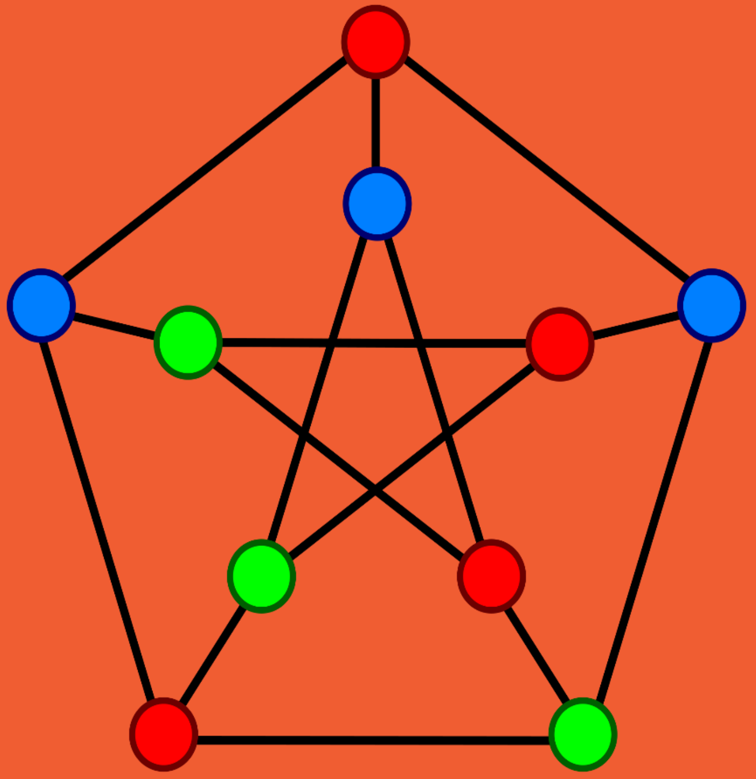
Lecture 2: Graphs (Adv.)



Joachim Gudmundsson

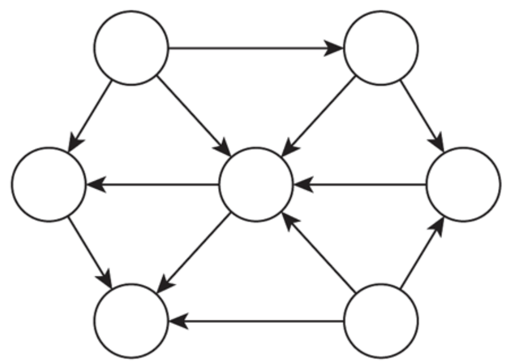
THE UNIVERSITY OF SYDNEY

3.5 Connectivity in Directed Graphs

Directed Graphs

Directed graph. G = (V, E)

– Edge (u, v) goes from node u to node v.



Example. Web graph - hyperlink points from one web page to another.

– Directedness of graph is crucial.
– Modern web search engines exploit hyperlink structure to rank web pages by importance.

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Graph search. BFS and DFS extend naturally to directed graphs.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

```
def BFS(G,s):

  layers = []
  current_layer = [s]
  next_layer = []
  “mark every vertex except s as not seen”
  while “current_layer not empty” :
    layers.append(current_layer)
    for u in current_layer:
        for v in “neighborhood of u”:
           if “haven’t seen v yet”:
              next_layer.append(v)
              “mark v as seen”
    current_layer = next_layer
    next_layer = []

  return layers
```

Strong Connectivity

Definition: Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

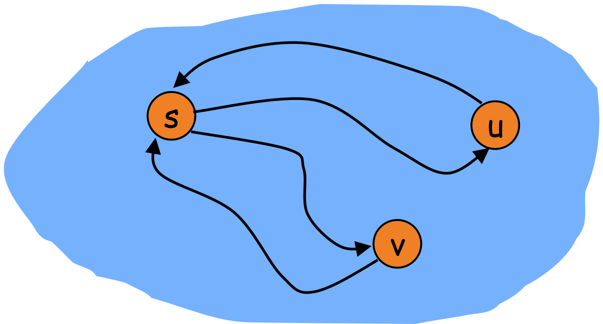
Definition: A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma: Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Proof: ($\Rightarrow$) Follows from definition.

