Depth First Search(DFS)

- Another graph traversal algorithm
- Unlike BFS, this one follows a path as deep as possible before *backtracking* .
- Where BFS is “queue-like,” DFS is “stack-like”.
- Vertices go through white, gray and black stages of color.
  - White : initially
  - Gray : when discovered first
  - Black : when finished i.e. the adjacency list of the vertex is completely examined.
- Also records timestamps for each vertex
  - d[v] (= start time) when the vertex is first discovered
  - f[v] (= finish time) when the vertex is finished

Depth First Search(DFS)

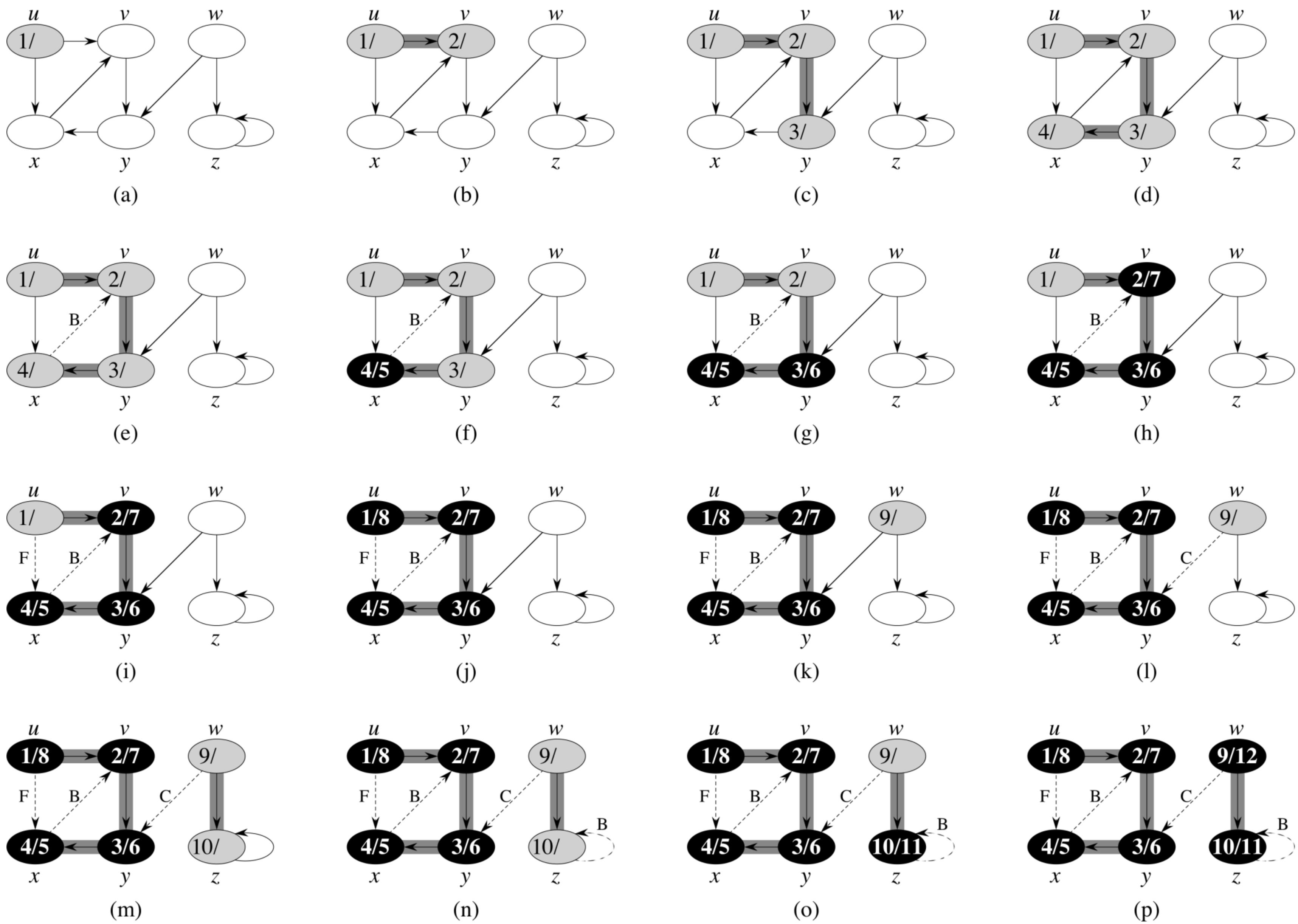
- Notes on timestamps:
  - Timestamps are integers in the range [1 .. 2n].
    (2n timestamp values are used because each node gets discovered once and finished once.)
  - For each node u, d[u] < f [u].
  - The color of u is white before d[u],
    gray between d[u] and f [u], and black thereafter.

Depth First Search(DFS)

```
DFS(G)
for each vertex u ∈ V[G]
      do color[u] ← WHITE
         π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
      do if color[u] = WHITE
            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] ← GRAY               ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]           ▷ Explore edge (u, v).
      do if color[v] = WHITE
            then π[v] ← u
                 DFS-VISIT(v)
color[u] ← BLACK              ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

DFS example



u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/
3/
x y z
(c)
u v w
1/ 2/
4/ 3/
x y z
(d)
u v w
1/ 2/
B
4/ 3/
x y z
(e)
u v w
1/ 2/
B
4/5 3/
x y z
(f)
u v w
1/ 2/
B
4/5 3/6
x y z
(g)
u v w
1/ 2/7
B
4/5 3/6
x y z
(h)
u v w
1/ 2/7
F B
4/5 3/6
x y z
(i)
u v w
1/8 2/7
F B
4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/
x y z
(n)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/11
x y z
(o)
u v w
1/8 2/7 9/12
F B C B
4/5 3/6 10/11
x y z
(p)

Running time of DFS

- Steps 1 and 2 (Initialization steps): O(n) time.
- DFS-Visit is called exactly once for each node.
- The call DFS-Visit(v) takes O (degree(v)) time.
- So, total time for all calls to DFS-Visit is

$$O\left(\sum_{v \in V} \mathrm{degree}(v)\right) = O(m).$$

- the overall running time of DFS is O(n + m).

Classification of edges into groups

- A <u>tree edge</u> is one in the depth-first forest
- A <u>back edge</u> (u, v) connects a vertex u to its ancestor v in the DF tree (includes self-loops)
- A <u>forward edge</u> is a nontree edge connecting a node to one of its DF tree descendants
- A <u>cross edge</u> goes between non-ancestral edges within a DF tree or between DF trees
- See labels in DFS example

- Example use of this property:
  A graph has a cycle iff DFS discovers a back edge
  (application: deadlock detection)
- When DFS first explores an edge (u, v), look at v's color:
  color[v] == white implies tree edge
  color[v] == gray implies back edge
  color[v] == black implies forward or cross edge

DFS Application: Cycle Detection

- DFS can be used to find out whether a graph or a digraph contains a cycle.
- Consider a digraph. It has a cycle if and only if the graph has a back edge. The same holds for graphs.
- Run DFS
- Check the nature of every edge
- If there is a back edge, then the graph has a cycle.
