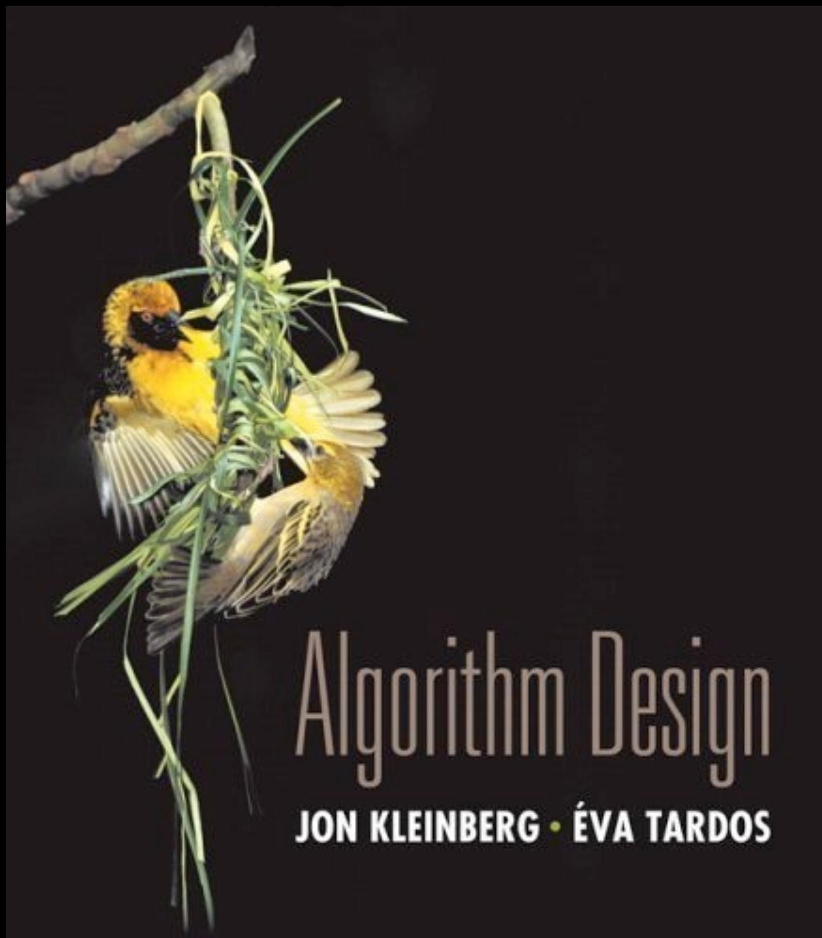
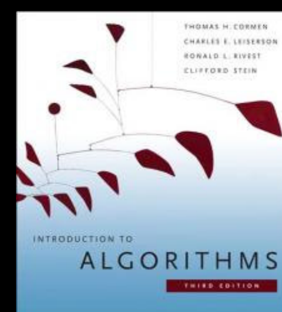


Chapter 3

Graphs

CLRS 12-13



Slides by Kevin Wayne.

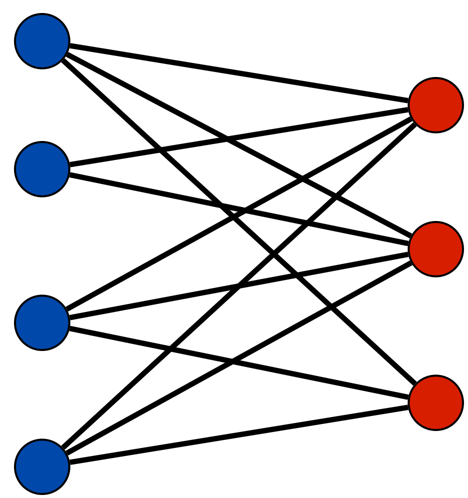
3.4 Testing Bipartiteness

Bipartite Graphs

Def. An undirected graph G = (V, E) is bipartite if the nodes can be coloured red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

