CPS196.2, Chapter 3: Trees

Note to djr: do spanning trees in sec4

1. Introduction and Preliminaries: Trees

In this chapter, we first examine trees, both as graphs and in their use as abstract data structures. Read also chapters 6 and 7 in J, DM (text).

A graph G is a pair, G = (X, E), with a set of vertices (or nodes), X, and a set of edges, E. Each (undirected) edge e in E is a vertex pair e = {x,y}, often written succinctly as e = xy. An x-y path of length k is a distinct set of vertices p = [x = x_0, x_1, x_2, ..., x_k = y] with x_i, x_{i+1} ∈ E, 0 ≤ i ≤ k − 1, except possibly for x = y and k ≥ 3 and then p is a cycle. A graph is connected if for every distinct pair x, y in X, there is an x-y path. If a graph is not connected, it has 1 ≤ k ≤ |X| connected components, which are connected subgraphs, G_k. A tree, T, is a connected acyclic graph, and a forest is an acyclic graph whose connected components are trees.

2. Tree Characterizations

Theorem T. The following are equivalent for a graph G = (X,E):

(T) G is a tree;
(P) for every pair of distinct vertices of x, y in X, there is a unique x-y path;
(EV1) G is connected with |X| = |E| + 1;
(EV2) G is acyclic with |X| = |E| + 1;
(UC) G_0 = (X, E ∪ xy), x, y ∈ X with xy ∉ E, has a unique cycle (this is sometimes written as G_0 = G + xy).

In addition we have the following characterization that provides a simple data structure for a tree. For x in X, let \( \text{adj}(x) = \{y \mid xy \in E\} \); for y ∈ adj(x), we say y is adjacent to x.

Theorem DT. T = (X, E) is a tree iff there is an integer ordering of the vertex set X, which we write as X = \( \bigcup_{i=k}^{n} x_i \), n − k + 1 = |X| (usually k = 1, and we may also call vertex x_i, vertex i), such that

(Order) each x_i is adjacent to a unique x_j with i<j, k ≤ i ≤ |X|. 
Example 1. The tree below has the order property:

![Diagram of a tree]

\[ T = [9, 6, 5, 5, 8, 10, 9, 9, 10] \]

Figure 1: Ordered Tree and Array Representation

Proof of ThmT:
We prove that the first 4 properties are equivalent and leave the 5th property as homework. In particular, we show (T) iff (P) and (P) implies (EV1) implies (EV2) implies (P).

(T) implies (P).
We show not(P) implies not(T). If for some distinct pair x,y, there is no x-y path, then G is not connected. On the other hand suppose there are two such x-y paths, p_{1} and p_{2}, and let z be the first vertex in p_{1} that is also in p_{2}. Then the vertices in the x-z subpath of p_{1} and the x-z subpath of p_{2} form a z-z cycle and G is not acyclic.

(P) implies (T).
Again we will do not(T) implies not(P). If G is not connected, then for some x, y, there is no x-y path so there is certainly not a unique one. If G has an x-x cycle c, there are two x-y paths in G for any y \neq x in c and uniqueness fails.

(P) implies (EV1).
Let e = xy be any edge. If we remove e and consider G-e, we first show that we will have a graph containing two trees T_{1} = (X_{1}, E_{1}) and T_{2} = (X_{2}, E_{2}) with x \in X_{1} and y \in X_{2}. We let X_{1} contain x and also any vertices in X such that there exists an x-z path in G not containing y. E_{1} contains the edges e = vw in E with u and v in E. Certainly T_{1} is acyclic since otherwise the cycle in T_{1} would be a cycle in G; it is connected by construction.
Hence $T_1$ is a tree. $T_2$ is similarly defined starting with $X_2$ which are the vertices in $X$ with a y-z path in $G$ not containing $x$. Notice that, by construction, $X_1$ and $X_2$ are disjoint; furthermore each vertex in $X$ is in $X_1$ or $X_2$ since there are (unique) x-w and y-w paths in $G$ for any $w$ in $X$.

Notice that we have just proved a little lemma that says that the removal of any tree edge breaks a tree into two tree components.

To finish the proof we use induction. $(P)$ implies $(EV1)$ is clear for $|X| = 1$; the induction hypothesis $(IH)$ asserts the implication for $|X| < n$. We consider $G$ with $|X| = n$ and remove any edge in $G-e$. By the lemma $G-e$ consists of two trees and by $(IH)$ both satisfy the edge-vertex relationship. Putting back the edge $e$, we see that $|X| = |E| + 1$ in $G$.

