COMP251: Data compression

Jérôme Waldispühl
School of Computer Science
McGill University
Based on slides from M. Langer (McGill) and (goodrich & Tamassia, 2009)

Information Theory

- When A communicates a message to B, A sends a bit string that encodes the message.
- The amount of information in the message depends not of the number of bits sent, but rather on the probability of that message being sent.
- How much information does a message transmit?

Data compression

- When A communicates to B, they first agree on a code.
- They choose a code such that are the most likely to be sent are encoded using fewer bits. This yields shorter messages on average.
- The length of the message should be approximately equal to the amount of information communicated (Shannon, 1948).

Codes & Codewords

Suppose you have a sample space $\Sigma$ (often called an alphabet).
Define a code to be a mapping:

$$C: \Sigma \rightarrow \{ \text{bit string} \}$$

For any $x \in \Sigma$, $C(x)$ is the codeword of $x$.
The length of a codeword is the number of bits in that codeword.

**Example:**

$\Sigma = \{ A, C, G, T \}$
$C(A) = 00$, $C(C) = 01$, $C(G) = 10$, $C(T) = 11$

Extension of a code

For an code $C$ on an alphabet $\Sigma$, we have a naturally defined code on sequences of elements from $\Sigma$.

**Example:**

$C(AGAT) = C(A) C(G) C(A) C(T) = 00 10 00 11$

Note: We concatenate the codewords of the elements (letters) of the sequence.

Fixed length code

All codewords have the same length.

**Example:**

- $\Sigma = \{ A, C, G, T \}$
  - $C(A) = 00$, $C(C) = 01$, $C(G) = 10$, $C(T) = 11$
- ASCII (8 bits), Unicode (16 bits)
Variable length code

Codewords can have different lengths.

Example: Morse code

Note: More common letters have shorter codewords.

Tree representation

Any code can be represented by a binary tree. Each codeword is a path from the root to a node representing the element (letter) encoded.

Principle: 0 for left child, 1 for right child.

Example: Morse code (short=0, long=1)

Prefix code

C is a prefix code if no codeword is a prefix of any other codeword.

Q: What does it mean for binary trees?

A: codewords are leaves!

Desambiguação

Prefix codes avoid the ambiguities that we saw with Morse code. How?

• Suppose B is sent a sequence of bits and B wants to decode this sequence.
• B wants to know the sequence of symbols that was encoded.
• If the code is a prefix code, then there is a simple method for decoding:
  o Repeatedly traverse the binary tree from root to leaf.
  o Each time B reaches a leaf, it reads off the symbol at the leaf, then returns to the root.

Desambiguación (Example)

Q: How to decode 0100110?

A:

0100110  A
0100110  AB
0100110  ABD
0100110  ABD

Result: ABD
Average code length

Given S, c, p, define:

\[
\bar{\lambda} = \sum_{s \in S} p(s) \cdot \lambda(s)
\]

Expected value of codework length \quad Length of codeword

Optimal prefix code: Given S and p(), choose a prefix code that minimizes the average code length.

Finding an optimal prefix code

First attempt: Recursively partition S into two subsets such that each split divides the probability in half as closely as possible.

Example:

<table>
<thead>
<tr>
<th>Event</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.32</td>
</tr>
<tr>
<td>B</td>
<td>0.25</td>
</tr>
<tr>
<td>C</td>
<td>0.2</td>
</tr>
<tr>
<td>D</td>
<td>0.18</td>
</tr>
<tr>
<td>E</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Finding an optimal prefix code

Q: How could you improve the code (i.e. reduce the average code length)?

A: Same the codewords of C & D!

Conclusion: This approach does not guarantee an optimal prefix code.

Optimal prefix code properties

Claim: For any optimal prefix code, internal node of the tree has two children.

Example: This tree cannot be optimal we could obtain a code with a lower average code length by contracting the red edge.

Optimal prefix code properties

Claim: For any optimal prefix code C, and any a, b \in S, if p(a) < p(b) then \lambda(a) \geq \lambda(b).

Proof: By contradiction.

• Suppose \lambda(a) < \lambda(b). Then swapping C(a) and C(b) reduces \lambda.
• Hence C was not optimal.
**Optimal prefix code properties**

**Claim:** For any optimal prefix code C, the two least probable elements have the same codeword length.

**Proof:** Otherwise we have a contradiction with previous properties.
- Let a, b have the smallest probability.
- Suppose \( \lambda(a) < \lambda(b) \).
- Then, b has a sibling d.
- We can reduce the average length by swapping a with d.

**Example**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>0.25</td>
<td>0.2</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>

From previous claim, D and E must have the same codeword length.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>0.25</td>
<td>0.2</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>

**Optimal prefix code properties**

**Claim:** There exists an optimal prefix code in which the two least probable elements of D have the same parent in the tree.

**Proof:**
- The 2 least probable elements have the same codeword length.
- So they are at the same level in the tree.
- Swapping codewords of same length does not change the average code length.

This claim suggests an algorithm (Huffman) for finding an optimal prefix code.

**Example**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>0.25</td>
<td>0.2</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>

We have now 4 elements (instead of 5). The next two least probable elements are C and (D,E). Thus, we make them siblings.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>0.25</td>
<td>0.2</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>

We have now 3 elements. The next two least probable elements are A and B, and we make them siblings.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.32</td>
<td>0.25</td>
<td>0.2</td>
<td>0.18</td>
<td>0.05</td>
</tr>
</tbody>
</table>
Huffman’s Algorithm

- Given a string $X$,
- Huffman’s algorithm constructs a prefix code
- that minimizes the size of the encoding of $X$.
- It runs in time $O(n + d \log d)$, where $n$ is the size of $X$ and $d$ is the number of distinct characters of $X$.
- A heap-based priority queue is used as an auxiliary structure.
- Greedy algorithm!

Algorithm: HuffmanEncoding($X$)

Input: string $X$ of size $n$

Output: optimal encoding trie for $X$

1. $C$ ← distinctCharacters($X$)
2. $Q$ ← new empty heap
3. for all $c$ ∈ $C$
   - $T_c$ ← new single-node tree storing $c$
   - $Q$.insert($\text{getFrequency}(c)$, $T_c$)
4. while $Q$.size() > 1
   - $f_1$ ← $Q$.minKey()
   - $T_1$ ← $Q$.removeMin()
   - $f_2$ ← $Q$.minKey()
   - $T_2$ ← $Q$.removeMin()
   - $T$ ← join($T_1$, $T_2$)
   - $Q$.insert($f_1 + f_2$, $T$)
5. return $Q$.removeMin()

Example

$X$ = abracadabra

Frequencies:

<table>
<thead>
<tr>
<th>Character</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
</tr>
<tr>
<td>r</td>
<td>2</td>
</tr>
</tbody>
</table>

Encoding:

- a: 0
- b: 110
- c: 100
- d: 101
- r: 111

Extended Huffman Tree Example