Announcements

- Homework 1 Due on Friday

Note

- "Polynomial time" means time $O(n^k)$ for some $k > 0$.
  - [for graph algorithms it means $O((|V|+|E|)^k)$]

Last Time

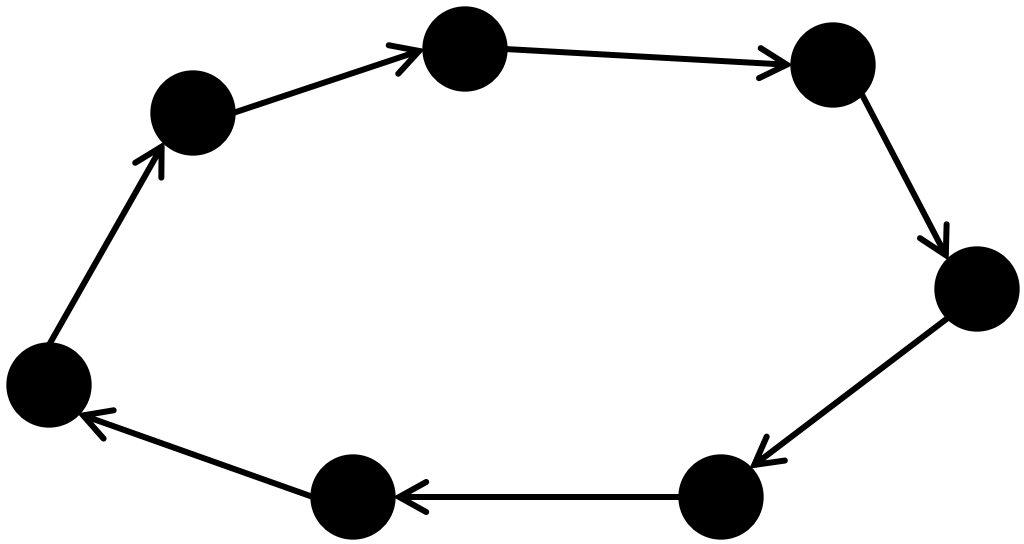
- Pre- and Post- orderings
  - Keep track of execution of DFS
  - Preorder when find a new vertex
  - Postorder when finish with vertex
- Directed graphs
  - Edges have direction
- Dependency and topological ordering
  - Order vertices so that for each edge (v,w) v comes before w

Today

- DAGs
- Topological Sort
- Strongly Connected Components

Cycles

Definition: A cycle in a directed graph is a sequence of vertices $v_1, v_2, v_3, \ldots, v_n$ so that there are edges $(v_1,v_2), (v_2,v_3), \ldots, (v_{n-1},v_n), (v_n,v_1)$



Obstacle

Proposition: If G is a directed graph with a cycle, then G has no topological ordering.

Proof:

- Have cycle $v_1, v_2, \ldots, v_n$.
- AFSOC we have an ordering.
- Find earliest v_i in the ordering.
- Note that v_i comes before v_{i-1}, in contradiction to the order property.

DAGs

Definition: A Directed Acyclic Graph (DAG) is a directed graph which contains no cycles.

Previous result said that *only* DAGs can be topologically ordered. Is the reverse true?

Surprisingly, yes.

Existence of Orderings

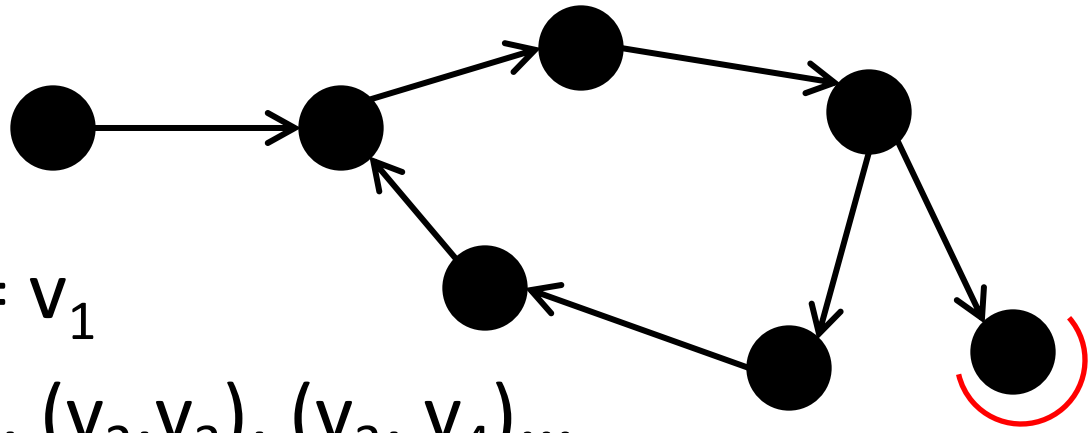
Theorem: Let G be a (finite) DAG. Then G has a topological ordering.

Proof:

- Consider the *last* vertex in the ordering.
  - Must be a *sink* (vertex with no outgoing edges).
- Idea: find a sink, put at end, order remaining.
- Question: Does G have a sink?

Sinks

Lemma: Every finite DAG contains at least one sink.



Proof:

- Start at vertex $v = v_1$
- Find edges (v_1,v_2), (v_2,v_3), (v_3, v_4)…
- Eventually either:
  - Some vertex repeats (create cycle)
  - Get stuck (found a sink)

Proof of Theorem

- Induction on |G|.
- Find sink v.
- Let G’ = G-v.
- Inductively order G’ (still a DAG).
- Add v to the end of the ordering.

Algorithm

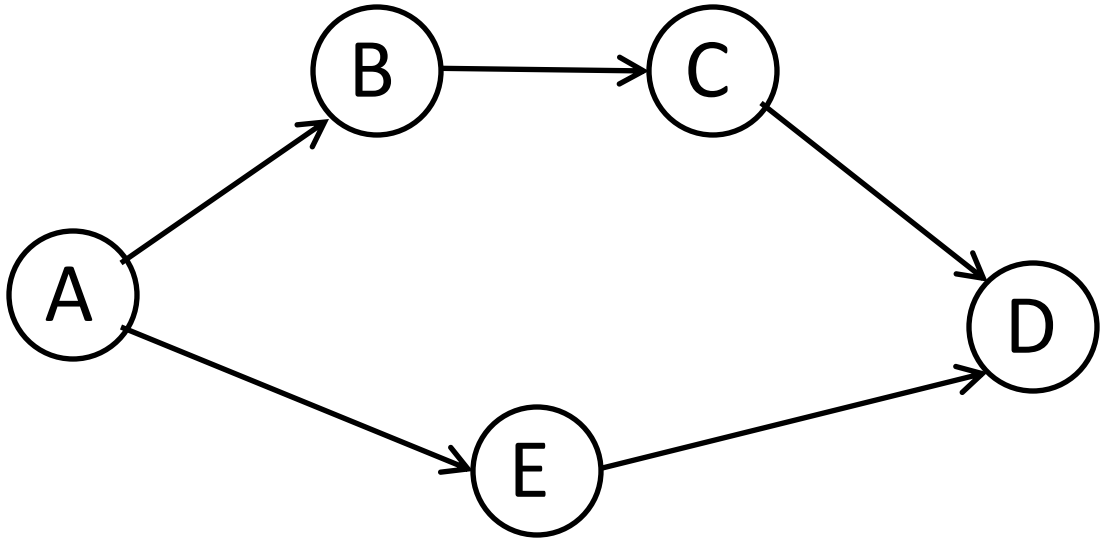
Problem: Design an algorithm that given a DAG G computes a topological ordering on G.

- Find sink v
  - Follow chain of vertices until stuck
- Compute ordering on G-v
- Place v at the end

Algorithm

```
Ordering(G)
   If |G|=0, Return {}
   Let v ∈ G
   While there is an edge (v,w)
      v ← w
   Return (Ordering(G-v),v)
```

Example



Final Ordering: A E B C D

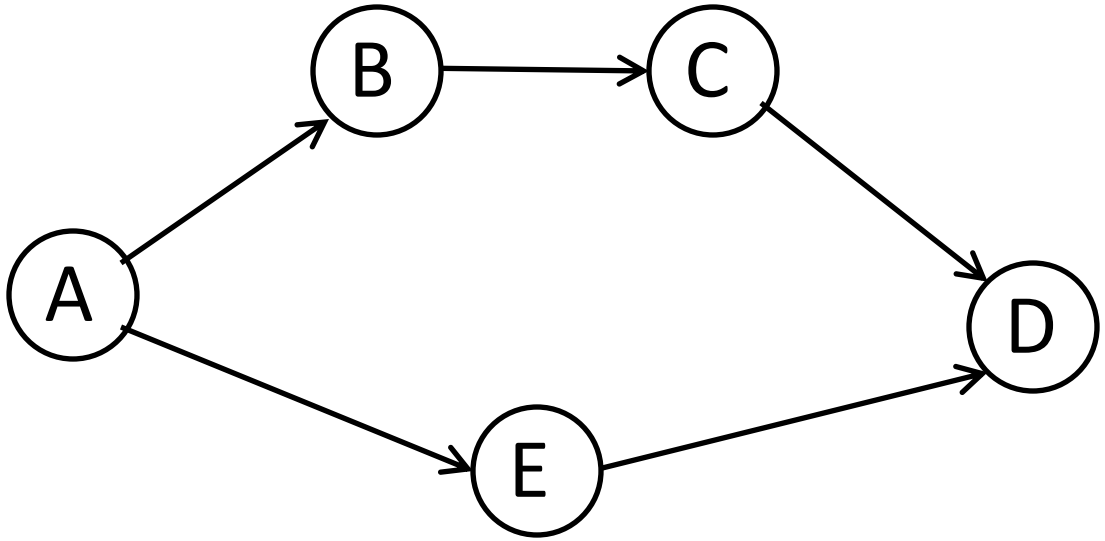
Runtime

(|V| time to find each sink) ·(|V| sinks)
= $O(|V|^2)$ runtime.