$(EV1)$ implies $(EV2)$.

**Define** $\text{deg}(x) = |\text{adj}(x)|$ and observe

**Proposition 1.** For any graph, $G = (X, E)$, $2 |E| = \sum_{x \in X} \text{deg}(x)$.

The truth of Prop1 follows from the observation that each edge has two vertices, which will both count in the degree sum. $(EV1)$ implies that $2(|X| - 1) = \sum_{x \in X} \text{deg}(x)$; hence in any tree with $|X| > 1$, there must be at least two vertices with $\text{deg} = 1$ (there can be no $\text{deg} = 0$ vertices by connectedness).

We now prove that $(EV1)$ implies $(EV2)$ by induction on $n = |X|$, the base being clear. Supposing the induction hypothesis $(IH)$ for $n-1$, let $G$ be such a tree with $n$ vertices, let $x$ have $\text{deg}(x) = 1$, $xy \in E$, and consider the graph $G - xy$. This reduced graph has $n-1$ vertices and $n-2$ edges and is connected. By $(IH)$ it is also acyclic. Now, since $\text{deg}(x) = 1$, $x$ cannot be part of a cycle in $G$; hence $G$ is also acyclic.

$(EV2)$ implies $(T)$.

We only need to show that $G$ is connected. If $G$ has $k$ connected components each is a acyclic and we have by applying $(EV2)$ to each component that $|X| = |E| + k$. But $(EV2)$ now says that $k = 1$; hence $G$ is connected.

**Exercise 1: prove (P) iff (UC).**

**Endproof.**

The proof of ThmDT is an easy induction and will be left as an exercise. Note that it is very much related to $(EV1)$ and $(EV2)$ and a proof might also use these properties. The importance of ThmDT is that we can use an array of size $|X| - 1$ to represent the tree as in Fig1.

**Exercise 2: Prove ThmDT.**
Is this array data structure for a tree a good one? That, of course, depends on what we want to do with it. Suppose we want to compute the unique x-y path, for example, the 2-3 path of Fig1. It is easy to see that we start with the smaller vertex, namely 2 and access the array to get 6. Then we continue inductively (or using a loop structure like while) and find that 3 is now the smaller vertex. At this point we have the partial paths \( p_1 = [2, 6] \) and \( p_2 = [3, 5] \). 5 is the smaller vertex so we augment \( p_2 = [3, 5, 8] \). Eventually both paths will reach 10 and we concatenate the paths without duplicating 10. The details are in

**Exercise 3:** Write an m-function in matlab to find the unique x-y path algorithm as above. Prove it correctly computes the unique x-y path of length \( k \) in \( k \) accesses to the tree array. How many comparisons?

As a final note, we observe that the array data structure is rather inflexible if we want to add or delete vertices in some manner that maintains a tree structure. In that case, we can replace the array by a linked list.

### 3. Rooted Trees

A tree \( T \), rooted at \( r \) can be defined recursively as follows: \( T_r = (X, E) \) with root \( r \) and children, \( C = \{c_1, \ldots, c_k\} \), which may be empty, such that, if \( C \neq \emptyset \), and each \( c_k \) is the root of a subtree, \( C_k \).

![rooted tree](image)

**Figure 2: rooted tree**

The vertices in any \( C_k \) are descendents of \( r \), and \( r \) is an ancestor of the vertices in any \( C_k \). \( r \) is also called the parent of its children. Children of the same parent are called siblings. A terminal vertex (or leaf) is a vertex without children. Clearly a vertex, \( v \), in a rooted tree is terminal if and only if \( \deg(v) = 1 \) (Why?). If \( v \) is not a terminal vertex and not the root, it is an internal vertex (note: Sometimes the root node is counted as an internal node; i.e., text). The height, \( h \), of a rooted tree is the length of the longest path from the root to a terminal vertex. Vertices, \( v \), that have the same \( v \)-r path length \( k \) are said to be on level \( k \). Therefore the height is the maximum level number.
Notice that in the array representation of the tree in Fig. 1, we can regard the tree as rooted at vertex 10. It is clear there are two subtrees since 10 appears twice in the array. Note that given any tree we can build a tree array with root implicitly represented as vertex $|V|$ (or vertex 1 if we want to go in decreasing order) using the construction implicit in the proof of ThmDT. Indeed if we order the vertices in $C_k, C_{k-1}, \ldots, C_1$, the array will capture more of the recursive structure. For example $T$ of Fig. 1 becomes $T = [7, 7, 5, 5, 6, 7, 10, 9, 10]$.

