CITS2200 Data Structures and Algorithms

Topic 22

# Data Compression

- Huffman Coding
- Lempel Ziv algorithms

## Data Compression Algorithms

Data compression algorithms exploit patterns in data files to compress the files. Every compression algorithm should have a corresponding decompression algorithm that can recover (most of) the original data.

Data compression algorihtms are used by programs such as WinZip, pkzip and zip. They are also used in the definition of many data formats such as pdf, jpeg, mpeg and .doc.

Data compression algorithms can either be *lossless* (e.g. for archiving purposes) or *lossy* (e.g. for media files).

We will consider some lossless algorithms below.

## Huffman coding

A nice application of a greedy algorithm is found in an approach to data compression called Huffman coding.

Suppose that we have a large amount of text that we wish to store on a computer disk in an efficient way. The simplest way to do this is simply to assign a binary code to each character, and then store the binary codes consecutively in the computer memory.

The ASCII system for example, uses a fixed 8-bit code to represent each character. Storing $n$ characters as ASCII text requires $8n$ bits of memory.

## Simplification

Let $C$ be the set of characters we are working with. To simplify things, let us suppose that we are storing only the 10 numeric characters 0, 1, . . ., 9. That is, set $C = \{0, 1, \cdots, 9\}$.

| Char | Code |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

## Non-random data

Consider the following data, which is taken from a Postscript file.

| Char | Freq |
|---|---|
| 5 | 1294 |
| 9 | 1525 |
| 6 | 2260 |
| 4 | 2561 |
| 2 | 4442 |
| 3 | 5960 |
| 7 | 6878 |
| 8 | 8865 |
| 1 | 11610 |
| 0 | 70784 |

Notice that there are many more occurrences of 0 and 1 than the other characters.

## A good code

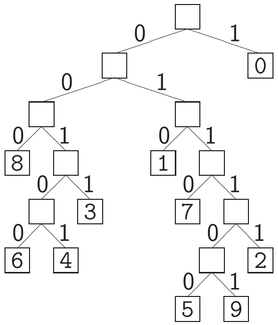
What would happen if we used the following code rather than the fixed length code?

| Char | Code |
|---|---|
| 0 | 1 |
| 1 | 010 |
| 2 | 01111 |
| 3 | 0011 |
| 4 | 00101 |
| 5 | 011100 |
| 6 | 00100 |
| 7 | 0110 |
| 8 | 000 |
| 9 | 011101 |

To store the string 0748901 we would get $0000011101001000100100000001$ using the fixed length code and $10110001010000111011010$ using the variable length code.

## Prefix codes

In order to be able to decode the variable length code properly it is necessary that it be a **prefix code** — that is, a code in which no codeword is a prefix of any other codeword.



## Cost of a tree

Now assign to each leaf of the tree a value, $f(c)$, which is the frequency of occurrence of the character $c$ represented by the leaf.

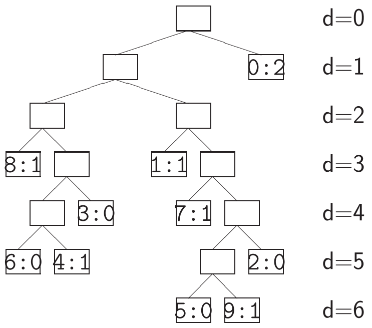
Let $d_T(c)$ be the depth of character $c$'s leaf in the tree $T$.

Then the number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the **cost** of the tree $T$.

For example, the number of bits required to store the string 0748901 can be computed from the tree $T$:



giving

$$B(T) = 2 \times 1 + 1 \times 3 + 1 \times 3 + 1 \times 4 + 1 \times 5 + 1 \times 6 = 23.$$

## Optimal trees

A tree representing an optimal code for a file is always a *full* binary tree — namely, one where every node is either a leaf or has precisely two children.

Therefore if we are dealing with an alphabet of $s$ symbols we can be sure that our tree has precisely $s$ leaves and $s - 1$ internal nodes, each with two children.

