Objectives

- Directed Graphs: Strong Connectivity
- Greedy Algorithms
  - Interval Scheduling

Review

- Compare and contrast directed and undirected graphs
- What is a topological ordering?
  - What does the graph represent?
    - What are the constraints on the graph?
  - What is the output?
  - How do we find the topological ordering?

Topological Order Runtime

```
Find a node v with no incoming edges  O(n)
Order v first O(1)
Delete v from G   O(n)
Recursively compute a topological ordering of G-{v}   O(n)
      and append this order after v  O(1)
```

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

Can we estimate better?

→ **$O(n^2)$**

Topological Sorting Algorithm: Running Time

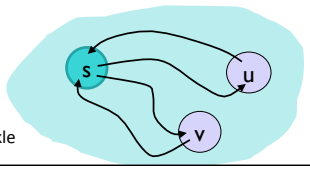
- Theorem. Find a topological order in O(m + n) time
- Pf.
  - Maintain the following information:
    - count[w] = remaining number of incoming edges
    - S = set of remaining nodes with no incoming edges
  - Initialization: O(m + n) via single scan through graph
  - Algorithm:
    - Select a node 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

Directed Graphs

STRONG CONNECTIVITY

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
  - ➢ We want to prove this…



Strong Connectivity

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

Strong Connectivity

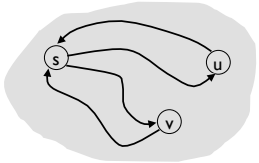
- If *u* and *v* are mutually reachable and *v* and *w* are mutually reachable, then *u* and *w* are mutually reachable.
- Proof. We need to show that there is a path from *u* → *w* and from *w* → *u*.
  - By defn of mutually reachable
    - There is a path *u* → *v* & a path *v* → *u*
    - There is 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 ⇐
    - for any nodes *u* and *v,* is there a path *u*→*v* and *v*→*u* ?

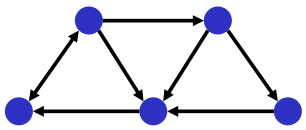
Strong Connectivity

- 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.
  - 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*

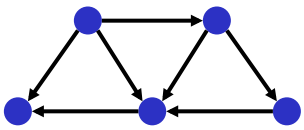


Strong Connectivity Problem

- **Claim**: We can 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}

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

  - Return true *iff* **both** BFS executions reach **all** nodes
  - Correctness follows immediately from previous lemma
    - All reachable from one node, *s* is reached by all

Strong Components

- Strong components: analogous to connected component in undirected graph
  - Set of mutually reachable nodes
- 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

- 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?

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

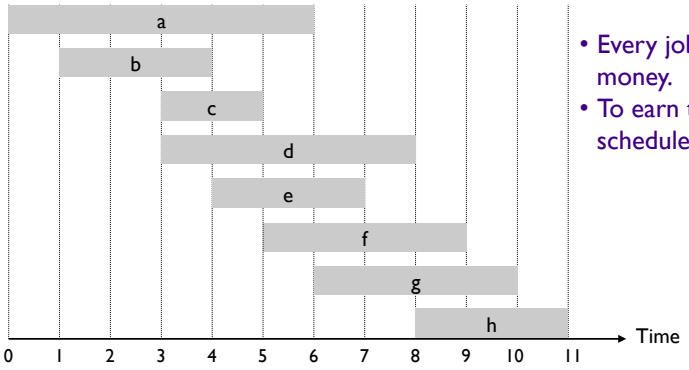
- Specifically, produce an **optimal** solution
- Approaches:
  - Greedy algorithm stays ahead
    - Does better than any other algorithm at each step
  - Exchange argument
    - Transform any solution into a greedy solution
  - Structural argument
    - Figure out some structural bound that all solutions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

Greedy algorithm stays ahead

INTERVAL SCHEDULING

Interval Scheduling

- Job j starts at s_j and finishes at f_j
- Two jobs are **compatible** if they don't overlap
- **Goal**: find maximum subset of mutually compatible jobs



- Every job is worth equal money.
- To earn the most money → schedule the most jobs

Greedy Algorithm Template

- Consider jobs (or whatever) in some order
  - Decision: What order is best?
- Take each job provided it's compatible with the ones already taken

What are options for orders? (rhetorical for now)

What is our goal?
What are we trying to minimize/maximize?

What is the worst case?

Greedy Algorithm Pseudo-Code

In some specified order

```
Set Greedy (Set candidate){
   solution = new Set( );
   while candidate.isNotEmpty()
      next = candidate.select() //use selection criteria,
      //remove from candidate and return value
      if solution.isFeasible(next) //constraints satisfied
         solution.union(next)
      if solution.solves()
         return solution

   //No more candidates and no solution
   return null
}
```


Greedy Algorithm Template

- Consider jobs (or whatever) in some order
  - Decision: What order is best?
- Take each job provided it's compatible with the ones already taken

What are options for orders?

What is our goal?
What are we trying to optimize?

What is the worst case?

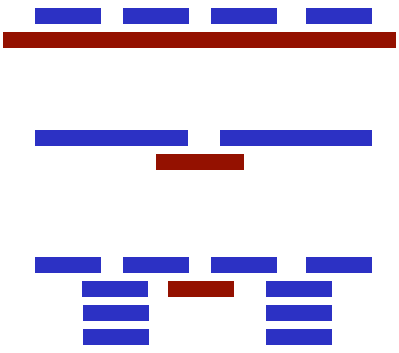
Interval Scheduling

- Earliest start time. Consider jobs in ascending order of start time s_j
  - Utilize CPU as soon as possible
- Earliest finish time. Consider jobs in ascending order of finish time f_j
  - Resource becomes free ASAP
  - Maximize time left for other requests
- Shortest interval. Consider jobs in ascending order of interval length $f_j - s_j$
- Fewest conflicts. For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j

Can we "break" any of these? i.e., prove they're not optimal?

Counterexamples to Optimality of Various Job Orders

Not optimal when …



breaks earliest start time

breaks shortest length

breaks fewest conflicts

Looking Ahead

- Problem Set 4 due Friday
- Exam given out on Friday