($\Leftarrow$) Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path.

▪



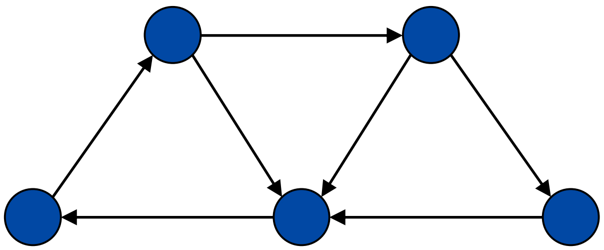
Strong Connectivity: Algorithm

Theorem: Can determine if G is strongly connected in O(m + n) time.

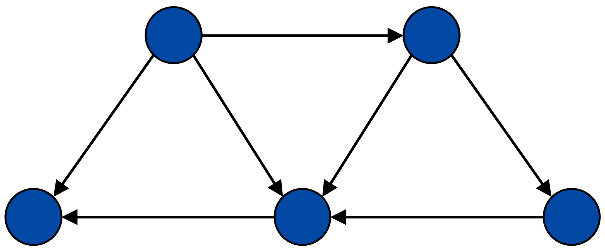
Proof:

– Pick any node s.
– Run BFS from s in G.
– Run BFS from s in G^{rev}.
– Return true iff all nodes reached in both BFS executions.
– Correctness follows immediately from previous lemma. ▪

reverse orientation of every edge in G



strongly connected



not strongly connected

Strong Connectivity

- Consider a graph G and let S1 and S2 be two strongly connected components in G of maximal size. Are S1 and S2 disjoint?

- Can we compute all the strongly connected components of a graph G efficiently?

Strong Connectivity

Algorithm by Kosaraju 1978 (unpublished)

STRONGLY-CONNECTED-COMPONENTS (G)

1. **Call** DFS(G) to compute finishing times f[u] for all *u*.
2. **Compute** G^{rev}
3. **Call** DFS(G^{rev}), but in the main loop, consider vertices in order of decreasing f[*u*] (as computed in first DFS)
4. **Output** the vertices in each tree of the depth-first forest formed in the second DFS as a separate strongly connected component.

Running time: O(n+m)

Correctness?

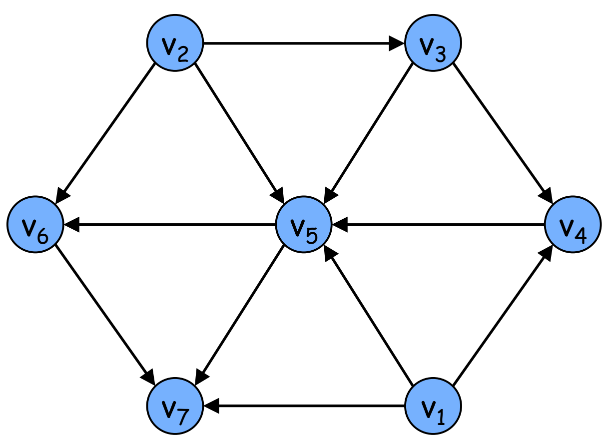
3.6 DAGs and Topological Ordering

Directed Acyclic Graphs (DAGs)

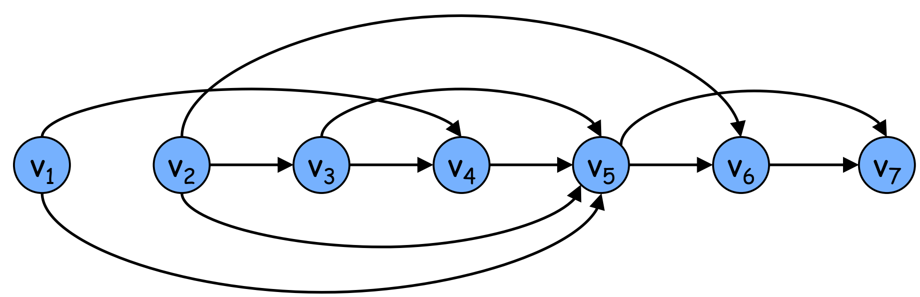
Definition: A DAG is a directed graph that contains no directed cycles.

