In this lecture we will discuss the greedy principle. Even though it is very hard to formally define what a greedy algorithms is, in general, greedy algorithms attempt to locally optimize a criterion at every step to the end of building an overall solution for the problem. This optimization is based on short-sighted decisions.

If a greedy algorithm is optimal for a problem then there is a local decision rule that one can use to construct optimal solutions. A typical approach for showing optimality of the greedy algorithm is by using an *exchange argument*: start with a general solution to the problem and gradually transform it into the solution found by the greedy algorithm, without decreasing its quality. By this argument the greedy algorithm finds a solution that is at least as good as any other solution.

The first greedy algorithm we will discuss is Huffman Coding. This algorithm arises in the area of **data compression**. Data compression deals with finding compact representations of data so that they occupy less space when stored. You probably encounter data compression standards daily: `jpeg` and `gif` for image transmission, `mp3` for audio content, `mpeg2` for video transmission, as well as utilities such as `gzip` and `bzip2`. All of these compression standards use the Huffman algorithm as a basic building block (as well as other techniques).

1 Symbol codes

Let's take a closer look at data representation. Computers operate on sequences of bits (0 and 1). For example, to represent the English letters in a computer we need a way to represent them by bits. A representation that takes as input English characters and outputs a binary string is called an encoding.

Now suppose we want to store the map of a chromosome. This consists of a sequence of hundreds of millions of nucleotide bases which, for our purposes, are just symbols of the form A, C, G and T. In particular, consider a chromosome whose map consists of 200 million bases. A straightforward way to represent this very long sequence of symbols in binary is by encoding every symbol that appears in the sequence separately by a *fixed length binary string*. We will call this binary string the *codeword* for the symbol and denote by $c(x)$ the codeword for symbol x.

For example, in the problem above, the input alphabet is $\{A, C, G, T\}$. Since there are 4 different symbols we can represent each of them with 2 bits and let $c(A) = 00$, $c(C) = 01$, $c(G) = 10$, $c(T) = 11$ be the codewords for $\{A, C, G, T\}$ respectively. The output of our encoding is the concatenation of the codewords for every symbol in the input sequence. For example, if 'ACGTAA' is a part of our input, then we encode it as

$$c(A)c(C)c(G)c(T)c(A)c(A) = 000110110000$$

The total length of the encoding for 'ACGTAA' $= 6 \cdot 2 = 12$.

A set of codewords as above provides a *symbol* code where every input symbol is encoded separately. We denote the above code for $\{A, C, G, T\}$ by C_0. Note that C_0 is a **fixed-length** symbol code: each one of its codewords has the same length. Another example of a fixed-length symbol code is the ASCII encoding system: every character and special symbol on the computer keyboard is encoded by a different 7-bit binary string. Thus there are $2^7 = 128$ symbols that can be represented by the ASCII code.

1.1 Unique decodability

Given C_0 (or any fixed-length symbol code), it is simple to decode for the original symbols. Read two bits (or a fixed number of bits) of the output string and output the symbol corresponding to this codeword; discard these bits and continue with the next two bits.

Also, it is easy to see that if $\mathbf{a} = a_1a_2a_3\ldots$ and $\mathbf{b} = b_1b_2b_2\ldots$ are two distinct input sequences of symbols from $\{A, C, G, T\}$, then they have distinct encodings: if $a_i \neq b_i$ for some $i \geq 1$, then positions $2i+1$ and $2i+2$ of the encodings of $\mathbf{a}$ and $\mathbf{b}$ will not be identical, since $c(a_i) \neq c(b_i)$.

Definition 1 *A symbol code is* **uniquely decodable** *if, for any two distinct sequences of input symbols, their encodings are distinct.*

Uniquely decodable codes allow for **lossless compression:** we may compress and decompress without errors. The Huffman algorithm gives a symbol code that achieves optimal lossless compression and operates on a **greedy** principle.

2 Prefix codes and optimal lossless compression

More concretely, a symbol code achieves optimal lossless compression if, given any input, it produces an encoded output (i.e., a binary string) of minimum size among all symbol codes, while maintaining the unique decodability property (so that compression is lossless).

