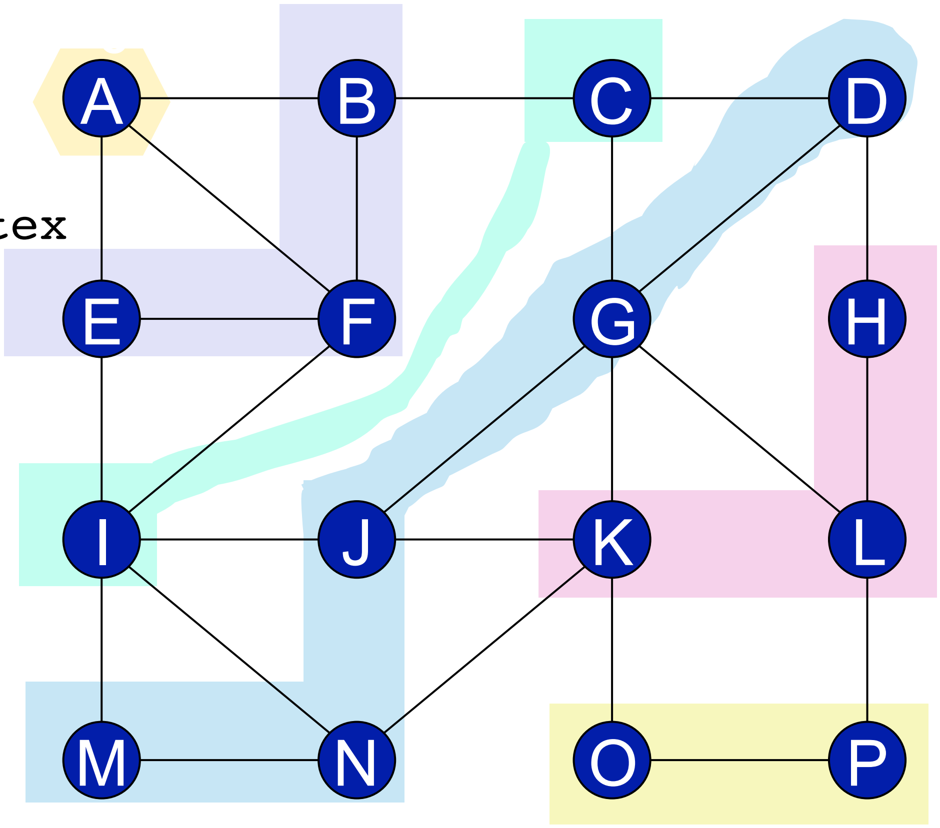
- Similar to level order tree traversal
- DFS is a last visited first explored strategy
  (uses a stack)
- BFS is a first visited first explored strategy
  (uses a queue)

BFS

```
bfs(in v:Vertex)
  q – queue of nodes to be processed
  q.enque(v)
  mark v as visited
  while(!q.isEmpty()) {
   w = q.dequeue()
   for (each unvisited vertex u adjacent to w) {
         mark u as visited
         q.enqueue(u)
   }
  }
```

Trace this example

```
bfs(in v:Vertex)
  q – queue of nodes
  q.enque(v)
  mark v as visited
  while(!q.isEmpty()) {
     w = q.dequeue()
     for (each unvisited vertex
           u adjacent to w) {
       mark u as visited
       q.enqueue(u)
       }
     }
```



Graph Traversal

- Properties of BFS and DFS:
  - Visit all vertices that are reachable from a given vertex
  - Therefore DFS(v) and BFS(v) visit a connected component
- Computation time for DFS, BFS for a connected graph: O(|V| + |E|)

  WHY?

Complexity BFS / DFS

- Each node is marked at most once, and visited at most once.
- The adjacency list of each node is scanned only once.
- Therefore time complexity for BFS and DFS is

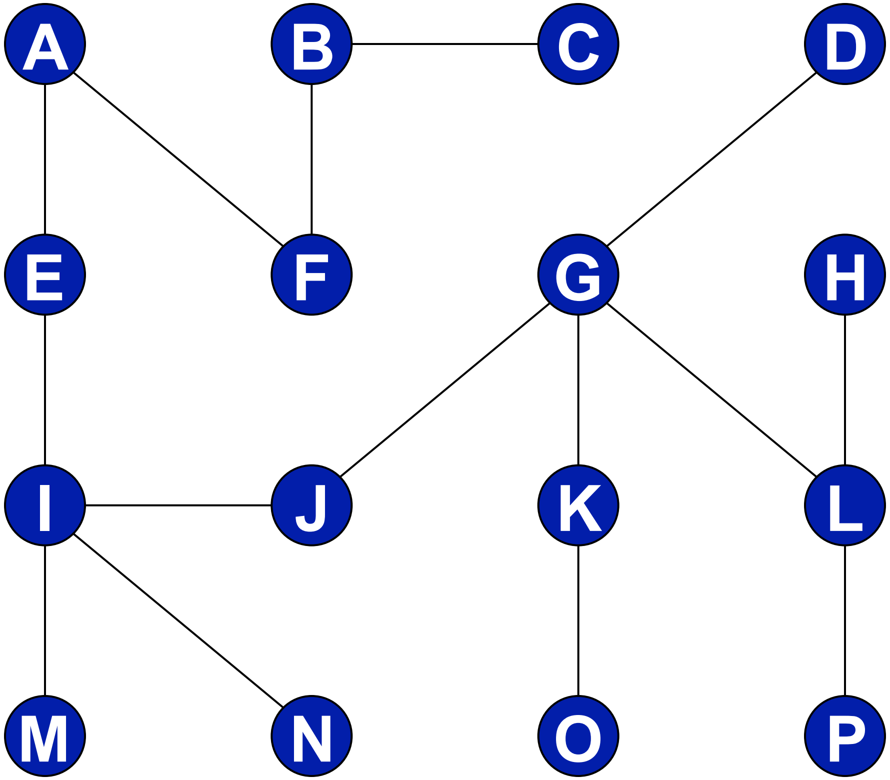
  $$O(|V|+|E|) \text{ or } O(n+m)$$

Reachability

- Reachability
  - *v* is reachable from *u*
    - if there is a (directed) path from *u* to *v*
  - solved using BFS or DFS
- Transitive Closure (G*)
  - G* has edge from *u* to *v* if v is reachable from u.

