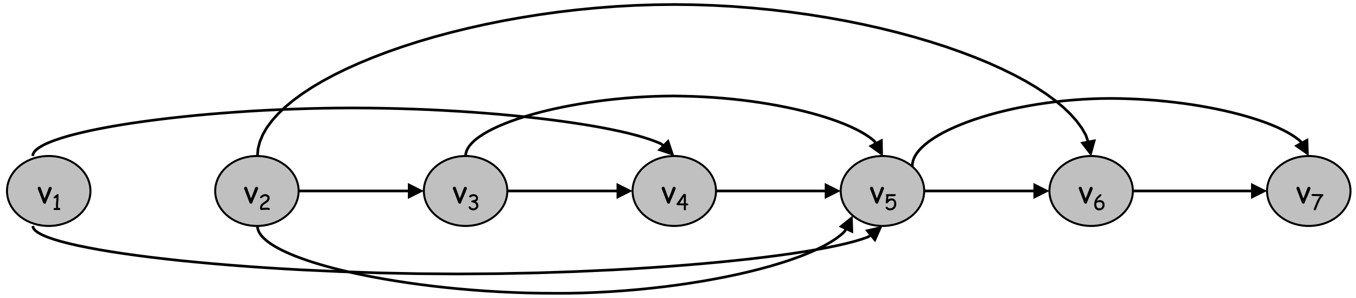
3.6 DAGs and Topological Ordering

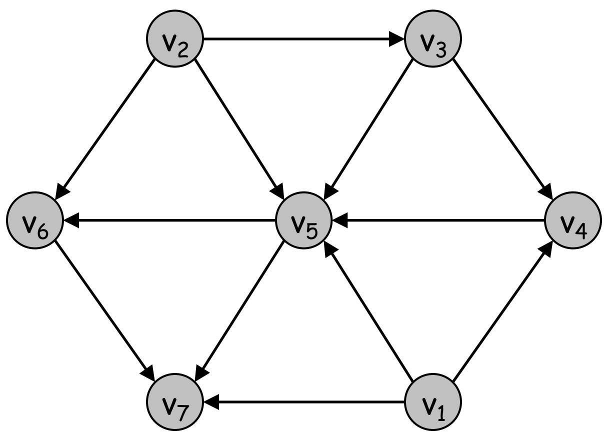


Directed Acyclic Graphs

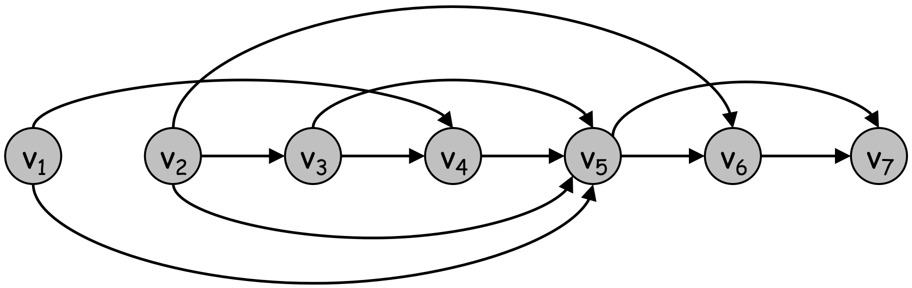
Def. A DAG is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j.

Def. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.

