COMP 355
Advanced Algorithms

Graphs: Topological Sort
Chapter 3 (KT)

Rhodes College
—1848—

Graph Search Algorithms

BFS and DFS almost the same for directed and undirected graphs

BFS on directed graphs: still O(m + n)

- It is possible for node s to have a path to a node t even though t has no path s
- Computing the set of all nodes t with the property that s has a path to t

DFS on directed graphs: still O(m + n)

- At node u, recursively launches depth-first search, in order, for each node to which u has an edge

Strong Connectivity (Directed Graphs)

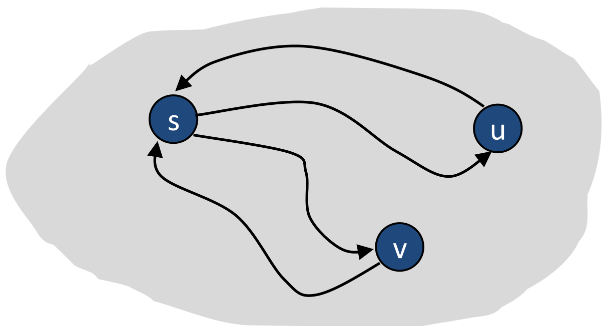
Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A directed 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.

- Pf. $\Rightarrow$ Follows from definition.
- Pf. $\Leftarrow$ 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. ▪

ok if paths overlap

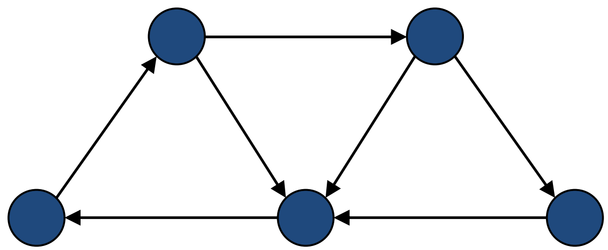


Strong Connectivity: Algorithm

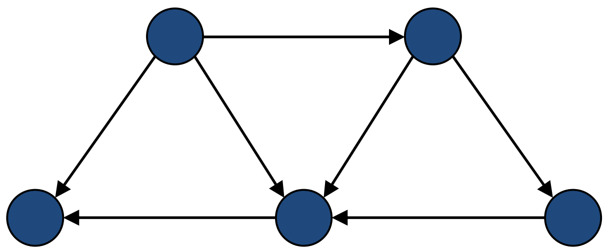
Theorem. Can determine if G is strongly connected in O(m + n) time.

Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}. reverse orientation of every edge in G
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.



strongly connected



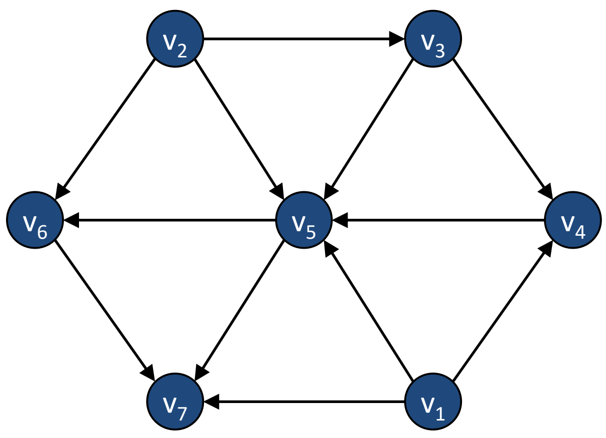
not strongly connected

Directed Acyclic Graphs

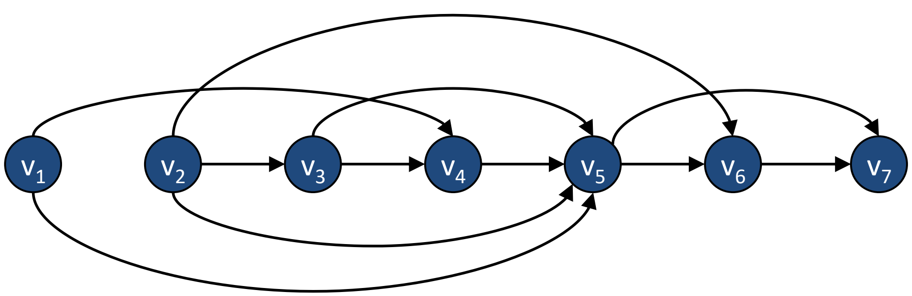
Def. An DAG is a directed graph that contains no directed cycles.

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

Precedence Constraints

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

Applications.

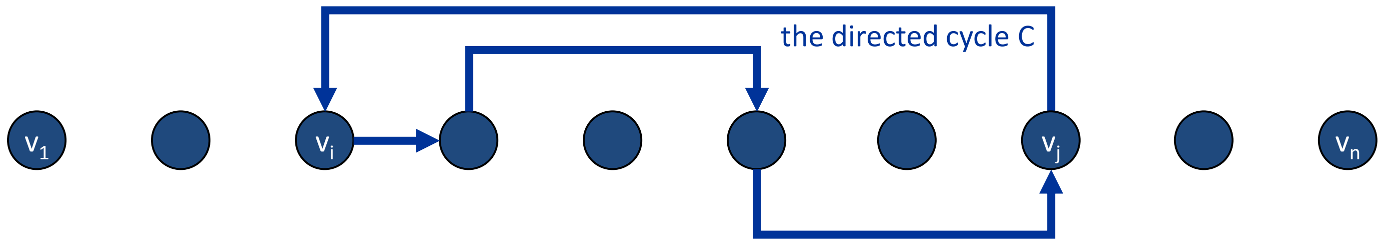
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j. Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

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.



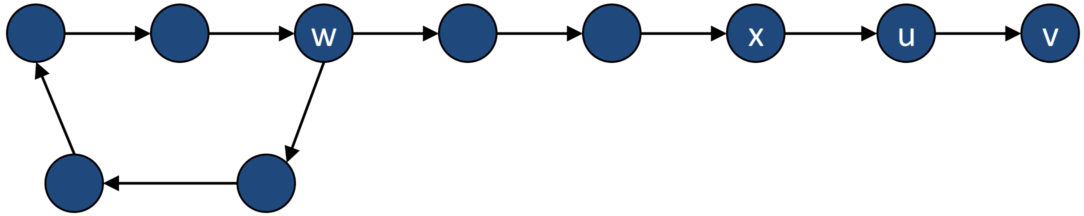
the supposed topological order: $v_1, \ldots, v_n$

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.



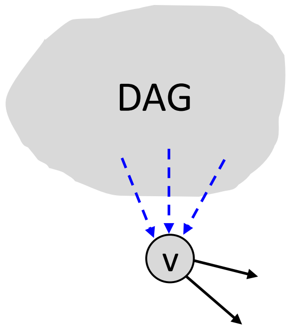
Directed Acyclic Graphs

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

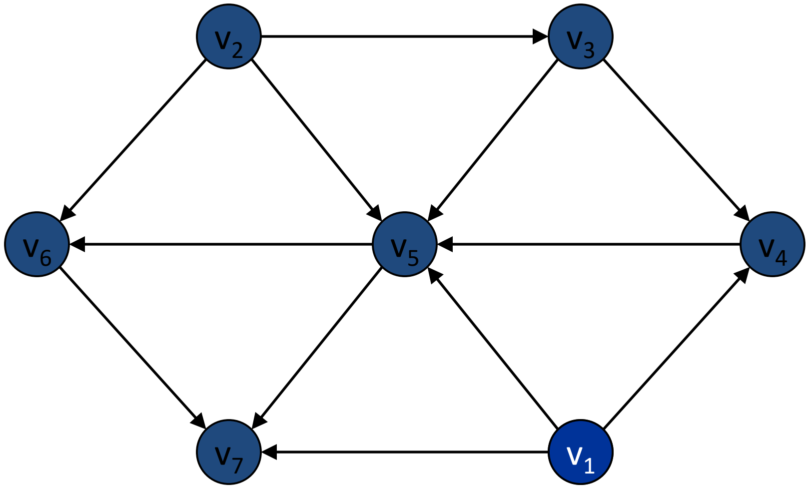
Pf. (by induction on n)

- 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
```

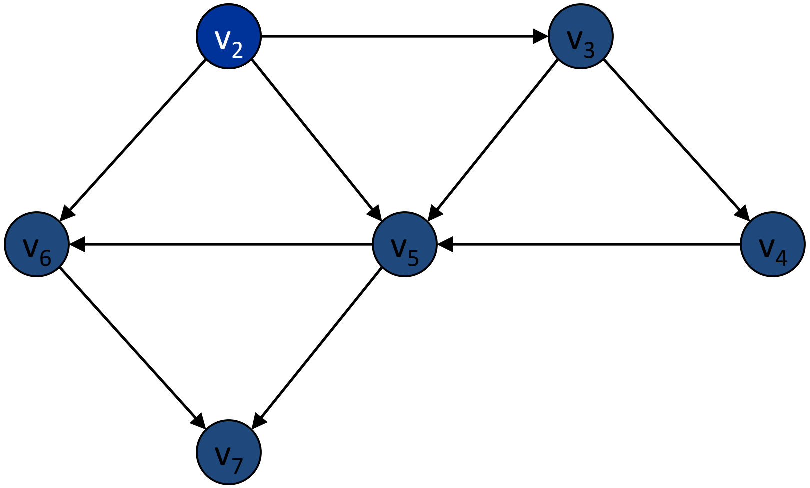


Topological Ordering Algorithm: Example



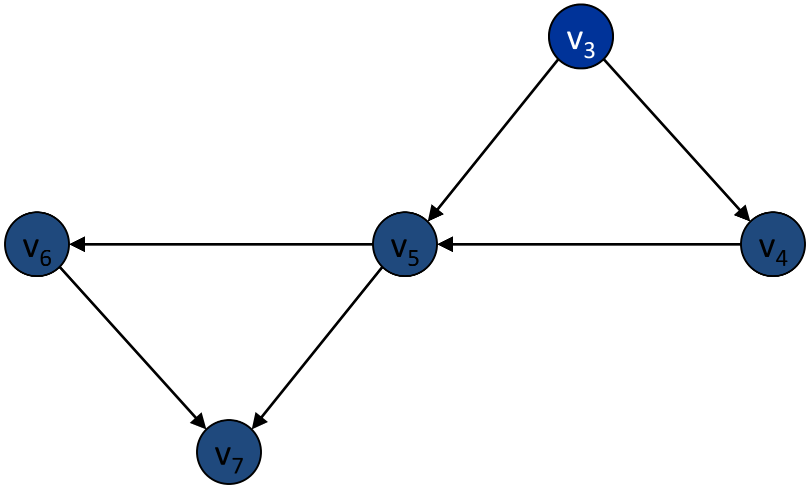
Topological order:

Topological Ordering Algorithm: Example



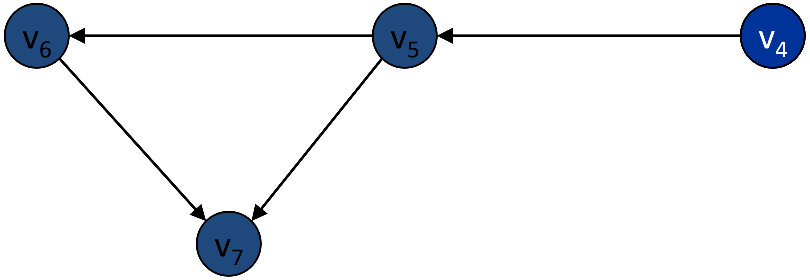
Topological order: v_1

Topological Ordering Algorithm: Example



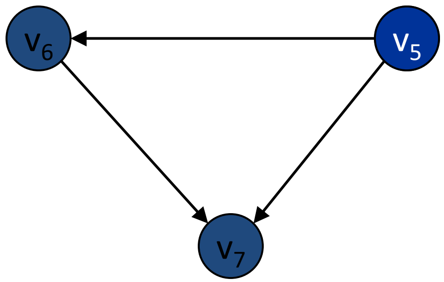
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



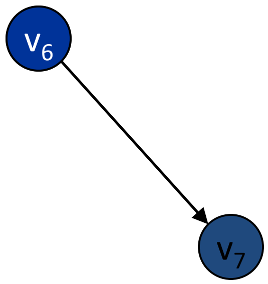
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

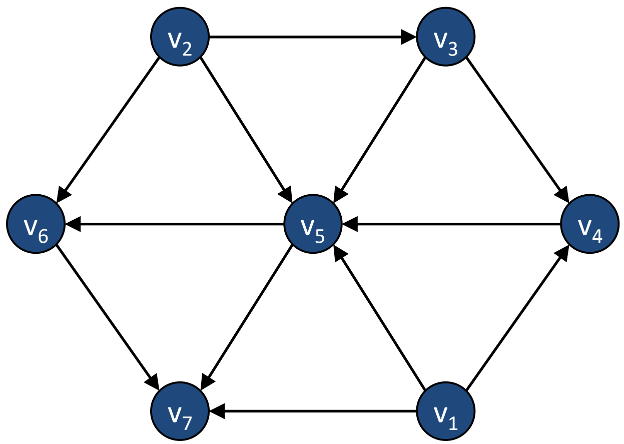


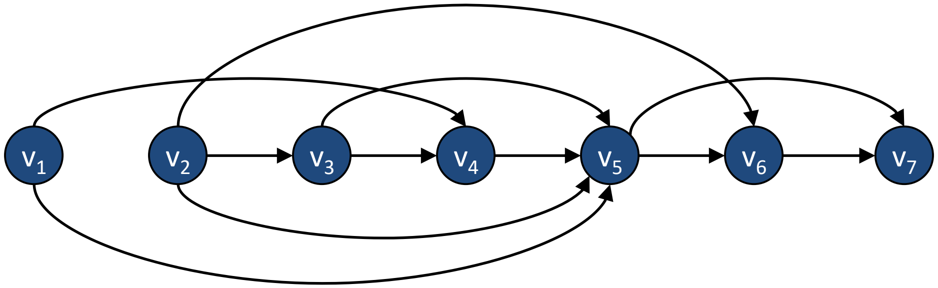
Topological order: v_1, v_2, v_3, v_4, v_5

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.

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 `count[w]` hits 0
  - this is $O(1)$ per edge