Trees as Graphs

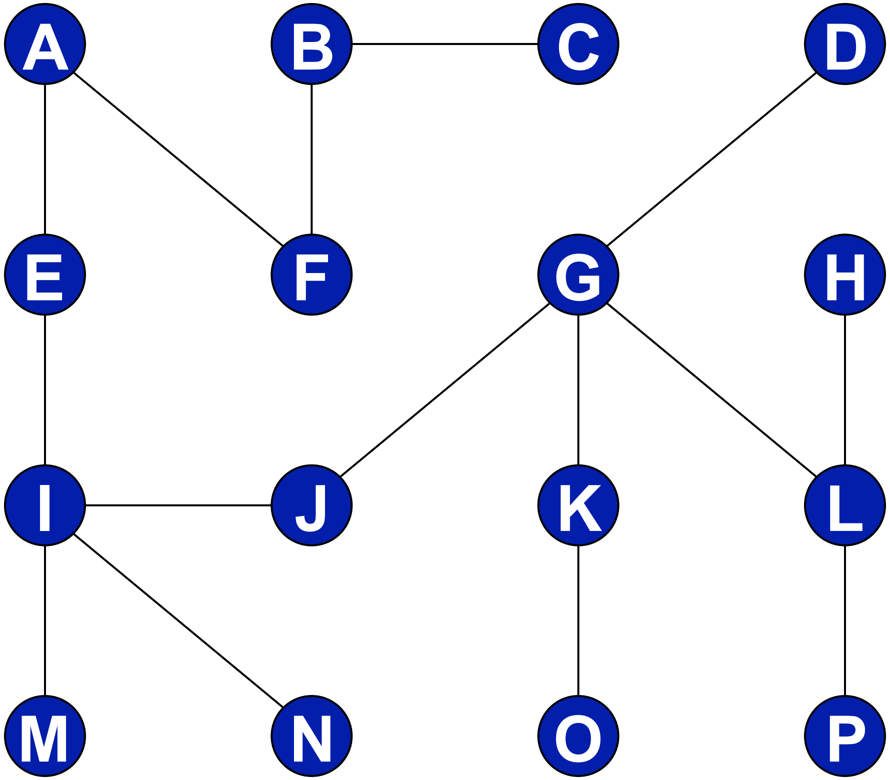
- Tree: an undirected connected graph that has no cycles.



Rooted Trees

- A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root

Example: Build rooted trees.



Question: Which node CANNOT be a root of this tree?

A. Node E   B. Node G   C. Node D   D. None

Trees as Graphs

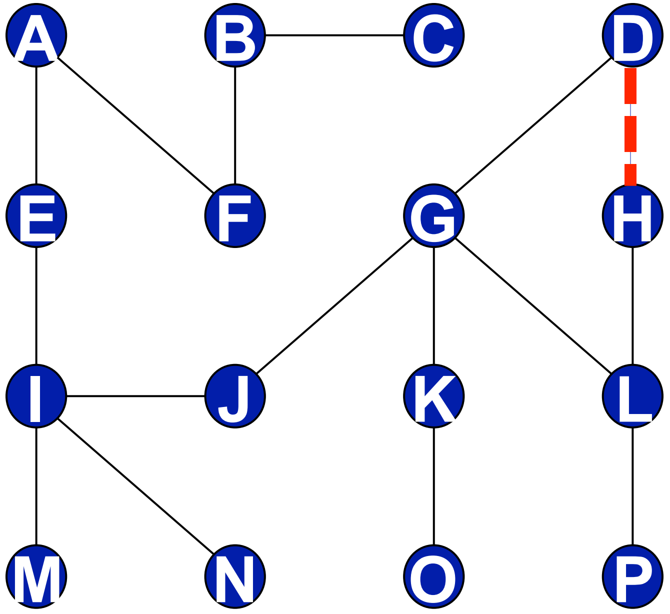
- Tree: an undirected connected graph that has no simple cycle.
- Cycle: a path that begins and ends at the same vertex and has length > 0
- Simple cycle: does not contain the same edge more than once

Theorems

A connected undirected graph with n vertices must have at least n-1 edges (otherwise some node in isolated.)

In a tree there is a unique path (no repeated nodes) between any two nodes (go up to common parent, go down to other node.)

A connected graph with n-1 edges is a tree. If we add one edge to a tree it gets a cycle, because there are then two paths between the incident nodes



Spanning Trees

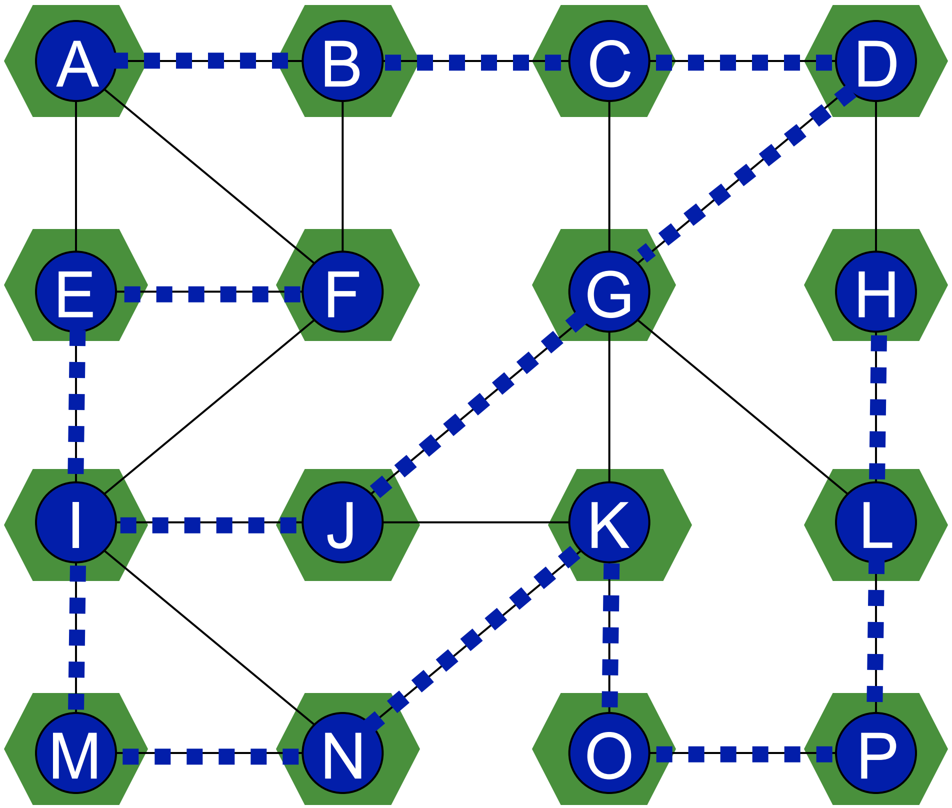
- **Spanning tree:** A sub-graph of a connected undirected graph G that contains all of G's vertices and enough of its edges to form a tree.

- How to get a spanning tree:
  - Remove edges until you get a tree, never disconnecting the nodes in the tree
  - Add edges until you have a spanning tree, never creating a cycle

Spanning Trees - DFS algorithm

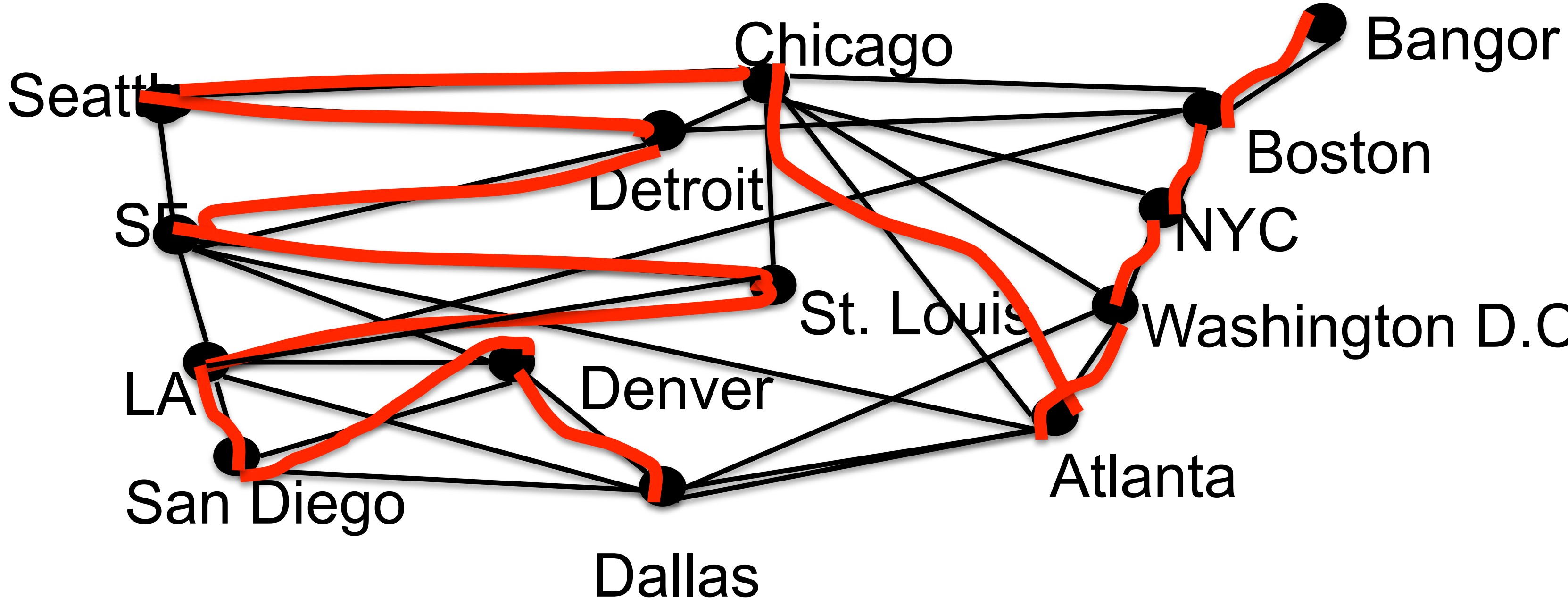
```
dfsTree(in v:vertex)
  Mark v as visited
  for (each unvisited vertex u adjacent to v)
     Mark the edge from u to v
     dfsTree(u)
```

Spanning Tree –
Depth First Search Example



Example

- Suppose that an airline must reduce its flight schedule to save money. If its original routes are as illustrated here, which flights can be discontinued to retain service between all pairs of cities (where might it be necessary to combine flights to fly from one city to another?)



Question

- Does Dijkstra's algorithm lead to the spanning tree with the minimal total distance?

- **No.**

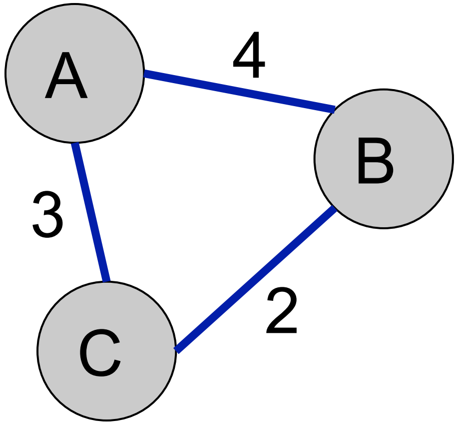
Question

- Does Dijkstra's algorithm lead to the spanning tree with the minimal total distance? Dijkstra determines the shortest path from a source to each node in the graph
- **No.**

  Counter example: (s=A)

  Shortest paths from A?

  Minimal total distance spanning tree?



