CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- SA4, SA5 due already
- SA6 out, due 11:59pm, Dec. 1
  - SA6 involves working through an example of the algo we covered last Monday
- PS4 due Nov. 30
  - No exercises beyond the Lookahead
- PS5 out Nov. 30, due Dec. 9
  - PS5 will be the final PS for the semester (there may be other SAs)
- PS3, SA4, SA5 grading update
- Project 4 out today
  - Due 11:59pm, Monday, Dec. 12
    - Severe late penalties if turned in during Exams—i.e., after 11:59pm Dec. 13
  - Intended team size: 4 (but talk to me if you'd prefer to work with a smaller team size)

All-Pairs Shortest Paths: A Vertex-Based Recursive Solution

- … We're still considering shortest path p from i to j with intermediate vertices in V_k
  - What's the relationship between p and the set of shortest paths from i to j with intermediate vertices in V_{k-1}?
- … Depends on whether or not vertex k is an intermediate vertex on path p
  - If not, then p is also a shortest path (i to j) with intermediate vertices in V_{k-1}
  - If so, then p can be broken down into sub-paths that are shortest paths with intermediate vertices in V_{k-1}
  - … one sub-path is from (i to k), the other is from (k to j) $i \overset{p_1}{\leadsto} k \overset{p_2}{\leadsto} j$
- Altogether, if W is the weights matrix, and $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k…

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \,. \end{cases}$$

"By the way, which one's Warshall?"

Floyd-Warshall Algorithm: Bottom-up All-Pairs Shortest Paths

- *Floyd-Warshall algorithm* for all-pairs shortest paths: the bottom-up method based on this decomposition
- Computes matrices $D^{(k)} = (d_{ij}^{(k)})$, where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL($W, n$)
  $D^{(0)} = W$
  for $k = 1$ to $n$
      let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
      for $i = 1$ to $n$
          for $j = 1$ to $n$
              $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
  return $D^{(n)}$
```

Note: This computes shortest path values, not the paths. See CLRS pages 695-697 about computing the paths themselves.

- What does this algorithm return? (What makes that a useful return value?)
- What is the running time of this algorithm?

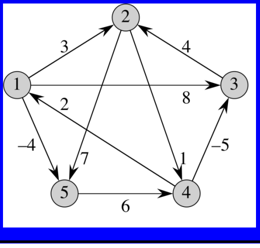
A Floyd-Warshall Example

See CLRS, Ch. 25.2

- Computes matrices D(k) = ($d_{ij}^{(k)}$), where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL(W, n)
  $D^{(0)} = W$
  for $k = 1$ to $n$
      let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
      for $i = 1$ to $n$
          for $j = 1$ to $n$
              $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
  return $D^{(n)}$
```

- What D matrices does it compute for this example graph?

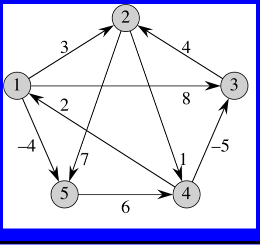


A Floyd-Warshall Example

- Computes matrices D(k) = ($d_{ij}^{(k)}$), where each $d_{ij}^{(k)}$ is the shortest path value from i to j using only intermediate vertices numbered up to k

```
FLOYD-WARSHALL(W, n)
  $D^{(0)} = W$
  for $k = 1$ to $n$
      let $D^{(k)} = (d_{ij}^{(k)})$ be a new $n \times n$ matrix
      for $i = 1$ to $n$
          for $j = 1$ to $n$
              $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
  return $D^{(n)}$
```

- What D matrices does it compute for this example graph?