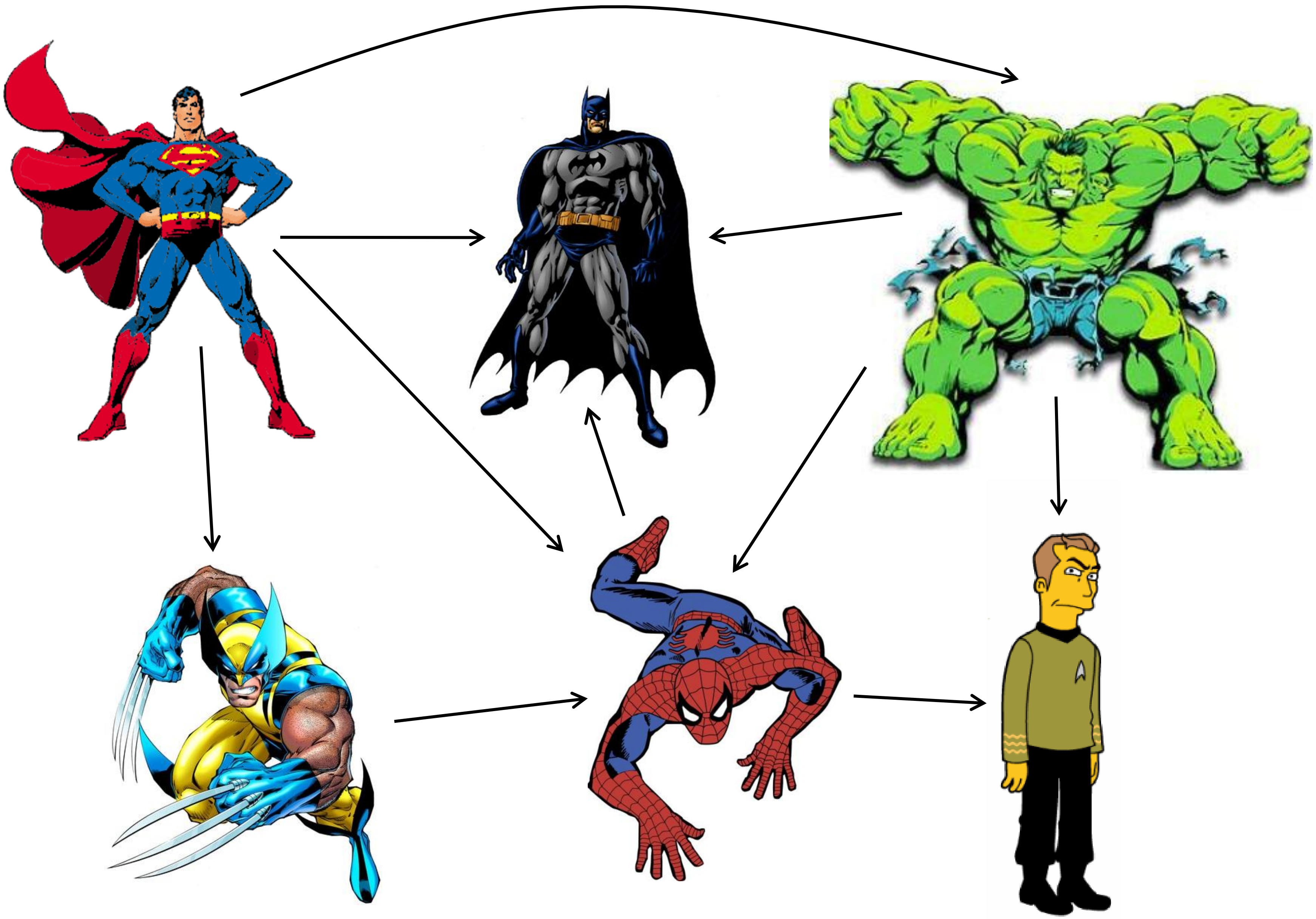


a DAG

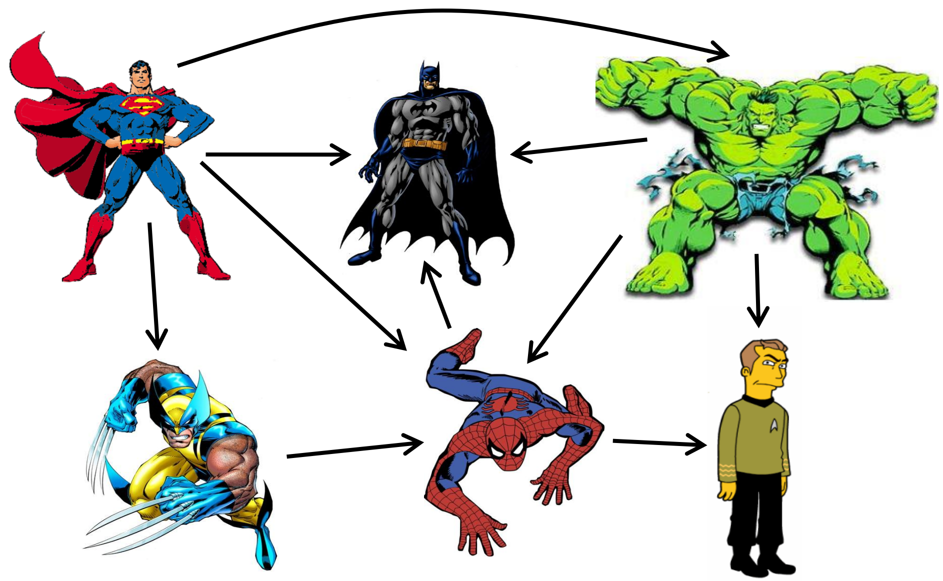


a topological ordering

Superhero Strength DAG



Superhero Strength Ordering

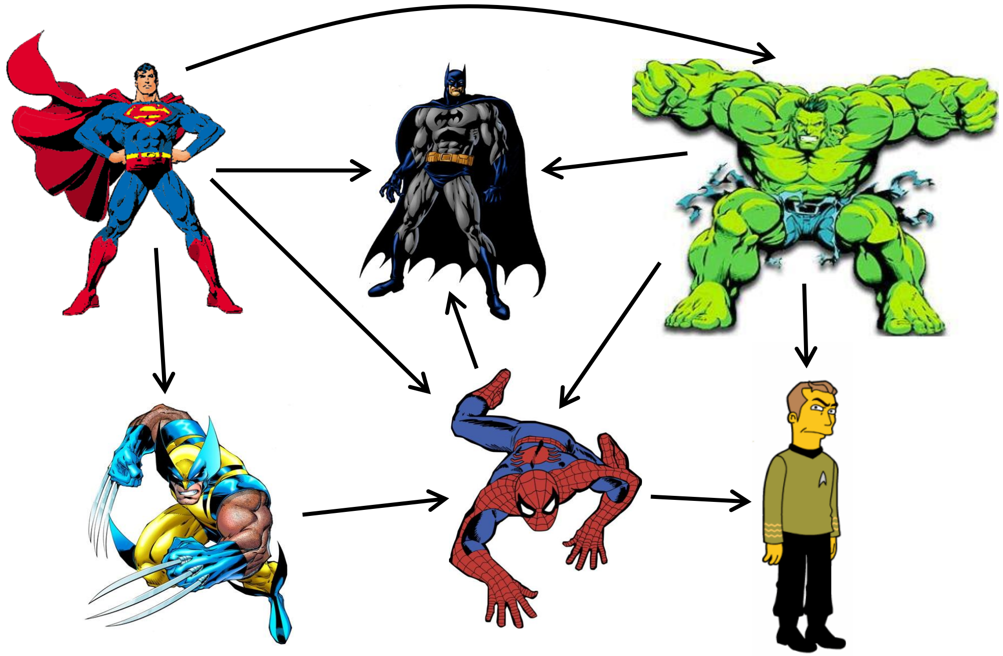


Precedence Constraints

Precedence constraints. Edge (v_i, v_j) means task v_i must occur before v_j.

