Lecture 16: Trees

We will wind up our chapter on relational thinking by discussing an important class of graphs — *trees* — along with some applications.

Here is the formal definition:

> A **tree** is a connected graph containing no cycles.

(Here, "connected" means that the graph consists of a single component as opposed to multiple islands, and that there is a well-defined path from every node to every other node.)

You have probably seen trees before; the concept of a *binary search tree* appears in sorting and searching applications. But binary trees are only special cases of trees; for example, both the line graph and the star graph defined previously are *also* examples of trees.

Trees are typically organized by defining a *root* (usually drawn at the top), *internal nodes*, and *leafs*. One can imagine a hierarchy where the root gives rise to some number of *children*, who in turn are parents of other child nodes, and so on. Leafs are typically drawn at the bottom, and are without any children. (Exercise: what is the degree of any leaf in a tree?). A binary tree is one in which every non-leaf node has two children.

The *level* of any given node denotes how far away it is from the root; so the root itself is Level 0, its children are Level 1, and so on.

Properties of trees

Let us prove a few simple properties about trees.

> **Theorem**. Designate any node as the "root". Then, there is a unique simple path from root to any other node.

We can prove this statement via contradiction as follows: suppose there were two *distinct* paths from the node. Imagine walking from the root to the node along the first path and walking back to the root along the second. This creates a cycle – which is a contradiction since a tree cannot have any cycles.

> **Theorem** Every tree with n nodes has $n - 1$ edges.

This can be proved as follows: by the above property, since there is a unique path from the root to any other node, one can imagine a "direction" being assigned to each edge in the tree. In other words, every non-root node has exactly one edge that "points" into it – that represents the end of the path pointing from the root to that node. There are $n - 1$ non-root nodes, and hence there are $n - 1$ non-root edges. Done!

> **Theorem**. Consider any *connected* subgraph of any tree. Then the subgraph itself is a tree.

We can prove this by contradiction as well. Suppose the connected subgraph under consideration is not a tree. Then, by definition it has to have cycles. But if it contains a cycle then that cycle has to be part of the bigger tree as well – which is a contradiction since trees cannot have cycles.

In this sense, trees obey a natural *recursive* property – every tree is composed of smaller trees. Thinking in terms of recursion is a subtle but important concept in all of computing. More about recursion in later lectures.

Applications

Trees are used all over the place in computer science and engineering.

Data Structures

Trees form a very important component of several types of data structures. You may have heard of binary search trees (or BSTs). A BST is a fundamental data structure used in search and sort applications.

BSTs are also used in applications such as *spell-check*. For example, suppose we are given a list of words that comprise a given dictionary. For simplicity, let us assume that the dictionary contains the following words:

> macchiato, complexify, clueless, jazzed, posit, phish, sheaves

Now, if a user types in a new word, your spell-check program needs to quickly identify whether the given word corresponds to one of the elements of the dictionary. A naive (but inefficient) way is to store the list as given, and walk down the list while comparing the user's word with each element in the list. A *better* way is to construct a binary tree by inserting elements either to the left or the right of a node based on whether it precedes or succeeds the word according to alphabetical order. In the above case, the binary tree would look like this:

- macchiato as the root,
- complexify as the left child of macchiato
- clueless as the left child of complexify
- jazzed as the right child of complexify
- posit as the right child of macchiato
- phish as the left child of posit
- sheaves as the right child of posit

The above is an example of a *balanced* binary search tree, since every node (except the leaves) has exactly two children.

Construction of a BST from a list of elements is achieved via the above procedure. The reverse — printing out the elements of a tree — can be achieved in a variety of ways. Essentially, the procedure involves traversing the tree, and printing the contents of each node that you visit. There are three common techniques:

- Pre-order traversal
- Post-order traversal
- In-order traversal

Pre-order traversal is defined via the following algorithm:

Input: Binary tree T

PreOrder(T):

if $T \neq \emptyset$:

- visit root of T
- PreOrder(left-subtree of root)
- PreOrder(right-subtree of root)

return

So, a pre-order traversal of the BST that we constructed above would produce:

macchiato, complexify, clueless, jazzed, posit, phish, sheaves.

Post-order traversal is identical, except that the order is different; we first call PreOrder(left-subtree), then PreOrder(right-subtree), then visit the root. In-order traversal is also similar except that visiting the root is in the middle. (Exercise: what do the post-order and in-order traversals of the above BST produce?)

The above 3 are different ways to do *depth first search*. There is also a different way to traverse trees known as *breadth first search*. We won't go into any of these in detail; see the textbook (Chapter 5 in particular) for a lengthier discussion.

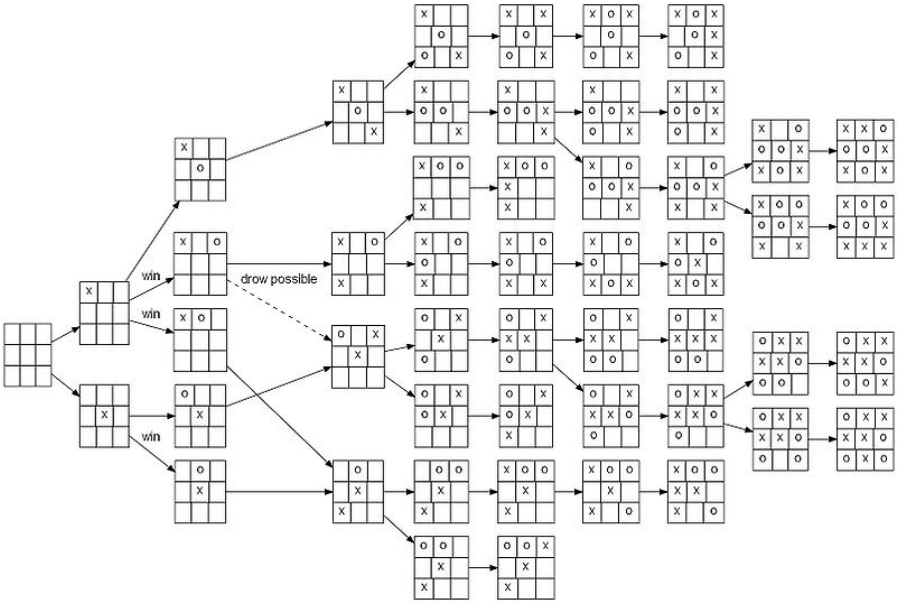

Figure 1: Source: wikipedia

Game trees

A particularly interesting application of trees is in designing strategies for game-playing computer programs, which is a challenge in a broader research area known as *artificial intelligence*. Suppose we are playing a two-player game (such as tic-tac-toe, or chess, or checkers, or Hearthstone). The idea is to model the evolution of any given game between two opponents as trees where nodes indicate positions (or states of the game-world) and edges indicate possible moves from each position. This is called a *game tree*. For example, (a small part of) the game tree for tic-tac-toe is shown above.

Given the state of any game, how does a game AI decide the next best move? There are different algorithms for this, but at some level they all do some kind of traversal of the game tree to figure out which move gives the best possible value. Tic-tac-toe has a relatively small game tree (which is why it is impossible to win in tic-tac-toe against a properly trained AI) but more complex games such as chess has far too many nodes for a search to properly work. (Stuff like depth-first-search are too complicated, so other algorithms are used.) In such cases, the AI searches a *partial* game tree, or as many levels as it can within a max available time limit.