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

Definition: 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 directed edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering

Precedence Constraints

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

Applications.

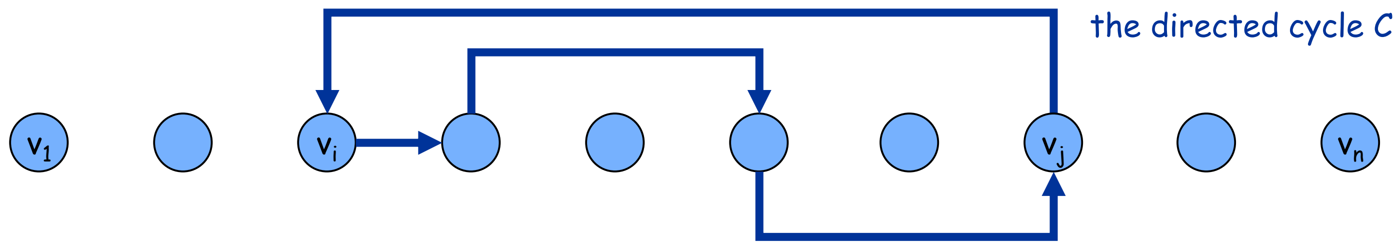
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

Directed Acyclic Graphs

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

Proof: (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 in C; 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: $v_1, \ldots, v_n$

Directed Acyclic Graphs

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

Question: Does every DAG have a topological ordering?

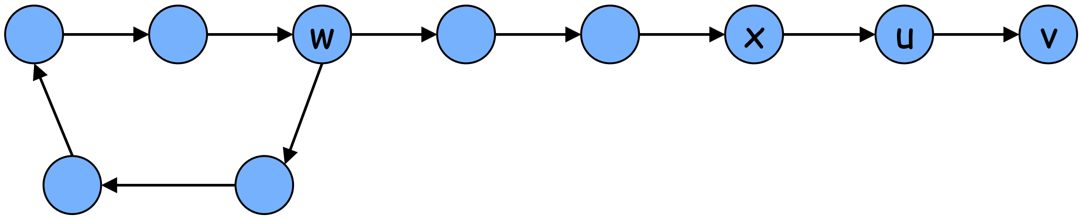
Question: If so, how do we compute one?

Directed Acyclic Graphs

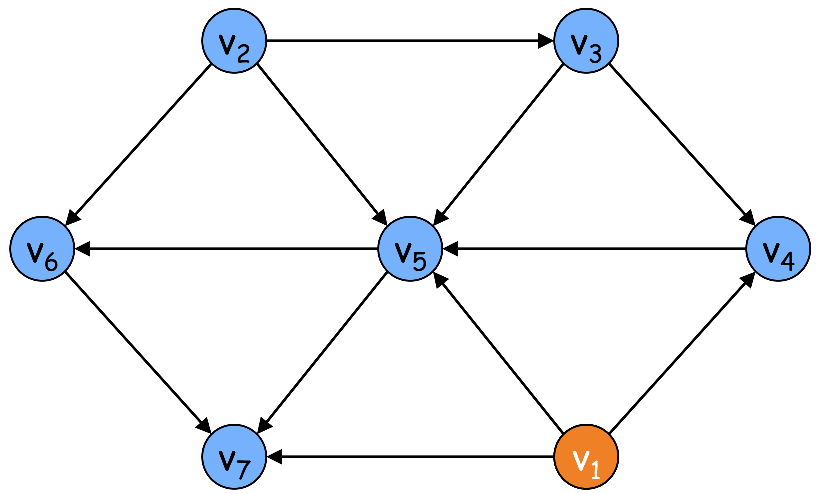
Lemma: If G is a DAG then G has a node with no incoming edges.