Applications.

- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

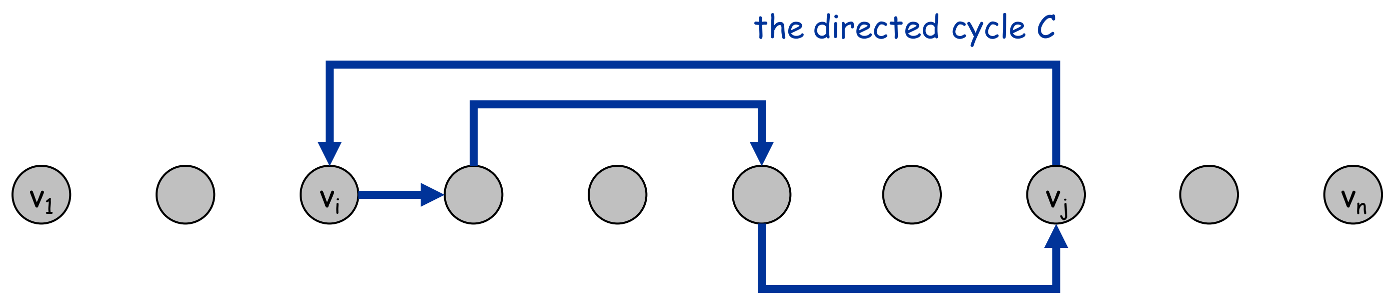


Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



the supposed topological order: $v_1, \ldots, v_n$

Lemma. If G has a topological order, then G is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

