Huffman Coding Trees

We usually encode strings by assigning **fixed-length codes** to all characters in the alphabet (for example, 8-bit coding in ASCII).

However, if different characters occur with different frequencies, we can save memory and reduce transmittal time by using **variable-length encoding**.

The idea is to assign shorter codes to characters that occur more often.

Huffman Coding Trees

We must be careful when assigning variable-length codes.

For example, let us encode **e** with 0, **a** with 1, and **t** with 01. How can we then encode the word **tea**?

The encoding is **0101**.

Unfortunately, this encoding is ambiguous. It could also stand for **eat**, **eaea**, or **tt**.

Of course this coding is unacceptable, because it results in loss of information.

Huffman Coding Trees

To avoid such ambiguities, we can use **prefix codes**. In a prefix code, the bit string for a character never occurs as the **prefix** (first part) of the bit string for another character.

For example, the encoding of **e** with 0, **a** with 10, and **t** with 11 is a prefix code. How can we now encode the word **tea**?

The encoding is **11010**.

This bit string is unique, it can only encode the word tea.

Huffman Coding Trees

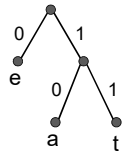
We can represent prefix codes using binary trees, where the characters are the labels of the leaves in the tree.

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 a character is the sequence of labels of the edges in the unique path from the root to the leaf labeled with this character.

Huffman Coding Trees

The tree corresponding to our example:



In a tree, no leaf can be the ancestor of another leaf. Therefore, no encoding of a character can be a prefix of an encoding of another character (prefix code).

Huffman Coding Trees

To determine the optimal (shortest) encoding for a given string, we first have to find the frequencies of characters in that string.

Let us consider the following string:

eeadfeejjeggebeeggddehhhecedddeciedee

It contains 1×a, 1×b, 3×c, 6×d, 15×e, 1×f, 4×g, 3×h, 1×i, and 2×j.

We can now use **Huffman's** algorithm to build the optimal coding tree.

Huffman Coding Trees

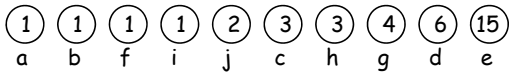
For an alphabet containing n letters, Huffman's algorithm **starts with n vertices**, one for each letter, labeled with that letter and its frequency.

We then determine the **two vertices** with the lowest frequencies and replace them with a **tree** whose root is labeled with the **sum** of these two frequencies and whose two children are the two vertices that we replaced.

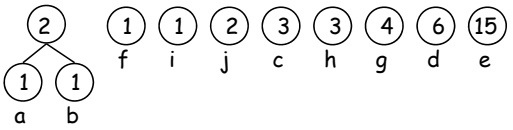
In the following steps, we determine the two lowest frequencies among the **single vertices and the roots of trees** that we already created.

This is repeated until we obtain a **single tree**.

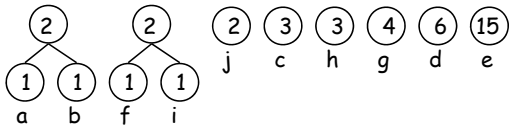
Huffman Coding Trees



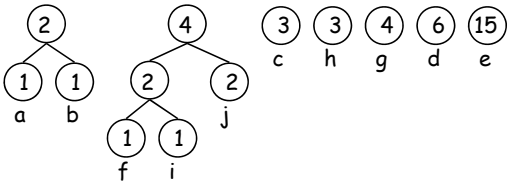
Huffman Coding Trees



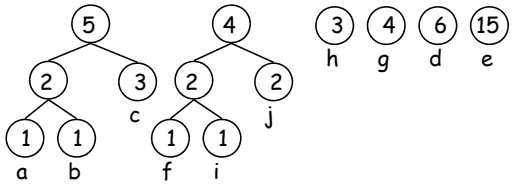
Huffman Coding Trees



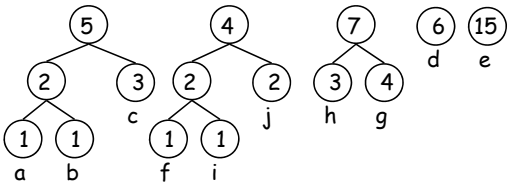
Huffman Coding Trees



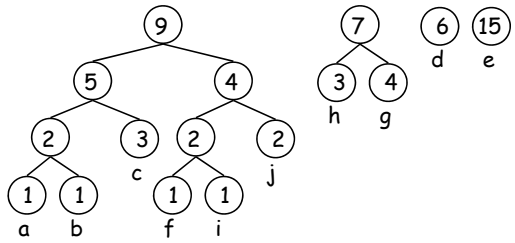
Huffman Coding Trees



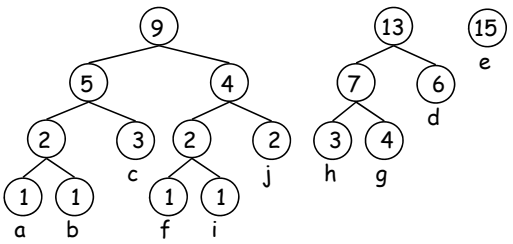
Huffman Coding Trees



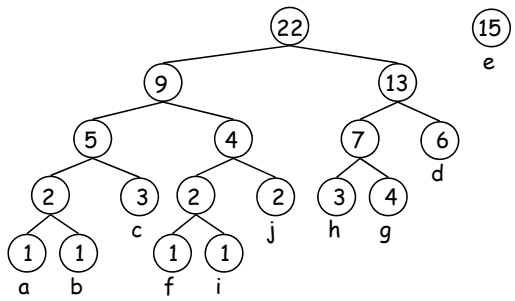
Huffman Coding Trees



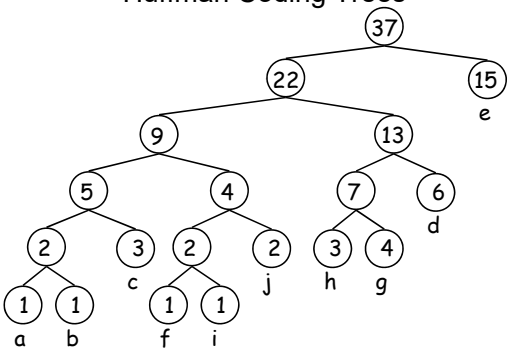
Huffman Coding Trees



Huffman Coding Trees

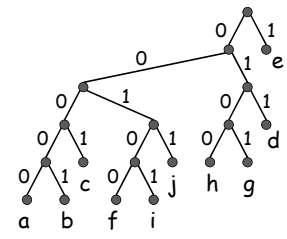


Huffman Coding Trees



Huffman Coding Trees

Finally, we convert the tree into a **prefix code tree**:



The **variable-length codes** are:

a (freq. 1): 00000
b (freq. 1): 00001
c (freq. 3): 0001
d (freq. 6): 011
e (freq. 15): 1
f (freq. 1): 00100
g (freq. 4): 0101
h (freq. 3): 0100
i (freq. 1): 00101
j (freq. 2): 0011

Huffman Coding Trees

If we encode the original string

eeadfeejjeggebeeggddehhhecedddeciedee

using a fixed-length code, we need four bits per character (for ten different characters). Therefore, the encoding of the entire string is 4·37 = 148 bits long.

With our variable-length code, we only need 1·5 + 1·5 + 3·4 + 6·3 + 15·1 + 1·5 + 4·4 + 3·4 + 1·5 + 2·4 = 101 bits.

Huffman Coding Trees

It can be shown that, for any given string, Huffman coding trees always produce a variable-length code with **minimum description length** for that string.

Since Huffman's algorithm is not covered in the textbook, please take a look at:

http://www.cs.duke.edu/csed/poop/huff/info/

Backtracking in Decision Trees

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

Decision trees can be used to model problems in which a **series of decisions** leads to a solution (compare with the "counterfeit coin" and "binary search tree" examples).

The possible **solutions** of the problem correspond to the paths from the root to the leaves of the decision tree.

Backtracking in Decision Trees

There are problems that require us to perform an **exhaustive search** of all possible sequences of decisions in order to find the solution.

We can solve such problems by constructing the **complete decision tree** and then find a path from its root to a leaf that corresponds to a solution of the problem.

In many cases, the efficiency of this procedure can be dramatically increased by a technique called **backtracking**.

Backtracking in Decision Trees

Idea: Start at the root of the decision tree and move downwards, that is, **make a sequence of decisions**, until you either reach a solution or you enter a situation from where no solution can be reached by any further sequence of decisions.

In the latter case, **backtrack to the parent** of the current vertex and take a different path downwards from there. If all paths from this vertex have already been explored, backtrack to its parent.

Continue this procedure until you **find a solution** or establish that **no solution exists** (there are no more paths to try out).

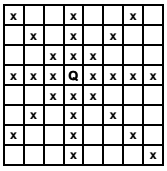
Backtracking in Decision Trees

Example: The n-queens problem

How can we place n queens on an n×n chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.



Decision Trees

Let us consider the **4-queens** problem.

Question: How many possible configurations of 4×4 chessboards containing 4 queens are there?

Answer: There are 16!/(12!·4!) = (13·14·15·16)/(2·3·4) = 13·7·5·4 = 1820 possible configurations.

Shall we simply try them out one by one until we encounter a solution?

No, it is generally useful to think about a search problem more carefully and discover **constraints** on the problem's solutions.

Such constraints can dramatically speed up the search process.

Backtracking in Decision Trees

Obviously, in any solution of the 4-queens problem, there must be **exactly one queen in each column** of the board.

Therefore, we can describe the solution of this problem as a **sequence of n decisions**:

Decision 1: Place a queen in the first column.

…

Decision 4: Place a queen in the fourth column.

For the **4-queens problem**, there are $4^4 = 256$ different sequences.

While this is already an efficiency gain, we will now see how backtracking can further improve it.

Backtracking in Decision Trees

empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen