Analysis of Algorithms, CSOR W4231

CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Data compression and huffman coding

Outline

Today

Motivation

Data compression: find compact representations of data

Data compression standards

- `jpeg` for image transmission
- `mp3` for audio content, `mpeg2` for video transmission
- utilities: `gzip`, `bzip2`

All of the above use the Huffman algorithm as a basic building block.

Data representation

- An organism's genome consists of *chromosomes* (giant linear DNA molecules)
- *Chromosome maps*: sequences of hundreds of millions of bases (symbols from $\{A, C, G, T\}$).
- **Goal:** store a chromosome map with 200 million bases.

How do we represent a chromosome map?

Data representation

- An organism's genome consists of *chromosomes* (giant linear DNA molecules)
- *Chromosome maps*: sequences of hundreds of millions of bases (symbols from $\{A, C, G, T\}$).
- **Goal:** store a chromosome map with 200 million bases.

How do we represent a chromosome map?

- Encode every symbol that appears in the sequence separately by a fixed length binary string.
- Codeword $c(x)$ for symbol x: a binary string encoding x of length $\ell(x)$

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example

- Input sequence: $ACGTAA$

Example code

- Alphabet $\mathcal{A} = \{A, C, G, T\}$ with 4 symbols
- Encode each symbol with 2 bits

alphabet symbol x	codeword $c(x)$
A	00
C	01
G	10
T	11

Output of encoding: the concatenation of the codewords for every symbol in the input sequence

Example

- Input sequence: $ACGTAA$
- Output: $c(A)c(C)c(G)c(T)c(A)c(A) = 000110110000$
- Total length of encoding $= 6 \cdot 2 = 12$ bits.

Today

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Examples of symbol codes

- $C_0 = \{00, 01, 10, 11\}$ is a symbol code for $\{A, C, G, T\}$.
- ASCII encoding system: every character and special symbol on the computer keyboard is encoded by a different 7-bit binary string.

Symbol codes

Symbol code: a set of codewords where every input symbol is encoded **separately**.

Examples of symbol codes

- $C_0 = \{00, 01, 10, 11\}$ is a symbol code for $\{A, C, G, T\}$.
- ASCII encoding system: every character and special symbol on the computer keyboard is encoded by a different 7-bit binary string.

Remark 1.

C_0 and ASCII are fixed-length symbol codes: each codeword has the same length.

Unique decodability

Decoding C_0?

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Remark 2.

- *This decoding algorithm works for ASCII (replace 2 by 7)*
- C_0*, ASCII: distinct input sequences have distinct encodings*

Unique decodability

Decoding C_0

- read 2 bits of the output;
- print the symbol corresponding to this codeword;
- continue with the next 2 bits.

Remark 2.

- *This decoding algorithm works for ASCII (replace 2 by 7)*
- *C_0, ASCII: distinct input sequences have distinct encodings*

Definition 1.

A symbol code is uniquely decodable if, for any two distinct input sequences, their encodings are distinct.

Lossless compression

- Lossless compression: compress and decompress without errors.
- Uniquely decodable codes allow for lossless compression.
- A symbol code achieves optimal lossless compression when it produces an encoding of minimum length for its input (among all uniquely decodable symbol codes).
- Huffman algorithm: provides a symbol code that achieves optimal lossless compression.

Fixed-length vs variable-length codes

Chromosome map consists of 200 million bases as follows:

alphabet symbol x	frequency $freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Fixed-length vs variable-length codes

Chromosome map consists of 200 million bases as follows:

alphabet symbol x	frequency $freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

- A appears much more often than the other symbols.

$\Rightarrow$ It might be best to encode A with fewer bits.

- Unlikely that the fixed-length encoding C_0 is optimal

Today

Prefix codes

Variable-length encodings

Code C_1

alphabet symbol x	codeword $c(x)$
A	0
C	00
G	10
T	1

Prefix codes

Variable-length encodings

Code C_1

alphabet symbol x	codeword $c(x)$
A	0
C	00
G	10
T	1

- C_1 is not unique decodable! E.g., 101110: *how to decode it?*

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

- C_2 is uniquely decodable.
- C_2 is such that no codeword is a prefix of another.