- Cycle detection = deadlock detection

Theorem

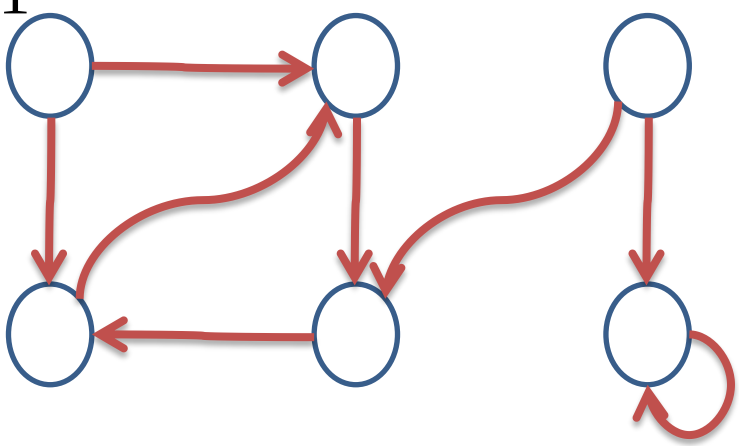
- Theorem: a directed graph G is acyclic iff a DFS of G yields no back edges:
  - => if G is acyclic, will be no back edges
    - Trivial: a back edge implies a cycle
  - <= if no back edges, G is acyclic
    - Proof by contradiction: G has a cycle $\Rightarrow \exists$ a back edge
      - Let *v* be the vertex on the cycle first discovered, and *u* be the predecessor of *v* on the cycle
      - When *v* discovered, whole cycle is white
      - Must visit everything reachable from *v* before returning from DFS-Visit()
      - So path from u$\rightarrow$v is gray$\rightarrow$gray, thus (*u*, *v*) is a back edge

DFS Application :Topological Sort