$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

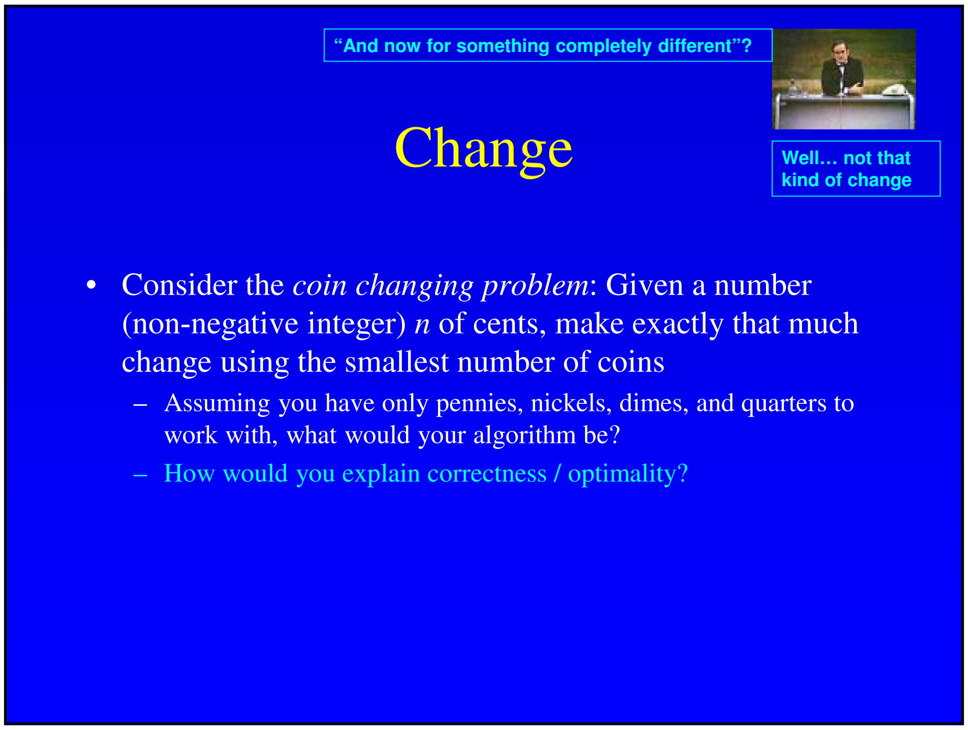
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

And Now For Something Completely Different

“It’s…”

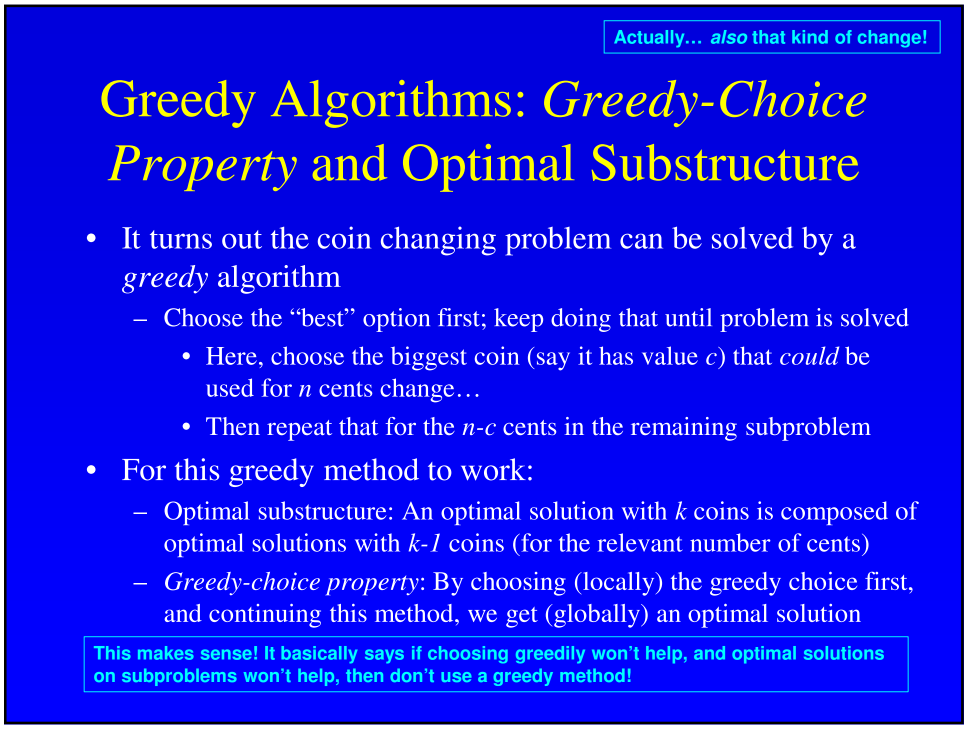


“And now for something completely different”?