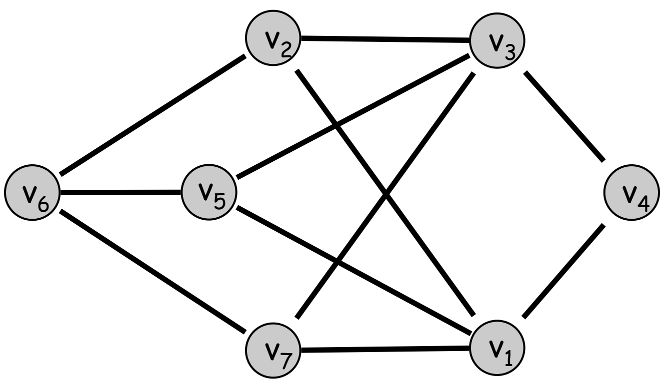


a bipartite graph

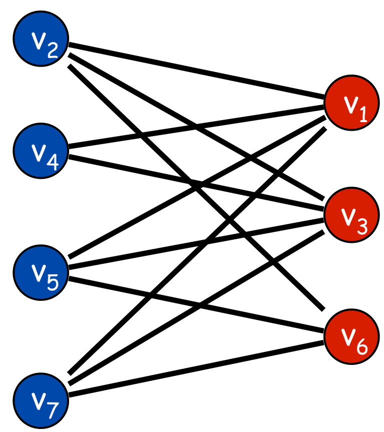
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand the structure of bipartite graphs.



a bipartite graph G

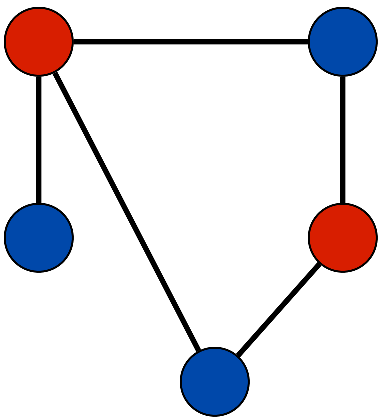


another drawing of G

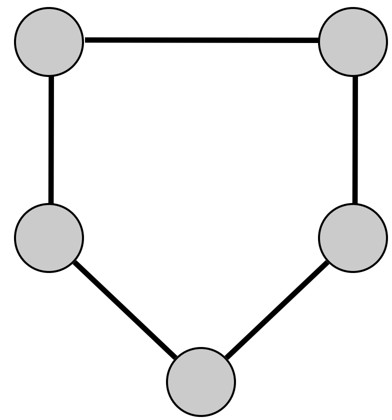
An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-colour the odd cycle, let alone G.



bipartite
(2-colorable)

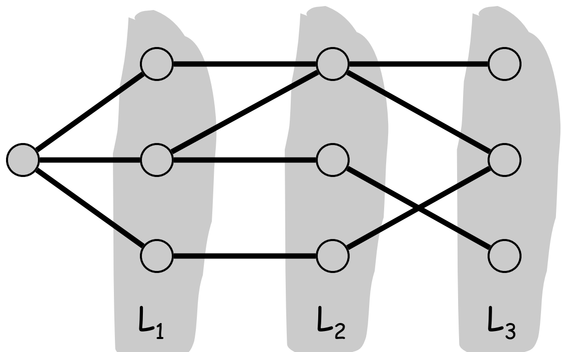


not bipartite
(not 2-colorable)

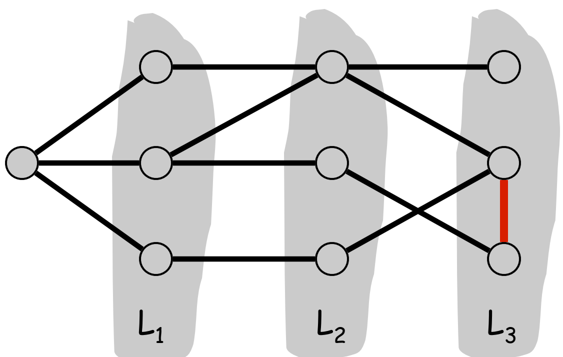
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