This is suboptimal.

Problem: After adding a sink to the end, the algorithm forgets the path that it took. Instead of backing up to start, just back up one step.

Example II



This is just DFS ordering!

Final Ordering: A E B C D

Algorithm II

```
TopologicalSort(G)
   Run DFS(G) w/ pre/post numbers
   Return the vertices in reverse
  postorder
```

Note: Can add vertices to list as postorder assigned.

Runtime: O(|V|+|E|).

Correctness

Proposition: If G is a DAG with an edge v → w then `w.post < v.post`.

Implies that ordering is correct.

Proof:

Break into cases based on which of v or w is discovered first.

Proof

- If v discovered first
  - w discovered while exploring v
  - w descendant of v
  - pre-post intervals nested
  - `w.post < v.post`
- If w discovered first
  - v cannot be a descendant (DAG)
  - pre-post intervals are disjoint
  - `w.post < v.post`

Topological Sort

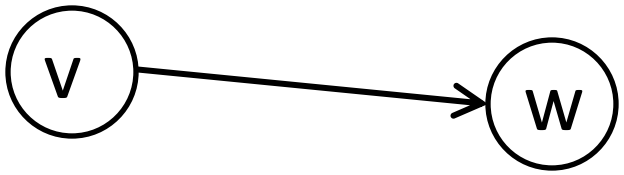
Useful algorithm.

- Many graph algorithms are relatively easy to find the answer for v if you've already found the answer for everything downstream of v.
  - Topologically sort G.
  - Solve for v in reverse topological order.

Connectivity in Digraphs

In undirected graphs, we had a very clean description of reachability: v was reachable from w if and only if they were in the same connected component.

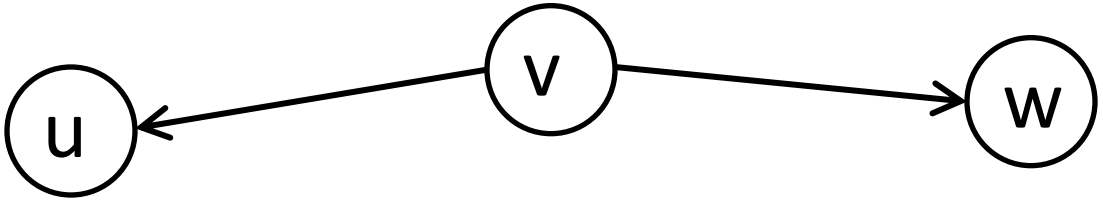
This no longer works for digraphs.



What is the right notion of connected components for digraphs?

Problems

- Reachability no longer symmetric
  - can reach w from v but not v from w.
- Maybe allow reachability in either direction?



- Maybe allow you to follow edges in either direction?
  - This basically treats digraph as undirected.

Strongly Connected Components

Definition: In a directed graph G, two vertices v and w are in the same Strongly Connected Component (SCC) if v is reachable from w *and* w is reachable from v.

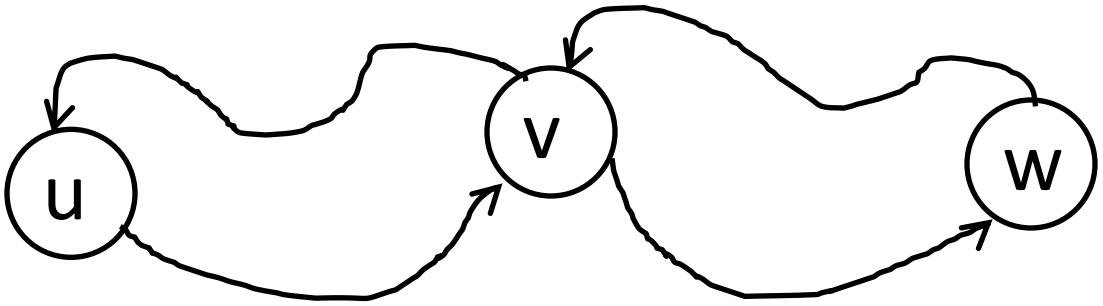
Question: Can you actually partition the vertices into such components?

Equivalence Relation

Let v ~ w if v reachable from w and visa versa.

Claim: This is an equivalence relation. Namely:

- v ~ v. (v reachable from itself)
- If v ~ w then w ~ v. (relation is symmetric)
- If u ~ v and v ~ w then u ~ w.



Components

Whenever you have an equivalence relation you can split a set into components (equivalence classes) so that $v \sim w$ if and only if v and w are in the same component.

Take any v, the set of all w so that $v \sim w$ is the component of v. Everything connects to everything else in this class and does not connect (both ways) to anything outside.

Question: SCCs

How many strongly connected components does the graph below have?

A) 1

B) 2

C) 3

D) 4

E) 5