Huffman invented a greedy algorithm to construct such an optimal tree.

The resulting code is called a **Huffman code** for that file.

## Huffman's algorithm

The algorithm starts by creating a forest of $s$ single nodes, each representing one character, and each with an associated value, being the frequency of occurrence of that character. These values are placed into a priority queue (implemented as a linear array).

5:1294 9:1525 6:2260 4:2561 2:4442

3:5960 7:6878 8:8865 1:11610 0:70784

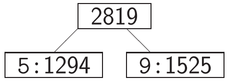
Then repeat the following procedure $s - 1$ times:

Remove from the priority queue the two nodes $L$ and $R$ with the lowest values, and create a internal node of the binary tree whose left child is $L$ and right child $R$.

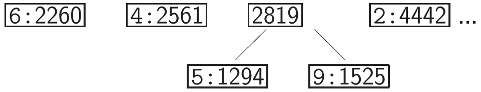
Compute the value of the new node as the sum of the values of $L$ and $R$ and insert this into the priority queue.

## The first few steps

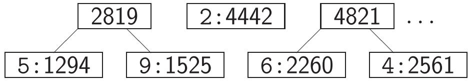
Given the data above, the first two entries off the priority queue are 5 and 9 so we create a new node



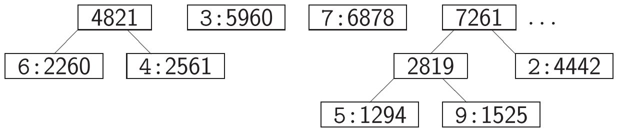
The priority queue is now one element shorter, as shown below:



The next two are 6 and 4 yielding



Now the smallest two nodes are 2 and the internal node with value 2819, hence we now get:



Notice how we are growing sections of the tree from the bottom-up (compare with the tree on slide 16).

See CLRS (page 388) for the pseudo-code corresponding to this algorithm.

## Why does it work?

In order to show that Huffman's algorithm works, we must show that there can be no prefix codes that are better than the one produced by Huffman's algorithm.

The proof is divided into two steps:

First it is necessary to demonstrate that the first step (merging the two lowest frequency characters) cannot cause the tree to be non-optimal. This is done by showing that any optimal tree can be reorganised so that these two characters have the same parent node. (see CLRS, Lemma 16.2, page 388)

Secondly we note that after making an optimal first choice, the problem can be reduced to finding a Huffman code for a smaller alphabet. (see CLRS, Lemma 16.3, page 391)

## Ziv-Lempel compression algorithms

The Ziv-Lempel compression algorithms are a family of compression algorithms that can be applied to arbitrary file types.

The Ziv-Lempel algorithms represent recurring strings with abbreviated codes. There are two main types:

- LZ77 variants use a buffer to look for recurring strings in a small section of the file.
- LZW variants dynamically create a dictionary of recurring strings, and assigns a simple code to each such string.

## Algorithms: LZ77

The **LZ77** algorithms use a *sliding window*. The sliding window is a buffer consisting of the last $m$ letters encoded ($a_0 ... a_{m-1}$) and the next $n$ letters to be encoded ($b_0 ... b_{n-1}$).

Initially we let $a_0 = a_1 = ... = a_{n-1} = w_0$ and output $\langle 0, 0, w \rangle$ where $w_0$ is the first letter of the word to be compressed

The algorithm looks for the longest prefix of $b_0 ... b_{n-1}$ appearing in $a_0 ... a_{m-1}$. If the longest prefix found is $b_0 ... b_{k-1} = a_i ... a_{i+k-1}$, then the entire prefix is encoded as the tuple

$$\langle i, k, b_k \rangle$$

where $i$ is the *offset*, $k$ is the *length* and $b_k$ is the *next character*.