Bipartite Graphs

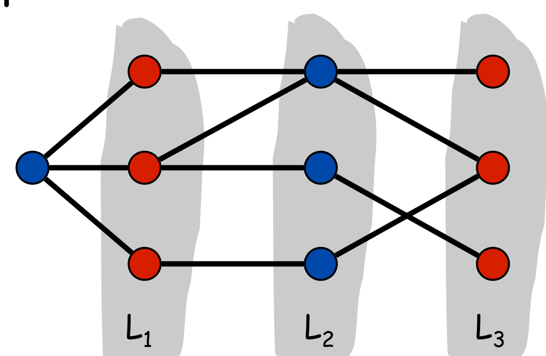
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in the same layer.
- By above property, this implies all edges join nodes on adjacent layers.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

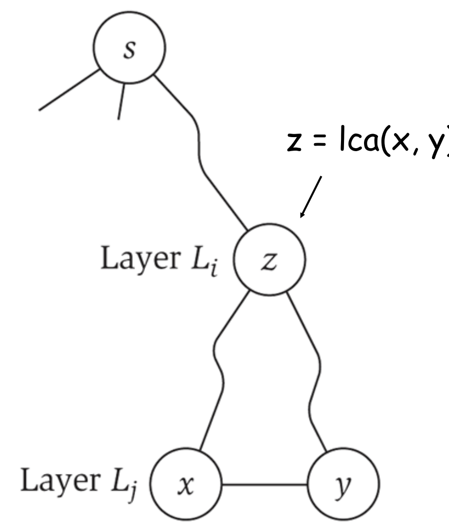
Lemma. Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

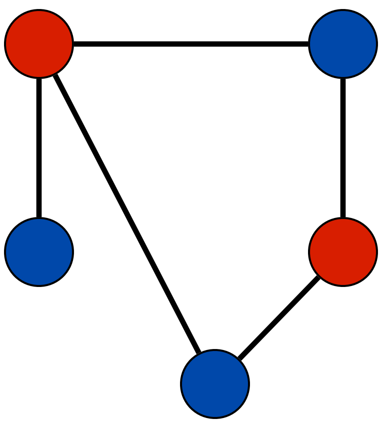
- Suppose (x, y) is an edge with x, y in same level L_j.
- Let $z = \text{lca}(x, y)$ = lowest common ancestor*.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path* from y to z, then path* from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd.



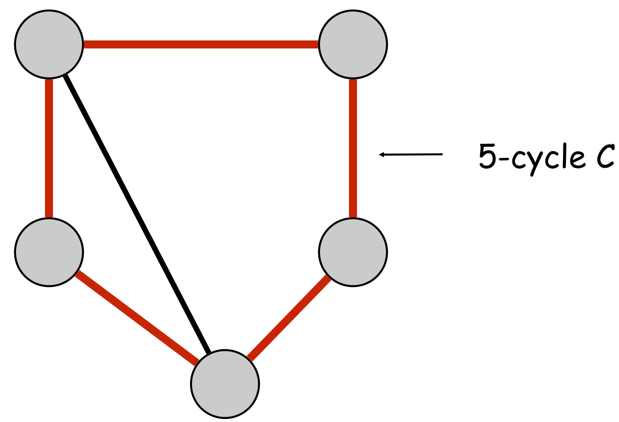
*<u>Consider only edges of the BFS tree.</u>

Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.



bipartite
(2-colorable)

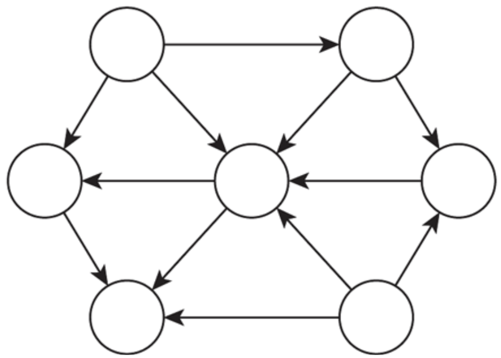


not bipartite
(not 2-colorable)

3.5 Connectivity in Directed Graphs

Directed Graphs

Directed graph. $G = (V, E)$
- Edge (u, v) goes from node u to node v.



Ex. 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 extends naturally to directed graphs.

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

Strong Connectivity

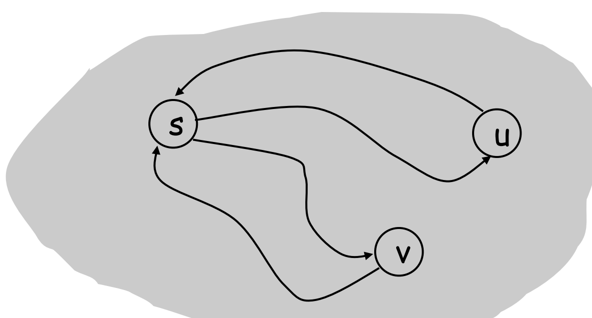
Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A 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

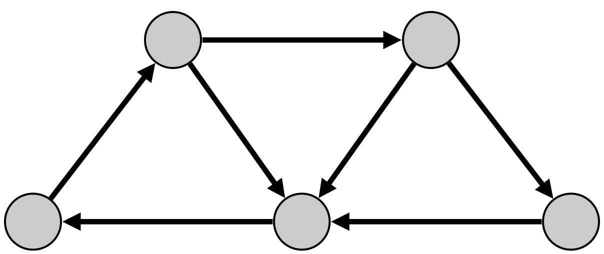


Strong Connectivity: Algorithm

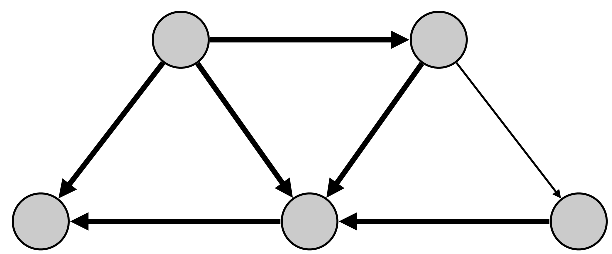
Theorem. Can determine if G is strongly connected in $O(m + n)$ time.
Pf.

- Pick any node s.
- Run BFS from s in G.
- Run BFS from s in G^{rev}.
- 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

3.6 DAGs and Topological Ordering

3.6 DAGs and Topological Ordering

What is the connection between computer science and algorithms?

I study CS and I hear a lot if you want to be a good programmer you must be good at algorithm,why ? and if it's true what algorithm should i read or study ?

Thomas Cormen, The C in CLRS.

Written Sep 12 · Upvoted by Siddarth Sampangi, UCSD B.S. in CS '14; UMass Amherst M.S. in CS '16, Bill Poucher, Baylor CS prof, ICPC Exec Director, Software: energy, synthetic genetics, etc., and Rohit RK

I'll tell you a little story. A true story.

Thomas Cormen, The C in CLRS.

Written Sep 12 · Upvoted by Siddarth Sampangi, UCSD B.S. in CS '14; UMass Amherst M.S. in CS '16, Bill Poucher, Baylor CS prof, ICPC Exec Director, Software: energy, synthetic genetics, etc., and Rohit RK

In the late 1970s and early 1980s, I worked at a startup that made systems for computer-aided design. Users could define parts and store them in a library of parts. Each part could include another part by reference, so that if you changed the definition of a part, then all of its uses would update automatically. Part A could include a reference to part B, which could include a reference to part C, and so on. Circular references were not allowed, as a part could not include itself.

We had a customer that wanted the library of parts written out to tape so that each part appeared on the tape before any other part that used it. I was the only person at the company who knew that what this customer wanted was a topological sort of a directed acyclic graph. I knew that there was an efficient algorithm for this problem, and I knew where I'd seen it (in Knuth). I didn't remember the details of the algorithm, and so I went to the library, got a copy of Knuth, and implemented the algorithm.

People at the company thought I was a god for knowing how to solve the problem, and how to solve it efficiently.

That's why you want to know about algorithms.

301.4k Views · View Upvotes · Answer requested by Killow

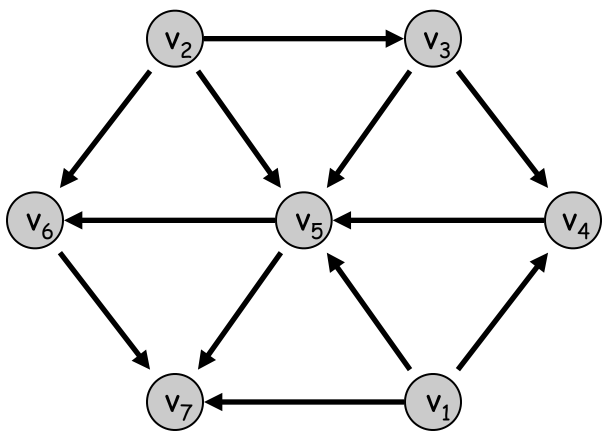
Upvote | 12.6k Downvote Comments 37+

Directed Acyclic Graphs

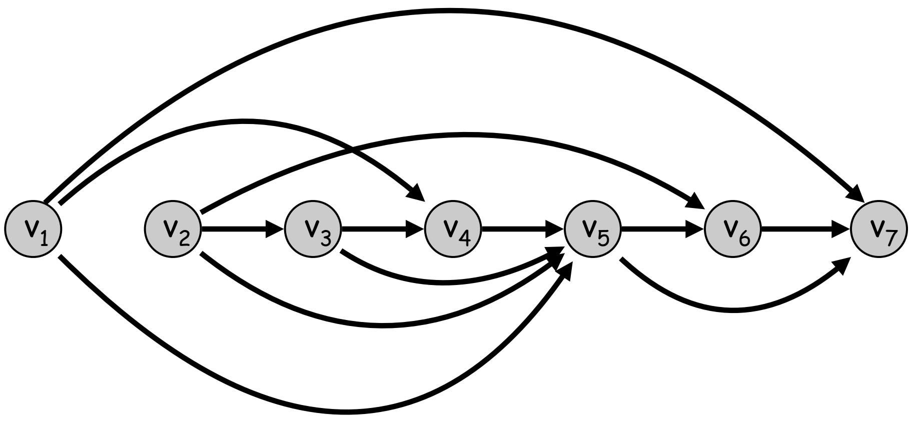
Def. A DAG is a directed graph that contains no directed cycles.

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

Precedence Constraints

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

Applications.

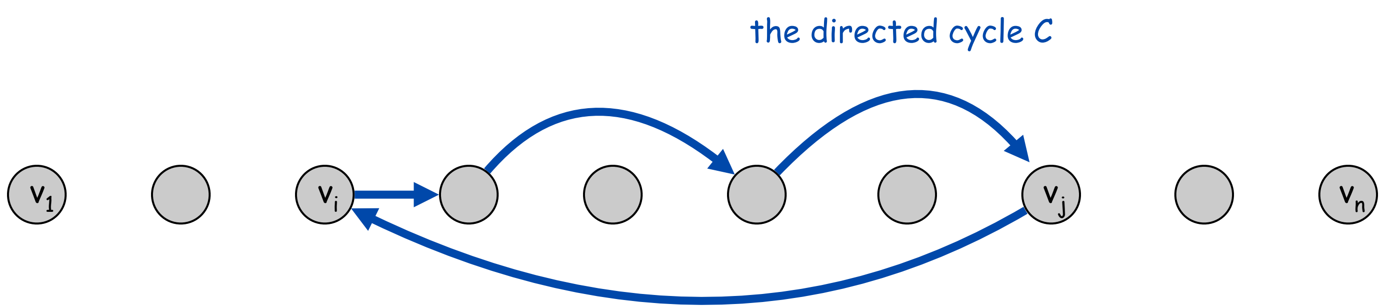
- Course prerequisite graph: course v_i must be taken before v_j.
- Compilation: module v_i must be compiled before v_j.
- Pipeline of computing jobs: output of job v_i needed to determine input of job v_j.

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 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.

