MCS-375: Algorithms: Analysis and Design Handout #G3
San Skulrattanakulchai
Gustavus Adolphus College Oct 27, 2014

Huffman Codes

CLRS: Ch 16.3

Ziv-Lempel is the most popular compression algorithm today. It is an adaptive algorithm, i.e., the code changes throughout the message. There are several variants of Ziv-Lempel, many of which are proprietary. Example apps using this compression method are Unix **compress**, **gzip**, and Windows' **winzip**. The encoded string has length proportional to the *entropy*, i.e., the information content of the source string.

Huffman's was an early compression algorithm. It is a non-adaptive code. The algorithm is implemented as Unix System V's **pack**, and **compact** (adaptive Huffman). It is also used within **bzip**, **JPEG**, and **MPEG**.

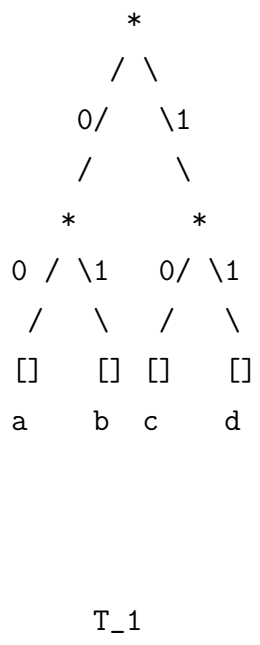
Prefix-free codes

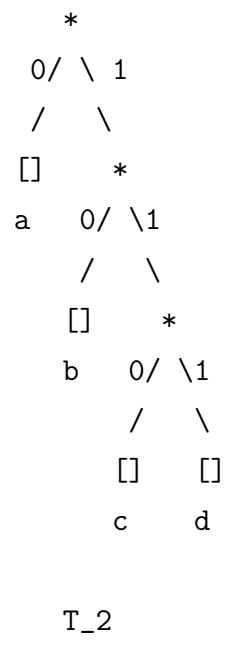
An alphabet of n characters can be encoded in strings that are $\lceil \log n \rceil$ bits long, e.g., for alphabet $\{ a, b, c, d \}$ we can use strings 00, 01, 10, 11 as code words for a, b, c, and d, respectively. Examples of such fixed-length codes in general use is ASCII, ISO 8859-1 (Latin-1), and Unicode.

To encode a given text, we can usually achieve shorter total length by giving shorter code words to frequently occurring characters. For example, if a occurs 51 times, and b, c and d each occurs 1 time, we can use code words 0 for a, 10 for b, 110 for c, and 111 for d.

The code should be *prefix-free*, i.e., no code word is a prefix of another. This condition is necessary and sufficient for an *instantaneous* code, i.e., one whose encoded message may be decoded on the fly while scanning through it.

A prefix-free code corresponds to a binary tree where each leaf represents a character of the alphabet. The code word for each character is determined by the character's access path, with left child corresponding to 0, and right child corresponding to 1.





For a prefix-free code, any message has ≤ 1 decoding.

Problem 1. Given a text, find the prefix-free code giving the shortest encoding.

Example. In *abacad*, a occurs 3 times, each of b, c, d occurs once. T_1 gives encoding length $(3 \times 2) + 2 + 2 + 2 = 12$, T_2 gives encoding length $(3 \times 1) + 2 + 3 + 3 = 11$.

Path length in trees. Consider a tree T with leaves L. Let $d(x)$ denote the depth of node x. Define the *external path length* of T to be the sum of the depths of all leaves

$$\sum \{\, d(\ell) : \ell \in L \,\}.$$

Let each leaf ℓ have weight $w(\ell)$. Define the *weighted external path length* (w.e.p.l) $P(T)$ of T to be the sum of the weighted depths of all leaves

$$P(T) = \sum \{\, w(\ell)d(\ell) : \ell \in L \,\}.$$

Example: Consider trees T_1 and T_2 above. Setting $w(a) = 3$ and $w(b) = w(c) = w(d) = 1$, we find $P(T_1) = 12$ and $P(T_2) = 11$. Note that the w.e.p.l then equals encoding length.

Problem 2. Given a set of items I and nonnegative weight function w on the items, find a binary tree T with leaves I and having minimum w.e.p.l.