## LZ77 Example

Suppose that $m = n = 4$ and we would like to compress the word $w = aababacbaa$

| Word | Window | Output |
|---|---|---|
| $aababacbaa$ | | $\langle 0, 0, a\rangle$ |
| $aababacbaa$ | $\underline{aa}aa\ \underline{aa}ba$ | $\langle 0, 2, b\rangle$ |
| $abacbaa$ | $aa\underline{ab\ a}\ \underline{aba}c$ | $\langle 2, 3, c\rangle$ |
| $baa$ | $a\underline{ba}c\ \underline{ba}a$ | $\langle 1, 2, a\rangle$ |

This outputs

$$\langle 0, 0, a\rangle\langle 0, 2, b\rangle\langle 2, 3, c\rangle\langle 1, 2, a\rangle$$

## LZ77 Example cont.

To decompress the code we can reconstruct the sliding window at each step of the algorithm. Eg, given

$$\langle 0, 0, a\rangle\langle 0, 2, b\rangle\langle 2, 3, c\rangle\langle 1, 2, a\rangle$$

| Input | Window | Output |
|---|---|---|
| $\langle 0, 0, a\rangle$ | | |
| $\langle 0, 2, b\rangle$ | $\underline{aa}aa\ aab?$ | $aab$ |
| $\langle 2, 3, c\rangle$ | $aa\underline{ab\ a}\ abac$ | $abac$ |
| $\langle 1, 2, a\rangle$ | $a\underline{ba}c\ baa?$ | $baa$ |

Note the trick with the third triple $\langle 2, 3, c\rangle$ that allows the look-back buffer to overflow into the look ahead buffer.

## Algorithms: LZW

The **LZW** algorithms use a *dynamic dictionary* The dictionary maps words to codes and is initially defined for every byte (0-255). The compression algorithm is as follows:

```
w = null
while(k = next byte)
    if wk in the dictionary
        w = wk
    else
        add wk to dictionary
        output code for w
        w = k
output code for w
```

## Algorithms: LZW

The decompression algorithm is as follows:

```
k = next byte
output k
w = k
while(k = next byte)
    if there's no dictionary entry for k
        entry = w + first letter of w
    else
        entry = dictionary entry for k
    output entry
    add w + first letter of entry to dictionary
    w = entry
```

## LZW Example

Consider the word $w = aababa$, and a dictionary $D$ where $D[0] = a$, $D[1] = b$ and $D[2] = c$. The compression algorithm proceeds as follows:

| Read | Do | Output |
|---|---|---|
| $a$ | $w = a$ | – |
| $a$ | $w = a, D[3] = aa$ | 0 |
| $b$ | $w = b, D[4] = ab$ | 0 |
| $a$ | $w = a, D[5] = ba$ | 1 |
| $b$ | $w = ab$ | – |
| $a$ | $w = a, D[6] = aba$ | 4 |
| $c$ | $w = c, D[7] = ac$ | 0 |
| $b$ | $w = b, D[8] = cb$ | 2 |
| $a$ | $w = ba$ | – |
| $a$ | $w = a, D[9] = baa$ | 5 |
| | | 0 |

## LZW Example cont.

To decompress the code $\langle 00140250 \rangle$ we initialize the dictionary as before. Then

| Read | Do | Output |
|---|---|---|
| 0 | $w = a$ | $a$ |
| 0 | $w = a, D[3] = aa$ | $a$ |
| 1 | $w = b, D[4] = ab$ | $b$ |
| 4 | $w = ab, D[5] = ba$ | $ab$ |
| 0 | $w = a, D[6] = aba$ | $a$ |
| 2 | $w = c, D[7] = ac$ | $c$ |
| 5 | $w = ba, D[8] = cb$ | $ba$ |
| 0 | $w = a, D[9] = baa$ | $a$ |

## Summary

1. Data Compression algorithms use pattern matching to find efficient ways to compress file.
2. Huffman coding uses a greedy approach to recode the alphabet with a more efficient binary code.
3. Adaptive Huffman coding uses the same approach, but with the overhead of precomputing the code.
4. LZ77 uses pattern matching to express segments of the file in terms of recently occuring segments.
5. LZW uses a hash function to store commonly occuring strings so it can refer to them by their key.