Applications of trees

Introduction

We will discuss 2 problems that can be studied using trees.

1) What series of decisions should be made to find an object with a certain property in a collection of objects of a certain type?

3) How should a set of characters be efficiently coded by bit strings?

Decision Trees

We have seen that rooted trees can be used to model problems in which a series of decisions leads to a solution.

For instance, a binary search tree can be used to locate items based on a series of comparisons, where each comparison tells us whether we have located the item, or whether we should go right or left in a subtree.

Decision Trees

A rooted tree in which each internal vertex corresponds to a decision (принятию решения), with a subtree at these vertices for each possible outcome of the decision, is called a decision tree.

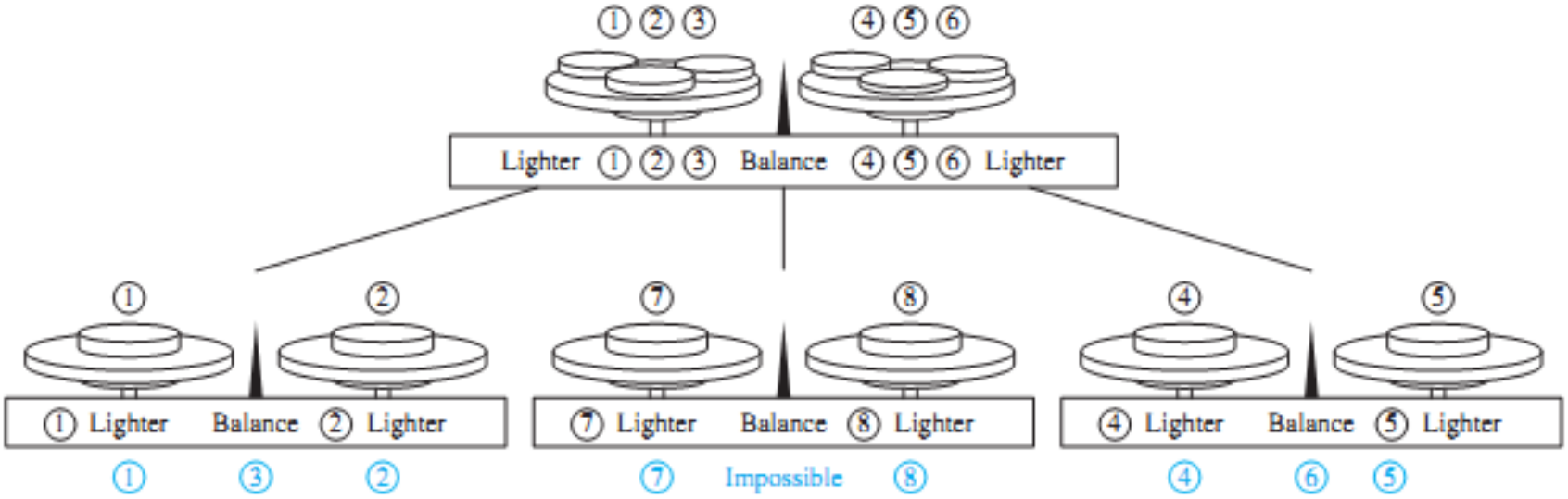
The possible solutions of the problem (возможные решения проблемы) correspond to the paths to the leaves of this rooted tree.

EXAMPLE

Suppose there are 8 coins: 7 with the same weight, and a counterfeit coin that weighs less than the others.

How many weighings are necessary using a balance scale to determine which of the 8 coins is the counterfeit one?

Give an algorithm for finding this counterfeit coin.



Solution: There are 3 possibilities for each weighing on a balance scale.

The 2 pans can have equal weight, the first pan can be heavier, or the second pan can be heavier.

Consequently, the decision tree for the sequence of weighings is a 3-ary tree.

There are at least 8 leaves in the decision tree because there are 8 possible outcomes (because each of the 8 coins can be the counterfeit lighter coin), and each possible outcome must be represented by at least one leaf.

The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree.

From Corollary 1 follows that the height of the decision tree is at least $\log_3 8 = 2$.

Hence, at least 2 weighings are needed.

It is possible to determine the counterfeit coin using 2 weighings.

The complexity of comparison–based sorting algorithms

Many different sorting algorithms have been developed.

To decide whether a particular sorting algorithm is efficient, its complexity is determined.

Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons can be found.

We can use decision trees to model sorting algorithms and to determine an estimate for the worst-case complexity of these algorithms.

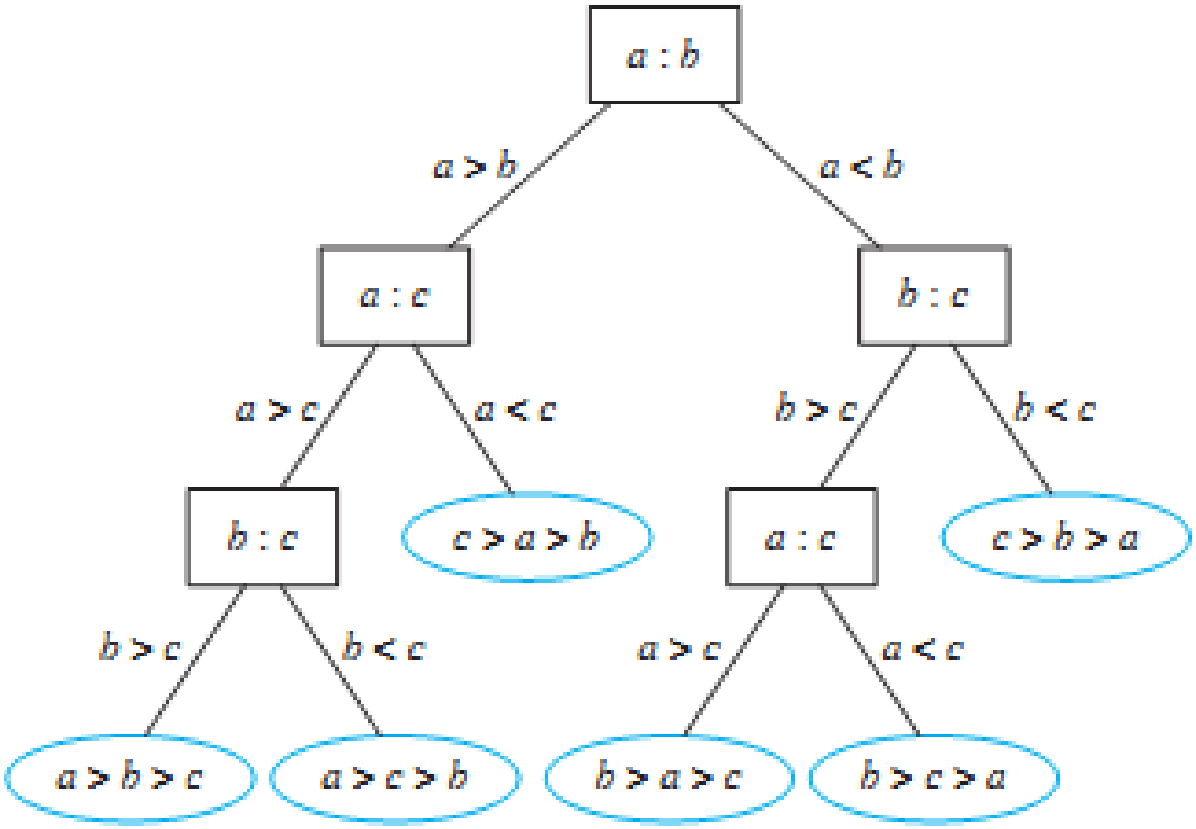
Note that given n elements, there are $n!$ possible orderings of these elements, because each of the $n!$ permutations of these elements can be the correct order.

Most commonly used sorting algorithms are based on binary comparisons, that is, the comparison of 2 elements at a time.

Thus, a sorting algorithm based on binary comparisons can be represented by a binary decision tree in which each internal vertex represents a comparison of 2 elements.

Each leaf represents one of the $n!$ permutations of n elements.

A Decision Tree for Sorting 3 Distinct Elements.



The complexity of a sort based on binary comparisons is measured in terms of the number of such comparisons used.

The largest number of binary comparisons ever needed to sort a list with n elements gives the worst-case performance of the algorithm.

The most comparisons used equals the longest path length in the decision tree representing the sorting procedure.

In other words, the largest number of comparisons ever needed is equal to the height of the decision tree.

Because the height of a binary tree with $n!$ leaves is $\geq \lceil \log n! \rceil$ (using Corollary 1), $\geq \lceil \log n! \rceil$ comparisons are needed, as stated in Theorem 1.

THEOREM 1 A sorting algorithm based on binary comparisons requires $\geq \lceil \log n! \rceil$ comparisons.

We can use Theorem 1 to provide a big-Omega (Ω) estimate for the number of comparisons used by a sorting algorithm based on binary comparison.

We need only note that $\lceil \log n! \rceil$ is $\Theta(n \log n)$,

one of the commonly used reference functions

for the computational complexity of algorithms.

Corollary 1 is a consequence of this estimate.

$O(\log_2(n!) = O(n\log n)$

(a)$O\log(n!) \in O(n\log n)$

$\log_2(n!) = \log_2(1) + \log_2(2) + \log_2(3) + \ldots \log_2(n-1) + \log_2(n)$

$\leq \log_2(n) + \log_2(n) + \log_2(n) + \ldots + \log_2(n) = n\log_2 n$

Let $c = 1, N_0 = 1 \quad \forall n \geq N_0 \; log_2(n!) \leq c \cdot n log_2(n)$

b) $\log_2(n!) = \log_2(n) + \log_2(n\text{-}1) + \ldots + \log_2(n/2) +$
$\log_2(n/2 - 1) + \ldots + \log_2(4) + \log_2(3) + \log_2(2) +$
$\log_2(1) > (n/2)\log_2(n/2) + (n/2)\log_2(2) =$

$= (n/2)\log_2(n) - (n/2)\log_2(2) + (n/2)\log_2(2) =$

$= (n/2)\log_2(n)$

$= (1/2)(n\log_2 n);$

$2\log_2(n!) > n\log_2(n).$

Let $c = 2$, $N_o = 1$ $\forall n \geq N_0$ $n\log_2(n) \leq c\log_2(n!)$

$N\log_2(n) \in O(\log_2(n!))$

COROLLARY 2 The number of comparisons used by a sorting algorithm to sort n elements based on binary comparisons is $\Omega(n \log n)$.

A consequence of Corollary 2 is that a sorting algorithm based on binary comparisons that uses $\Theta(n \log n)$ comparisons, in the worst case, to sort n elements is optimal, in the sense that no other such algorithm has better worst-case complexity.

Prefix Codes

Consider the problem of using bit strings to encode the letters of the English alphabet (where no distinction is made between lowercase and uppercase letters).

We can represent each letter with a bit string of length 5, because there are only 26 letters and there are 32 bit strings of length 5.

The total number of bits used to encode data is 5 times the number of characters in the text when each character is encoded with 5 bits.

Is it possible to find a coding scheme of these letters such that, when data are coded, fewer bits are used?

We can save memory and reduce transmittal time if this can be done.

Consider using bit strings of different lengths to encode letters.

Letters that occur more frequently should be encoded using short bit strings, and longer bit strings should be used to encode rarely occurring letters.

When letters are encoded using varying numbers of bits, some method must be used to determine where the bits for each character start and end.

For instance, if *e* were encoded with 0, *a* with 1, and t with *01*, then the bit string 0101 could correspond to eat, tea, eaea, or tt.

One way to ensure that no bit string corresponds to > 1 sequence of letters is to encode letters so that the bit string for a letter never occurs as the first part of the bit string for another letter.

Codes with this property are called **prefix** codes.

For instance, the encoding of *e* as 0, *a* as 10, and *t* as 11 is a prefix code.

A word can be recovered from the unique bit string that encodes its letters.

For example, the string 10110 is the encoding of ate.

To see this, note that the initial 1 does not represent a character, but 10 does represent *a* (and could not be the first part of the bit string of another letter).

Then, the next 1 does not represent a character, but 11 does represent *t*.

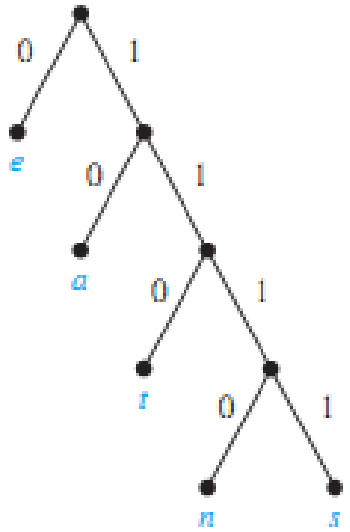
The final bit, 0, represents *e*.

A prefix code can be represented using a binary tree, where the characters are the labels of the leaves in the tree.

The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1.

The bit string used to encode character is the sequence of labels of the edges in the unique path from the root to the leaf that has this character as its label.

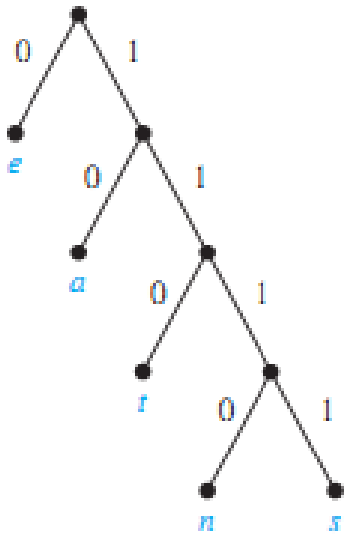
For instance, the tree bellow represents the encoding of e by 0, *a* by 10, *t* by 110, *n* by 1110, and *s* by 1111.