Proof: (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. ▪

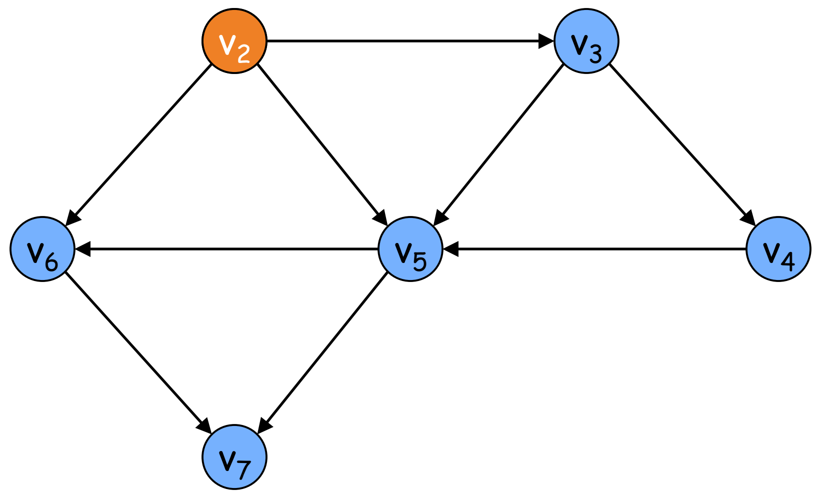


Topological Ordering Algorithm: Example



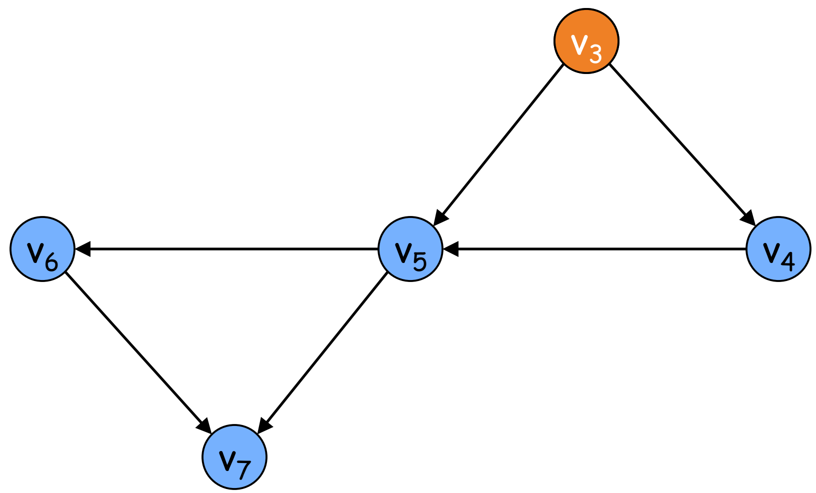
Topological order:

Topological Ordering Algorithm: Example



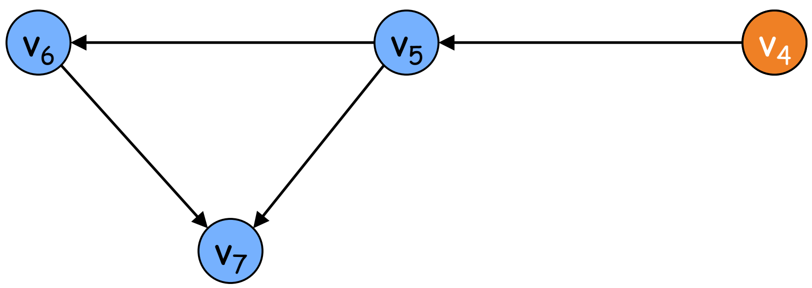
Topological order: v_1

Topological Ordering Algorithm: Example



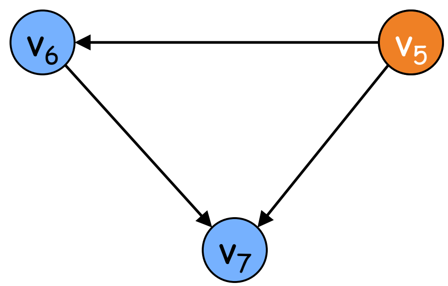
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



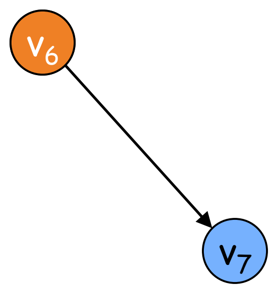
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

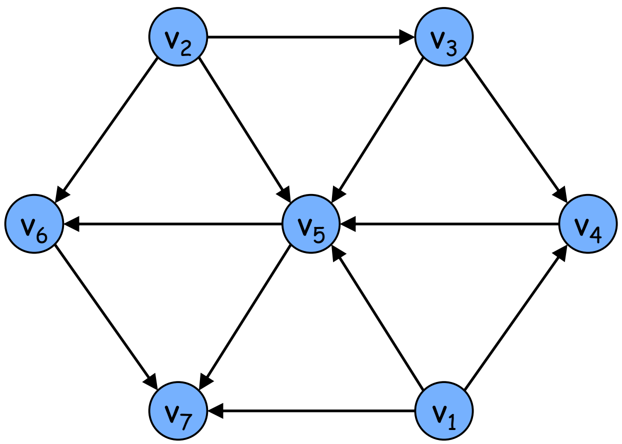


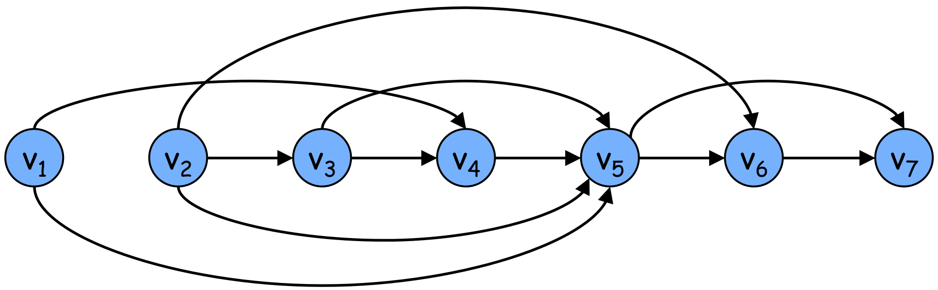
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

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

Topological Ordering Algorithm: Example





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

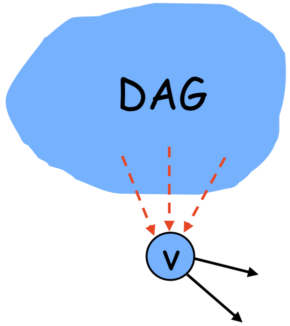
Directed Acyclic Graphs

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

Proof: (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

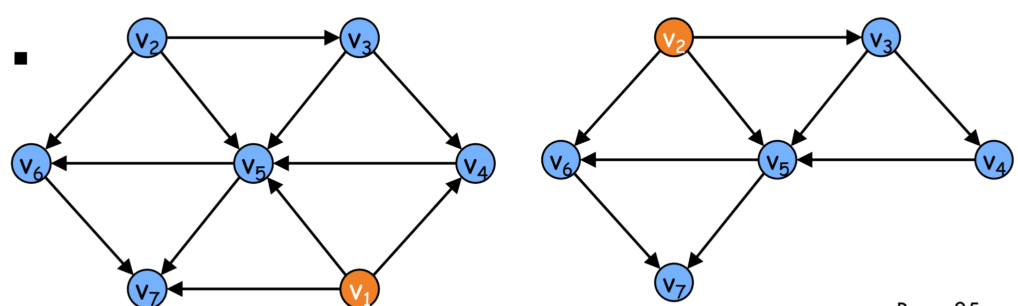


Topological Sorting Algorithm: Running Time

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

Proof:

- 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 ▪



Summary: Graphs

- Connectivity in directed graphs
- DAGs
- Topological sort