Q. Does every DAG have a topological ordering?

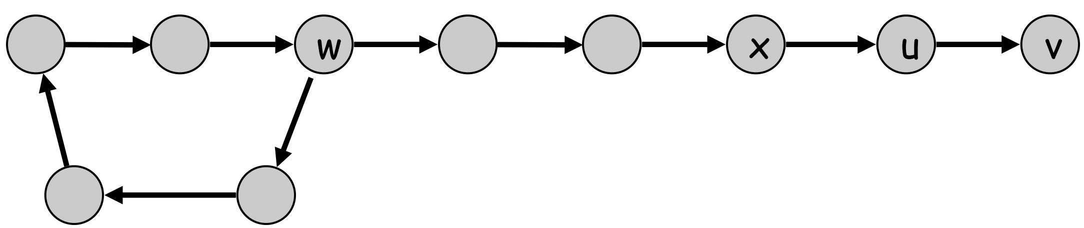
Q. If so, how do we compute one?

Directed Acyclic Graphs

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

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. ▪



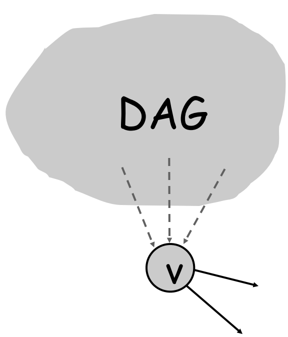
Directed Acyclic Graphs

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

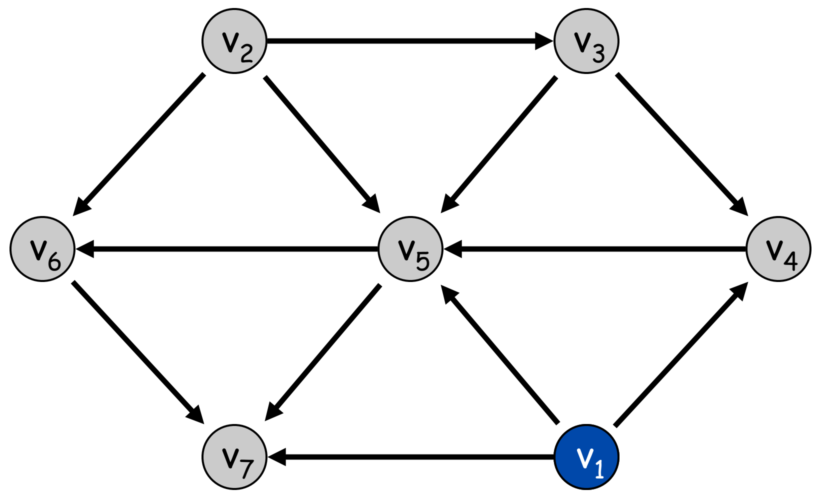
Pf. (by induction on n)

