Breadth-First Search

Input $G(V, E)$ [a connected graph]
v [start vertex]

Algorithm Breadth-First Search
visit v
$V' \leftarrow \{v\}$ [V' is the vertices already visited]
Put v on Q [Q is a queue]
repeat while $Q \neq \emptyset$
  $u \leftarrow head(Q)$ [$head(Q)$ is the first item on Q]
  for $w \in A(u)$ [$A(u) = \{w | \{u, w\} \in E\}$]
    if $w \notin V'$
    then visit w
      Put w on Q
      $V' \leftarrow V' \cup \{w\}$
    endif
  endfor
  Delete u from Q

If all edges have equal length, we can extend this algorithm to find the shortest path length from v to any other vertex:

- Store the path length with each node when you add it.
- Length(v) = 0.
- Length(w) = Length(u) + 1

Depth-First Search

Input $G(V, E)$ [a connected graph]
v [start vertex]

Algorithm Depth-First Search
visit v
$V' \leftarrow \{v\}$ [V' is the vertices already visited]
Put v on S [S is a stack]
$u \leftarrow v$
repeat while $S \neq \emptyset$
if $A(u) - V' \neq \emptyset$
then Choose $w \in A(u) - V'$
  visit w
  $V' = V' \cup \{w\}$
  Put w on stack
  $u \leftarrow w$
else $u \leftarrow top(S)$ [Pop the stack]
endif
endrepeat

Binary Trees

In a binary tree, each node has at most two children (i.e., has outdegree at most two).

- We call one of them the *left* child and the other the *right* child.

Binary trees are useful because:

- Many things (like sorting, arithmetic evaluation, etc.) can be expressed as binary trees
- because each node has at most two children, they can be represented efficiently in a computer.

Traversing Binary Trees

Three standard methods:

- Preorder traversal:
  - Process the root
  - Traverse the left subtree (by preorder)
  - Traverse the right subtree (by preorder)
- Inorder traversal:
  - Traverse the left subtree (by Inorder)
  - Process the root
  - Traverse the right subtree (by Inorder)
- Postorder traversal:
  - Traverse the left subtree (by Postorder)
  - Traverse the right subtree (by Postorder)
  - Process the root

Example

Preorder Inorder Postorder

- Preorder is essentially depth-first search
- Postorder is the obvious way to process the arithmetic expression

Algorithm for Preorder Traversal

Input $G(V, E)$ [a binary tree]

Algorithm Preorder
  procedure traverse(u)
    process u
    if u has left child lc **then** traverse(lc)
    if u has right child rc **then** traverse(rc)
  endproc
  traverse($root$)

Spanning Trees

A *spanning tree* of a connected graph $G(V, E)$ is a connected acyclic subgraph of G, which includes all the vertices in V and only (some) edges from E.

Think of a spanning tree as a "backbone"; a minimal set of edges that will let you get everywhere in a graph.

- Technically, a spanning tree isn't a tree, because it isn't directed.

Constructing a Spanning Tree

Theorem: Every connected graph has a spanning tree.

Proof: Do the obvious thing: start a some node and grow the tree. The following algorithm does it.

Input $G(V, E)$ [a connected graph]

Algorithm SpanTree
  Choose a vertex v in G
  $V' \leftarrow \{v\}$ [Initialize spanning tree $T(V', E')$]
  $E' \leftarrow \emptyset$
  repeat while $V' \neq V$
    Pick $c \in V'$ and $c' \in V - V'$ such that $\{c, c'\} \in E$
    $V' \leftarrow V' \cup \{c'\}$
    $E' \leftarrow E' \cup \{\{c, c'\}\}$
  endrepeat
Output $T(V', E')$

Why does this work?

- After each iteration of the loop, T is a tree which spans the subgraph containing the vertices in V'.
- If $V' \neq V$, you can always find a new vertex to add to V', since G is connected.

- Notice: we could also use DFS and BFS to find spanning trees, as well as Dijkstra's shortest path algorithm.

Minimum Spanning Trees

If we have weights on the edges, we often want to find a spanning tree with minimum weight. A slight modification of the previous algorithm does it.

Input $G(V, E)$ [a connected graph]
  $w(e)$ for all $e \in E$ [Weights on edges]

Algorithm MinSpanTree
  Choose a vertices v, v' in G
    such that $\{v, v'\}$ has minimal weight
    (break ties arbitrarily)
  $V' \leftarrow \{v, v'\}$ [Initialize spanning tree $T(V', E')$]
  $E' \leftarrow \{\{v, v'\}\}$
  repeat while $V' \neq V$
    Pick $c \in V'$ and $c' \in V - V'$ such that $\{c, c'\} \in E$
      and $\{c, c'\}$ *has minimal weight*
    $V' \leftarrow V' \cup \{c'\}$
    $E' \leftarrow E' \cup \{\{c, c'\}\}$
  endrepeat
Output $T(V', E')$

MinSpanTree: Correctness

For simplicity, suppose the edges weights are all unique.

Lemma: If the vertices of $G(V, E)$ are divided into two disjoint sets V_1 and V_2, then any minimum spanning tree of G contains the minimum weight edge e connecting a vertex in V_1 to a vertex in V_2.

Proof (sketch): Suppose not. Then there is a min-weight spanning tree T that doesn't contain e. Add e to try. There must now be a cycle containing e. Take away some other edge e' that a "bridge" between V_1 and V_2. This gives you a spanning tree T' with less weight than T. That's a contradiction.

Proof of Algorithm's Correctness: At every stage, we're adding the edge of minimum weight between the vertices in V' and those in $V - V'$, so these edges must all be on the spanning tree.

Game Trees

Trees are particularly useful for representing and analyzing games.

Example: *Daisy* is a game where players alternate picking petals from a daisy. A player gets to pick 1 or 2 petals. Whoever picks the last one wins. (There's another version where whoever takes the last one loses; both get analyzed the same way.)

Here's the game tree for 4-petal daisy: