Lecture 19 — Compression

Storing large data sets

Two different types of goals

1. Maximize ease of access, manipulation, processing

2. Minimize size — reduces resource utilization for storage and transmission

Today will examine this set of goals

In general, data cannot be compressed

Imagine wanting to represent all m-bit strings using (m-t) bits

- We want each m-bit string to have a unique representation
- There are \(2^m\) m-bit strings
- But only \(2^{m-t}\) \((m-t)\)-bit strings to represent them

\(\Rightarrow\) So hard to compress if every string is used.

- If only a relatively small number of the possible strings appear — could compress.
- If the same "long" strings appear repeatedly, could represent them with a "short" string
- If could relax requirement that each string have unique representation, then compression might work but make "similar" strings identical.
Input Document $\xrightarrow{\text{COMPRESS}}$ Compressed Document $\xrightarrow{\text{DE-COMPRESS}}$ Re-constituted Document $\xrightarrow{\text{Huffman Coding, Lempel-Ziv}}$.

Lossless compression: $D = D'$

Lossy compression: $D'$ is close enough but not necessarily identical to $D$

Algorithms can also be non-adaptive or adaptive.

Non-adaptive: assumes knowledge of data (e.g., character frequencies).

Adaptive: assumes no knowledge of data, but builds such knowledge.

Framework

Input $\xleftarrow{\text{known alphabet}}$ (English characters: a, b, c, ...

Sequence of characters from known alphabet ("Hello World")

Seek Binary Codes ("Codewords") that represent each character as a binary string [Code = \{code words\}]

Output - Concatenated string of code words representing string of characters.
Fixed-length code example

- Each character assigned a code word of some length

  - Alphabet: \{a, b, c, d, e, f, g, h\} 8 letters

    \[
    \begin{array}{llllll}
    000 & 001 & 010 & 011 & 100 & 101 & 110 & 111
    \end{array}
    \]

    Need 3 bits to represent 8 characters

  - A file of 1 million characters requires 3 million bits to store

  - A code like this is simple to encode and decode

    \[ba ba = 001'000'001'000\]

    \[\text{t-need to keep track of register}\]

Run-length encoding

Imagine given a string of bits \[00000011100011000000\]

- Store as 6, 3, 3, 2, 5

- \(\rightarrow\) Fax machine transmission

- \(\rightarrow\) Also used in jpeg
Variable-length code

Variable length - code words may be different length
How might you choose to use short versus long code words?

IDEA: Use shorter code words for more frequent characters,
use longer code words for rarer ones.

Example

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
</table>
Frequency | 45% | 13% | 12% | 16% | 9% | 5% |
Code | 0 | 101 | 100 | 111 | 1101 | 1100 |
1 bit | 3 bits | 4 bits |

\[ \left( \frac{45}{1} + \frac{13}{3} + \frac{12}{3} + \frac{16}{3} + \frac{9}{4} + \frac{5}{4} \right) \times 8 \text{ bits} = 224 \text{ bits} \]

Compare to 3-bit code using 300 bits

Why didn't I use "1" or "01" or "10", which would have made even shorter encoding?

Here we did "prefix coding": No code word is a prefix of another code word.

All word decoding:

101111110001001101

\[ b d f a c e \]
Coding Optimization

- alphabet \{a_i\} with frequencies \( f(a_i) \)
- binary code words \( c(a_i) \) \ldots \( c(a_n) \) s.t. minimize \# bits to represent data
  \[
  B(c) = \sum_{i=1}^{n} f(a_i) \cdot |c(a_i)|
  \]
- can represent as trees
  let \( d(a_i) \) be the depth of leaf \( a_i \). Find binary tree \( T \) with \( n \) leaves labeled \( a_i \ldots a_n \) that minimizes
  \[
  B(T) = \sum_{i=1}^{n} f(a_i) \cdot d(a_i) \quad \text{subject to} \quad d(a_i) = 1 \forall a_i
  \]

**Huffman Code - Algorithm for constructing optimal prefix trees**

Given:
- \( C \) is an alphabet of \( n \) characters
  - Each character \( c \in C \) has \( c \cdot \text{freq} \) attribute