- 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
```

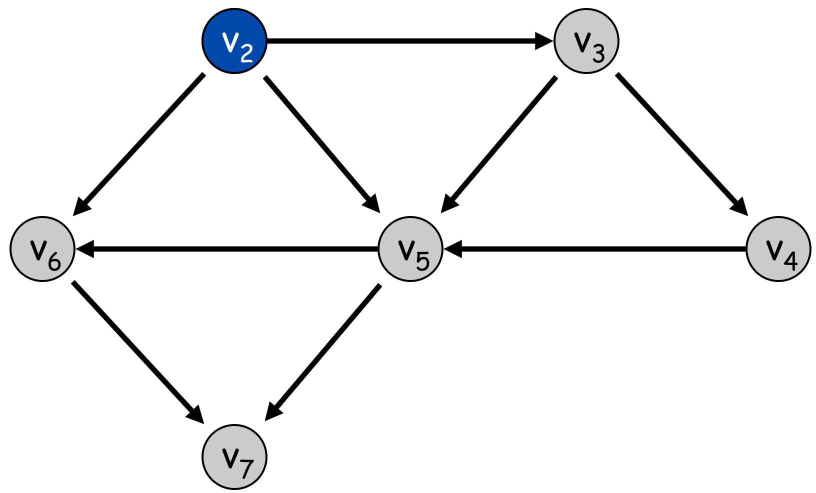


Topological Ordering Algorithm: Example



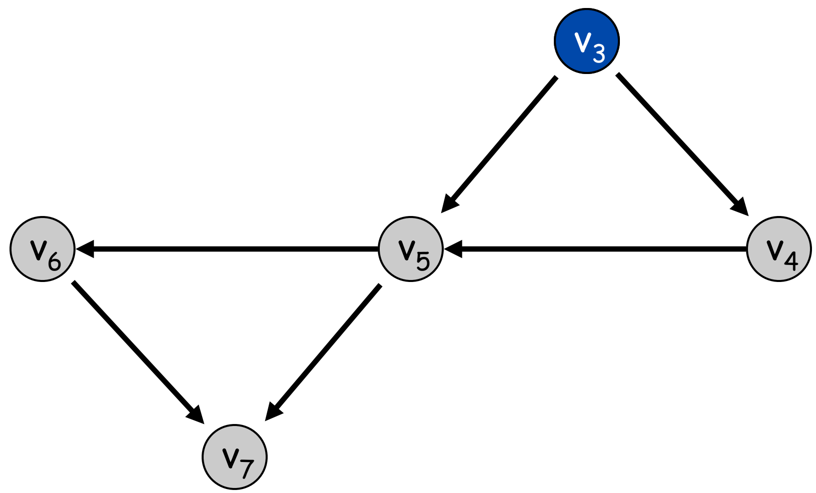
Topological order:

Topological Ordering Algorithm: Example



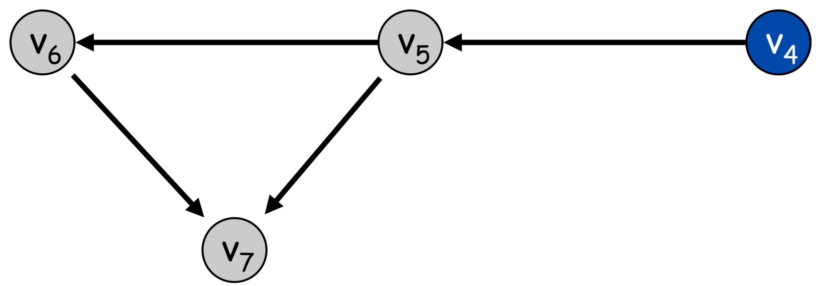
Topological order: v_1

Topological Ordering Algorithm: Example



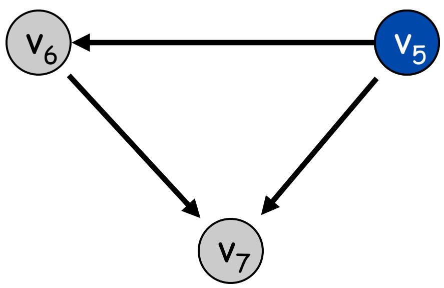
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



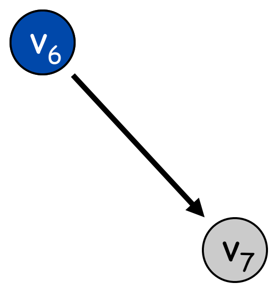
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

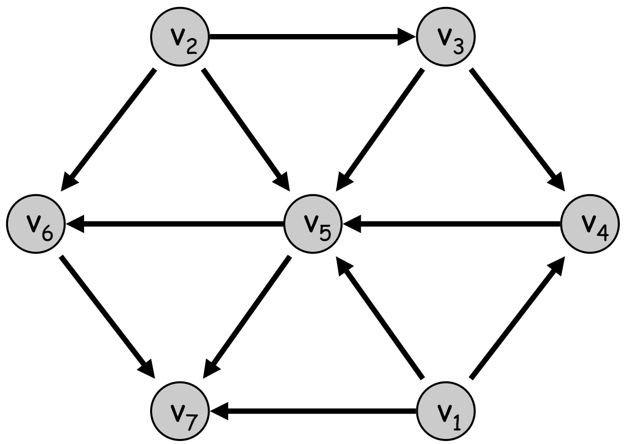


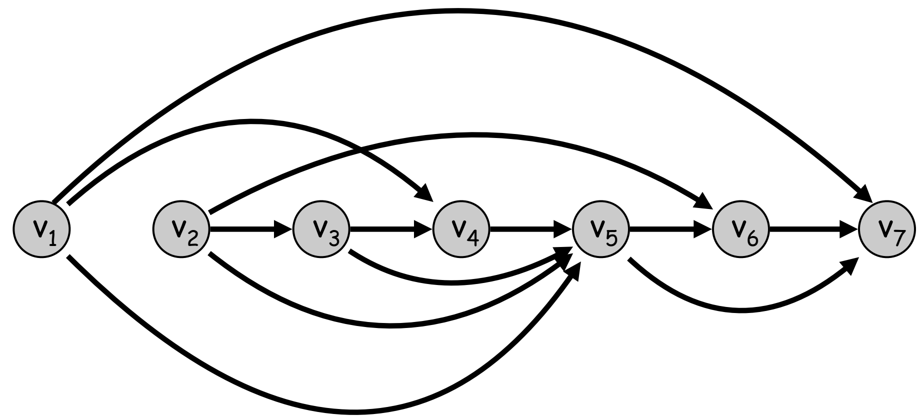