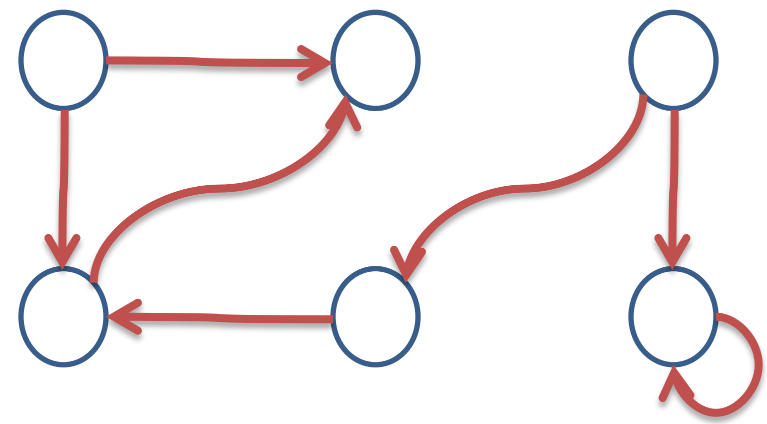
- Topological sort of a Directed Acyclic Graph (DAG):
- Definition: Given a DAG G, a topological sort of G is a linear arrangement of the nodes so that for each directed edge (u; v), u appears before v.
- A topological sort is a listing of the nodes on a line so that each directed edge goes from left to right.
- Such an ordering is not possible if the directed graph contains a cycle.
- Topological sort is used in situations where a set of events needs to be ordered given some precedence constraints.

DAG(Directed Acyclic Graph)

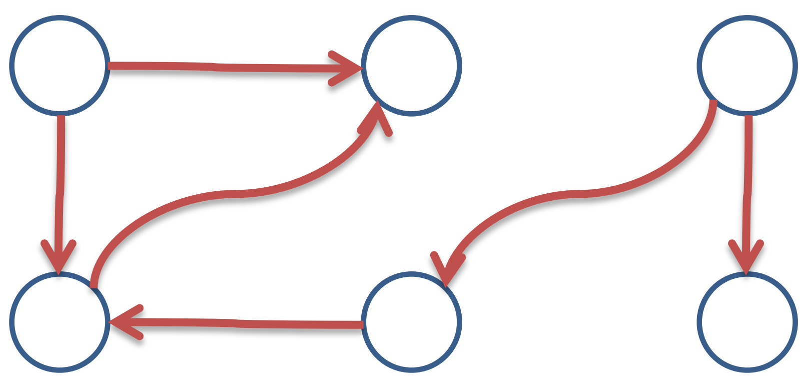
G1



G2



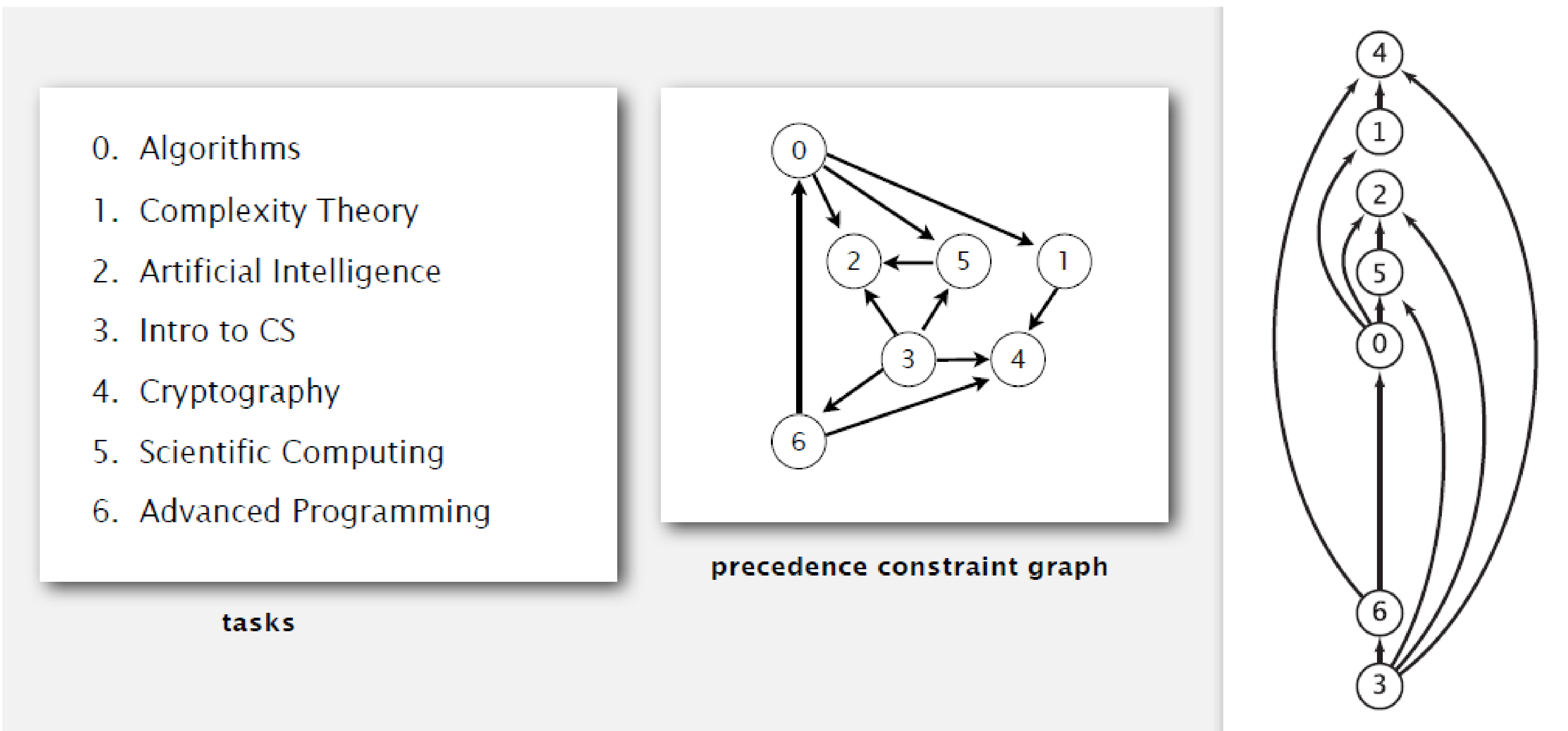
G3



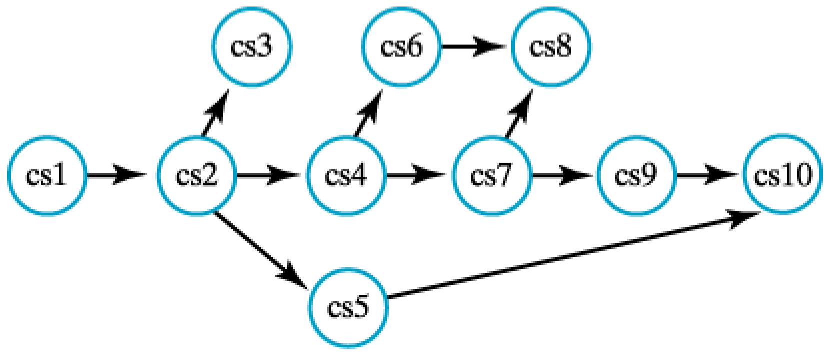
Which graph is DAG?