For example, suppose we knew that in our chromosome map, A appears 110 million times, C 5 million times, G 25 million times and T 60 million times. Then, intuitively, it seems unlikely that the fixed-length encoding suggested above offers the most compact way to represent the input. The reason is that A appears much more often than the other symbols so we might have significant gains if we used fewer bits to encode it. Hence it seems appropriate to use **variable-length** encodings instead of fixed-length ones. For example, we could use code C_1 where $c(A) = 0$, $c(C) = 00$, $c(G) = 10$, $c(T) = 1$. But then we don't have unique decodability anymore, because consider the encoding 101110. We cannot even decode for the first symbol because we do not know where its codeword ends!

Now consider code C_2 as follows:

$$c(A) = 0, c(C) = 111, c(G) = 110, c(T) = 10$$

This code has the nice property that no codeword is a prefix of another codeword. In other words, no codeword contains or is contained in another codeword. Such a code is called *prefix* (or prefix-free) code.

Definition 2 *A symbol code is a prefix code if no codeword is a prefix of another.*

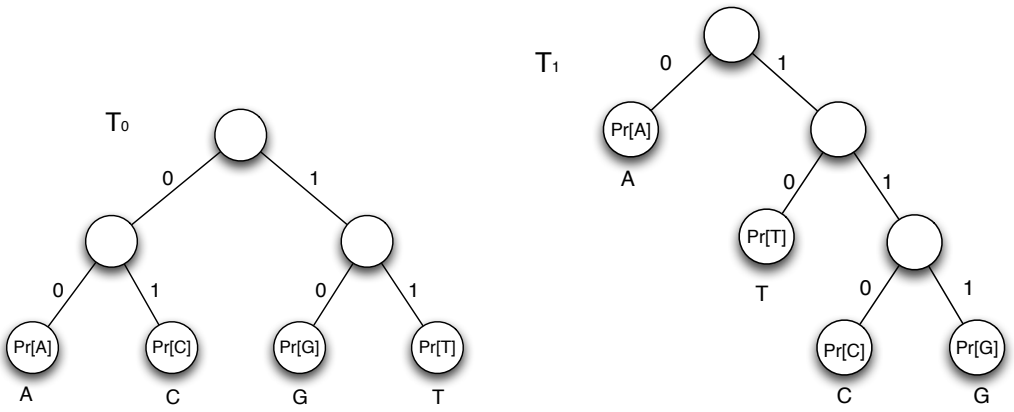
For example, C_0 is a prefix code but C_1 is not. It is easy to decode the binary string output by a prefix code using the following algorithm:

- Scan the binary string from left to right until you've seen enough bits to match a codeword; then output the symbol corresponding to this codeword. (Since no other codeword is a prefix of or contains this codeword, this sequence of bits can't be used to encode any other symbol.)
- Continue starting with the next bit of the bit string.

Prefix codes are uniquely decodable: codes that are not uniquely decodable do not allow one to identify where one codeword ends and another begins; prefix codes not only allow one to identify where a codeword ends, they do that *soon as* the codeword ends.[1]

Given that decoding a prefix code is so easy, it would be nice if, to achieve optimal lossless compression using symbol codes, we could focus entirely on prefix codes (instead of all uniquely decodable symbol codes). Fortunately a fact from information theory guarantees that we do not lose any performance by restricting attention to prefix codes. So from now on we will solely focus on prefix codes to achieve optimal compression.

[1] Because the end of a codeword is immediately recognizable, they are also called *instantaneous* codes.



(a) Binary tree for prefix code C_0

(b) Binary tree for prefix code C_1

Figure 1: Binary trees for the prefix codes C_0 and C_1.

Let's revisit code C_3: have we gained anything in terms of compression by using this variable-length prefix code? Using C_3 to encode the 200 million symbol input file, we obtain an output of

$$110 \cdot 1 + 2 \cdot 60 + 3 \cdot 25 + 3 \cdot 5 = 320 \text{ million bits.}$$

On the other hand, using the fixed length encoding C_0, the length of the encoded file is 400 million bits. Hence we obtained an improvement of 20% by using the variable-length encoding!

2.1 Optimal lossless compression using prefix codes

We start with a somewhat more formal definition for our framework. Consider an input alphabet $\mathcal{A}$ consisting of symbols $\{a_1, \ldots, a_n\}$, each having probability $\{p_1, \ldots, p_n\}$; we denote the latter set by P. In the chromosome example above, we have $\mathcal{A} = \{A, C, G, T\}$. The frequencies of A, C, G, T define the set of empirical probabilities $P = \{110/200, 5/200, 25/200, 60/200\}$. Let C be a binary prefix code for $(\mathcal{A}, P)$ with codewords $c(a_i)$. Let ℓ_i be the length of codeword $c(a_i)$. The expected length $L(C)$ of C is

$$L(C) = \sum_{a_i \in \mathcal{A}} p_i \cdot \ell_i$$

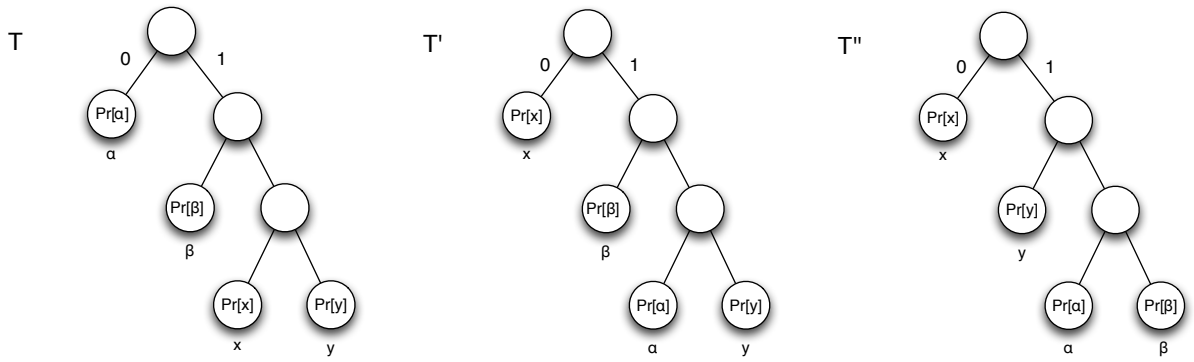
Our goal is to find the prefix code C^* that has the minimum length $L(C^*)$ for the pair $(\mathcal{A}, P)$. For example, we have $L(C_0) = 2$ while $L(C_2) = 1.6$. This is actually the optimal encoding for our pair $(\mathcal{A}, P)$ and it is what the Huffman algorithm outputs. In the next few sections we show that the Huffman algorithm is an optimal prefix code, meaning it achieves the optimal compression among all prefix codes.

2.2 Prefix codes and trees

A natural way to think about prefix codes is to picture them as **binary trees**.

Definition 3 *A binary tree T is a rooted tree such that each node that is not a leaf has at most two children.*

In a binary tree that represents a prefix code, a branch to the left represents a 0 in the encoding and a branch to the right a 1. For example, the binary trees that correspond to C_0 and C_1 are given in Figure 1.



(a) Optimal T with α, β not siblings (b) T' with α at max depth (c) T'' with α, β at max depth

Figure 2: Exchange argument for optimal tree T'' where α, β are siblings at maximum depth.

Note that there can be more than one optimal encoding: just flip the left and right-hand side of T_1. Also, since the binary tree represents a prefix code, the symbols must appear at the leaves, otherwise the prefix property would be violated. Hence the tree has n leaves and the codewords are given by root-to-leaf paths. Therefore $\ell_i = \text{depth}_T(i)$ and we can now express the **expected length** of the prefix code corresponding to tree T as

$$L(T) = \sum_{a_i \in \mathcal{A}} p_i \cdot \ell_i = \sum_{1 \le i \le n} p_i \cdot \text{depth}_T(i).$$

We will now focus on the tree corresponding to the optimal prefix code. First note that such a tree must be **full**: all internal nodes must have exactly two children (otherwise delete the internal node with one child and obtain a better code). Also, the following property of the tree for the optimal prefix code will be useful for our subsequent analysis.

Claim 1 *There is an optimal prefix code, with corresponding tree T^*, in which the two lowest frequency characters are assigned to leaves that are siblings in T^* at maximum depth.*

Proof. We will prove the claim by using an exchange argument. That is, we will start with a tree for an optimal prefix code and transform this tree into T^*.

Let T be the tree for the optimal prefix code. Let α and β be the two characters with the smallest probabilities such that $\Pr[\alpha] \le \Pr[\beta] \le \Pr[s]$ for all $s \in \mathcal{A} - \{\alpha, \beta\}$. Let x and y be the two siblings at maximum depth in T; see Figure 2(a).

Now $\Pr[\alpha] \le \Pr[x]$ and $\Pr[\beta] \le \Pr[y]$. We exchange α with x and construct tree T' as in Figure 2(b). How do the expected lengths of the two trees compare?

$$\begin{aligned}
L(T) - L(T') &= \sum_{a_i \in \mathcal{A}} \Pr[a_i] \cdot \text{depth}_T(i) - \sum_{a_i \in \mathcal{A}} \Pr[a_i] \cdot \text{depth}_{T'}(i) \\
&= \Pr[\alpha] \cdot \text{depth}_T(\alpha) + \Pr[x] \cdot \text{depth}_T(x) - \Pr[\alpha] \cdot \text{depth}_{T'}(\alpha) - \Pr[x] \cdot \text{depth}_{T'}(x) \\
&= \Pr[\alpha] \cdot \text{depth}_T(\alpha) + \Pr[x] \cdot \text{depth}_T(x) - \Pr[\alpha] \cdot \text{depth}_T(x) - \Pr[x] \cdot \text{depth}_T(\alpha) \\
&= (\Pr[\alpha] - \Pr[x]) \cdot (\text{depth}_T(\alpha) - \text{depth}_T(x)) \ge 0
\end{aligned}$$

The third line follows because the depth of α in tree T' is the same as the depth of x in tree T, since they were exchanged. Hence the expected length of T' is no larger than that of T. The same analysis shows that if we exchange β and y in T', as shown in Figure 2(c), $L(T') - L(T'') \ge 0$. Combining the two inequalities leads to $L(T'') \le L(T)$, but since T is optimal we conclude that $L(T'') = L(T)$ and T'' is also optimal. The claim follows by taking T^* to be T''. □

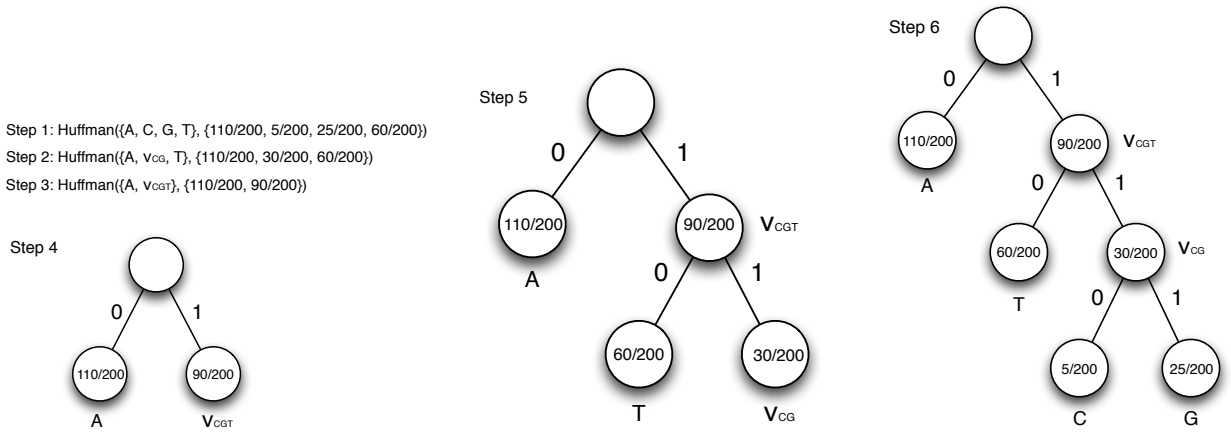


Figure 3: Execution of recursive Huffman algorithm for the chromosome example in Section 2. The execution progresses top down, from left to right. The algorithm starts constructing the tree at the third recursive call.

2.3 Huffman coding

Claim 1 tells us how to build the optimal tree greedily. Take the two characters (symbols) with the lowest probabilities, delete them from the alphabet, and replace them with a new meta-character; this new meta-character will be the parent of the two deleted characters in the tree. Recursively construct the tree using this process. We obtain the following recursive algorithm.

Huffman$(\mathcal{A}, P)$

if $|\mathcal{A}| = 2$ **then**

Encode one character using 0 and the other character using 1

end if

Let α and β be the two characters with the lowest probabilites (break ties arbitrarily).

Form a new alphabet $\mathcal{A}_1 = \mathcal{A} - \{\alpha, \beta\} + \{\nu\}$ where ν is a new meta-character with probability $\Pr[\alpha] + \Pr[\beta]$; let P_1 be the new set of probabilities over $\mathcal{A}_1$.

$T_1 = \text{Huffman}(\mathcal{A}_1, P_1)$ *//since ν is a meta-character, thus a character in alphabet $\mathcal{A}_1$, ν is a leaf in T_1; T_1 gives the prefix code for the pair $\mathcal{A}_1, P_1$.*

Return the following tree T: start with T_1, make ν an internal node, and add two children labelled α and β below node ν.

Note that this algorithm returns a a binary tree T; interpreting this tree as a prefix code gives the code for the pair $(\mathcal{A}, P)$. Figure 3 shows the execution of this algorithm step by step and the resulting prefix code for our chromosome map example from Section 2.

Next we show that the prefix tree returned by Huffman coding is indeed optimal.

2.4 Analysis of Huffman coding: correctness

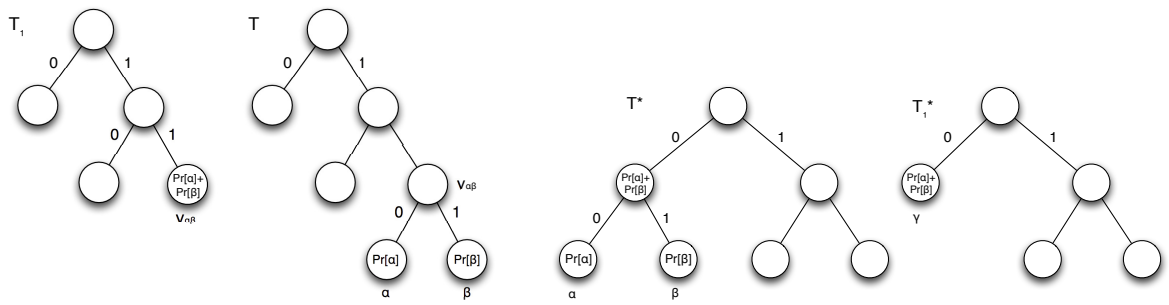
We will prove optimality by induction on $n \geq 2$, the size of our alphabet.

Proof. *Base case.* For an alphabet of two characters, Huffman is obviously optimal.

Hypothesis. Assume that Huffman returns the optimal prefix code for an alphabet of n characters.

Induction Step. We will show that Huffman is optimal for an alphabet of size $n + 1$. Let $\mathcal{A}$ be the alphabet of size $n + 1$, $P = \{p_1, \ldots, p_{n+1}\}$ the corresponding probabilities. Let T be the tree our algorithm constructs for $(\mathcal{A}, P)$. We want to show that T is optimal.

Our algorithm constructs T as follows: it chooses the two characters α, β with the smallest



(a) Tree T returned by Huffman coding for $(\mathcal{A}, P)$.

(b) Optimal tree T^* for $(\mathcal{A}, P)$.

Figure 4: Trees T and T^* in proof of optimality for Huffman. T is returned by the Huffman algorithm while T^* is the optimal tree for the pair $(\mathcal{A}, P)$.

probabilities and replaces them with a new meta-character ν such that $\Pr[\nu] = \Pr[\alpha] + \Pr[\beta]$. Let $\mathcal{A}_1 = \mathcal{A} - \{\alpha, \beta\} + \{\nu\}$, $P_1 = P - \{p_\alpha, p_\beta\} + \{p_\nu\}$.

Now our algorithm will recursively call itself on input $(\mathcal{A}_1, P_1)$. By the induction hypothesis, since $|\mathcal{A}_1| = n$, the algorithm returns an optimal tree T_1 for $(\mathcal{A}_1, P_1)$. Next it replaces the leaf node ν of T_1 by two children α and β and returns this new tree T as its prefix code; see Figure 4. Note that

$$\begin{aligned} L(T) &= L(T_1) + (\Pr[\alpha] + \Pr[\beta]) \cdot (\text{depth}_{T_1}(\nu) + 1) - (\Pr[\alpha] + \Pr[\beta]) \cdot \text{depth}_{T_1}(\nu) \\ &= L(T_1) + \Pr[\alpha] + \Pr[\beta]. \end{aligned} \tag{1}$$

We claim that T is optimal and will show this by contradiction.
Suppose that T is not optimal. Then there is an optimal tree T^* for $(\mathcal{A}, P)$ such that

$$L(T^*) < L(T). \tag{2}$$

By Claim 1, we may assume w.l.o.g. that in T^*, α and β are siblings at maximum depth. Now suppose that in T^*, we replace α and β by a meta-character γ such that $\Pr[\gamma] = \Pr[\alpha] + \Pr[\beta]$. Let T_1^* be the resulting tree; then

$$\begin{aligned} L(T_1^*) &= L(T^*) - \Pr[\alpha] \cdot \text{depth}_{T^*}(\alpha) - \Pr[\beta] \cdot \text{depth}_{T^*}(\beta) + (\Pr[\alpha] + \Pr[\beta]) \cdot \text{depth}_{T^*}(\gamma) \\ &= L(T^*) - \Pr[\alpha] - \Pr[\beta] \qquad (\textit{since } \text{depth}_{T^*}(\gamma) = \text{depth}_{T^*}(\alpha) - 1 = \text{depth}_{T^*}(\beta) - 1)) \\ &< L(T) - \Pr[\alpha] - \Pr[\beta] \qquad (\textit{by equation } (2)) \\ &= L(T_1) \qquad (\textit{by equation } (1)) \end{aligned}$$

Hence $L(T_1^*) < L(T_1)$. But this contradicts the optimality of T_1, so T is optimal. □

2.5 Analysis of Huffman: Running Time

A simple implementation of the algorithm in Section 2.3 requires time $O(n^2)$: there are $n-1$ recursive calls (the size of the input alphabet shrinks by one at every call) and we need $\Theta(n)$ time to identify the two lowest probability characters and merge them into a single character with the combined probability.

However, if a data structure known as a *priority queue* is used to store the alphabet, then the Huffman algorithm requires $O(n \log n)$ time. A priority queue (discussed in Chapter 6 in your textbook) is a data structure that maintains a set of elements, each with a numerical key. Here we can maintain the alphabet, using the probabilities as keys. A priority queue allows for the following operations:

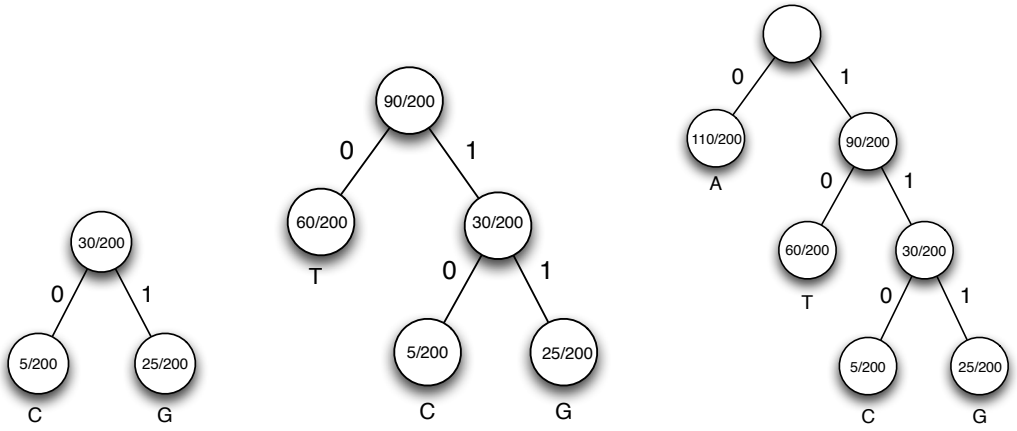


Figure 5: Execution of iterative Huffman algorithm for the chromosome example in Section 1.1. The execution progresses from left to right.

- Initialize the queue as an empty binary heap in $O(n)$ time.
- Extract the minimum element from the queue in $O(\log n)$ (this is performed twice per recursive call).
- Insert an element in the queue in $O(\log n)$ (this operation is performed once per recursive call).

It is easy to write the above recursive algorithm as an iterative one (you can find the pseudocode in your textbook but think about it on your own). In practice, we use the iterative approach. An execution of the iterative Huffman algorithm for the chromosome map example from Section 2 is shown in Figure 5.

2.6 Beyond Huffman coding

We argued that the Huffman algorithm provides an optimal symbol code for the ensemble $(\mathcal{A}, P)$. However this does not imply that it achieves better compression than codes that do not encode every input symbol separately. Indeed, variable-length codes that encode larger blocks of input symbols achieve better compression. Examples of such codes, known as *stream* codes, are arithmetic coding and the Lempel-Ziv algorithm.

Also, if a bit from the output of the compressor is flipped, then decompression cannot carry through for any of the above methods. So if compressed files are to be stored or transmitted over noisy media, we need to use error correcting codes on top of data compression.