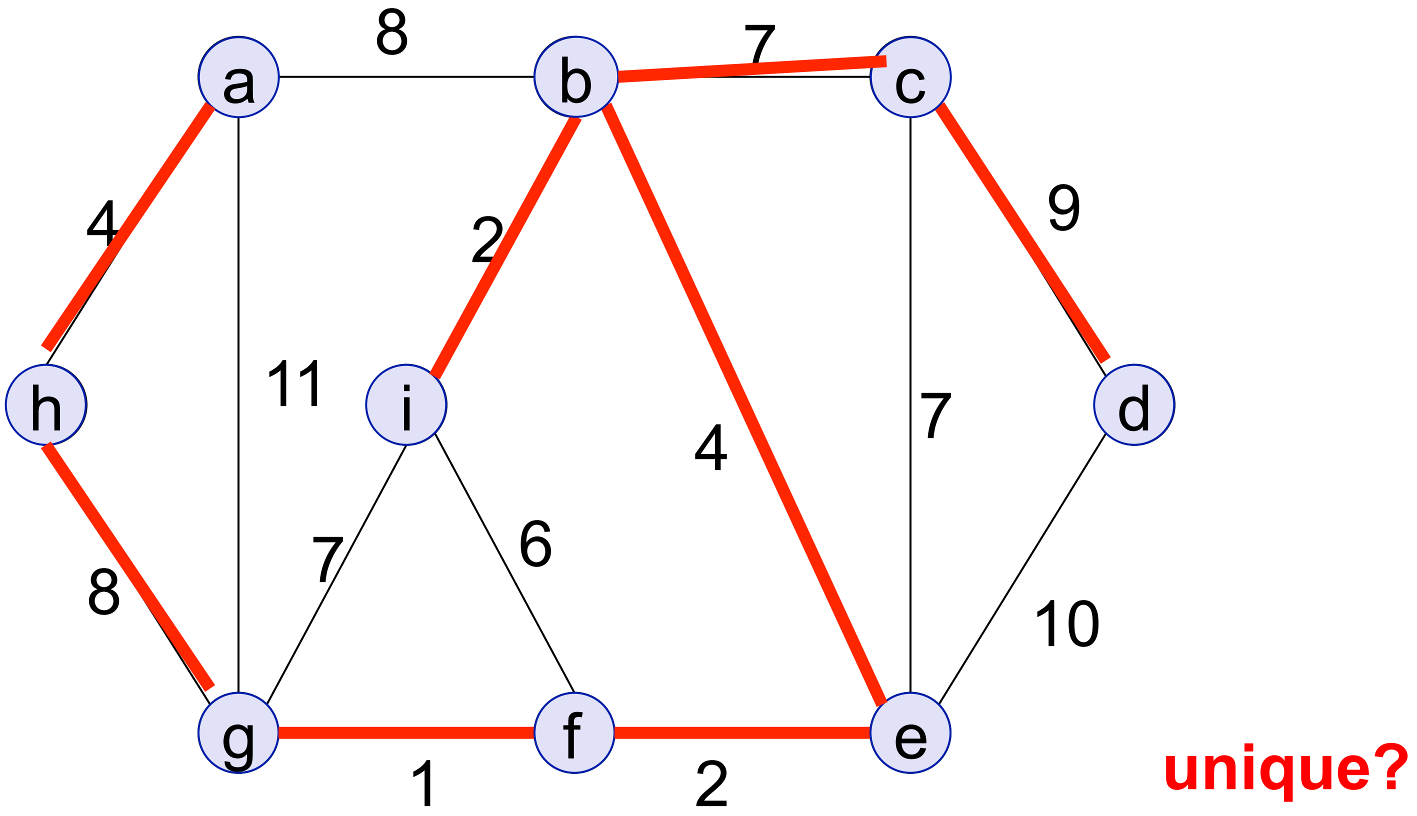
Minimum Spanning Tree

- Minimum spanning tree
  - Spanning tree **minimizing the sum of edge weights**
- Example: Connecting each house in the neighborhood to cable
  - Graph where each house is a vertex.
  - Need the graph to be connected, and minimize the cost of laying the cables.

Prim's Algorithm

- Idea: incrementally build spanning tree by adding the least-cost edge to the tree
  - Weighted graph
  - Find a set of edges
    - Touches all vertices
    - Minimal weight
    - Not all the edges may be used

Prim's Algorithm: Example starting at d



{(d,c),(c,b), (b,i), (b,e), (e,f), (f,g), (g,h), (h,a) }

Prim's Algorithm

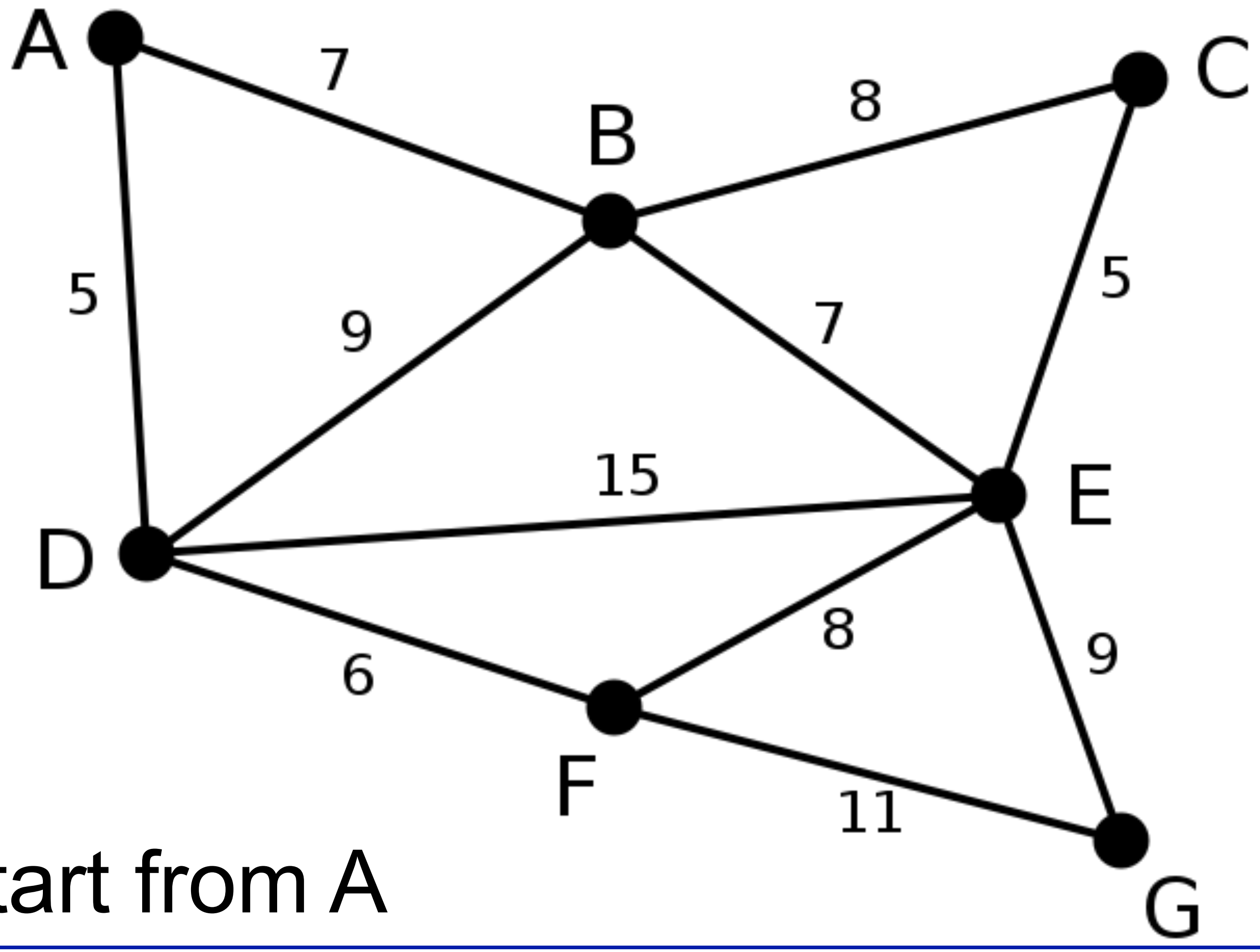
```
prims(in  v:Vertex)
  // Determines a minimum spanning tree for a weighted
  // connected, undirected graph whose weights are
  // nonnegative, beginning with any vertex v.
  Mark vertex v as visited and include it in
  the minimum spanning tree
  while (there are unvisited vertices) {
      find the least-cost edge (v, u) from a visited
            vertex v to some unvisited vertex u
      Mark u as visited
      Add vertex u and the edge (v, u) to the
            minimum spanning tree
  }
  return minimum spanning tree
```

Prim vs Dijkstra

- Prim's MST algorithm is very similar to Dijkstra's SSSP algorithm.

- What is the difference?

Do SSSP, do MST



Start from A

Graphs Describing Precedence

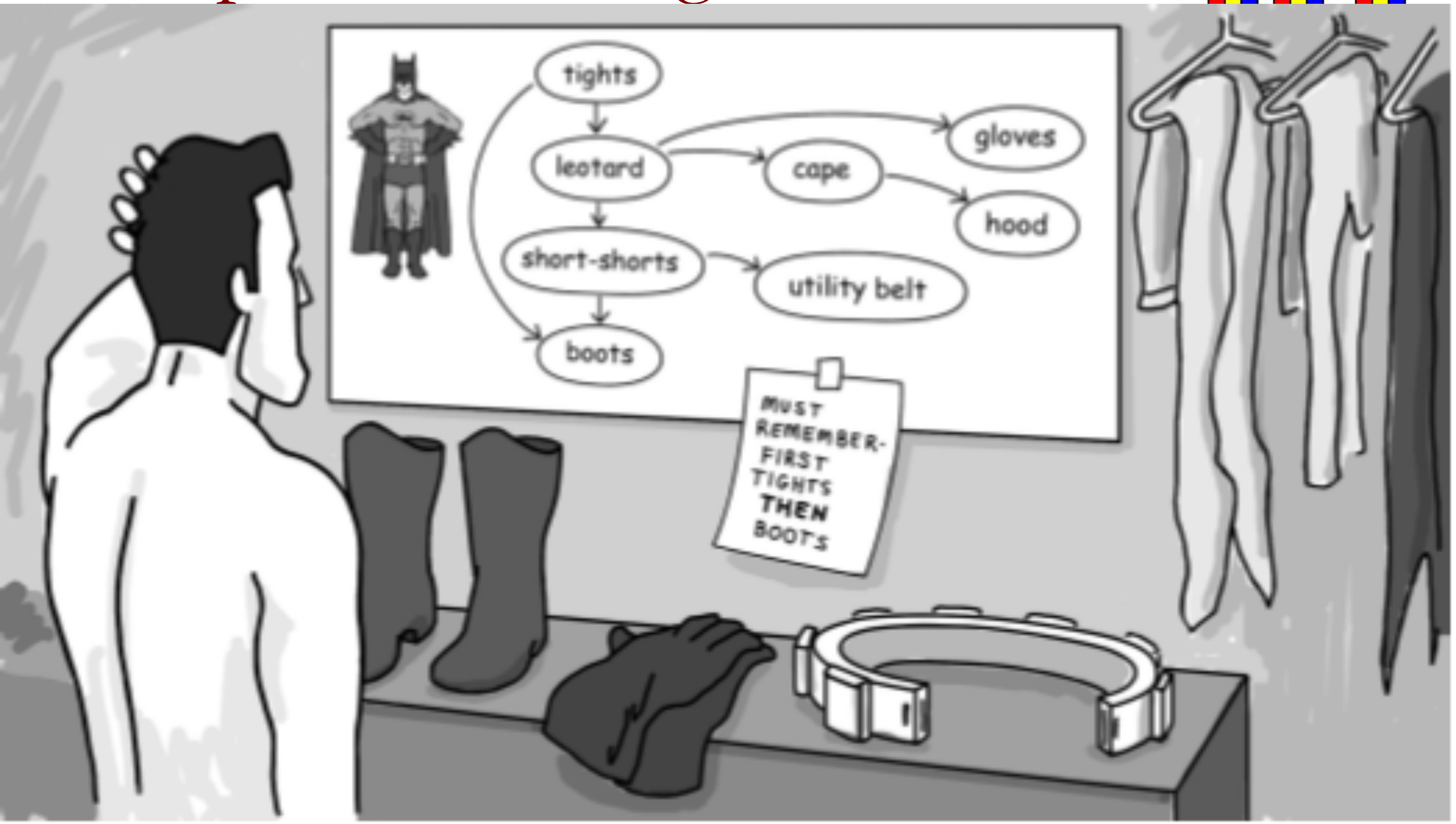
- Edge from x to y indicates x should come before y, e.g.:
  - prerequisites for a set of courses
  - dependences between programs
  - dependences between statements

    a = 10

    b = 20

    c = a+b
  - set of tasks

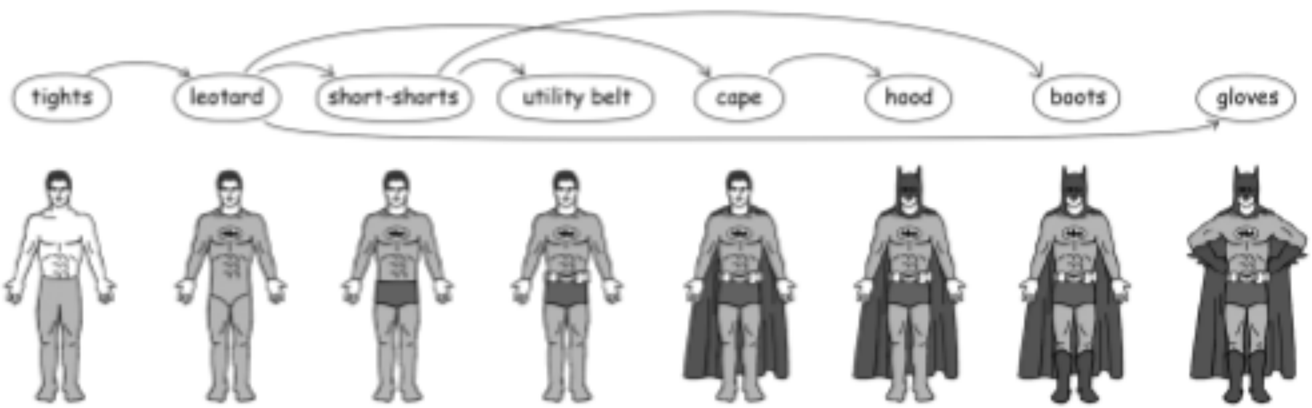
Graphs Describing Precedence



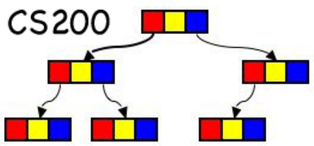
Batman images are from the book "Introduction to bioinformatics algorithms"

