CMPSCI 250: Introduction to Computation

Lecture #17: Trees
David Mix Barrington
2 April 2013

Trees

- Examples of Tree Structures
- Simple Graphs With No Cycles
- Rooted Tree Classes
- Recursion and Induction for Trees
- Numerical Properties of Trees
- Applications of Trees

Examples of Tree Structures

- In Lecture #14 we discussed a number of tree structures used in computer science: **rooted trees**, **binary trees**, and **expression trees**. Each class of objects had a recursive definition, and we could prove facts about them by induction.

- Rooted trees model hierarchies of various kinds, with a **root** at the top and a single **parent** for every element except the root. A leaf is a node that has no children, and a non-leaf is also called an internal node. The hierarchies model file systems, the object hierarchy in Java, and the reporting trees of organization -- as long as in each case an element cannot have more than one parent. If no parent can have more than two children, the tree is **binary**.

- Expression trees had leaves labeled by elements and internal nodes labeled by operators. The value of an expression tree is defined inductively.

Simple Graphs With No Cycles

- Rosen defines a **circuit** to be a non-trivial path from a vertex to itself. A **simple circuit** is one that does not reuse an edge. He also defines the **cycle graph** C_n (for $n \geq 3$) to be the simple graph with n vertices and n edges, with a circuit containing all the edges.

- If a simple graph contains a simple circuit, it may still reuse vertices. But if any simple circuit exists, there also exists a simple circuit that does not. Since the edges of this circuit form a subgraph that is isomorphic to some C_n, we will call such a circuit a cycle as well.

- If a simple graph has no simple circuit, we call it a **forest**. The connected components of a forest are called trees, and a single graph is a **tree** if it is both connected and has no cycle (no simple circuit).

- Note that we have many definitions of “tree” -- these are **graph-theoretic**.

Unique Simple Paths

- Here is a fundamental property of a graph-theoretic tree. If u and v are any two vertices in a tree T, there is a **unique simple path** in T from u to v.
- In fact, if G is any simple graph, G is a tree *if and only if* for every u and v, there is a unique simple path from u to v. Let's prove this.
- By definition, G is connected if and only for any u and v, there is *at least one* path from u to v. So we must show that if G is connected, there is a cycle in G if and only there exist u and v with *more than one* simple path from u to v.
- If C is a cycle in G, let u and v any two distinct vertices on C. We have two paths along C from u to v, one in each direction.
- The converse is a bit more complicated.

Finishing the Unique Simple Paths Proof

- We assume that G is a simple graph, that u and v are vertices, and that there are at least two different simple paths from u to v. We want to prove that there exists a cycle in G.

- The first idea would be to take the circuit that goes from u along the first path to v, then back along the second path to u. But this may not be a simple cycle, as the two paths could share edges and even vertices.

- Rosen proves in the solution to Exercise 10.4.59 that in this case we have a cycle. Start at u and follow the paths together as long as they share edges. Let s be the first node at which they split. (If s = v, whichever path is not finished is a simple circuit from v to v.) Now follow the first path until it hits the second path again (which might or might not be at v). Then follow the other path back to s. This is a simple circuit, and if a simple circuit exists then a cycle must exist.

Rooted Tree Classes

- What's the connection between graph-theoretic trees and the rooted trees from Lecture #14?

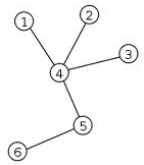
- Let T be a graph-theoretic tree and let r be any vertex in T. If we declare r to be the root of T, we can redraw T with r at the top (at level 0), r's neighbors at level 1, r's neighbors' neighbors at level 2, and so forth. Every other vertex v has a unique simple path to r, and the length of this path tells us what level v is at.

- This is a proper rooted tree because every node x (except r) has a parent in the tree. This is the only one of its neighbors that is closer to r, and is the next node on the unique simple path from x to r.

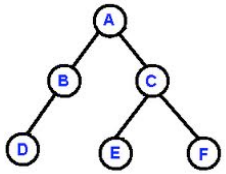
- A single graph-theoretic tree gives a different rooted tree for each root.

iClicker Question #1: Rooted Trees

- **Are these two trees isomorphic (as simple graphs) to one another**?