Prefix codes

Variable-length encodings

Code C_2

alphabet symbol x	codeword $c(x)$
A	0
C	110
G	111
T	10

- C_2 is uniquely decodable.
- C_2 is such that no codeword is a prefix of another.

Definition 2 (prefix codes).

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

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
   - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
    - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Thus prefix codes allow for

- unique decoding;
- fast decoding (the end of a codeword is instantly recognizable).

Decoding prefix codes

1. Scan the binary string from left to right until you've seen enough bits to match a codeword;
2. Output the symbol corresponding to this codeword.
    - Since no other codeword is a prefix of this codeword or contains it as a prefix, this sequence of bits cannot be used to encode any other symbol.
3. Continue starting from the next bit of the bit string.

Thus prefix codes allow for

- unique decoding;
- fast decoding (the end of a codeword is instantly recognizable).

Examples of prefix codes: C_0, C_2

Prefix codes and optimal lossless compression

- Decoding a prefix code is very fast.

⇒ Would like to focus on prefix codes (rather than **all** uniquely decodable symbol codes) for achieving optimal lossless compression.

- Information theory guarantees this: for every uniquely decodeable code, exists a prefix code with the **same codeword lengths**
- So we can solely focus on prefix codes for optimal compression.

Compression gains from variable-length prefix codes

Chromosome map: *do we gain anything by using C_2 instead of C_0 when compressing the map of 200 million bases?*

Input

symbol x	$freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

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

Compression gains from variable-length prefix codes

Chromosome map: *do we gain anything by using C_2 instead of C_0 when compressing the map of 200 million bases?*

Input

symbol x	$freq(x)$
A	110 million
C	5 million
G	25 million
T	60 million

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

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

- C_0: 2 bits $\times 200$ million symbols $= 400$ million bits
- C_2: $1 \cdot 110 + 3 \cdot 5 + 3 \cdot 25 + 2 \cdot 60 = 320$ million bits
- Improvement of 20% in this example

The optimal prefix code problem

Input:

- Alphabet $\mathcal{A} = \{a_1, \ldots, a_n\}$
- Set $P = \{p_1, \ldots, p_n\}$ of empirical probabilities over $\mathcal{A}$ such that $p_i = \Pr[a_i]$

Output: a binary prefix code $C^* = \{c(a_1), c(a_2), \ldots, c(a_n)\}$ for $(\mathcal{A}, P)$, where codeword $c(a_i)$ has length ℓ_i and is such that its expected length

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

is **minimum** among all binary prefix codes.

Example

Chromosome example

Input

symbol x	$\Pr(x)$
A	$110/200$
C	$5/200$
G	$25/200$
T	$60/200$

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11

Code C_2

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

- $L(C_0) = 2$
- $L(C_2) = 1.6$
- *Coming up:* C_2 is the output of the Huffman algorithm, hence an optimal encoding for $(\mathcal{A}, P)$.

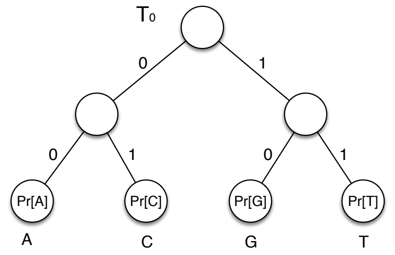
Today

Prefix codes and trees

- A binary tree T is a rooted tree such that each node that is not a leaf has at most two children.
- Binary tree for a prefix code: a branch to the left represents a 0 in the encoding and a branch to the right a 1.

Code C_0

x	$c(x)$
A	00
C	01
G	10
T	11



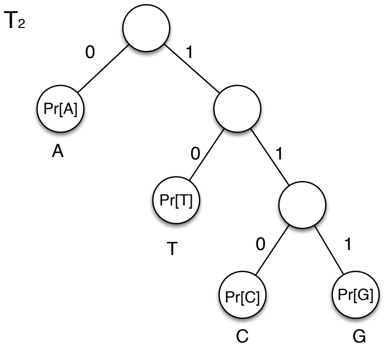
Binary tree for prefix code C_0

Prefix codes and trees

- A binary tree T is a rooted tree such that each node that is not a leaf has at most two children.
- Binary tree for a prefix code: a branch to the left represents a 0 in the encoding and a branch to the right a 1.

Code C_2

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



Binary tree for prefix code C_2

Properties of binary trees representing prefix codes

1. *Where do alphabet symbols appear in the tree?*
2. *What do codewords correspond to in the tree?*
3. *Consider the tree corresponding to the optimal prefix code. Can it have internal nodes with one child?*

Properties of binary trees representing prefix codes

1. Symbols must appear at the leaves of the tree T (*why?*)

   $\Rightarrow$ T has n leaves.

2. Codewords $c(a_i)$ are given by root-to-leaf paths.

   Recall that ℓ_i is the length of the codeword $c(a_i)$ for input symbol a_i. Therefore, on the tree T, ℓ_i corresponds to the depth of a_i (we assume that the root is at depth 0).

   $\Rightarrow$ Can rewrite the **expected length** of the prefix code as:

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

3. Optimal tree must be full: all internal nodes must have exactly two children (*why?*).

More on optimal tree

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.

More on optimal tree

Claim 1.

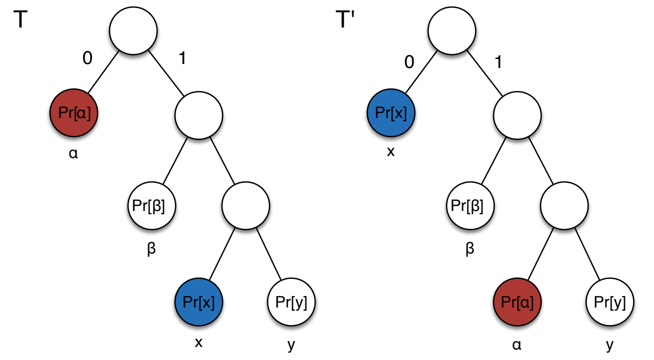
There is an optimal prefix code, with corresponding tree T^*, in which the two lowest frequency characters are assigned to leaves that are siblings in T^* at maximum depth.

Proof.

By an exchange argument: start with a tree for an optimal prefix code and transform it into T^*. □

Proof of Claim 1

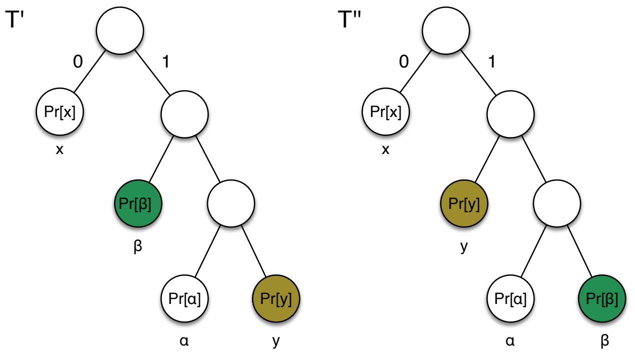
- Let T be the tree for the optimal prefix code.
- Let α, β be the two symbols with the smallest probabilities, that is, $\Pr[\alpha] \leq \Pr[\beta] \leq \Pr[s]$ for all $s \in \mathcal{A} - \{\alpha, \beta\}$.
- Let x and y be the two siblings at maximum depth in T.



We want L(T) ≥ L(T′)

Proof of Claim 1

- Let T be the tree for the optimal prefix code.
- Let α, β be the two symbols with the smallest probabilities, that is, $\Pr[\alpha] \leq \Pr[\beta] \leq \Pr[s]$ for all $s \in \mathcal{A} - \{\alpha, \beta\}$.
- Let x and y be the two siblings at maximum depth in T.



We want L(T′) ≥ L(T′′)

How do the expected lengths of the two trees compare?

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

- The third equality follows from the exchange.
- Similarly, exchanging β and y in T' yields $L(T') - L(T'') \geq 0$.
- Hence $L(T) - L(T'') \geq 0$.
- Since T is optimal, it must be $L(T) = L(T'')$.
- So T'' is also optimal.

The claim follows by setting T^* to be T''.

Building the optimal tree

Claim 1 tells us how to build the optimal tree greedily!

1. Find the two symbols with the lowest probabilities.
2. Remove them from the alphabet and replace them with a new meta-character with probability equal to the sum of their probabilities.
   - **Idea:** this meta-character will be the parent of the two deleted symbols in the tree.
3. Recursively construct the optimal tree using this process.

Greedy *algorithms: make a local (myopic) decision at every step that optimizes some criterion and eventually show that this is the optimal way for building the entire solution.*

Today

1. Data compression
2. Symbol codes and optimal lossless compression
3. Prefix codes
4. Prefix codes and trees
5. The Huffman algorithm

Huffman algorithm

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

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

Encode one symbol using 0 and the other using 1

end if

Let α and β be the two symbols with the lowest probabilities

Let ν be a new meta-character with probability $\Pr[\alpha] + \Pr[\beta]$

Let $\mathcal{A}_1 = \mathcal{A} - \{\alpha, \beta\} + \{\nu\}$

Let P_1 be the new set of probabilities over $\mathcal{A}_1$

$T_1 =$ `Huffman`$(\mathcal{A}_1, P_1)$

`return` T as follows: replace leaf node ν in T_1 by an internal node, and add two children labelled α and β below ν.

Remark 3.

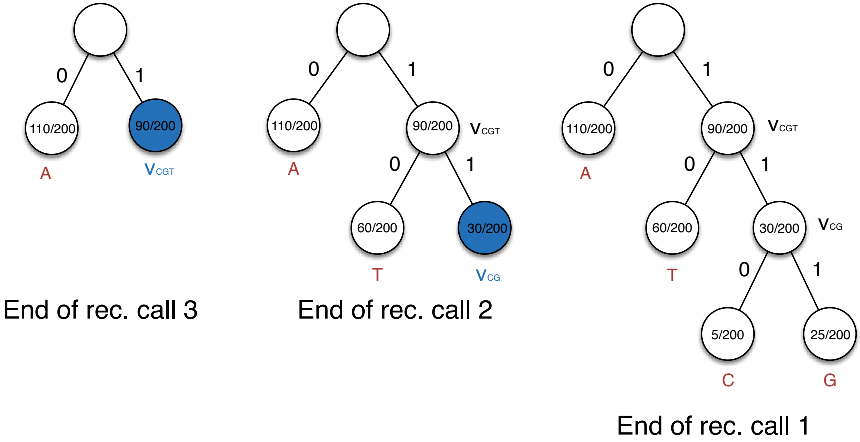
Output of `Huffman` *procedure is a binary tree* T*; the code for* $(\mathcal{A}, P)$ *is its corresponding prefix code.*

Example: recursive Huffman for chromosome map

Recursive call 1: $\texttt{Huffman}(\{A, C, G, T\}, \{\frac{110}{200}, \frac{5}{200}, \frac{25}{200}, \frac{60}{200}\})$

Recursive call 2: $\texttt{Huffman}(\{A, \nu_{CG}, T\}, \{\frac{110}{200}, \frac{30}{200}, \frac{60}{200}\})$

Recursive call 3: $\texttt{Huffman}(\{A, \nu_{CGT}\}, \{\frac{110}{200}, \frac{90}{200}\})$



Correctness

Proof: by induction on the size of the alphabet $n \geq 2$.

- **Base case.** For $n = 2$, Huffman is optimal.
- **Hypothesis.** Assume that Huffman returns the optimal prefix code for alphabets of n symbols.
- **Induction Step.** Let $\mathcal{A}$ be an alphabet of size $n + 1$, P the corresponding set of probabilities.

  Let T_1 be the optimal (by the hypothesis) tree returned by our algorithm for $(\mathcal{A}_1, P_1)$, where $\mathcal{A}_1, P_1, T_1$ as in the pseudocode. Let T be the final tree returned for $(\mathcal{A}, P)$ by our algorithm. We claim that T is optimal.

  We will prove the claim by contradiction. **Assume** T^* is the optimal tree for $(\mathcal{A}, P)$ such that

$$L(T^*) < L(T). \tag{1}$$

A useful fact

Fact 3.

Let T be a binary tree representing a prefix code. If we replace sibling leaves α, β in T by a meta-character ν where $\Pr[\nu] = \Pr[\alpha] + \Pr[\beta]$, we obtain a tree T_1 such that

$$L(T) = L(T_1) + (\Pr[\alpha] + \Pr[\beta]).$$

Proof.

Notation: $d_T(a_i) = \mathrm{depth}_T(a_i)$

- α, β are sibling leaves in T, hence $d_T(\alpha) = d_T(\beta)$.
- T differs from T_1 only in that α, β are replaced by ν. Since $d_{T_1}(\nu) = d_T(\alpha) - 1$, we obtain

$$\begin{aligned} L(T) - L(T_1) &= \Pr[\alpha] d_T(\alpha) + \Pr[\beta] d_T(\beta) - (\Pr[\alpha] + \Pr[\beta]) d_{T_1}(\nu) \\ &= \Pr[\alpha] + \Pr[\beta]. \end{aligned} \quad (2)$$

□

Correctness (cont'd)

- Claim 1 guarantees there is such an optimal tree for $(\mathcal{A}, P)$ where α, β appear as siblings at maximum depth.
- W.l.o.g. assume that T^* is such an optimal tree. By Fact 3, if we replace siblings α, β in T^* by ν' where $\Pr[\nu'] = \Pr[\alpha] + \Pr[\beta]$, the resulting tree T_1^* satisfies $L(T^*) = L(T_1^*) + (\Pr[\alpha] + \Pr[\beta])$.
- Similarly, the tree T returned by the Huffman algorithm satisfies $L(T) = L(T_1) + (\Pr[\alpha] + \Pr[\beta])$.
- By the induction hypothesis, we have $L(T_1^*) \geq L(T_1)$ since T_1 is optimal for alphabets of size n. Hence

$$L(T^*) = L(T_1^*) + \Pr[\alpha] + \Pr[\beta] \geq L(T_1) + \Pr[\alpha] + \Pr[\beta] = L(T), \quad (3)$$

  where the inequality follows from the induction hypothesis.
- Equation (3) contradicts Assumption (1). Thus T must be optimal.

Implementation and running time

1. Straightforward implementation: $O(n^2)$ time
2. Store the alphabet symbols in a min priority queue implemented as a binary min-heap with keys their probabilities
   - **Operations:** `Initialize` ($O(n)$), `Extract-min` ($O(\log n)$), `Insert` ($O(\log n)$)

   Total time: $O(n \log n)$ time

For an iterative implementation of Huffman, see your textbook.

Example: iterative Huffman for chromosome map

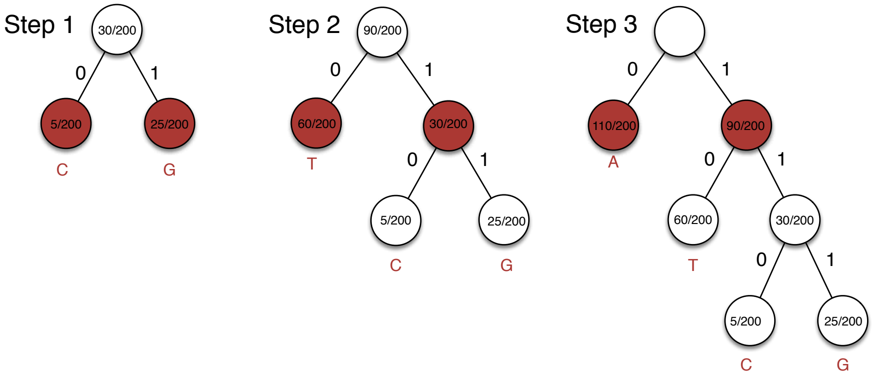
Input $(\mathcal{A}, P)$

symbol x	$\Pr(x)$
A	$110/200$
C	$5/200$
G	$25/200$
T	$60/200$

$\rightarrow$

Output code

symbol x	$c(x)$
A	0
C	110
G	111
T	10



Beyond Huffman coding

- Huffman algorithm provides an optimal symbol code.
- Codes that encode larger blocks of input symbols might achieve better compression.
- Storage on noisy media: *what if a bit of the output of the compressor is flipped?*
  - Decompression cannot carry through.
  - Need error correction on top of compression.