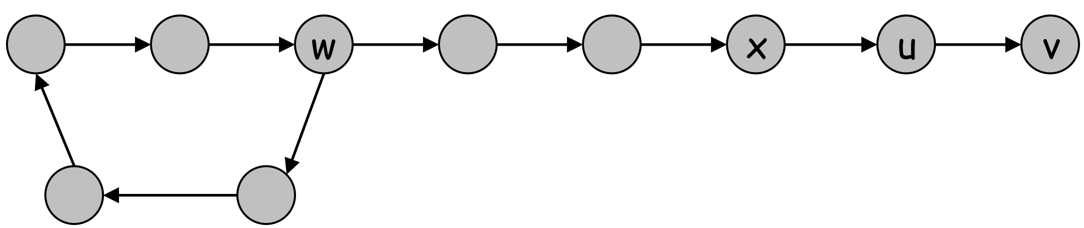
Superman Lemma

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



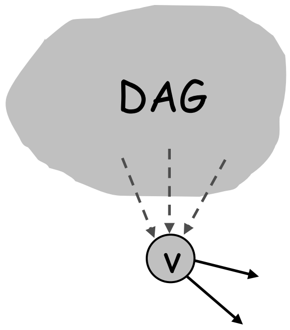
Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a topological ordering.

Pf. (by induction on n)

- Base case: true if n = 1.
- Given DAG on n > 1 nodes, find a node v with no incoming edges.
- G - { v } is a DAG, since deleting v cannot create cycles.
- By inductive hypothesis, G - { v } has a topological ordering.
- Place v first in topological ordering; then append nodes of G - { v }
- in topological order. This is valid since v has no incoming edges. ▪

To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of $G-\{v\}$
  and append this order after v



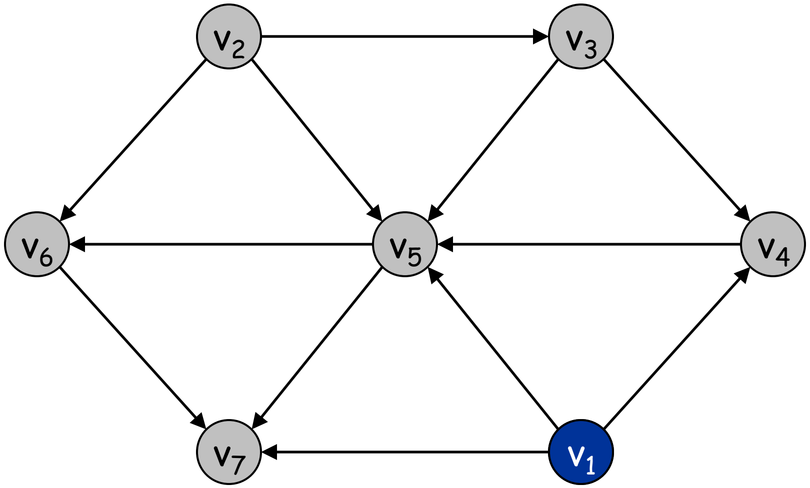
Topological Sorting Algorithm: Running Time

Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

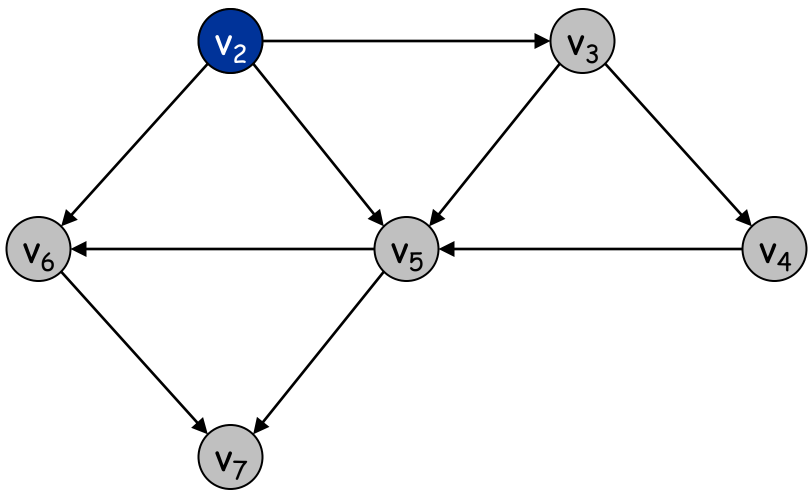
- Maintain the following information:
  - `count[w]` = remaining number of incoming edges
  - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
  - remove v from S
  - decrement `count[w]` for all edges from v to w, and add w to S if c `count[w]` hits 0
  - this is $O(1)$ per edge ▪

