7.4. Decision Trees, Tree Isomorphisms

7.4.1. Decision Trees. A *decision tree* is a tree in which each vertex represents a question and each descending edge from that vertex represents a possible answer to that question.

Example: The Five-Coins Puzzle. In this puzzle we have five coins C_1, C_2, C_3, C_4, C_5 that are identical in appearance, but one is either heavier or lighter that the others. The problem is to identify the bad coin and determine whether it is lighter or heavier using only a pan balance and comparing the weights of two piles of coins. The problem can be solved in the following way. First we compare the weights of C_1 and C_2. If C_1 is heavier than C_2 then we know that either C_1 is the bad coin and is heavier, or C_2 is the bad coin and it is lighter. Then by comparing say C_1 with any other the other coins, say C_5, we can determine whether the bad coin is C_1 and is heavier (if C_1 it is heavier than C_5) or it is C_2 and is lighter (if C_1 has the same weight as C_5). If C_1 is lighter than C_2 we proceed as before with "heavier" and "lighter" reversed. If C_1 and C_2 have the same weight we can try comparing C_3 and C_4 in a similar manner. If their weight is the same then we know that the bad coin is C_5, and we can determine whether it is heavier or lighter by comparing it to say C_1. The corresponding decision tree is the following:

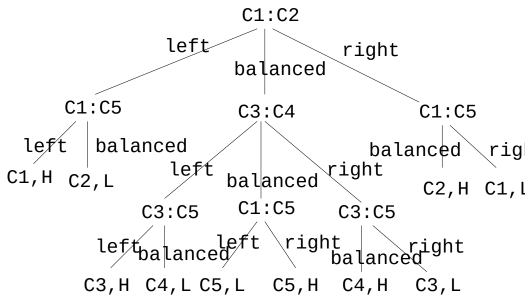

FIGURE 7.15. Decision tree for the 5 coins puzzle.

In each vertex "$C_i : C_j$" means that we compare coins C_i and C_j by placing C_i on the left pan and C_j on the right pan of the balance, and each edge is labeled depending on what side of the balance is heavier. The terminal vertices are labeled with the bad coin and whether it is heavier (H) or lighter (L). The decision tree is optimal in the sense that in the worst case it uses three weighings, and there is no way to solve the problem with less than that—with two weighings we can get at most nine possible outcomes, which are insufficient to distinguish among ten combinations of 5 possible bad coins and the bad coin being heavier or lighter.

7.4.2. Complexity of Sorting.

Sorting algorithms work by comparing elements and rearranging them as needed. For instance we can sort three elements a_1, a_2, a_3 with the decision tree shown in figure 7.16

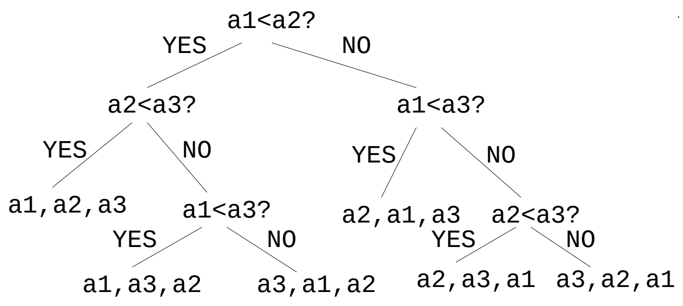

FIGURE 7.16. Sorting three elements.

Since there are $3! = 6$ possible arrangements of 3 elements, we need a decision tree with at least 6 possible outcomes or terminal vertices. Recall that in a binary tree of height h with t terminal vertices the following inequality holds: $t \leq 2^h$. Hence in our case $6 < 2^h$, which implies $h \geq 3$, so the algorithm represented by the decision tree in figure 7.16 is optimal in the sense that it uses the minimum possible number of comparisons in the worst-case.

More generally in order to sort n elements we need a decision tree with $n!$ outcomes, so its height $h(n)$ will verify $n! \leq 2^{h(n)}$. Since $\log_2(n!) = \Theta(n \log_2 n)$,[1] we have $h(n) = \Omega(n \log_2 n)$. So the worse case complexity of a sorting algorithm is $\Omega(n \log_2 n)$. Since the merge-sort algorithm uses precisely $\Theta(n \log_2 n)$ comparisons, we know that it is optimal.

