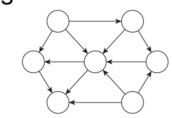
Objectives

- Topological Orderings of DAGs
- Introducing Greedy Algorithms

Review: Directed Graphs G = (V, E)

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

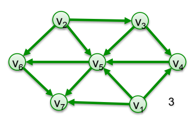


- Representation
  - Maintain both in and out edges of each node

Review: Directed Acyclic Graphs

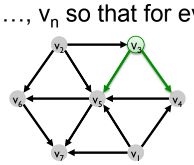
- Def. A ***DAG*** is a directed graph that contains no directed cycles.
- Example. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j
  - Course prerequisite graph: course v_i must be taken before v_j
  - Compilation: module v_i must be compiled before v_j
  - Pipeline of computing jobs: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

a DAG:

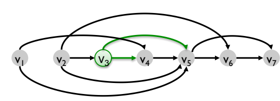


Review: Topological Ordering

- 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"

Towards a Topological Ordering

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

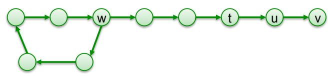
Do we know there is always a node with no incoming edges?

Towards a Topological Ordering

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



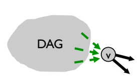
Creating a Topological Order

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

Ideas?

Directed Acyclic Graphs

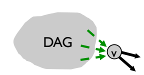
- Lemma. If G is a DAG, then G has a topological ordering.
- Pf. (by induction on n)
  - Base case: true if n = 1
  - Given DAG on n > 1 nodes, find a node v with no incoming edges
  - G - { v } is a DAG, since deleting v cannot create cycles
  - By inductive hypothesis, G - { v } has a topological ordering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



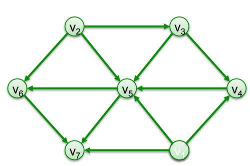
Topological Ordering Algorithm

- 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
```

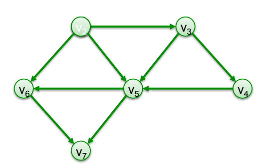


Topological Ordering Algorithm: Example



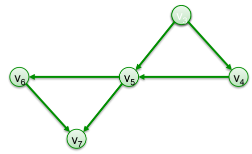
Topological order:

Topological Ordering Algorithm: Example



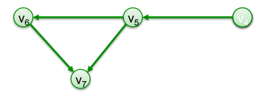
Topological order: v_1

Topological Ordering Algorithm: Example



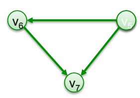
Topological order: v_1, v_2

Topological Ordering Algorithm: Example



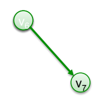
Topological order: v_1, v_2, v_3

Topological Ordering Algorithm: Example



Topological order: v_1, v_2, v_3, v_4

Topological Ordering Algorithm: Example

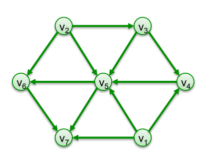


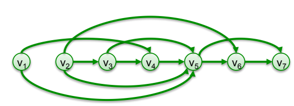
Topological order: v_1, v_2, v_3, v_4, v_5

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                    O(n)
Order v first
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

→ **O(n²)**

Can we do better?

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

INTRODUCING 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: c1 < c2 < … < cn.

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

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

Coin-Changing: Analysis of Greedy Algorithm

- 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. ▪

k	c_k	All optimal solutions must satisfy	Max value of coins 1, 2, …, k-1 in any OPT
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

Which coin — Coin value — If don't take c_k

Coin-Changing: Analysis of Greedy Algorithm

- Observation. Greedy algorithm is sub-optimal for US postal denominations:
  - 500 100 98 79 78 64 44 28 17 2 1
- Counterexample. 158¢.
  - Greedy: 100, 44, 2, 2, 2, 2, 2, 2, 2.
  - Optimal: 79, 79.

Proving Greedy Algorithms Work

- Specifically, produce an **optimal** solution
- Two approaches:
  - Greedy algorithm stays ahead
    - Does better than any other algorithm at each step
  - Exchange argument
    - Transform any solution into a greedy solution

Looking Ahead

- Wiki due Wednesday
  - Chapter 3 through 3.5
- Problem Set 3 due Friday
- Midterm handed out on Friday
- Extra Credit Opportunities
  - Jeopardy! Challenger: IBM's Watson
  - SSA conference
  - Talk by Jan Cuny
  - Problems