The tree representing a code can be used to decode a bit string.

For instance, consider the word encoded by 111110111100 using the code bellow.

This bit string can be decoded by starting at the root, using the sequence of bits to form a path that stops when a leaf is reached.



HUFFMAN CODING

We now introduce an algorithm that takes as input the frequencies (which are the probabilities of occurrences) of symbols in a string and produces as output a prefix code that encodes the string using the fewest possible bits, among all possible binary prefix codes for these symbols.

This algorithm, known as Huffman coding, was developed by David Huffman in a term paper he wrote in 1951 while a graduate student at MIT.

(Note that this algorithm assumes that we already know how many times each symbol occurs in the string, so we can compute the frequency of each symbol by dividing the number of times this symbol occurs by the length of the string.)

Huffman coding is a fundamental algorithm in data compression, the subject devoted to reducing the number of bits required to represent information.

Huffman coding is extensively used to compress bit strings representing text and it also plays an important role in compressing audio and image files.

Algorithm 2 presents the Huffman coding algorithm.

Given symbols and their frequencies, the goal is to construct a rooted binary tree where the symbols are the labels of the leaves.

The algorithm begins with a forest of trees each consisting of 1 vertex, where each vertex has a symbol as its label and where the weight of this vertex equals the frequency of the symbol that is its label.

At each step, we combine 2 trees having the least total weight into a single tree by introducing a new root and placing the tree with larger weight as its left subtree and the tree with smaller weight as its right subtree

Furthermore, we assign the sum of the weights of the two subtrees of this tree as the total weight of the tree.

The algorithm is finished when it has constructed a tree, that is, when the forest is reduced to a single tree.

ALGORITHM 2 Huffman Coding.

procedure Huffman(C: symbols a_i with frequencies w_i , $i = 1, ..., n$)

F := forest of n rooted trees, each consisting of the single vertex a_i and assigned weight w_i

while F is not a tree

Replace the rooted trees T and T' of least weights from F with $w(T) \geq w(T')$ with a tree having a new root that has T as its left subtree and T' as its right subtree.

Label the new edge to T with 0 and the new edge to T' with 1.

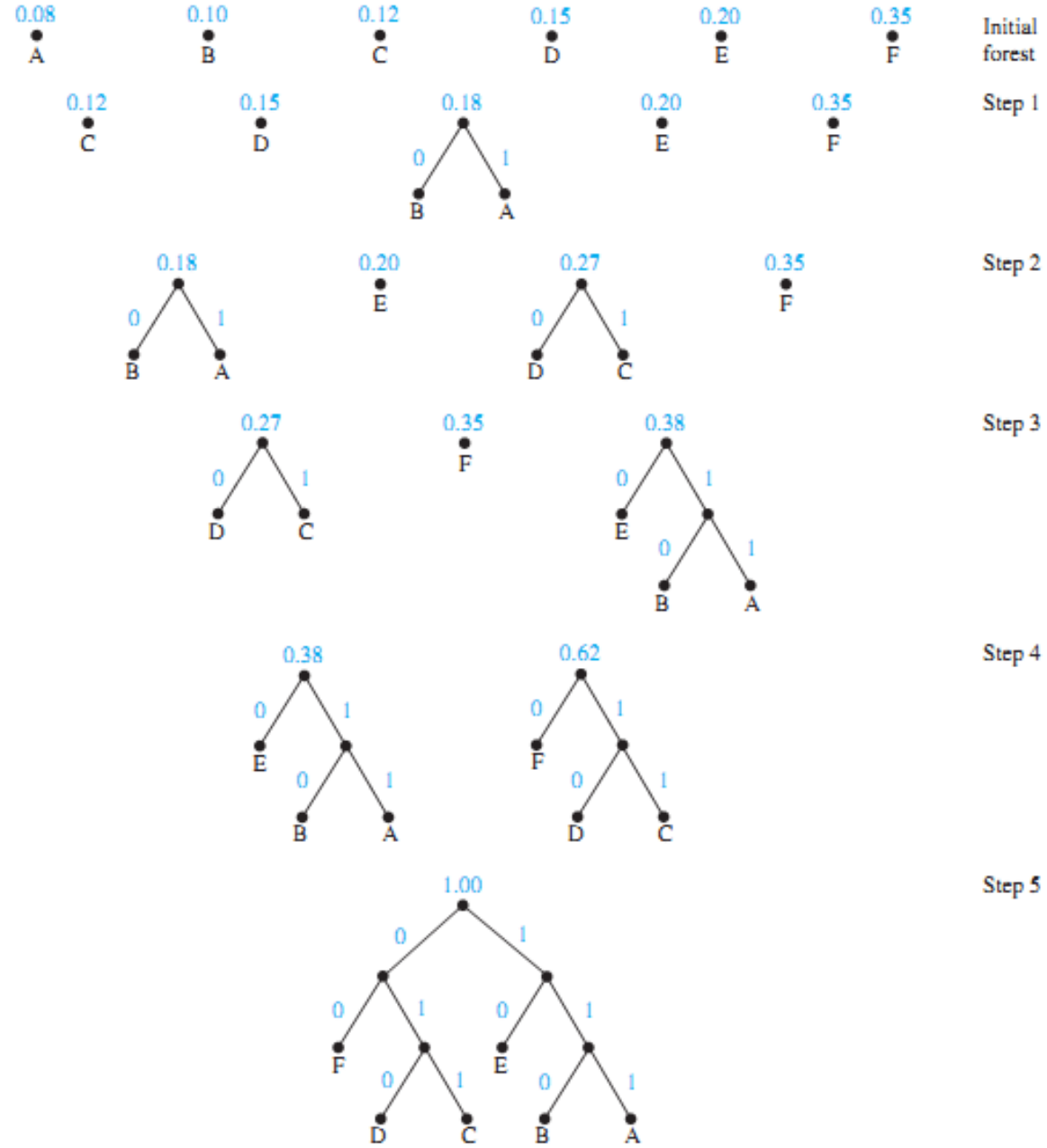
Assign $w(T) + w(T')$ as the weight of the new tree.

{the Huffman coding for the symbol a_i is the concatenation of the labels of the edges in the unique path from the root to the vertex a_i }

EXAMPLE Use Huffman coding to encode the following symbols with the frequencies listed:

A: 0.08, *B*:0.10, *C*: 0.12, *D*: 0.15, *E*: 0.20, *F*: 0.35.

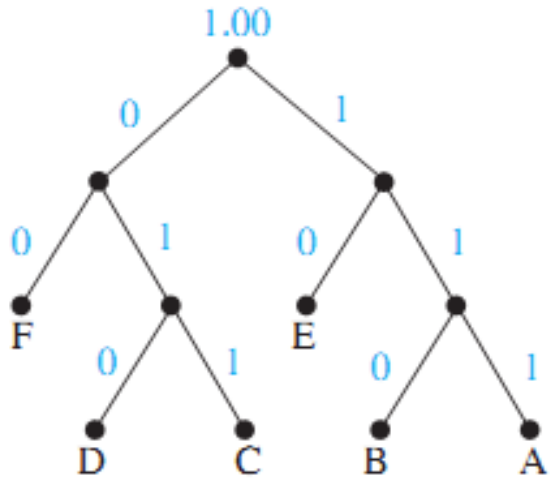
What is the average number of bits used to encode a character?


0.08
A
0.10
B
0.12
C
0.15
D
0.20
E
0.35
F
Initial forest
0.12
C
0.15
D
0.18
0
1
B
A
0.20
E
0.35
F
Step 1
0.18
0
1
B
A
0.20
E
0.27
0
1
D
C
0.35
F
Step 2
0.27
0
1
D
C
0.35
F
0.38
0
1
E
0
1
B
A
Step 3
0.38
0
1
E
0
1
B
A
0.62
0
1
F
0
1
D
C
Step 4
1.00
0
1
0
1
F
0
1
E
0
1
D
C
B
A
Step 5

Solution: The encoding produced encodes *A* by 111, *B* by 110, *C* by 011, *D* by 010, *E* by 10, and *F* by 00.

The average number of bits used to encode a symbol using this encoding is

$3 \cdot 0.08 + 3 \cdot 0.10 + 3 \cdot 0.12 + 3 \cdot 0.15 + 2 \cdot 0.20 + 2 \cdot 0.35 = 2.45$.



Note that Huffman coding is a greedy algorithm.

Replacing the 2 subtrees with the smallest weight at each step leads to an optimal code in the sense that no binary prefix code for these symbols can encode these symbols using fewer bits.

There are many variations of Huffman coding.

We can use more than 2 symbols to encode the original symbols in the string.

Also, instead of encoding single symbols, we can encode blocks of symbols of a specified length, such as blocks of 2 symbols.

Doing so may reduce the number of bits required to encode the string.