- (a) No, because the bottom one has a root and the top one does not.
- (b) No, because the top one has a node of degree four and the bottom one does not.



- (c) Yes, because each one has six nodes and five edges.
- (d) Yes, because if you make 4 the root of the top one and redraw it you can get a copy of the bottom one.

Rooted Tree Vocabulary

- Remember that the **degree** of a node in a rooted tree is the number of its children (not its degree as a member of the simple graph).
- A rooted tree is **k-ary** if no node has degree more than k, and **binary** if no node has degree more than 2. It is **full k-ary** if every internal node has degree exactly k, and **balanced** if every leaf is at the same level.
- Sometimes we care about the order of the children of an internal node, and sometimes we don't. Often in a binary tree we distinguish the **left** and the **right** children of a node.
- For each tree class, an **isomorphism** between two trees is a bijection of the vertices that preserves the important structure, like the parent-child relationship and, if order of children matters, the order of children.

Recursion and Induction on Trees

- As we saw in Lecture #14, we have recursive definitions for rooted trees, k-ary trees, and full k-ary trees, with or without order on the children of a node.
- Each recursive definition gives us a **Law of Induction** for the class. For example, we defined a rooted tree to be either a single node or a root node with one or more children, each of which is the root of a distinct rooted tree.
- If P(T) is a property of rooted trees, we can prove $\forall T{:}P(T)$ by induction. The base case is to prove P(R) for all single-node trees R.
- For the inductive case, we assume that T is a tree whose root has children that are the roots of $T_1,\ldots, T_k$, and that $P(T_i)$ holds for all i from 1 to k. We then need to prove P(T) from these assumptions.

Numerical Properties of Trees

- A bit of observation of trees should convince you that the number of edges in any tree is always exactly one less than the number of vertices. How can we prove this? Let's prove it by induction on rooted trees, since any graph-theoretic tree can be made into a rooted tree by choosing a root.

- The base case is for a one-node tree, which by definition has one vertex and no edges. So our equation $e = v - 1$ holds for the base case.

- The general case has T consisting of a root r with edges to the roots of k subtrees $T_1,..., T_k$. If each subtree T_i has e_i edges and v_i vertices, the IH tells us that $\forall i: e_i = v_i - 1$. Let's count the total edges and vertices of T. We have one root plus all the vertices in all the subtrees, for $1 + \sum(v_i)$ vertices. We have k edges from the root and all the edges in all the subtrees, for $k + \sum(e_i)$ edges. Applying the IH, we have $e = k + \sum(v_i - 1) = k + (\sum(v_i)) - k = \sum(v_i) = v - 1$. We have shown $e = v - 1$ for T and completed the induction.

Counting Nodes and Edges in Balanced k-ary Trees

- Balanced k-ary trees have a different recursive definition. The full balanced k-ary tree of height 0 is a single vertex. The FBkT of height i+1 consists of a root with k children, each of which is the root of an FBkT of height i. So an induction over FBkT's is really just an ordinary induction on the height.

- How many vertices and edges does an FBkT of height n have? Let's call these two functions V(n) and E(n). For the base case, we have one vertex and no edges, so V(0) = 1 and E(0) = 0. For the inductive case, our definition gives us the equations V(n+1) = 1 + kV(n) and E(n+1) = k + kE(n).

- For binary trees (k = 2) we can use these equations to prove $V(n) = 2^{n+1} - 1$ and $E(n) = 2^{n+1} - 2$. In general we get $V(n) = (k^{n+1} - 1)/(k - 1)$ and again E(n) = V(n) - 1.

- It's easier to count leaves, and we get just L(0) = 1, L(n+1) = kL(n), and $L(n) = k^n$.

iClicker Question #2: Full k-ary Trees

- A **full k-ary tree** is either a single node or a root with edges to *exactly k* roots of distinct full k-ary trees. Rosen proves that a full k-ary tree with n total vertices and i internal nodes must satisfy the equation $n = ki + 1$. If we prove this by induction, **what is our inductive step**?

- (a) For a one-node tree, $n = 1$ and $i = 0$ so $n = ki + 1$ is true.

- (b) Every child tree T_j satisfies $n_j = ki_j + 1$.