Graphs Describing Precedence

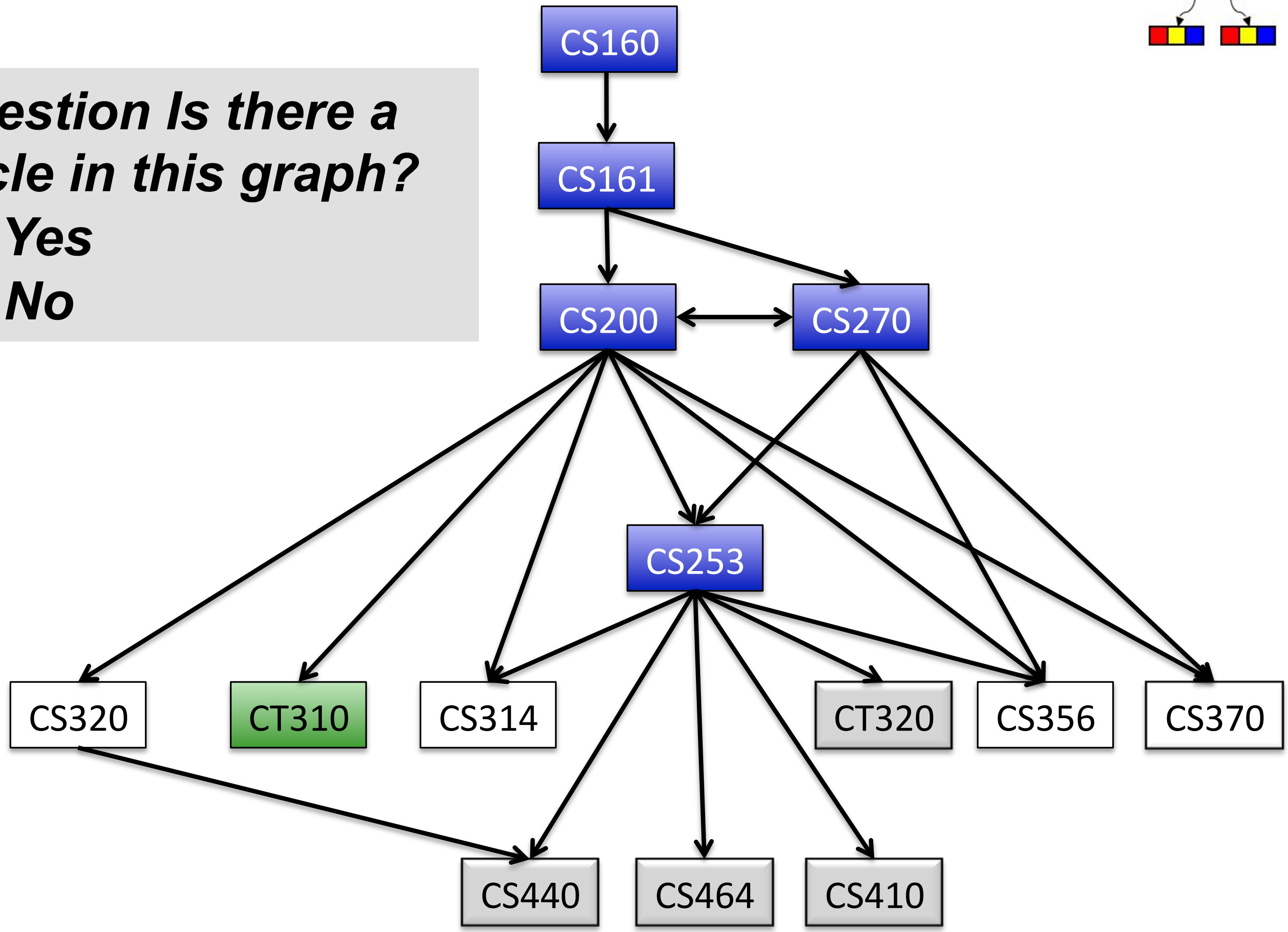
- Want an ordering of the vertices of the graph that respects the precedence relation
  - Example: An ordering of CS courses
- The graph must not contain cycles. **WHY?**



CS Courses Required for CS and ACT Majors



Question Is there a cycle in this graph?
A. Yes
B. No

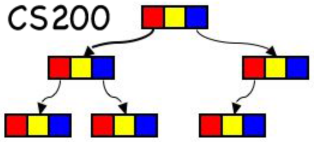


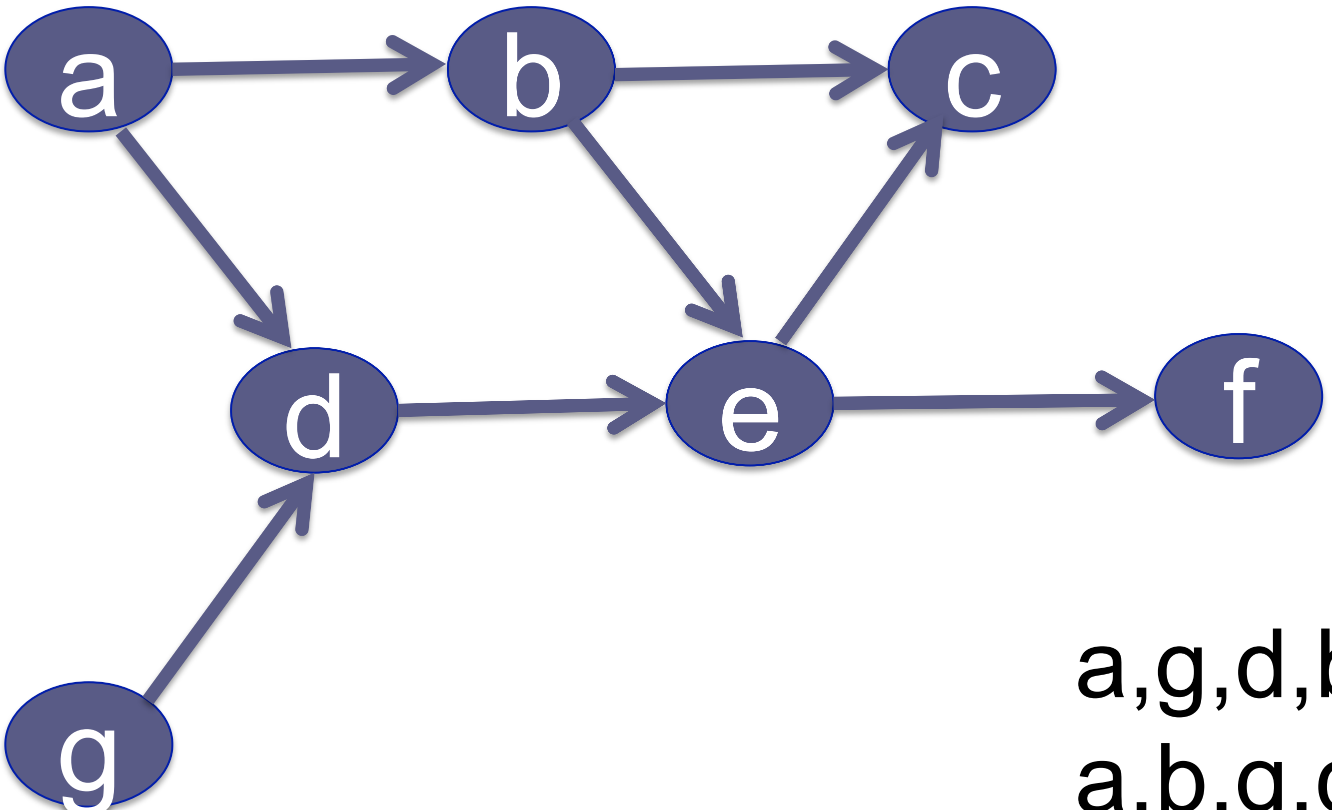
Topological Sorting of DAGs

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

Question: Is a topological sort unique?