7.4.3. Isomorphisms of Trees.

Assume that T_1 is a tree with vertex set V_1 and T_2 is another tree with vertex set V_2. If they are rooted trees then we call their roots r_1 and r_2 respectively. We will study three different kinds of tree-isomorphisms between T_1 and T_2.

[1]According to Stirling's formula, $n! \approx n^n e^{-n} \sqrt{2\pi n}$, so taking logarithms $\log_2 n! \approx n \log_2 n - n \log_2 e + \frac{1}{2} \log_2 (2\pi n) = \Theta(n \log_2 n)$.

1. Usual graph-isomorphism between trees: T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency, i.e., $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.

2. Root-tree-isomorphism: T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency and the root vertex, i.e.:
   (a) $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.
   (b) $f(r_1) = r_2$.

3. Binary-tree-isomorphism: Two binary trees T_1 and T_2 are isomorphic if there is a bijection $f : V_1 \to V_2$ that preserves adjacency, and the root vertex, and left/right children, i.e.:
   (a) $f(v)$ is adjacent to $f(w)$ if and only if v is adjacent to w.
   (b) $f(r_1) = r_2$.
   (c) $f(v)$ is a left child of $f(w)$ if and only if v is a left child of w.
   (d) $f(v)$ is a right child of $f(w)$ if and only if v is a right child of w.

Example: Figure 7.17 shows three trees which are graph-isomorphic. On the other hand as rooted trees T_2 and T_3 are isomorphic, but they are not isomorphic to T_1 because the root of T_1 has degree 3, while the roots of T_2 and T_3 have degree 2. Finally T_2 and T_3 are not isomorphic as binary trees because the left child of the root in T_2 is a terminal vertex while the left child of the root of T_3 has two children.

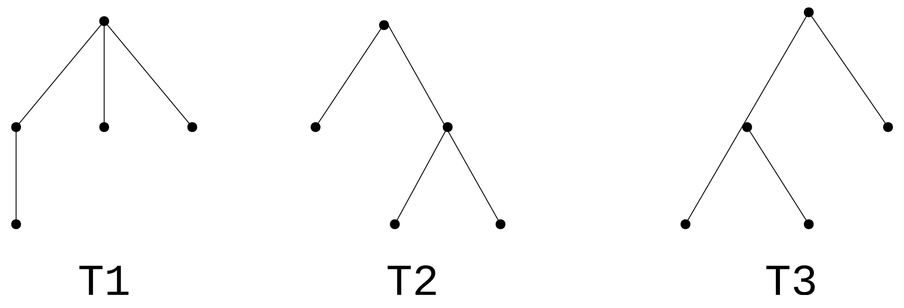

FIGURE 7.17. Trees with different kinds of isomorphisms.

Exercise: Find all non-isomorphic 3-vertex free trees, 3-vertex rooted trees and 3-vertex binary trees. *Answer*: Figure 7.18 shows all 5 non-isomorphic 3-vertex binary trees. As rooted trees T_2–T_5 are isomorphic, but T_1 is not isomorphic to the others, so there are 2 non-isomorphic 3-vertex rooted trees represented for instance by T_1 and T_2. All of them are isomorphic as free trees, so there is only 1 non-isomorphic 3-vertex free tree.

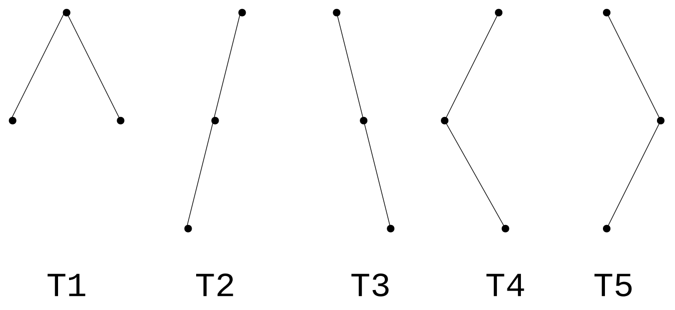

FIGURE 7.18. Non-isomorphic binary trees.

