Lecture 19

Compression and Huffman Coding

Supplemental reading in CLRS: Section 16.3

19.1 Compression

As you probably know at this point in your career, **compression** is a tool used to facilitate storing large data sets. There are two different sorts of goals one might hope to achieve with compression:

- *Maximize* ease of access, manipulation and processing
- *Minimize* size—especially important when storage or transmission is expensive.

Naturally, these two objectives are often at odds with each other. In this lecture we will focus on the second objective.

In general, data cannot be compressed. For example, we cannot losslessly represent all m-bit strings using $(m-1)$-bit strings, since there are 2^m possible m-bit strings and only 2^{m-1} possible $(m-1)$-bit strings. So when is compression possible?

- If only a relatively small number of the possible m-bit strings appear, compression is possible.

- If the same "long" substring appears repeatedly, we could represent it by a "short" string.

- If we relax the requirement that every string have a unique representation, then compression might work but make "similar" strings identical.

19.1.1 Lossless and lossy

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Initial}\\ \text{Document}\\ D\end{array}} \arrow[r, "\text{Compress}"] & \boxed{\begin{array}{c}\text{Compressed}\\ \text{Document}\\ C\end{array}} \arrow[r, "\text{Decompress}"] & \boxed{\begin{array}{c}\text{Reconstituted}\\ \text{Document}\\ D'\end{array}}
\end{tikzcd}$$

In **lossless** compression, we require that $D = D'$. This means that the original document can always be recovered exactly from the compressed document. Examples include:

- Huffman coding
- Lempel–Ziv (used in `gif` images)

In **lossy** compression, D' is close enough but not necessarily identical to D. Examples include:

- `mp3` (audio)
- `jpg` (images)
- `mpg` (videos)

19.1.2 Adaptive and non-adaptive

Compression algorithms can be either adaptive or non-adaptive.

- *Non-adaptive* – assumes prior knowledge of the data (e.g., character frequncies).
- *Adaptive* – assumes no knowledge of the data, but builds such knowledge.

19.1.3 Framework

For the remainder of this lecture, we consider the following problem:

Input: Known alphabet *(e.g., English letters a,b,c,...)*
Sequence of characters from the known alphabet *(e.g., "helloworld")*

We are looking for a **binary code**—a way to represent each character as a binary string (each such binary string is called a **codeword**).

Output: Concatenated string of codewords representing the given string of characters.

19.1.4 Fixed-length code

In a *fixed-length code*, all codewords have the same length. For example, if our alphabet is

$$\{\text{a, b, c, d, e, f, g, h}\},$$

then we can represent each of the 8 characters as a 3-bit string, such as

$$\left\{\begin{array}{cccccccc} \text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h} \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{array}\right\}.$$

Such a code is easy to encode and decode:

$$\text{baba} = 001|000|001|000$$

Just as in DNA encoding and decoding, it is important to keep track of register: the deletion or insertion of a single bit into the binary sequence will cause a frame shift, corrupting all later characters in the reconstituted document.

19.1.5 Run-length encoding

Imagine you are given the string of bits

$$\underbrace{000000}_{6}\underbrace{111}_{3}\underbrace{000}_{3}\underbrace{11}_{2}\underbrace{00000}_{5}.$$

Rather than create codewords, we could simply store this string of bits as the sequence $\langle 6,3,3,2,5\rangle$. This strategy is used in fax-machine transmission, and also in `jpeg`.

19.1.6 Variable-length code

If we allow our codewords to have different lengths, then there is an obvious strategy:

> Use shorter codewords for more frequent characters; use longer codewords for rarer characters.

For example, consider a six-letter alphabet with the following character frequencies:

Character	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	0	101	100	111	1101	1100
	(1 bit)		(3 bits)		(4 bits)	

Using this code, the average number of bits used to encode 100 characters is

$$(45)1+(13)3+(12)3+(16)3+(9)4+(5)4=224.$$

Compare this to a 3-bit fixed-length code, in which it would take 300 bits to encode 100 characters.

Notice that I didn't use 1, 01, or 10 as codewords, even though they would have made the encoding shorter. The reason for this is that we want our code to be **uniquely readable**: it should never be ambiguous as to how to decode a compressed document.[1] One standard way to make uniquely readable codes is to use **prefix coding**: no codeword occurs as a prefix (initial substring) of another codeword. This allows unambiguous, linear-time decoding:

$$\underbrace{101}_{\text{b}}\,\underbrace{111}_{\text{d}}\,\underbrace{1100}_{\text{f}}\,\underbrace{0}_{\text{a}}\,\underbrace{100}_{\text{c}}\,\underbrace{1101}_{\text{e}}$$

Prefix coding means that we can draw our code as a binary tree, with the leaves representing codewords (see Figure 19.1).

19.2 The Huffman Algorithm

The problem of §19.1.3 amounts to the following. We are given an alphabet $\{a_i\}$ with frequencies $\{f(a_i)\}$. We wish to find a set of binary codewords $C=\{c(a_1),\ldots,c(a_n)\}$ such that the average number of bits used to represent the data is minimized:

$$B(C)=\sum_{i=1}^{n} f(a_i)\left|c(a_i)\right|.$$

Equivalently, if we represent our code as a tree T with leaf nodes $a_1,\ldots,a_n$, then we want to minimize

$$B(T)=\sum_{i=1}^{n} f(a_i)\,d(a_i),$$

where $d(a_i)$ is the depth of a_i, which is also equal to the number of bits in the codeword for a_i.

The following algorithm, due to Huffman, creates an optimal prefix tree for a given set of characters $C=\{a_i\}$. Actually, the Huffman code is optimal among *all* uniquely readable codes, though we don't show it here.

[1] For an example of non-unique readibility, suppose we had assigned to "d" the codeword 01 rather than 111. Then the string 0101 could be decoded as either "ab" or "dd."

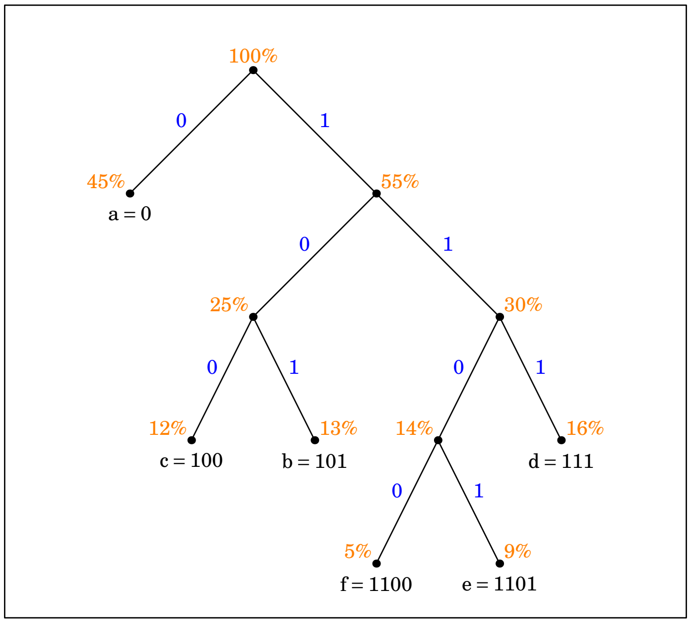

Figure 19.1. Representation of a binary code as a binary tree.

Algorithm: HUFFMAN-TREE(C)

```
 1  $n \leftarrow |C|$
 2  $Q \leftarrow C$ ▷ a min–priority queue keyed by frequency
 3  for $i \leftarrow 1$ to $n-1$ do
 4      Allocate new node $z$
 5      $z.left \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
 6      $z.right \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
 7      $z.freq \leftarrow x.freq + y.freq$
 8      $Q$.INSERT($z$)
 9  ▷ Return the root of the tree
10  return EXTRACT-MIN($Q$)
```

19.2.1 Running time

Initially, we build a minqueue Q with n elements. Next, the main loop runs $n-1$ times, with each iteration consisting of two EXTRACT-MIN operations and one INSERT operation. Finally, we call EXTRACT-MIN one last time; at this point Q has only one element left. Thus, the running time for Q a binary minheap would be

$$\underbrace{\Theta(n)}_{\text{BUILD-QUEUE}} + \underbrace{\Theta(n \lg n)}_{\text{Loop}} + \underbrace{O(1)}_{\text{EXTRACT-MIN}} = \Theta(n \lg n).$$

If instead we use a van Emde Boas structure for Q, we achieve the running time

$$\Theta(n \lg \lg n).$$

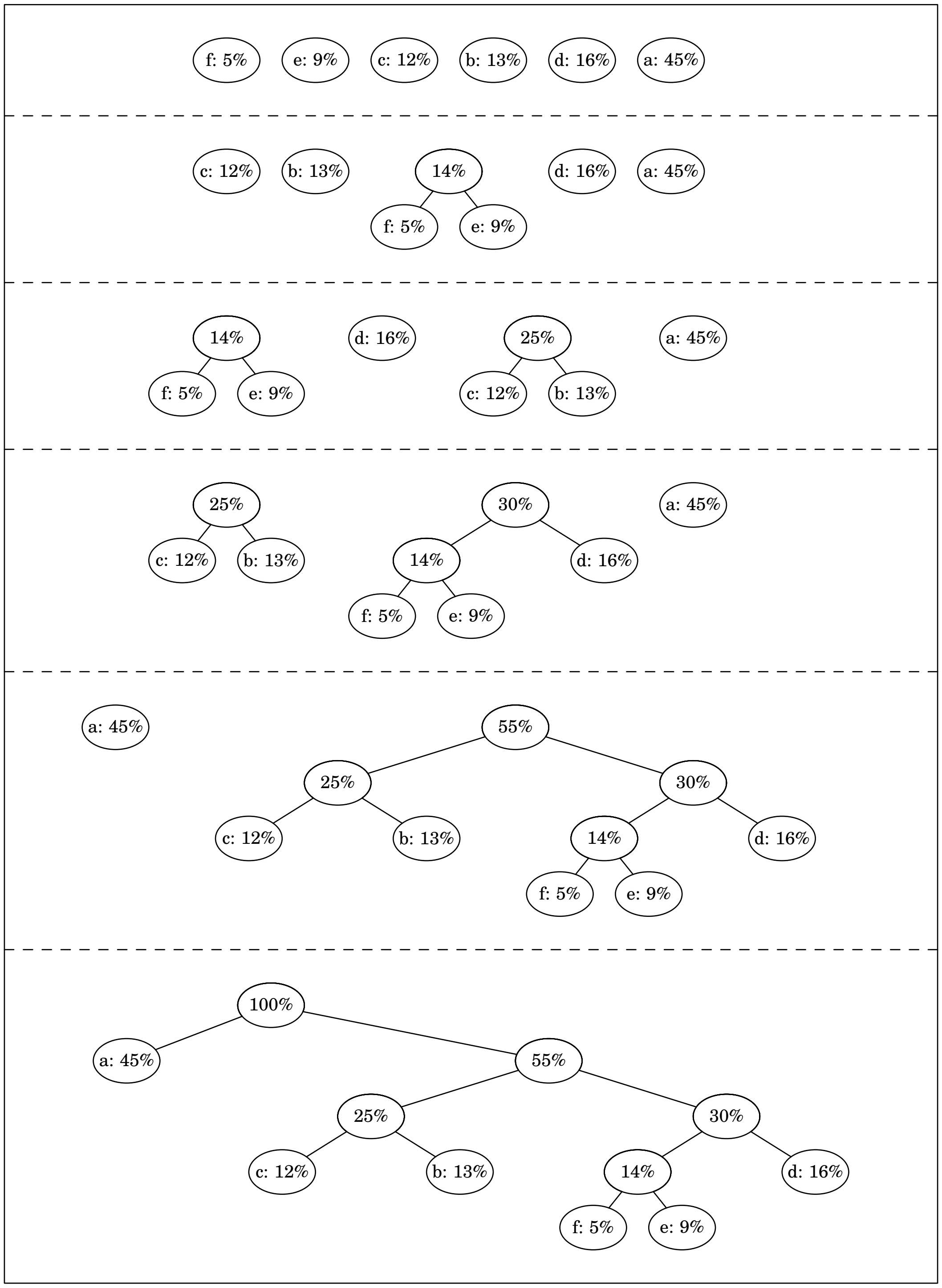

Figure 19.2. Example run of the Huffman algorithm. The six rows represent the state of the graph each of the six times line 3 is executed.

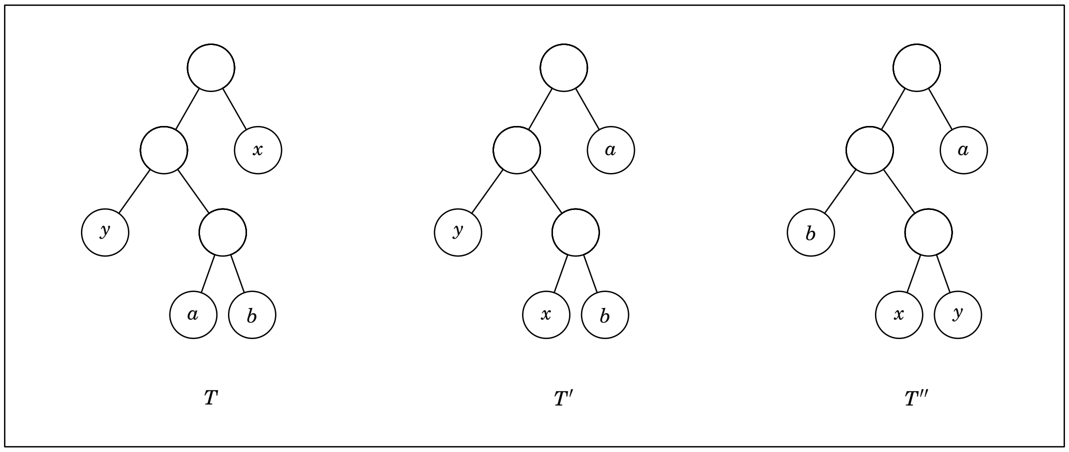

Figure 19.3. Illustration of the proof of Lemma 19.1.

19.2.2 Correctness

The Huffman algorithm is a greedy algorithm: at each stage, we merge together the two nodes of lowest frequency.

Lemma 19.1 (CLRS Lemma 16.2)**.** *Suppose x, y are the two most infrequent characters of C (with ties broken arbitrarily). Then there exists an optimal prefix code for C with codewords for x and y of the same length and differing only in the last bit. (In other words, there exists an optimal tree in which x and y are siblings.)*

Proof. Let T be an optimal tree. Let a be a leaf of maximal depth. If a has no sibling, then deleting a from the tree (and using a's parent to represent the character formerly represented by a) produces a better code, contradicting optimality. So a has a sibling b, and since a has maximum depth, b is a leaf. So a and b are nodes of maximal depth, and without loss of generality we can say $a.\mathit{freq} \le b.\mathit{freq}$.

Also without loss of generality, say $x.\mathit{freq} \le y.\mathit{freq}$. Switch the leaves a and x; call the resulting tree T' (see Figure 19.3). Then $B(T') \le B(T)$, seeing as $x.\mathit{freq} \le a.\mathit{freq}$. So it must be the case that $x.\mathit{freq} = a.\mathit{freq}$ and T' is optimal as well. Similarly, switch the leaves b and y in T'; call the resulting tree T''. It must be the case that $y.\mathit{freq} = b.\mathit{freq}$ and T'' is optimal. In T'', the leaves x and y are siblings. □

Lemma 19.2 (CLRS Lemma 16.3)**.** *Given an alphabet C, let:*

- *x, y be the two most infrequent characters in C (with ties broken arbitrarily)*
- *z be a new symbol, with $z.\mathit{freq} \leftarrow x.\mathit{freq} + y.\mathit{freq}$*
- *$C' = \big(C \setminus \{x, y\}\big) \cup \{z\}$*
- *T' be an optimal tree for C'.*

Then T is an optimal tree for C, where T is a copy of T' in which the leaf for z has been replaced by an internal node having x and y as children.

Proof. Suppose T is not optimal. By Lemma 19.1, let $\mathscr{T}$ be an optimal tree for C in which x and y are siblings. Delete x and y from $\mathscr{T}$, and label their parent (now a leaf) with the symbol z. Call the resulting tree $\mathscr{T}'$. Notice that $\mathscr{T}'$ is a tree for C', and furthermore,

$$\begin{aligned} B(\mathscr{T}') &= B(\mathscr{T}) - x.\mathit{freq} - y.\mathit{freq} \\ &< B(T) - x.\mathit{freq} - y.\mathit{freq} \\ &= B(T'). \end{aligned}$$

This contradicts the optimality of T'. We conclude that T must have been optimal in the first place. □

Corollary 19.3. *The Huffman algorithm is correct.*

Proof sketch. Let T be the tree produced by the Huffman algorithm. Construct a new tree U as follows. Initially, let U consist of just one vertex. Then, perform a series of transformations on U. In each transformation, append two children to what was previously a leaf node. In this way, we can eventually transform U into T. Moreover, Lemma 19.2 guarantees that, at each stage, U is an optimal tree for the alphabet formed by its leaves. □