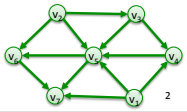
Objectives

- Topological Orderings of DAGs

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:



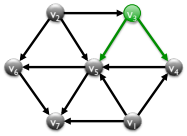
Problem: Valid Ordering

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

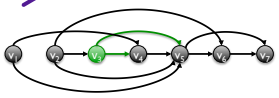
- Example: Getting dressed
  - What tasks are involved?
  - What tasks depend on other tasks?

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 $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$.



a DAG



a topological ordering
All edges point "forward"

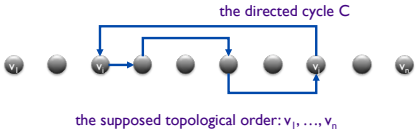
Coordinating labeling of nodes, but numbering is not known for just DAG

Towards a Solution

- Start by showing that if G has a topological order, then G is a DAG
- Eventually, we'll show the other direction: if G is a DAG, then G has a topological order

Directed Acyclic Graphs

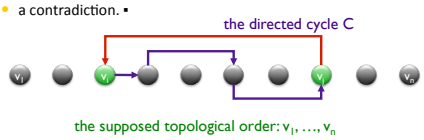
- Lemma. If G has a topological order, then G is a DAG.
- Proof plan: Try to show that G has a topological order even though G has a cycle



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. ▪



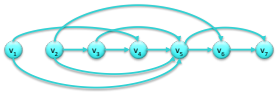
the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

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

Directed Acyclic Graphs

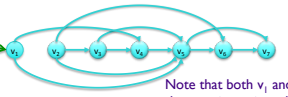
- Does every DAG have a topological ordering?
  - If so, how do we compute one?
- What do we need to be able to create a topological ordering?
  - What are some characteristics of this graph?



Directed Acyclic Graphs

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

Need a place to start: a node with no incoming edges (no dependencies)



Note that both v_1 and v_2 have no incoming edges

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?

Towards a Topological Ordering

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

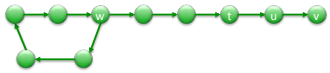
How to prove?

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
  - What contradiction are we looking for?

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 step n+1 (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 ▪



Putting it all together: Creating a topological order

Ideas?

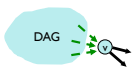
Topological Ordering Algorithm

```
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 because deleting v cannot create cycles
  - By inductive hypothesis, G - { v } has a topological ordering
  - Place v first in topological ordering;
  - Append nodes of G - { v } in topological order.
    - valid since v has no incoming edges. ▪

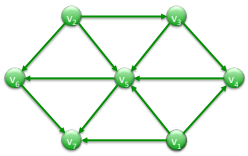


Topological Ordering Algorithm

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

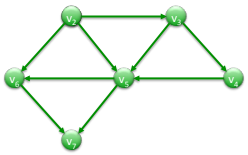
```
Find a node v with no incoming edges
Order v first
Delete v from G
Recursively compute a topological ordering of G-{v}
      and append this order after v
```

Topological Ordering Algorithm: Example



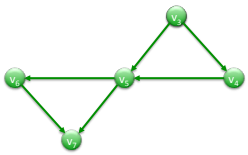
Topological order:

Topological Ordering Algorithm: Example



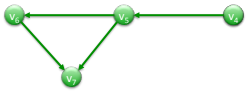
Topological order: v_1

Topological Ordering Algorithm: Example



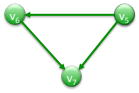
Topological order: v_1, v_2

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

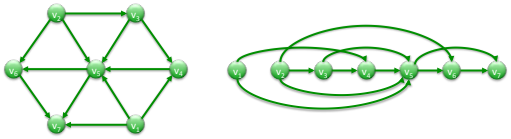
Topological Ordering Algorithm: Example

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

GREEDY ALGORITHMS

Greedy Algorithms

At each step, take as much as you can get
→ "local" optimizations

- Need a proof to show that the algorithm finds an optimal solution
- A counter example shows that a greedy algorithm does not provide an optimal solution

Example of Greedy Algorithm

- How do you make change to give out the *fewest* coins?
- Determine for 34¢

Example of Greedy Algorithm

- How do you make change to give out the *fewest* coins?

```
while change > 0:
        if change >= 25:
                print "Quarter"
                change -= 25
        elif change >= 10:
                print "Dime"
                change -= 10
        …
```

Let's generalize …

- Ex: 34¢.

Coin Changing

- Goal. Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.
- Ex: 34¢.

- Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.
- Ex: $2.89.

Coin-Changing: Greedy Algorithm

- Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Sort coins' denominations by value: $c_1 < c_2 < \ldots < c_n$.

coins selected

```
S = ϕ
while x ≠ 0
   let k be largest integer such that c_k ≤ x
   if k = 0
      return "no solution found"   ← How could this happen?
   x = x - c_k
   S = S ∪ {k}
return S
```

Is cashier's algorithm ***optimal***?

Coin-Changing: Analysis of Greedy Algorithm

extra: not covered in class

- Theorem. Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100
- Pf. (by induction on x)
  - Consider optimal way to change $c_k \le x < c_{k+1}$
    - Greedy takes coin k
  - Any optimal solution must also take coin k
    - If not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
    - Table below indicates no optimal solution can do this
  - Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by greedy algorithm. ▪

Which coin	k	Coin value c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT	If don't take c_k
	1	1	P ≤ 4	-	
	2	5	N ≤ 1	4	
	3	10	N + D ≤ 2	4 + 5 = 9	
	4	25	Q ≤ 3	20 + 4 = 24	
	5	100	no limit	75 + 24 = 99	

Coin-Changing: Analysis of Greedy Algorithm

- Observation. Greedy algorithm is sub-optimal for US postal denominations:
  - 500 300 200 100 86 85 79 78 66 65 46 44 33 32 20 4 3 2 1
- Counterexample. 158¢.
  - Greedy: 100, 44, 4, 4, 4, 2.
  - Optimal: 79, 79.

Proving Greedy Algorithms Work

- Specifically, produce an **optimal** solution
- Approaches:
  - Greedy algorithm stays ahead
    - Does better than any other algorithm at each step
  - Exchange argument
    - Transform any solution into a greedy solution
  - Structural argument
    - Figure out some structural bound that all solutions must meet

Looking Ahead

- Exam 1: due at 5 p.m. today
- Monday: Wiki
  - Sections 3.3-3.6
- Problem Set 4 due Friday