A directed graph without cycles





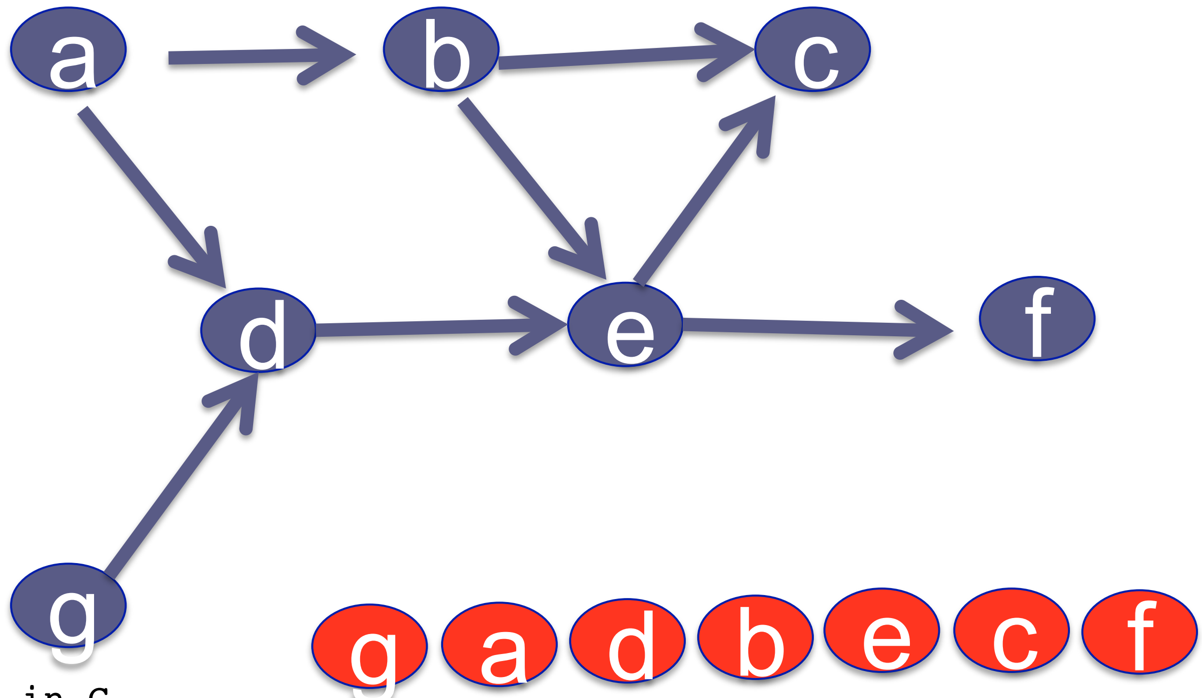
a,g,d,b,e,c,f
a,b,g,d,e,f,c

Topological Sort - Algorithm 1

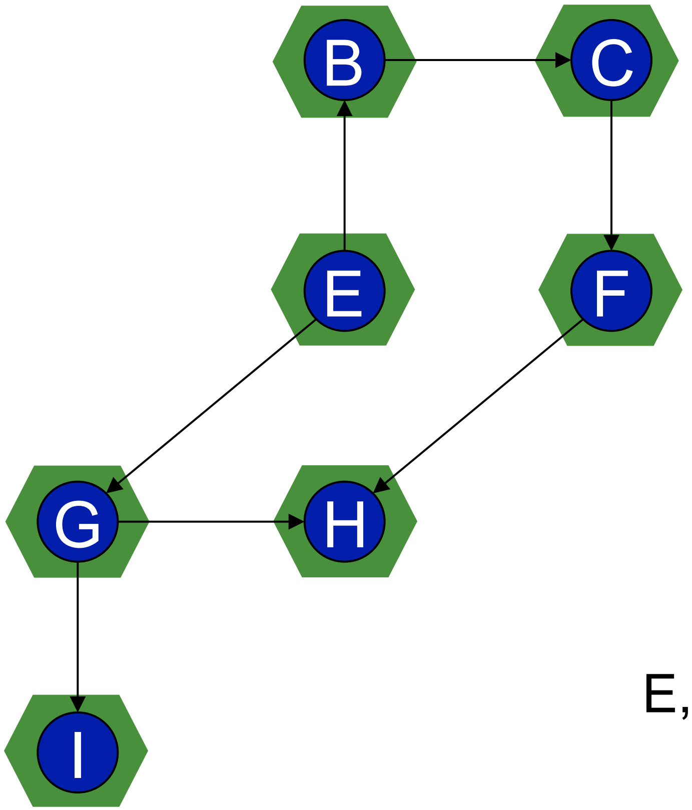
```
topSort1(in G:Graph)
  n= number of vertices in G
  for (step =1 through n)
     select a vertex v that has no successors
     aList.add(0,v)
     Delete from G vertex v and its edges
  return aList
```

- Algorithm relies on the fact that in a DAG there is always a vertex that has no successors.
- Destructively modifies the graph.

Topological Sort - Algorithm 1




```
topSort1(in G:Graph)
   n= number of vertices in G
   for (step =1 through n)
        select a vertex v that has no successors
        aList.add(0,v)
        Delete from G vertex v and its edges
   return aList
```