- (c) Given that each child tree T_j satisfies $n_j = ki_j + 1$, $n = ki + 1$.

- (d) Assume that $m = ki + 1$ for all m such that $m < n$.

Applications of Trees

- Let's conclude by looking at some of the uses of trees in computer science, some of them familiar from CMPSCI 187 and some not.

- In 187 we looked at **binary search trees**, where each node of the tree stores an object from some ordered class. If x is the object at an internal node v, every object in x's left subtree is ≤ x and every object in the right subtree is ≥ x. We can find any object in the tree by using the internal node values to guide us from the root to where it either is or isn't.

- We also saw **heaps**, where the element at a node must be ≥ the elements at each of its children. These let us implement **priority queues** using O(log n) adjustments to add a new element or remove the largest element.

- In each case, we use the tree structure to get short paths to every element.

Lower Bounds From Decision Trees

- In a **decision tree**, each node represents a position during the execution of an algorithm. Each internal node is labeled with a question, and there is a child for each answer to the question. A leaf represents a position where the algorithm has come to a conclusion.

- If x is the number of possible answers to the question, the decision tree must have at least x leaves. If the degree of the tree is k and its depth is d, this gives us the inequality $k^d \geq x$ which translates to $d \geq \log_k x$.

- In a **comparison-based sorting algorithm**, k = 2 because when we compare two elements there are two possible answers. Since there are n! possible orders for n elements, the number of comparisons used in the worst case must be at least $\log_2(n!)$ which can be shown to be at least O(n log n). We call this a **lower bound** on the performance of any such sorting algorithm.

iClicker Question #3: Decision Trees

- You have n coins, all weighing the same except for one which is slightly heavier. You have a balance scale that will tell you whether one group of coins is heavier, the same weight, or lighter than another group. **Which of these four statements is false**?

- (a) If $n > 3^k$, you **cannot** do it with k weighings.

- (b) If $n = 3^k$, you **can** do it with k weighings.

- (c) If $n = 2^k$, you **can** do it with k weighings.

- (d) If $n > 2^k$, you **cannot** do it with k weighings.

Prefix Codes and Huffman Coding

- The ASCII and Unicode alphabets each represent letters by bit strings, and use strings of the same length for each letter. This makes it easy to take a stream of bits and group them into the string for each letter.

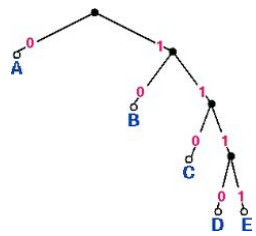
- A **variable-length code** can be more efficient than these **fixed-length codes**, if more common letters are represented by shorter strings. But if the codes for letters are different lengths, we need a way to break up a string correctly.

- A set of strings is called a **prefix code** if no string is a proper prefix of another. We can arrange the strings of a binary prefix code as the leaves of a binary tree, with an internal node for each proper prefix of a code word. When we read the stream of code bits, we travel down the tree until we hit a leaf, at which time we know we have a code word and return to the root.

- **Huffman's algorithm** (presented in Rosen) is a way to get the tree for the most efficient code for a given alphabet, given the probability of each letter.

iClicker Question #4: A Prefix Code

- The tree at right represents a prefix code for converting strings over {A,B,C,D,E} into binary strings and vice versa. In this code, what is the translation of the string “BEAD”?



- (a) 10011111110
- (b) 1101111001110
- (c) 11100111110
- (d) 10111101110

Game Trees

- We've seen a number of two-player games in the course so far. We can model each game by a **game tree**, where a node represents a position and is labeled by which player is next to move. The root is the start position and a leaf represents a position where the game is over. Leaves are also labeled by the final score of the game (if you like, how much Black must pay to White).

- We've used the idea that in any **deterministic** game of **perfect information**, each player has an **optimal strategy** and the game has a **value** that each player can either achieve or exceed against any opposing strategy.

- We prove this result by induction on game trees. The base case is a one-node tree, where no one has a move and the value is fixed. For the inductive case, one player has a choice of moves, and by the IH each move has a value. So the optimal move is to choose the value best for the moving player, and the value of the new game is the best of those values for that player.