Topological Ordering Algorithm: Example



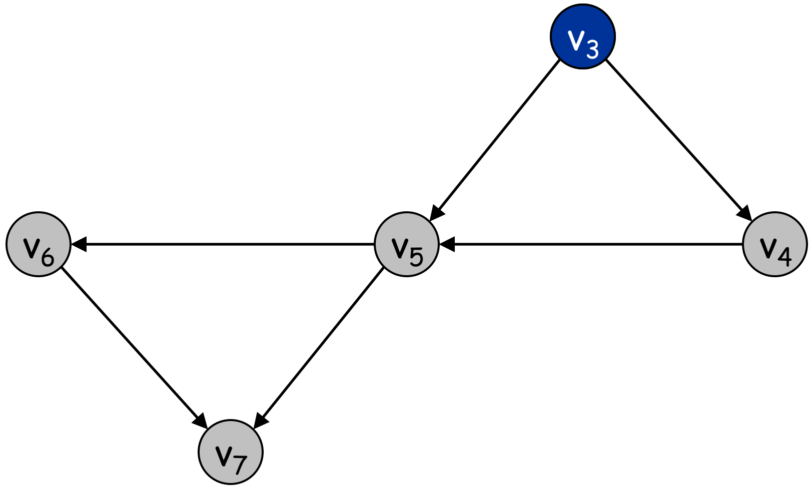
Topological order:

Topological Ordering Algorithm: Example



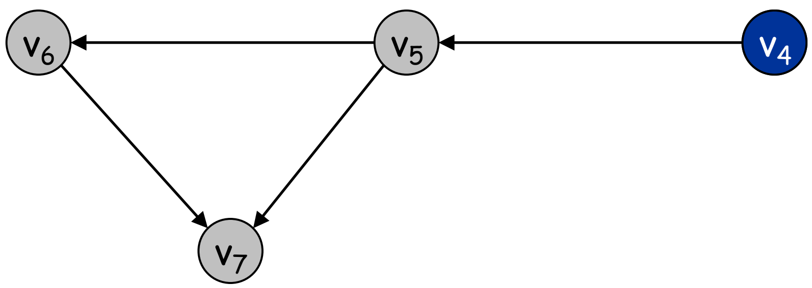
Topological order: v_1

Topological Ordering Algorithm: Example



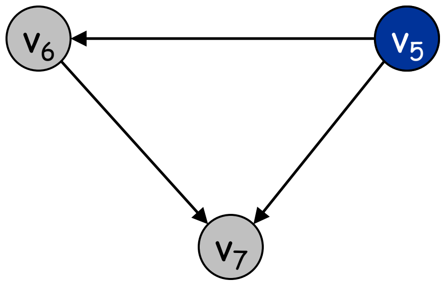
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



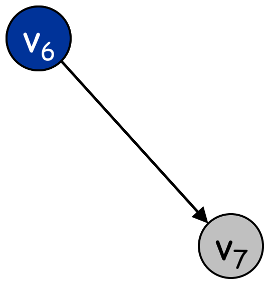
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

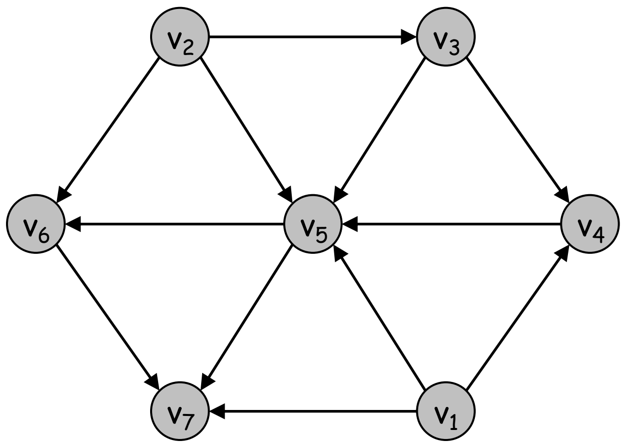


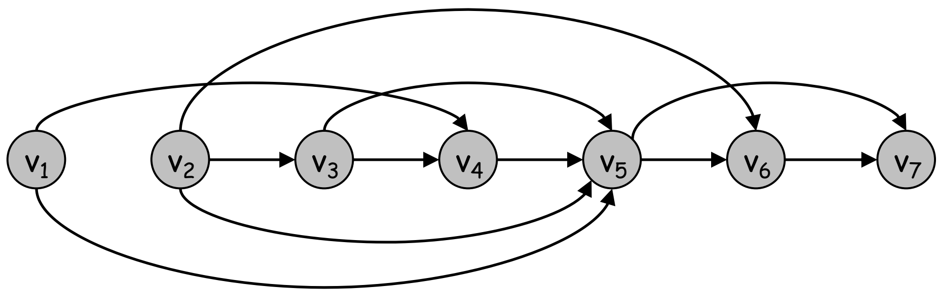
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