7.4.4. Game Trees. Trees are used in the analysis of some games. As an example we study the following game using a tree: Initially there are two piles with 3 coins and 1 coin respectively. Taking turns two players remove any number of coins from one of the piles. The player that removes the last coin loses. The following tree represents all possible sequences of choices. Each node shows the number of coins in each pile, and each edge represents a possible "move" (choice) from one of the players. The first player is represented with a box and the second player is represented with an circle.

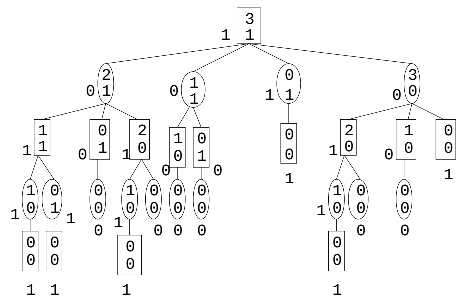

FIGURE 7.19. Tree of a game.

The analysis of the game starts by labeling each terminal vertex with "1" if it represents a victory for the first player and "0" if it represents a victory for the second player. This numbers represent the "value" of each position of the game, so that the first player is interested in making it "maximum" and the second player wants to make it "minimum". Then we continue labeling the rest of the vertices in the following way. After all the children of a given vertex have been labeled, we label the vertex depending on whether it is a "first player" position (box) or a "second player" position (circle). First player positions are labeled with the maximum value of the labels of its children, second player positions are labeled with the minimum value of the labels of its children. This process is called the *minimax procedure*. Every vertex labeled "1" will represent a position in which the first player has advantage and can win if he/she works without making mistakes; on the other hand, vertices labeled "0" represent positions for which the second player has advantage. Now the strategy is for the first player to select at each position a children with maximum value, while the second player will be interested in selecting children with minimum value. If the starting position has been labeled "1" that means that the first player has a winning strategy, otherwise the second player has advantage. For instance in the present game the first player has advantage at the initial position, and only one favorable movement at that point: $\binom{3}{1} \rightarrow \binom{0}{1}$, i.e., he/she must remove all 3 coins from the first pile. If for any reason the first player makes a mistake and removes say one coin from the first pile, going to position $\binom{2}{1}$, then the second player has one favorable move to vertex $\binom{0}{1}$, which is the one with minimum "value".

Alpha-beta pruning. In some games the game tree is so complicated that it cannot be fully analyzed, so it is built up to a given depth only. The vertices reached at that depth are not terminal, but they can be "evaluated" using heuristic methods (for instance in chess usually losing a knight is a better choice than losing the queen, so a position with one queen and no knights will have a higher value than one with no queen and one knight). Even so the evaluation and labeling of the vertices can be time consuming, but we can bypass the evaluation of many vertices using the technique of *alpha-beta pruning*. The idea is to skip a vertex as soon as it becomes obvious that its value will not affect the value of its parent. In order to do that with a first player (boxed) vertex v, we assign it an *alpha value* equal to the maximum value of its children evaluated so far. Assume that we are evaluating one of its children w, which will be a second player (circled) position. If at any point a children of w gets a value less than or equal to the alpha value of v then it will become obvious that the value of w is going to be less than the current alpha value of v, so it will not affect the value of v and we can stop the process of evaluation of w (prone the subtree at w). That is called an *alpha cutoff*. Similarly, at a second player (circled) vertex v, we assign a *beta value* equal to the minimum value of its children evaluated so far, and practice a *beta cutoff* when one of its grandchildren gets a value greater than or equal to the current beta value of v, i.e., we prone the subtree at w, where w is the parent of that grandchildren.

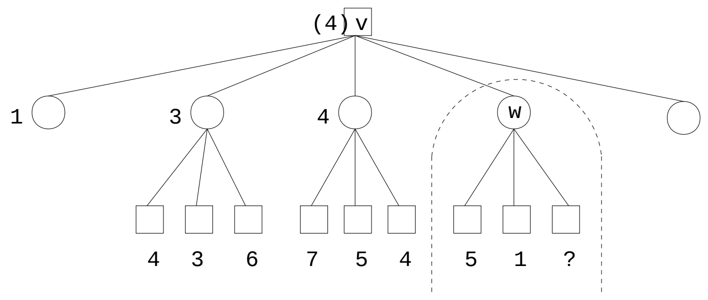

FIGURE 7.20. Alpha cutoff.