Data Compression

Data compression can be *lossless* or *lossy*.

Lossless compression retains full reversibility — on de-compression the original data is completely recovered. Lossy compression is not fully reversible; some information is lost from the original data.

Examples of lossless compression: Huffman coding (treated in detail in this lecture), gzip. Fact: no lossless compression algorithm can successfully obtain a positive compression on all data.

Examples of lossy compression: JPEG, MPEG.

Lossy Compression

This is actually a very old idea.

Can you read the following?

Lov is a many spl dor d thing

ngland xp cts v ry man to do his duty

F ll m ny fl w r s b rn t bl sh ns n

Vowels are highly redundant (in Hebrew writing they are omitted!). These examples show that in much of human communication, not all aspects of data are equally important, as either people can interpolate missing information or they are less sensitive to some dimensions. The latter is exploited in the lossy compression of, e.g., JPEG.

Lossless Compression

For many applications however, data degradation is intolerable: e.g., legal documents, accounting reports, manuals, tables, data bases. So any compression here must be lossless.

Lossless data representation compression exploits the *occurence frequency* of data items being represented. One idea is to index repetitions of the same string rather than repeat it, compressing text into a "dictionary" of (short) indices instead. The other idea is code very frequently occurring symbols with short codes. It is used to help the first idea too.

We concentrate on lossless compression here. The key lossless algorithm is *Huffman coding*, which plays a role even in most lossless compression but also in lossy ones.

Code Length

Using binary codes for C characters (printable, unprintable) needs $\lceil log\ C \rceil$ bits if all characters are encoded into a fixed length. For 7 bits (plus 1 more for parity check), we can encode 128 characters.

This encoding does not exploit the fact that in English text the letter E occurs most frequently, while letters like Q — which in fact is almost always followed by a U except in $QANTAS$ — are much less frequent.

In the previous sentence, E occurs 20 times, and Q four times. A non-fixed length code that assigns a short code to E and a longer one to Q will therefore represent text more concisely in binary.

Weights

In fact, if we can estimate the frequency of occurence of every letter in the alphabet in typical English text, then perhaps we can arrive at an *optimal* compression scheme for its binary encoding.

Studies have been conducted to get these frequencies. Notice that we also have to encode *blanks*, *end of line*, *new line*, etc., and these have relatively high frequencies. In the literature, frequencies are also called *weights* — and we will often do that too. High weight data is therefore "more important" as they occur frequently.

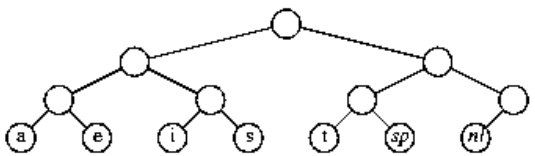
If a compression scheme based on this is applied to text files, the saving in space can be large.

A simple toy example

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
Total			174

A standard coding scheme

A Trie is a tree with data only in the leaves



Representation of the original code by a tree

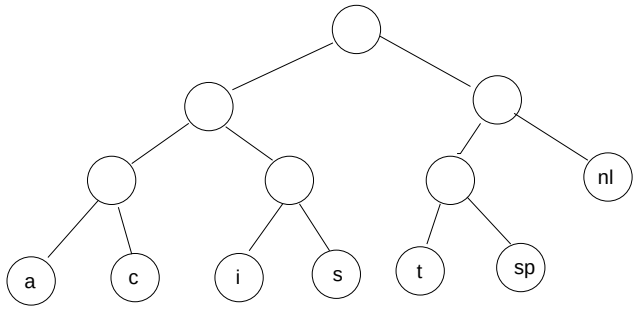
Prefix Codes

Notice that in the previous code table, no code is the prefix of another. Such codes are called *prefix codes*, and have the property that a binary sequence of coded symbols can be uniquely decoded into the symbols by processing the sequence from the left.

Moreover, the tree representing the prefix code must be a *trie*, as is seen in the preceding tree where all symbols are in the leaves. It is easy to show that any tree representing a prefix code is a trie, and vice-versa.

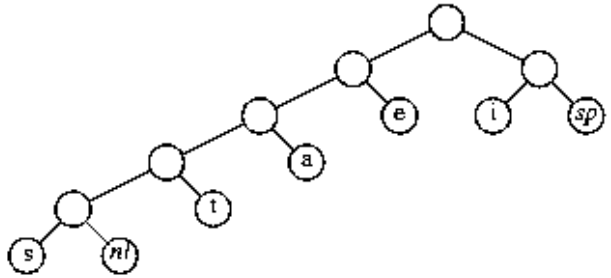
Not all codes used in computing or telecommunications are prefix codes.

Shorter path lengths = shorter codes



A slightly better tree

Optimal Trie



Optimal prefix code tree

Codes corresponding to the optimal tree (trie)

Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
Total			**146**

Optimal prefix code

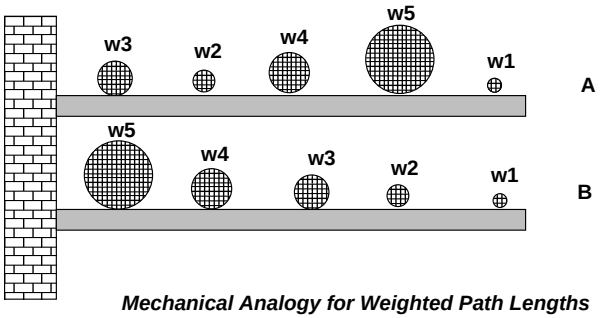
Huffman's Algorithm

How is the optimal code tree obtained? Huffman's algorithm is the standard way to construct it. The input to the algorithm is a set of symbols or characters to be binary coded, and their weights. The formal problem is the following:

Given a set $\{< s_1, w_1 >, < s_2, w_2 >, \ldots, < s_n, w_n >\}$ of symbol-weight pairs, find a trie with the s_i in the leaves that *minimizes* $\sum_{i=1}^{n} w_i l_i$ where l_i is the path length from root to the leaf with symbol s_i.

This is informally described as *minimizing the (sum of) weighted path lengths.*

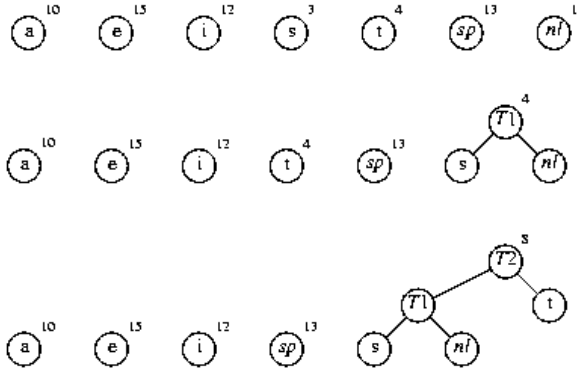
Analogy

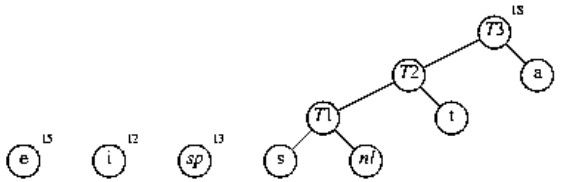


Mechanical Analogy for Weighted Path Lengths
= Turning Moment of the Weights

B < A (so, better for big weights to be nearer pivot)

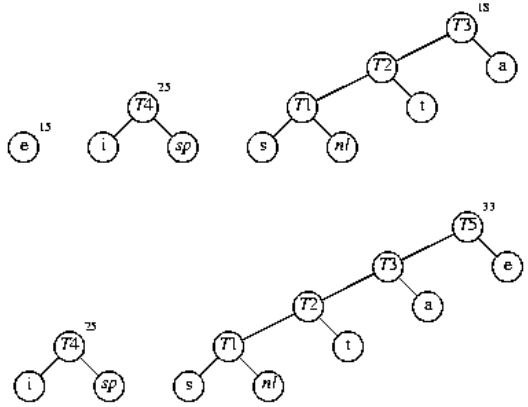
Construction of Huffman Tree



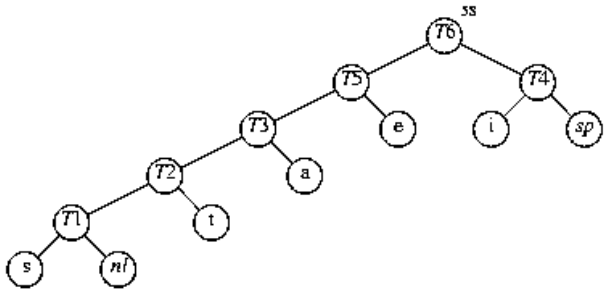


Huffman's algorithm after each of first three merges

More merging



Done



Huffman's algorithm after each of last three merges

You can picture the above tree as pivotted by its root on a wall, looking at the whole assembly from above. Then the optimality is equivalent to the placement of the weights so as to minimize the turning moment of the tree structure about its pivot.

Proof of Correctness

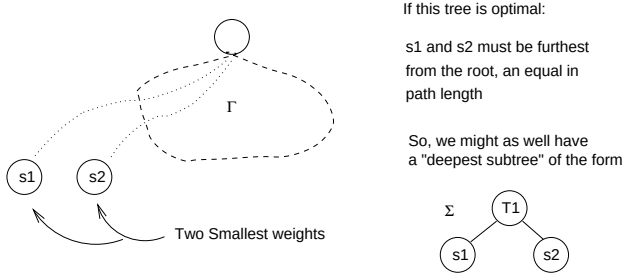
The proof of correctness of the Huffman algorithm, i.e. that it produces a minimal weighted path length tree (and therefore *optimal*), is via these observations:

1. The two smallest items (weight-wise) must be in a sub-tree furthest from the root. (If more than 2 are smallest, pick any two.)
2. Form this subtree, say its root is named T, with the two symbols s_i and s_j as its children.
3. Let Θ be a tree (whatever shape it is) w.r.t. the original $< s1, w_1 >, \ldots, < s_n, w_n >$ specifications, and suppose it has a deepest sub-tree consisting of $Theta$ and its two nodes s_i and s_j.

Let Θ' be the tree in which that subtree in Θ is replaced by just T (with weight $w_i + w_j$).

4. Θ is minimal iff Θ' is minimal.
5. The preceding observation is essentially the loop invariant in the iterative loop in the Huffman algorithm, which terminates when all symbols are used up.

Picture of the proof idea



Key idea of loop invariant – subtree Σ will be in some optimal tree, whatever the tree Γ turns out to be.

Efficiency

If the weights are sorted, then the Huffman construction is $O(N)$ for N weights. Can you see why?

On decoding a Huffman-compressed binary string of length N, because the code has the prefix property the de-compression is also $O(N)$. Can you outline the procedure to do it?

Thus, it is as good as it gets, since the information is of essentially size N and we require no information loss.

What is meant by the *optimality* of this code? (COMP2711 — can you provide a frequentist intepretation of it?)

JPEG and MPEG uses Huffman coding at the end

The lossy JPEG compression algorithm for static images goes thru 4 steps.

- Extract 8x8 pixel block from the image, converted into a luminance/chrominance color space, exploiting relative eye insensitivity to chrominance to discard information about it;
- Apply discrete cosine transform (DCT) to the elements of the block to convert them into "spatial frequency" curves defined by DCT coefficients; [For Comp Eng and EE majors, this is like moving from time (analogy: color space) to the frequency (analogy: color spectra) domains.]

JPEG + Huffman, cont'd

- Quantize these curves by discretizing the DCT coefficients — this is where loss of image information occurs; smaller coefficients are discarded, and larger coeffients are rounded; [For Comp Eng and EE majors, this is like discrete band-pass filtering.]
- Huffman encoding is then used to compress the quantized coefficients for storage. (To recover for display, apply the inverse DCT.) Alternative compression scheme is arithmetic encoding but it is slower.

In the lossless compression gzip, Huffman coding is also used in the second phase after commonly occurring strings are indexed by indirection.