Topological order: v_1, v_2, v_3, v_4, v_5, v_6

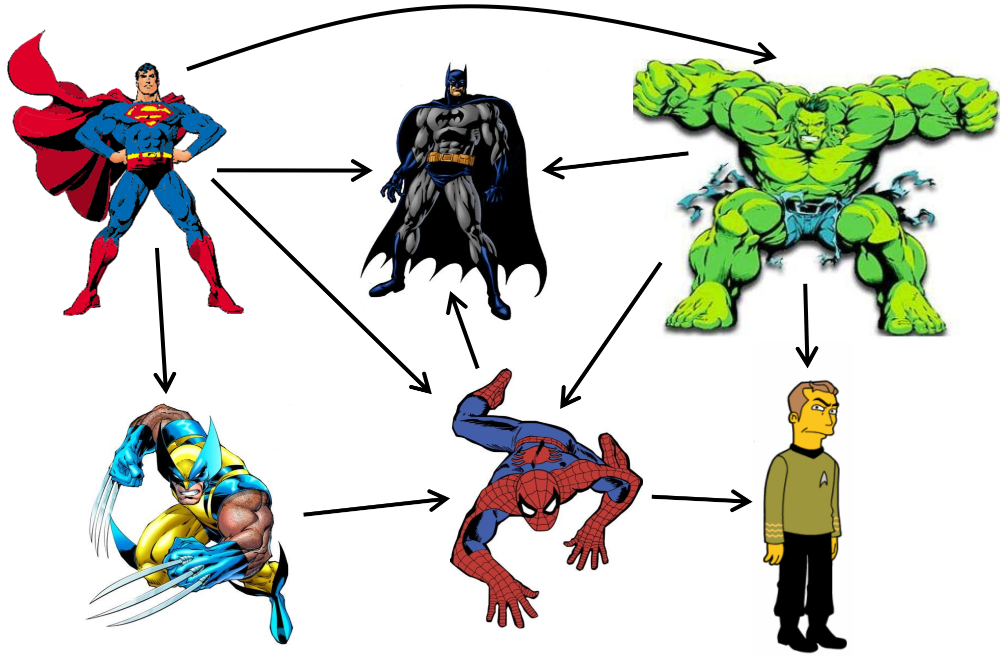
Topological Ordering Algorithm: Example



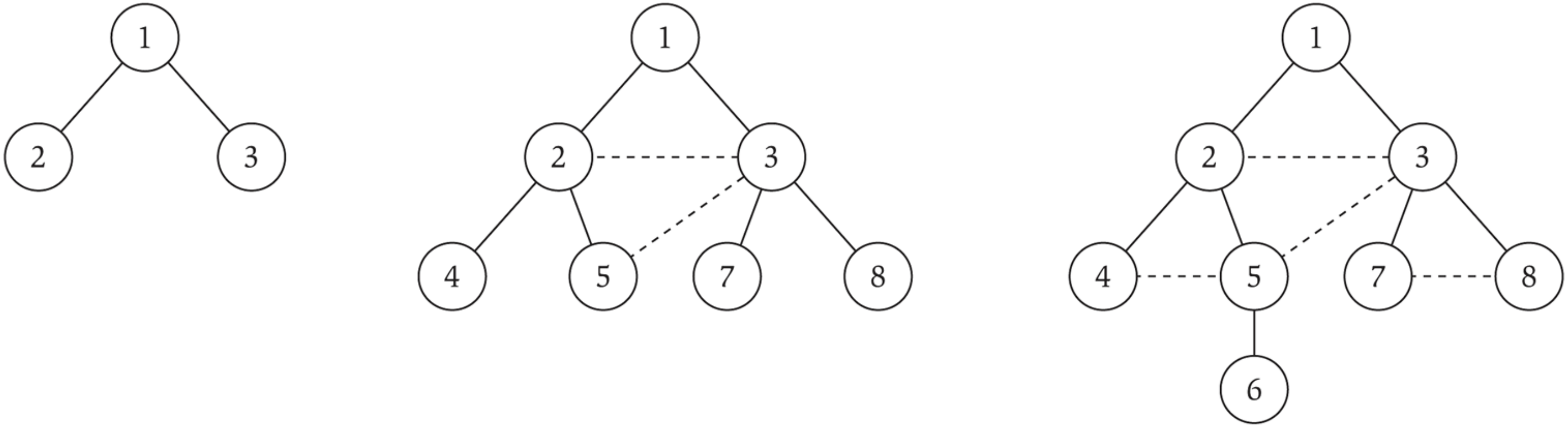


Topological order: $v_1, v_2, v_3, v_4, v_5, v_6, v_7$.

Superhero Strength Ordering



Some meditations on search

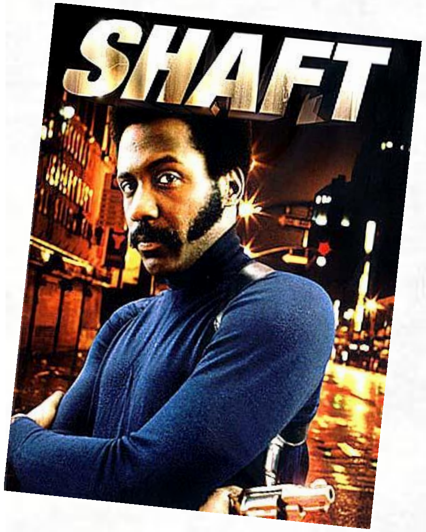


(a) (b) (c)

Depth-First Search

```
DepthFirstSearch (graph G, vertex v):
   count := 0;
   for each vertex w in G do
        Explored(w) = false;
   DFS(v);

procedure DFS(vertex v):
   Explored(v) = true;
   count++;
   d[v] := count;
   for each neighbor w of v do
      if !Explored(w) then
          Parent(w) = v;
          DFS(w);
```



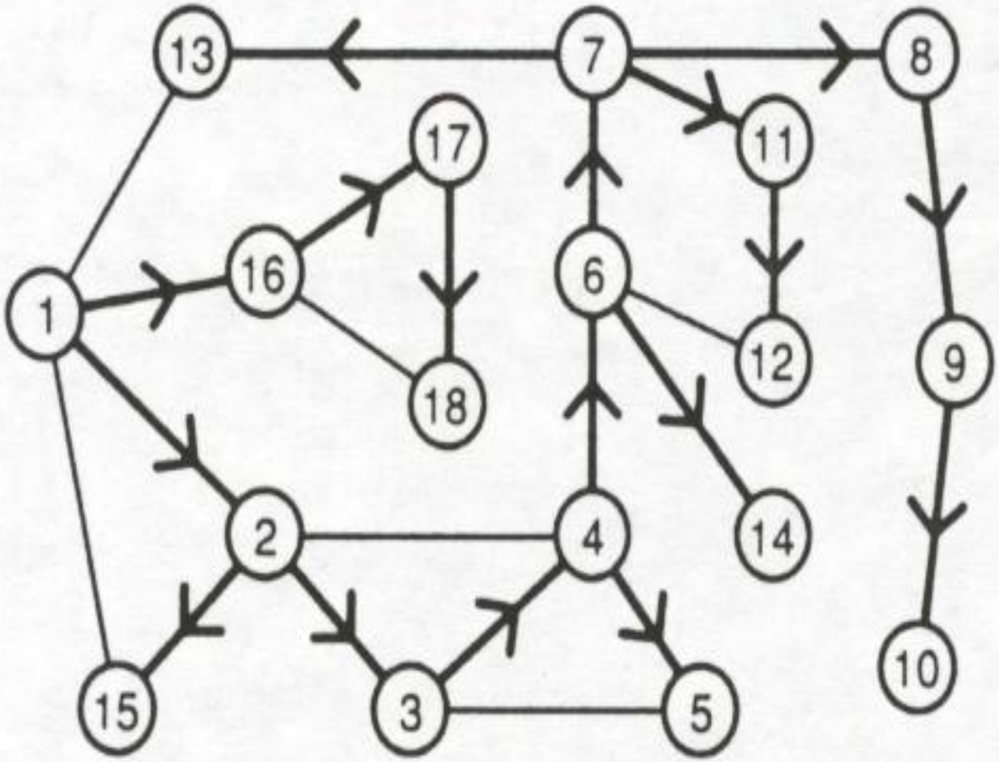


Figure 12.7 Depth-first search in an undirected graph. The search begins at the vertex labelled 1, and the vertices are labelled in the order they are encountered during the search. Followed edges are drawn with heavy lines and skipped edges with light lines; the arrows on the followed edges indicate the direction in which the edge was followed. When the skipped edges are deleted, the result is a tree; if vertex 1 is taken as the root of the tree then skipped edges join only ancestrally related vertices.