Note:
- Optimal tree has \( n \) leaves (one for each character)
  and \( n-1 \) internal nodes

[CLRS § 16.3]
Algorithm

\[ n = |C| \]

\[ Q = C \leftarrow \text{place characters in min-priority queue by frequency attribute} \]

for \( i = 1 \) to \( n-1 \)

allocate new node \( z \)

\[ z \text{.left} = x = \text{Extract-Min}(Q) \]

\[ z \text{.right} = y = \text{Extract-Min}(Q) \]

\[ z \text{.freq} = x \text{.freq} + y \text{.freq} \]

\( \text{Insert}(Q, z) \)

return \( \text{Extract-Min}(Q) \) // return the root of the tree

Example

\[ \begin{array}{ccccccc}
    f: 5 & e: 9 & c: 12 & b: 13 & d: 16 & a: 45 \\
    \\
    c: 12 & b: 13 & 14 & d: 16 & a: 45 \\
    \\
    14 & d: 16 & 25 & a: 45 \\
    \\
    f: 5 & e: 9 & 25 \\
    \\
    25 & 30 & 14 & d: 16 \\
    \\
    c: 12 & b: 13 & f: 5 & e: 9 \\
\end{array} \]
Running Time:
Initialize \( Q \) with \( n \) elements
Loop \( (n-1) \) times with 2 Extract-Min's & 1 INSERT
1 Final Extract-Min

Implementation:
Binary Min-Heap \( \Rightarrow \) Initialize \( O(n) \); Loop \( O(n \log n) \)
\( \text{Final} \Rightarrow O(n \log \log n) \)

Van Emde Boas \( \Rightarrow O(n \log \log n) \)

Correctness - Note this is a greedy algorithm. At each step, greedy choice is to make tree from two lowest frequency characters at equal depth at lowest level of tree.
Lemma 16.2: \( x, y \) are 2 most infrequent characters of \( C \). Then exists optimal prefix code for \( C \) with code words for \( x \) and \( y \) of same length & differing only in the last bit.

\[
\begin{align*}
T_{\text{optimal}} & \quad \rightarrow \quad T' \\
& \quad \rightarrow \quad T''
\end{align*}
\]

\[
\begin{align*}
x \cdot \text{freq} & \leq a \cdot \text{freq} \\
\text{so } T' \text{ is also optimal}
\end{align*}
\]

\[
\begin{align*}
y \cdot \text{freq} & \leq b \cdot \text{freq} \\
\text{so } T'' \text{ is also optimal}
\end{align*}
\]

Lemma 16.3: \( C \) is alphabet with least frequent elements \( x, y \)

\[
\begin{align*}
C' = C - x \cdot y \cdot z + x \cdot z \cdot \bar{z} + y \cdot z \cdot \bar{z} & \quad \text{& } z \cdot \text{freq} = x \cdot \text{freq} + y \cdot \text{freq}
\end{align*}
\]

Then

\[
T' \text{ represents optimal code for } C'
\]

\[
T \text{ is an optimal code for } C \text{ with } T' \text{ being } T' \text{ with } \bar{z} \text{ leaf \ replaced by internal node having } x \text{ & } y \text{ as children.}
\]
If $T$ not optimal, then consider $\overline{T'}$ that is optimal and has $x,y$ as siblings by 16.2

\[ B(\overline{T'}) = B(T') + x \cdot \text{freq} + y \cdot \text{freq} \]
\[ B(T') = B(T) - x \cdot \text{freq} - y \cdot \text{freq} \]

Consider $\overline{T'}$, in which $x,y$ replaced by parent $z$ as leaf

\[ B(\overline{T'}) = B(\overline{T}) - x \cdot \text{freq} - y \cdot \text{freq} \]
\[ < B(T) - x \cdot \text{freq} - y \cdot \text{freq} \]
\[ = B(T') \leftarrow \text{contradiction to} \]
\[ T' \text{ being optimal} \]

so $T$ must represent the optimal prefix code for $C$. 