Topological order: v_1, v_2, v_3, v_4, v_5

Topological Ordering Algorithm: Example

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

Topological Ordering Algorithm: Example





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

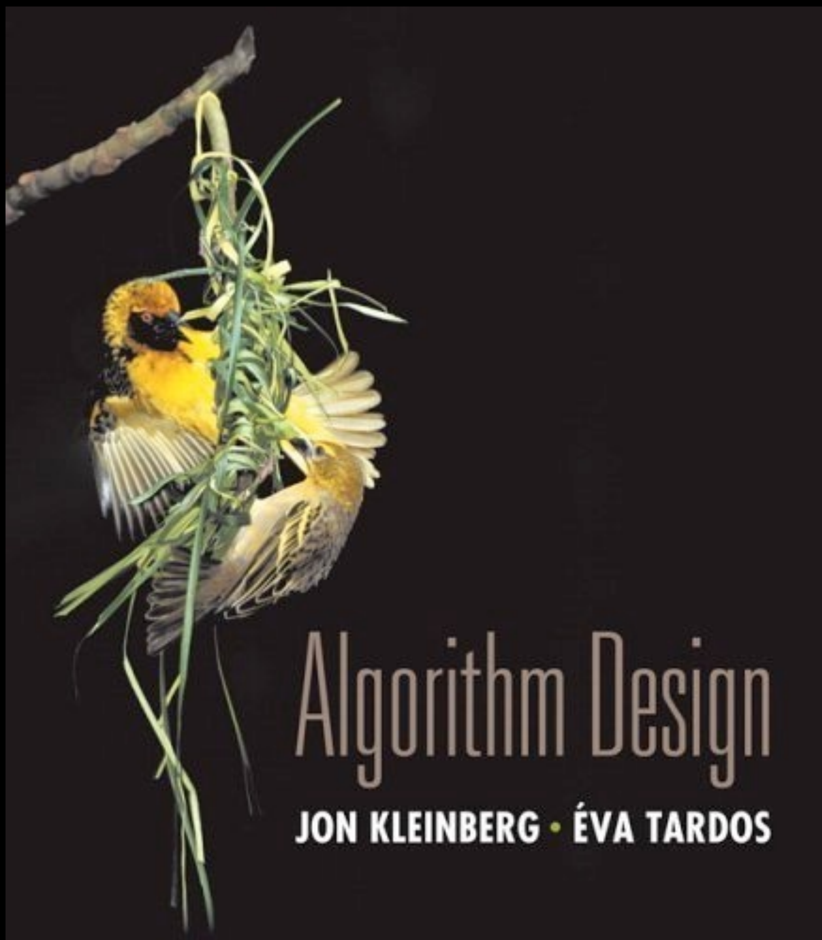
Topological Sorting Algorithm: Running Time

```
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
```

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

Pf.

- Maintain the following information:
  - for each node w, `count[w]` = number of remaining 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 ▪



Chapter 3

Graphs

CLRS 12-13