Objectives

- Directed graphs
- Topological Orderings of DAGs

Review

- Thoughts on Dr. Danner's talks?
- Journals
- What do we know about graphs?

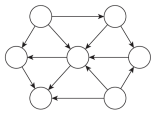
Review

- What do we know about graphs?
  - Space
  - Connectivity
    - BFS, DFS
  - Bipartite graphs
    - How to color?
    - When know not colorable?

DIRECTED GRAPHS

Directed Graphs G = (V, E)

- Edge (u, v) goes from node u to node v

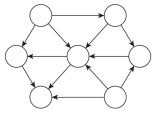


How can we represent directed graphs? What information do we want?

- 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

Representing Directed Graphs

- Edge (u, v) goes from node u to node v



- For each node, keep track of
  - Out edges (where links go)
  - In edges (from where links come in)
- Could only store *out* edges
  - Figure out *in* edges with increased computation/ time
  - Useful to have both *in* and *out* edges

CONNECTIVITY IN DIRECTED GRAPHS

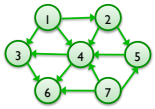
Graph Search

- How does ***reachability*** change with directed graphs?

- Example: Web crawler
  1. Start from web page s.
  2. Find all web pages linked from s, either directly or indirectly.

Graph Search

- Directed reachability. Given a node *s*, find all nodes reachable from *s*.
- Directed *s-t* shortest path problem. Given two nodes *s* and *t*, what is the length of the shortest path between *s* and *t*?
  - Not necessarily the same as t→s shortest path
- Graph search. BFS and DFS extend naturally to directed graphs
  - Trace through out edges
  - Run in O(m+n) time



Problem

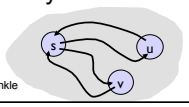
- Find all nodes with paths **to** *s*
  - Rather than paths from *s* to other nodes

Problem/Solution

- Problem. Find all nodes with paths **to** *s*
- Solution. Run BFS on *in edges* instead of out edges

Strong Connectivity

- Def. Node *u* and *v* are ***mutually reachable*** if there is a *path* from *u* → *v* and also a *path* from *v* → *u* (not necessarily a direct edge)
- 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



Strong Connectivity

- **Prove:** If *u* and *v* are mutually reachable and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable

Strong Connectivity

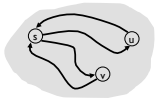
- Claim. If *u* and *v* are mutually reachable and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable.
- Proof. We need to show that there is a path from *u* → *w* and from *w* → *u*.
  - By defn of mutually reachable
    - There is a path *u* → *v* & a path *v* → *u*
    - There is a path *v* → *w*, and a path *w* → *v*
  - Take path *u*→*v* and then from *v* → w
    - Path from *u*→*w*
  - Similarly for *w*→*u*

Strong Connectivity

- 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.
  - 1st prove ⇒
  - 2nd prove ⇐

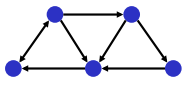
Strong Connectivity

- 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. ⇒ Follows from definition of strongly connected
- Pf. ⇐ For any nodes *u* and *v*, make path *u*→*v* and *v*→*u*
  - *u*→*v* : concatenating *u*→*s* with *s*→*v*
  - *v* →*u*: concatenate *v*→*s* with *s*→*u*

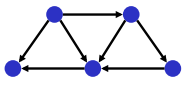


Strong Connectivity Problem

- Determine if G is strongly connected in O(m + n) time



strongly connected



not strongly connected

Hint: Can we leverage any algorithms we know have O(m+n) time?

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
    - All reachable from one node, *s* is reached by all

reverse orientation of every edge in G
Or, the BFS using the *in* edges

Strong Components

- **Prove:** For any two nodes *s* and *t* in a directed graph, their strong components are either identical or disjoint

Hint: Consider a node in common…

Strong Components

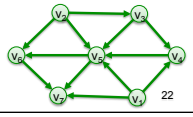
- Claim. For any two nodes *s* and *t* in a directed graph, their strong components are either identical or disjoint
- Proof.
  - Consider *v* in both strong components
    - s→ v; v → s; v→t; t→v ➔ t→s, s→t (mutually reachable)
    - As soon as there is one common node, then have identical strong components
  - On the other hand, consider *s* and *t* are not mutually reachable
    - No node *v* that is in the strong component of each
      - What would it mean if there were?

DAGS AND TOPOLOGICAL ORDERING

Directed Acyclic Graphs

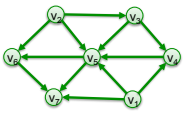
- Def. A ***DAG*** is a directed graph that contains no directed cycles.
- Example. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j
  - Course prerequisite graph: course v_i must be taken before v_j
  - Compilation: module v_i must be compiled before v_j
  - Pipeline of computing jobs: output of job v_i needed to determine input of job v_j

a DAG:



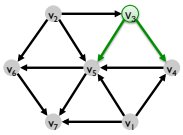
Problem: Valid Ordering

- Given a set of tasks with dependencies, what is a valid order in which the tasks could be performed?

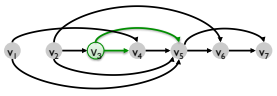


Topological Ordering

- Problem: Given a set of tasks with dependencies, what is a valid order in which the tasks could be performed?
- Def. A ***topological order*** of a directed graph G = (V, E) is an ordering of its nodes as v_1, v_2, …, v_n so that for every edge (v_i, v_j) we have i < j.



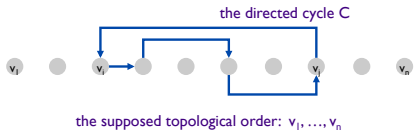
a DAG



a topological ordering
All edges point "forward"

Directed Acyclic Graphs

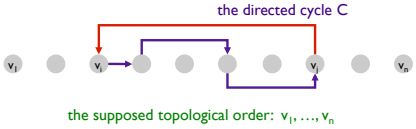
- Lemma. If G has a topological order, then G is a DAG.
- Proof plan: Try to show that G has a cycle



Why isn't this valid?

DAGs & Topological Orderings

- 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 v_i be the lowest-indexed node in C, and let v_j be the node on C just before v_i; thus (v_j, v_i) is an edge
  - By our choice of i (lowest-indexed node), i < j
  - 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. ▪

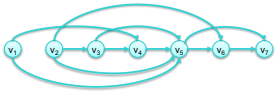


Directed Acyclic Graphs

- Does every DAG have a topological ordering?
  - If so, how do we compute one?

Directed Acyclic Graphs

- Does every DAG have a topological ordering?
  - If so, how do we compute one?
- What would we need to be able to create a topological ordering?
  - What are some characteristics of this graph?



Directed Acyclic Graphs

- Does every DAG have a topological ordering?
  - If so, how do we compute one?
- What would we need to be able to create a topological ordering?
  - What are some characteristics of this graph?

Towards a Topological Ordering

Goal: Find an algorithm for finding the TO
Idea: 1st node is one with no incoming edges

Do we know there is always a node with no incoming edges?

Directed Acyclic Graphs

- Lemma. If G is a DAG, then G has a node with no incoming edges
  - This is our starting point of the topological ordering

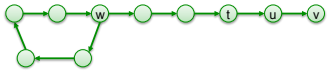
How to prove?

Towards a Topological Ordering

- Lemma. If G is a DAG, then G has a node with no incoming edges
- Proof idea: consider if there is no node without incoming edges

Towards a Topological Ordering

- 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
  - Pick any node v, and follow edges backward from v.
    - Since v has at least one incoming edge (u, v), we can walk backward to u
  - Since u has at least one incoming edge (t, u), we can walk backward to t
  - Repeat until we visit a node, say w, twice
    - Has to happen at least by n+1 steps (Why?)
  - Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle, which is a contradiction to G is a DAG ▪



Creating a Topological Order

- With a node with no incoming edges, can create a topological ordering

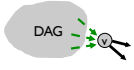
Ideas?

Topological Ordering Algorithm

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

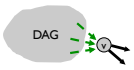
```
Find a node v with no incoming edges
Order v first
Delete v from G
Recursively compute a topological ordering of G-{v}
      and append this order after v
```

How do we know this works?



Directed Acyclic Graphs

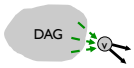
- Lemma. If G is a DAG, then G has a topological ordering.
- Pf. (by induction on n)
  - Base case: true if n = 1
  - Given DAG on n > 1 nodes, find a node v with no incoming edges
  - G - { v } is a DAG, since deleting v cannot create cycles
  - By inductive hypothesis, G - { v } has a topological ordering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



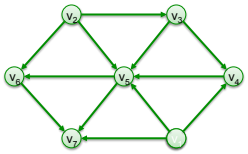
Topological Ordering Algorithm

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

```
Find a node v with no incoming edges
Order v first
Delete v from G
Recursively compute a topological ordering of G-{v}
      and append this order after v
```

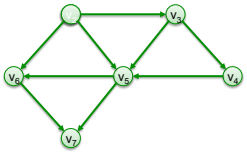


Topological Ordering Algorithm: Example



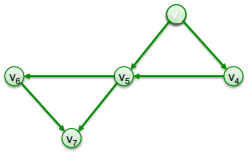
Topological order:

Topological Ordering Algorithm: Example



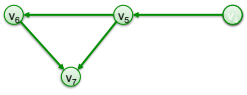
Topological order: v_1

Topological Ordering Algorithm: Example



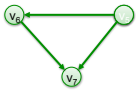
Topological order: v_1, v_2

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

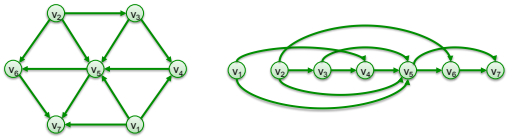
Topological Ordering Algorithm: Example

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 Order Runtime

```
Find a node v with no incoming edges
Order v first
Delete v from G
Recursively compute a topological ordering of G-{v}
      and append this order after v
```

- Where are the costs?
- How would we implement?

Topological Order Runtime

```
Find a node v with no incoming edges
Order v first                                  O(n)
Delete v from G
Recursively compute a topological ordering of G-{v}   O(n)
      and append this order after v
```

- Find a node without incoming edges and delete it: **O(n)**
- Repeat on all nodes

Can we do better?

→ **$O(n^2)$**

Topological Sorting Algorithm: Running Time

- Theorem. Find 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
  - Algorithm:
    - Select a node v from S, remove v from S
    - Decrement count[w] for all edges from v to w
      - Add w to S if count[w] = 0

Looking Ahead

- Problem Set 3 due Friday
- Danner talk write up due Friday
- Exam 1 handed out on Friday