Problem 1 reduces to Problem 2. To solve Problem 1, we scan the text, counting up each character's frequency. We then solve Problem 2 by defining the weight of a character to be its frequency. As illustrated above, w.e.p.l. equals encoding length.

Optimum Merge Trees

We wish to merge 2 sorted files into 1 sorted file.

Example 1: Merging 2, 4, 8, 16 and 1, 3, 9, 10, 11 gives 1, 2, 3, 4, 8, 9, 10, 11, 16.

A merging algorithm works by comparing the first numbers of both files, outputting the smaller number, and recursively merging the remaining files.

The time to merge a file of size m with a file of size n is $O(m+n)$.

The General Merging Problem

We are given n sorted files of varying lengths. We wish to merge them into 1 sorted file by repeatedly merging 2 files until 1 file remains.

A sequence of merges corresponds to a binary tree.

Example 2: Here are 2 ways to merge 3 files of length 10, 20, 30:

```
1st merge takes time 30
2nd merge takes time 60
total merge time 90
```

```
1st merge takes time 50
2nd merge takes time 60
total merge time 110
```

Definition. An *optimum merge pattern* (optimum merge tree) minimizes the total merge time, i.e., it minimizes the length of all files output by a merge.

Problem 3. Given the lengths of n sorted files, find an optimum merge pattern.

Lemma. Optimum merge trees minimize w.e.p.l.

Proof Sketch. Consider a binary tree T with leaves L and weight $w(\ell)$, where $\ell \in L$. Interpreting T as a merge tree, define the *weight* of an interior node to be the sum of the weights of its children. (The weight of an interior node is the time for the merge at that node.)

claim: w.e.p.l. equals the total interior node weight (i.e., total merge time) since a leaf ℓ contributes $d(\ell)w(\ell)$ to both quantities.

The lemma follows from the claim. □

Problem 4. Given a set of items I and nonnegative weight function w on the items, find a binary tree T with leaves I and having minimum total interior node weight.

By the definition of interior node weight, we see that Problem 3 reduces to Problem 4. We previously showed Problem 1 reduces to Problem 2. The above lemma shows Problem 2 is equivalent to Problem 4. So when designing or proving the correctness of a solution, we may assume the solution is to Problem 2 or Problem 4.

Greedy algorithm for merge patterns

Problem 3 has an obvious greedy algorithm: repeatedly merge the shortest files. We will show this greedy algorithm is correct.

Correctness

Theorem. The optimum merge pattern algorithm (i.e., Huffman's algorithm) is correct.

Lemma 1. Some optimal tree has the 2 smallest items in the 2 sibling leaves of maximum depth.

Proof Sketch. Consider any binary tree whose leaves are the items. Suppose the 2 smallest items aren't siblings of maximum depth. Swap them into that position. The swap doesn't increase the w.e.p.l. So if we start with an optimum tree, we'll get another optimum tree that satisfies the Lemma. □

To complete the proof of correctness, we must show that after Huffman's algorithm combines the 2 smallest items, it can proceed recursively.

Proof. 1. The first merge of it is valid, by Lemma 1.

2. After the first merge, the problem is to find another optimum merge pattern so the same strategy works!

So Huffman's algorithm is correct. □

Huffman's Algorithm

let a and b be 2 items of smallest weight
replace them by an item p of weight $w(a) + w(b)$
recursively find the minimum tree R for the new items
return T the tree R with a and b as children of p

Example

Let's be given six items with weights 2, 3, 3, 4, 6, 12.

[MISSING FIGURE HERE]

Implementation and Timing

1. For efficiency we implement Huffman's algorithm iteratively rather than recursively.

2. Instead of maintaining the items and the combined items as a sorted list as in the figure, we maintain them in a priority queue.

Pseudocode for Huffman's Algorithm

$Q \leftarrow$ empty priority queue
for $i \leftarrow 1$ **to** n **do** INSERT item i in Q
for $i \leftarrow 1$ **to** $n - 1$ **do** {
  $a \leftarrow$ EXTRACT_MIN(Q)
  $b \leftarrow$ EXTRACT_MIN(Q)
  combine a and b into a new item, and INSERT it in Q
}
/* now Q contains 1 item, representing the Huffman tree */

Total time $= O(n) \times O(\log n) = O(n \log n)$ since each iteration does $O(1)$ priority queue operations and each priority queue operation takes $O(\log n)$ time.