582487 Data Compression Techniques

Lectures, Spring 2012, period III

Juha Kärkkäinen

0. Introduction

Data compression deals with encoding information in as few bits as is possible or reasonable. The purpose is (usually) to reduce resource requirements:

- Permanent storage: hard disk, optical disk, flash memory, etc.
- Transmission: bandwitdh and/or time.
- Processing: main memory, cache memory.

Data compression can reduce the cost of dealing with data or enable things that would not otherwise be possible with available resources.

In addition, data compression has connections to machine learning and data mining, for example, in MDL (Minimum Description Length) principle.

Compressed data is often useful only for storage and transmission and needs to be decompressed before doing anything else with it.

- Compression (encoding) is the process of transforming the original (uncompressed) data into compressed data.
- Decompression (decoding) is the reverse process of recovering the original data from the compressed data.

Compression and decompression are often performed by different parties and one must be aware of what information apart from the compressed data is available to both parties. It is often helpful to think compression as communication with the compressed data as the message to be sent.

A recent trend are *compressed data structures* that support certain operations on the data without decompression.

- Typically, they need more space than fully compressed data but much less space than a corresponding uncompressed data structure.
- The operations are usually slower than on the uncompressed data structure.

There are two broad types of compression:

- Lossy compression throws away some information. The data produced by decompression is similar but not an exact copy of the original data.
  - Very effective with image, video and audio compression.
  - Removes unessential information such as noise and undetectable details.
  - Throwing away more information improves compression but reduces the quality of the data.
- Lossless compression keeps all information. The data produced by decompression is an exact copy of the original data.
  - Removes redundant data, i.e., data that can be reconstructed from the data that is kept.
  - Statistical techniques: minimize the average amount of data.

Bit is the basic unit of information or size of data or computer storage. International Electrotechnical Commission (IEC) recommends:

- Bit is abbreviated bit. The abbreviation b is best avoided to prevent confusion with B, which stands for byte (8 bits).

- The prefixes kilo (k), mega (M), giga (G), etc. refer to powers of 10. The corresponding binary prefixes referring to powers of two are called kibi (Ki), mebi (Mi), gibi (Gi), etc.. (Note that the abbreviation for kibi is capitalized but the one for kilo is not.)

For example, 8 kbit means 8000 bits and 1 KiB means 8192 bits.

The level of compression is usually expressed in one the following ways.

- Compression ratio is c/u, where c is the size of the compressed data and u is the size of the original, uncompressed data.

- Bit rate is the average number of bits per some basic unit of data, such as bits per character for text or bits per second for audio.

For example, a bit rate of 3 bits/character on standard ASCII text means compression ratio $3/8 = 37.5$ %.

About this course

- This course covers only lossless compression techniques. Lossy compression techniques are quite specific to the type of data and often rely on a deep understanding of human senses.

- The focus is on text compression. However, many of the basic techniques and principles are general and applicable to other types of data.

- The course concentrates on practical algorithms and techniques. Some key concepts of information theory, the mathematical basis of data compression, will be introduced but not covered in detail.

1. Variable-length encoding

Definition 1.1: Let Σ be the *source alphabet* and Γ the *code alphabet*. A *code* is defined by an injective mapping

$$C : \Sigma \to \Gamma^*$$

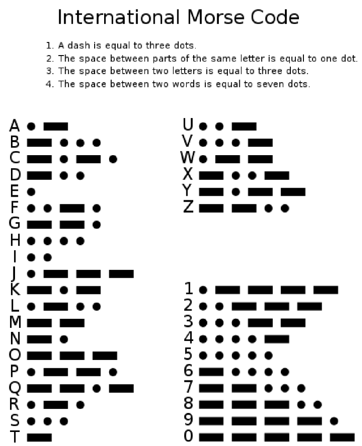
The code is extended to sequences over Σ by concatenation:

$$C : \Sigma^* \to \Gamma^* : s_1 s_2 \ldots s_n \mapsto C(s_1)C(s_2)\ldots C(s_n)$$

- We will usually assume that Γ is the binary alphabet $\{0, 1\}$. Then the code is called a *binary code*.