A **binary tree** is a rooted tree, say $T_{\text{root}}$, with at most two (left and right) subtrees, $T_l$ and $T_r$, respectively. As in the definition of a rooted tree, each subtree is again a binary tree. Binary trees are usually represented by linked data structures, each node containing the key (element from a totally ordered set) and three pointers: $p$, $l$ and $r$, to the parent, left child, and right child respectively. For example consider the binary tree in Fig. 3 below.

![Figure 3](image)

If we use $P$, $L$, and $R$ arrays (which could have been combined into fields of a linked list) for the pointers, we have:

- $P = [\text{nil}, 1, 1, 2, 2, 3, 3, 5, 5, 6, 6]$;
- $L = [2, 4, 6, \text{nil}, 8, 10, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}]$;
- $R = [3, 5, 7, \text{nil}, 9, 11, \text{nil}, \text{nil}, \text{nil}, \text{nil}, \text{nil}]$.

Notice that $P$ is essentially our $T$ array data structure but each node points to the unique node of lower number (which, of course, is equivalent).
Binary trees are used to store information in nodes in a field or a pointer, which is a co-field of the key. We assume that the key is a unique identifier. The first given key is placed in the root. Then, given a subsequent key, we search from the root placing the new key in the first vacant left child if newkey < key(parent) or the first vacant right child if newkey > key(parent). See text.

When binary trees are used in this manner, they are called search trees. Clearly, the complexity to find (or not find) any key, in the worst case, is related to the height, and we would like to use binary trees that are “balanced” in a sense that the height is minimized.

A full binary tree is one where each node has two children or none. A complete binary tree with k levels is a full binary tree where all vertices on level j, 0 \leq j \leq k − 1, have exactly two children and level k vertices are leaves. Clearly, a complete k level binary tree has height h = k and n = 2^{h+1} - 1 vertices.

**Proposition 2.** A full binary tree with n nodes has one root node, |I| internal nodes, and |T| terminal nodes where |T| = |I| + 2 and n = 2|I| + 3 = 2|T| - 1.

**Proof.**

By Proposition 1, \( \sum_{x \in \text{nodes}} \deg(x) = 2|E| \). Let |T| and |I| be the number of terminal nodes and internal nodes, respectively. Then

\[
\sum_{x \in T} \deg(x) + \sum_{x \in I} \deg(x) + \deg(\text{root}) = 2|E| = 2(n - 1) = 2(|T| + |I|).
\]

Since each terminal node has deg = 1, each internal node has deg = 3, and deg(root) = 2, we obtain |T| + 3|I| + 2 = 2(|T| + |I|).

**End proof.**

**Proposition 3.** \( \log_2(|T|) \leq h \).

**Proof.**

We first note that any tree of height h cannot have more vertices than a complete binary tree of height h; that is; \( n \leq 2^{h+1} - 1 \). From Prop 1 we obtain \( n = 2|T| - 1 \); hence, |T| \leq 2^h.

**End proof.**

Notice that a binary tree such as in Fig1 can be traversed in three basic ways.

**preorder**

order the root;
if nonempty, preorder left subtree
else if nonempty, preorder rightsubtree;

For Fig3, preorder = [1, 2, 4, 5, 8, 9, 3, 6, 10, 11, 7].

We omit “nonempty” for convenience from now on.

**inorder**

from root
inorder leftsubtree;
order root;
inorder rightsubtree;

For Fig3, inorder = [4, 2, 8, 5, 9, 1, 10, 6, 11, 3, 7].

**postorder**

from root
postorder leftsubtree;
postorder rightsubtree;
order root;

For Fig3, postorder = [4, 8, 9, 2, 10, 11, 6, 7, 3, 1]

**Example:** arithmetic expression tree

![arithmetic expression tree](image)

* Figure 4: $T_{ae}$
If we traverse this tree we see that

\[ \text{inorder}(T_{ae}) = [A, +, B, *, C, -, D, /, E], \]

and we need parentheses to disambiguate this to \((A + B) \cdot C - D / E\), assuming that \(*\) and \(/\) have precedence over \(+\) and \(-\).

We see also that

\[ \text{postorder}(T_{ae}) = [A, B, +, C, *, D, E, /, -], \]

and we use the “enter key” as
A enter, B + C * D, enter E / -.

---

4. Spanning Trees

Definition
Bfs
Dfs
General