Precedence scheduling

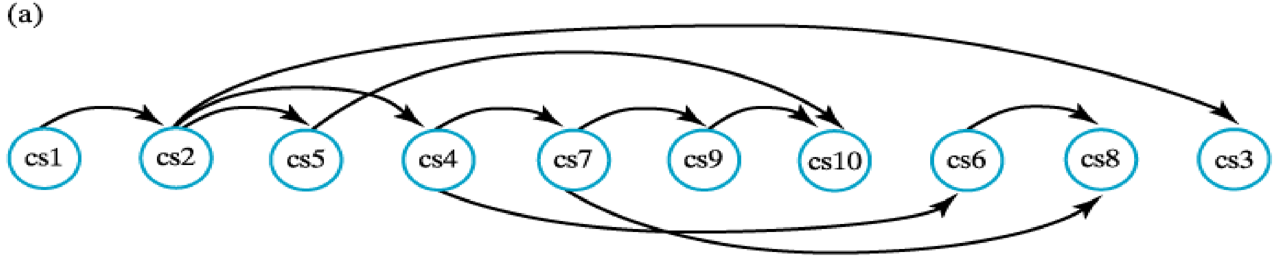
- Goal: Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?
- Digraph model: vertex = task, edge = precedence constraint.



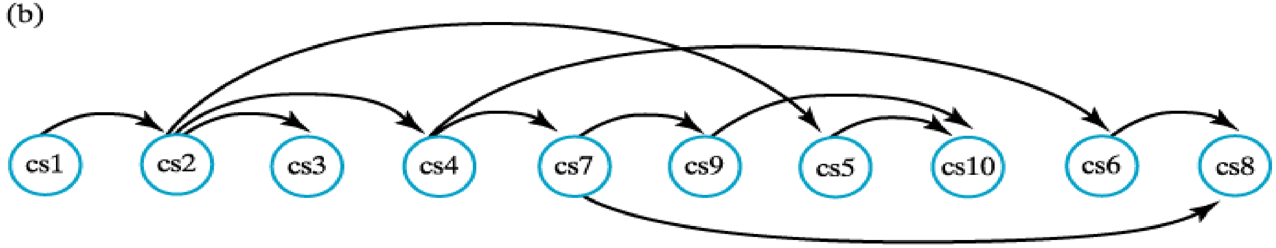
Topological Order Example



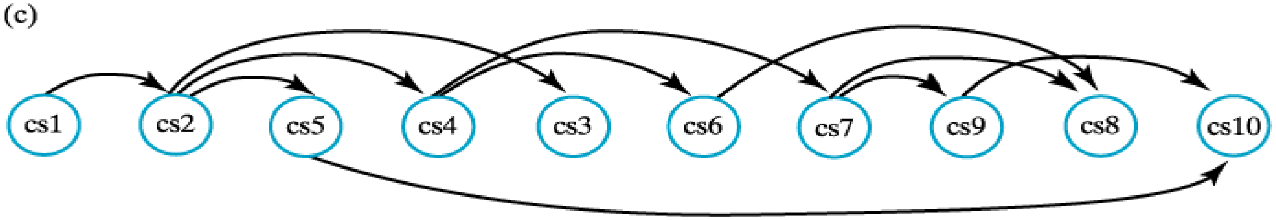
(a)



(b)



(c)



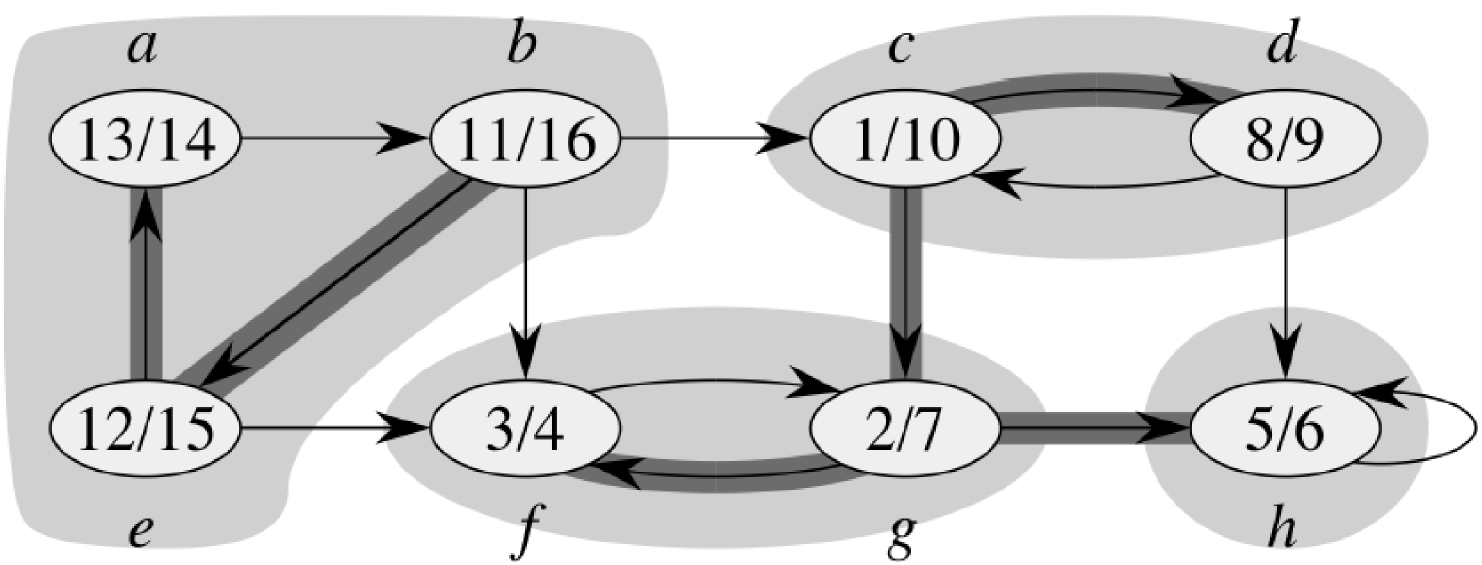
Topological Sort

- How to topological sort a DAG?
  1. Call DFS algorithm on DAG G
  2. As each vertex is finished, insert it to the front of a linked list
  3. Return the linked list of vertices.

- Thus topological sort is a descending sort of vertices based on DFS finishing times
- What is the time complexity?

DFA Application:
Strongly Connected Components(SCC)

- Given a directed graph G = (V, E), a strongly connected component (SCC) of G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices $u, v \in C$, u is reachable from v and v is reachable from u



SCC Algorithm

1. Call DFS algorithm on G
2. Compute G^T
3. Call DFS algorithm on G^T, looping through vertices in order of decreasing finishing times from first DFS call
4. Each DFS tree in second DFS run is an SCC in G

* Transpose of G is denoted by G^T
  G^T is simply G with edges reversed

SCC Application

- packaging software modules
- Software module dependency digraphs construct directed graph of which modules call which other modules
- An SCC is a set of mutually interacting modules
- pack together those in the same SCC