Important Properties of DFS

For a given recursive call DFS(u), all nodes that are marked “Explored” between the invocation and end of this recursive call are descendents of u in the DFS tree T.

Most important property of DFS on undirected graphs

Every edge is either a tree edge or an edge between an ancestor and a descendent in the tree.

Let (u,v) be an edge.
Suppose the call to DFS(u) occurs before the call to DFS(v).
Then before DFS(u) completes, v will be visited.

DFS Analysis

Running time: O(n+m)

- call DFS on each node exactly once (afterwards Explored is true)
- each edge is examined twice (once from each endpoint)

assuming adjacency list representation of graph

IN CLASS EXERCISE!!!

Network reliability -- how would you determine efficiently if there is a single point of failure in your network?

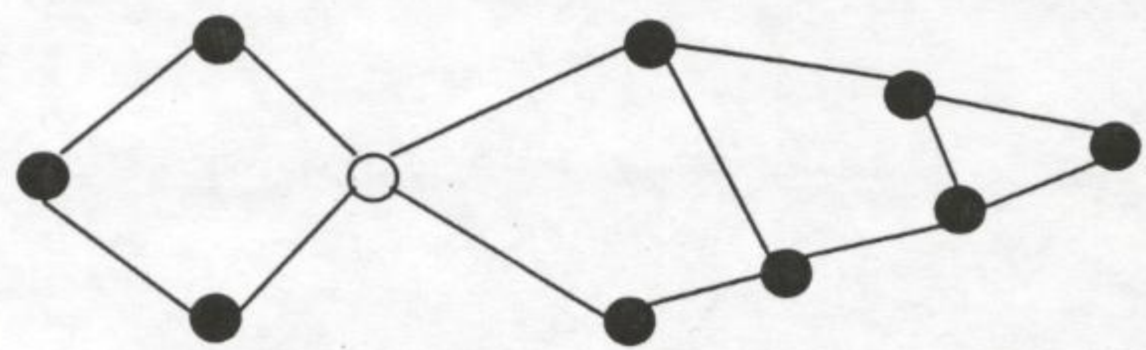


Figure 4–10. An articulation vertex is the weakest point in the graph

Network reliability -- how would you determine efficiently if there is a single point of failure in your network?

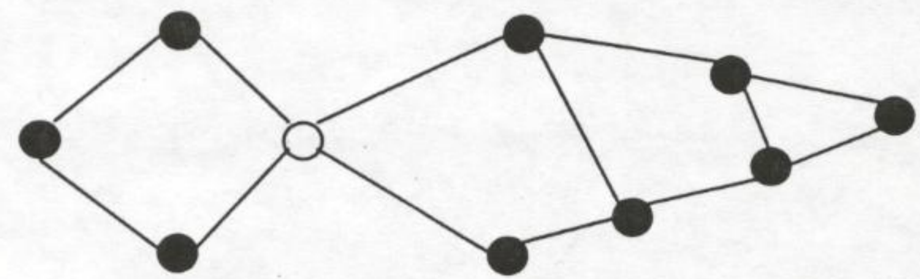


Figure 4–10. An articulation vertex is the weakest point in the graph

```
DepthFirstSearch (graph G, vertex v):
   count := 0;
   for each vertex w in G do
        Explored(w) = false;
   DFS(v);

procedure DFS(vertex v):
   Explored(v) = true;
   count++;
   d[v] := count;
   for each neighbor w of v do
      if !Explored(w) then
          Parent(w) = v;
          DFS(w);
```

Depth-First Search

```
DepthFirstSearch (graph G, vertex v):
   count := 0;
   for each vertex w in G do
        Explored(w) = false;
   DFS(v);

procedure DFS(vertex v):
   Explored(v) = true; count++;
   d[v] := count; low[v] := d[v];
   for each neighbor w of v do
      if !Explored(w) then
          Parent(w) = v; DFS(w); low[v] := min(low[v], low[w]);
          if low[w] >= d[v]  then  v is an articulation point;
      else  low[v] := min(low[v], d[w]);
    count++;
```

Modeling the Problem

An art, but key to applying algorithm design techniques to real-world problems.

Most algorithms designed to work on rigorously defined abstract structure.

Another Problem

You need to coordinate the motion of a pair of mobile robots in a given building. Each has a radio transmitter it uses to communicate with a base station. If the two robots get closer than distance r from each other, there are problems with interference among their transmitters.

How do you plan the motion of the robots so each gets from its starting point to its intended destination as quickly as possible, but in the process, the robots don't come close enough together to cause interference?