Well… not that kind of change

Change

- Consider the *coin changing problem*: Given a number (non-negative integer) *n* of cents, make exactly that much change using the smallest number of coins
  - Assuming you have only pennies, nickels, dimes, and quarters to work with, what would your algorithm be?
  - How would you explain correctness / optimality?

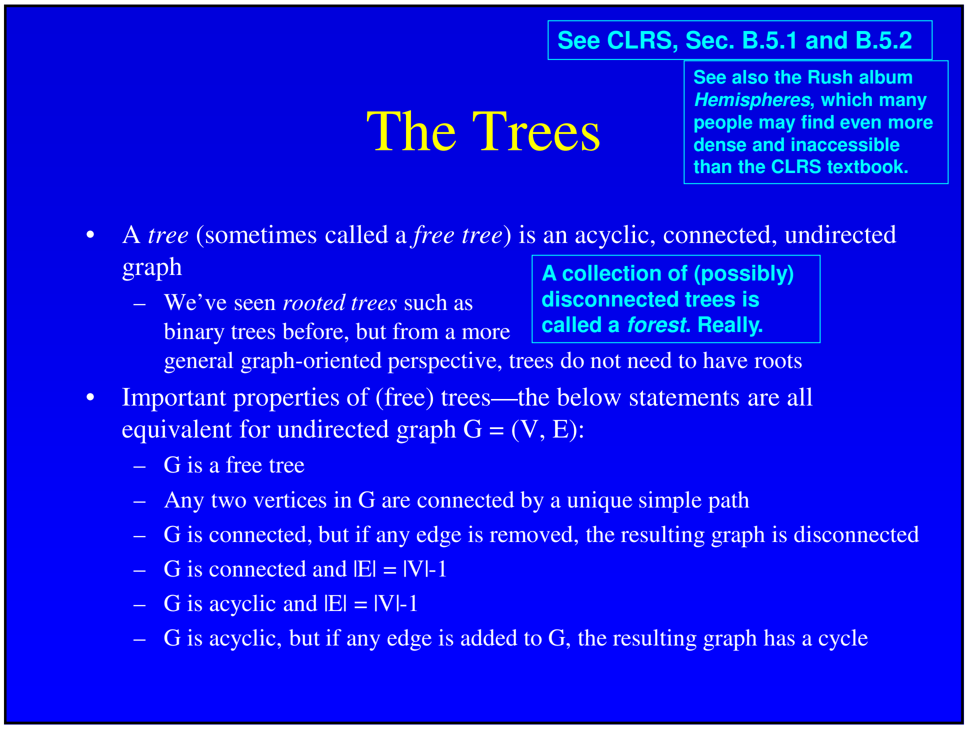


Actually… *also* that kind of change!

Greedy Algorithms: *Greedy-Choice Property* and Optimal Substructure

- It turns out the coin changing problem can be solved by a *greedy* algorithm
  - Choose the “best” option first; keep doing that until problem is solved
    - Here, choose the biggest coin (say it has value *c*) that *could* be used for *n* cents change…
    - Then repeat that for the *n-c* cents in the remaining subproblem
- For this greedy method to work:
  - Optimal substructure: An optimal solution with *k* coins is composed of optimal solutions with *k-1* coins (for the relevant number of cents)
  - *Greedy-choice property*: By choosing (locally) the greedy choice first, and continuing this method, we get (globally) an optimal solution

This makes sense! It basically says if choosing greedily won’t help, and optimal solutions on subproblems won’t help, then don’t use a greedy method!



See CLRS, Sec. B.5.1 and B.5.2

See also the Rush album *Hemispheres*, which many people may find even more dense and inaccessible than the CLRS textbook.

The Trees

- A *tree* (sometimes called a *free tree*) is an acyclic, connected, undirected graph
  - We've seen *rooted trees* such as binary trees before, but from a more general graph-oriented perspective, trees do not need to have roots

A collection of (possibly) disconnected trees is called a *forest*. Really.

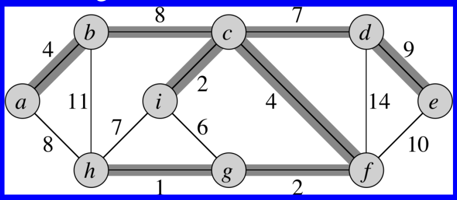
- Important properties of (free) trees—the below statements are all equivalent for undirected graph G = (V, E):
  - G is a free tree
  - Any two vertices in G are connected by a unique simple path
  - G is connected, but if any edge is removed, the resulting graph is disconnected
  - G is connected and $|E| = |V|-1$
  - G is acyclic and $|E| = |V|-1$
  - G is acyclic, but if any edge is added to G, the resulting graph has a cycle

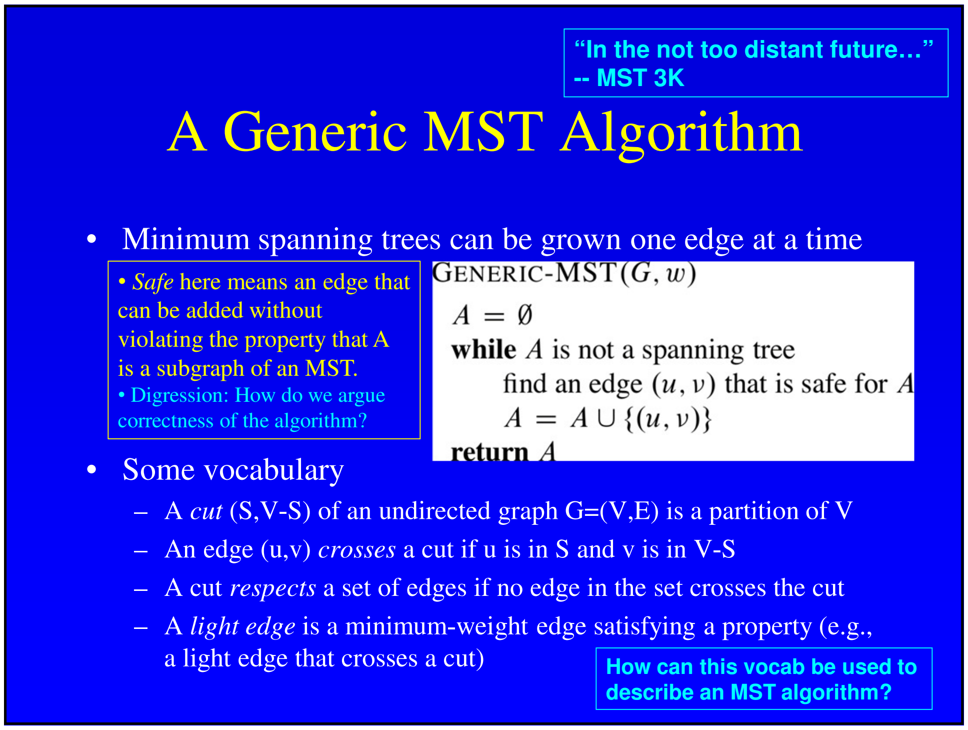
More Trees

- With rooted trees, recall vocabulary:
  - Ancestor / descendant
  - Parent / child / sibling
  - Internal node / external node / leaf
  - Depth / height of a node in the tree
- Given a connected undirected graph G = (V,E), many subgraphs of G are trees
  - Consider an acyclic subgraph that connects all the vertices in V
  - How do we know it's a tree?
  - How many edges must it have?