E, B, G, C, F, H, I

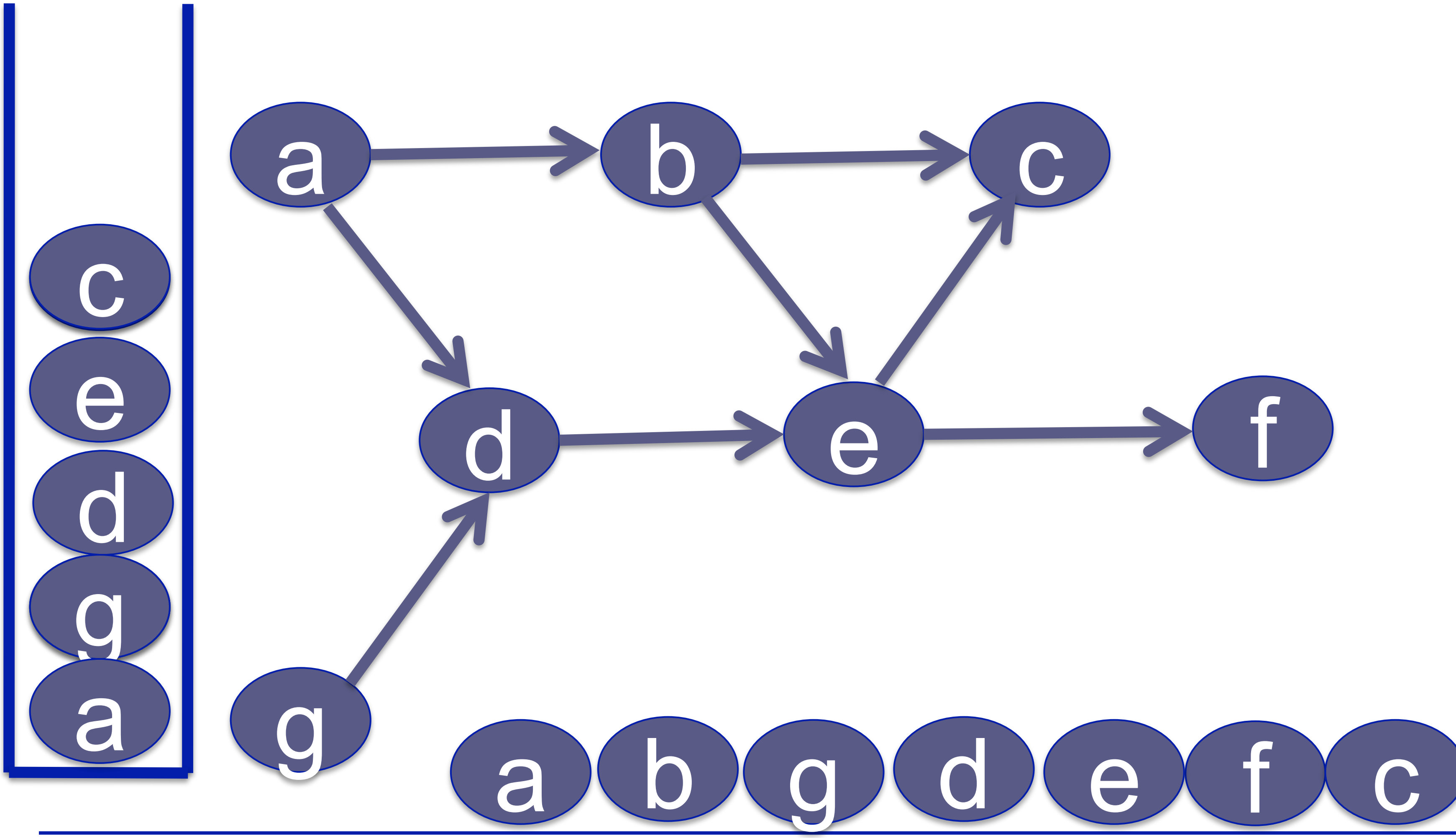
Topological Sort - Algorithm 2

- 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 2

```
topSort2( in theGraph:Graph):List
   s.createStack()
   for (all vertices v in the graph theGraph)
        if (v has no predecessors)
                s.push(v)
                Mark v as visited
   while (!s.isEmpty())
        if (all vertices adjacent to the vertex on top of
                the stack have been visited)
                v = s.pop()
                aList.add(0, v)
        else
                Select an unvisited vertex u adjacent to vertex on top
                        of the stack
                s.push(u)
                Mark u as visited
   return aList
```

Algorithm 2: Example 1

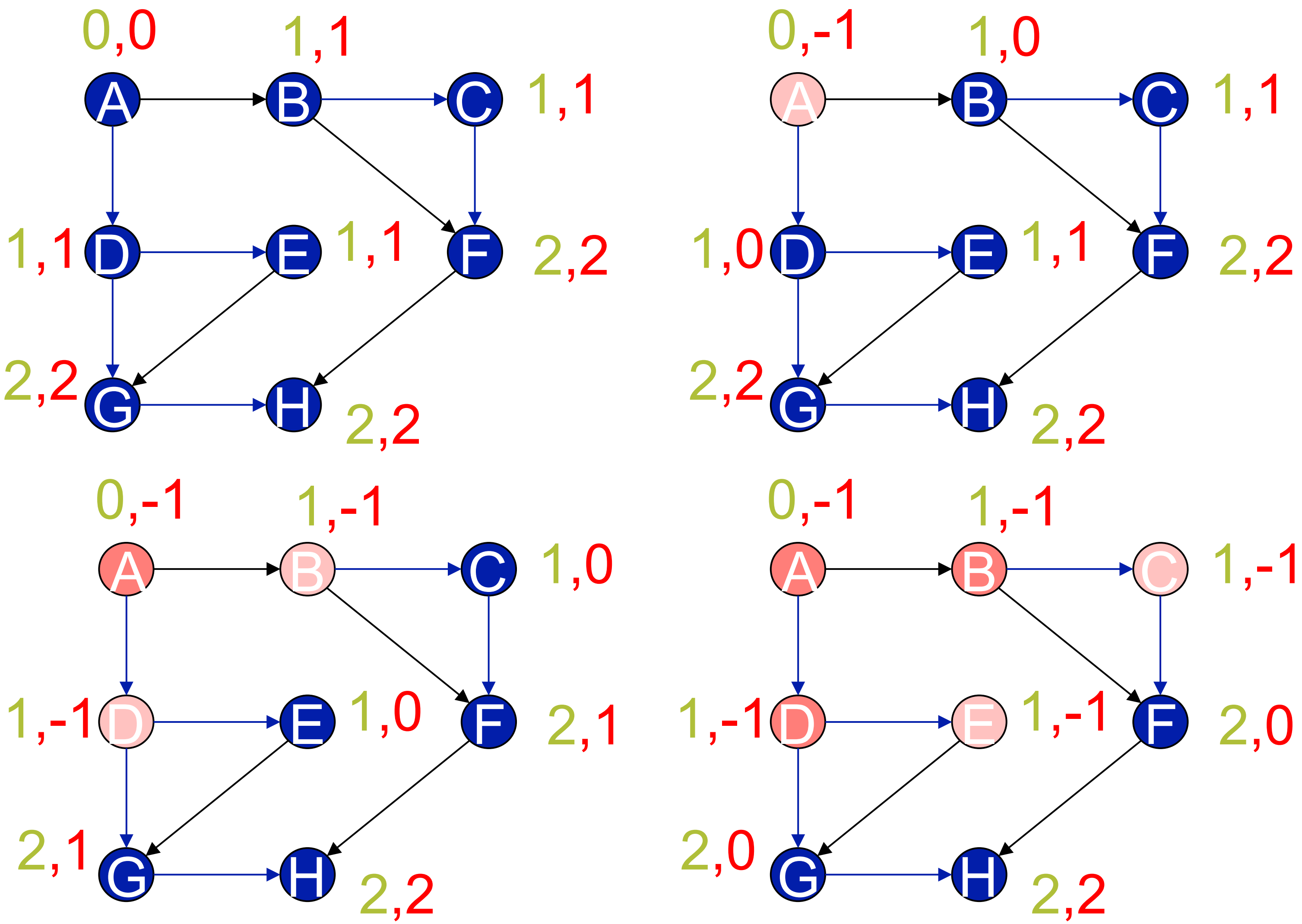


Third topological sort algorithm

- First two topological sort algorithms found nodes without successors and then backtracked
- **Forward** algorithm based on inDegrees
  - Copy all inDegrees to temporary inDegree tID
  - Repeat until all visited:

1. Find new nodes without predecessors (tID 0)
2. Put these in a list, or print them out (P5), making sure they will not be selected again (e.g. set their tID to -1)
3. Subtract 1 from tID of all successors of the nodes from step 2

forward topological sort



A

B D

C E

finish the animation