Minimum Spanning Trees (MSTs)

- Given a connected undirected graph G = (V,E), an acyclic subgraph that connects all the vertices in V is a *spanning tree* of G
  - It's a tree; and it covers ("spans") all the vertices of G
  - For network G, represents unique connections / paths between each pair of nodes in G
- Consider the *minimum spanning tree* (MST) problem: given weighted, undirected, connected graph G, find a spanning tree T with minimal total weight over all edges in T





"In the not too distant future…"
-- MST 3K

A Generic MST Algorithm

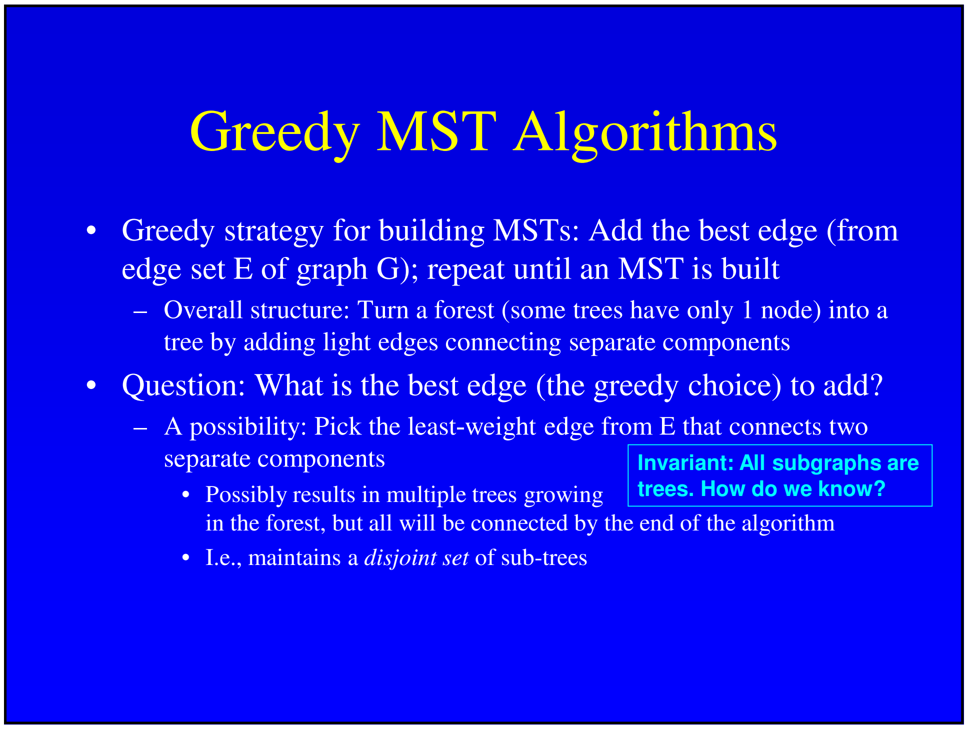
- Minimum spanning trees can be grown one edge at a time

• *Safe* here means an edge that can be added without violating the property that A is a subgraph of an MST.
• Digression: How do we argue correctness of the algorithm?

```
GENERIC-MST(G, w)
  A = ∅
  while A is not a spanning tree
      find an edge (u, v) that is safe for A
      A = A ∪ {(u, v)}
  return A
```

- Some vocabulary
  - A *cut* (S,V-S) of an undirected graph G=(V,E) is a partition of V
  - An edge (u,v) *crosses* a cut if u is in S and v is in V-S
  - A cut *respects* a set of edges if no edge in the set crosses the cut
  - A *light edge* is a minimum-weight edge satisfying a property (e.g., a light edge that crosses a cut)

How can this vocab be used to describe an MST algorithm?



Greedy MST Algorithms

- Greedy strategy for building MSTs: Add the best edge (from edge set E of graph G); repeat until an MST is built
  - Overall structure: Turn a forest (some trees have only 1 node) into a tree by adding light edges connecting separate components
- Question: What is the best edge (the greedy choice) to add?
  - A possibility: Pick the least-weight edge from E that connects two separate components
    - Possibly results in multiple trees growing in the forest, but all will be connected by the end of the algorithm
    - I.e., maintains a *disjoint set* of sub-trees

Invariant: All subgraphs are trees. How do we know?