- Σ is an arbitrary set, possibly even an infinite set such as the set of natural numbers.

Example 1.2: Morse code is a type of variable length code.



Let us construct a binary code based on Morse code.

First attempt: Use zero for dot and one for dash.

$$\texttt{A} \mapsto 01 \qquad \texttt{B} \mapsto 1000 \qquad \texttt{C} \mapsto 1010 \qquad \texttt{D} \mapsto 100 \qquad \ldots$$

However, when applied to sequences, we have a problem:

$$\texttt{CAT} \mapsto 1010011 \qquad \texttt{KIM} \mapsto 1010011 \qquad \texttt{TETEETT} \mapsto 1010011 \qquad \ldots$$

The problem is that the code has no representation for the longer spaces between letters.

- The code would work if we had some kind of a *delimiter* to separate letters from each other, but the codes we are interested in here do not support such delimiters. They are sometimes called *self-delimiting codes*.
- If all the letter codes had the same length, there would be no need for a delimiter. The ASCII code is an example of such a *fixed-length code*. However, here we are more interested in *variable-length codes*.

The kind of code we need can be formalized as follows.

Definition 1.3: A code $C : \Sigma \to \Gamma^*$ is *uniquely decodable* if for all $S, T \in \Sigma^*$, $S \neq T$ implies $C(S) \neq C(T)$.

Determining whether a code is uniquely decodable is not always easy. Also, unique decodability does not necessarily mean that decoding is easy. However, there are easily recognizable and decodable subclasses:

- A fixed-length code that maps every source symbol to a code sequence of the same length is always uniquely decodable.
- A much larger subclass are prefix codes.

Definition 1.4: A code $C : \Sigma \to \Gamma^*$ is a *prefix code* if for all $a, b \in \Sigma$, $a \neq b$ implies that $C(a)$ is not a prefix of $C(b)$.

Theorem 1.5: Every prefix code is uniquely decodable.

Proof. Exercise.

Let us return to the Morse code example and define a binary prefix code based on the Morse code:

- Encode dots and dashes with ones and the spaces with zeros.
- The number of zeros and ones represents the length in time: dot is a single one, dash is three ones, etc..
- The space between letters is included in the code for the preceding letter.

$$\texttt{A} \mapsto 10111000 \quad \texttt{B} \mapsto 111010101000 \quad \texttt{C} \mapsto 11101011101000 \quad \ldots$$

$$\texttt{CAB} \mapsto 11101011101000101110001110101010000$$

In any encoded sequence, three consecutive zeros always marks the end of a symbol code, acting as a kind of a delimiter and ensuring that the code is a prefix code.

The code on the previous slide is not ideal for compression because the encoded sequences tend to be quite long.

A straightforward optimization is to use two ones for a dash and two zeros for a long space.

$$\texttt{A} \mapsto 101100 \quad \texttt{B} \mapsto 1101010100 \quad \texttt{C} \mapsto 11010110100 \quad \ldots$$

$$\texttt{CAB} \mapsto 110101101001011001101010100$$

The unnecessarily long codes are a form of *redundancy*.

- Removing redundancy is one way to achieve compression.
- In the original Morse code, the length of dashes and long spaces is a compromise between compression (for faster transmission) and avoiding errors.
- Some amount of redundancy is necessary if we want to be able to detect and correct errors. However, such error-correcting codes are not covered on this cause.

Huffman coding

Let P be a *probability distribution* over the source alphabet Σ. We want to find a binary prefix code C for Σ that minimizes the *average code length*

$$\sum_{a \in \Sigma} P(a)|C(a)|$$

Such a code is called *optimal*.

The probability distribution P may come from

- a probabilistic model of the data
- a statistical study of the type of data (for example, letter frequencies in English)
- actual symbol frequencies in the data to be encoded.

If the probability distribution is known to the decoder, the decoder can construct the code the same way the encoder does. Otherwise, the code (or the distribution) must be included as part of the compressed data.

We assume that all probabilities are positive. A symbol with a zero probability should be removed from the alphabet.

Intuitively, to minimize the average code length, we should use shorter codes for more frequent symbols and longer codes for rare symbols.

- Morse code has this property. The shortest codes are assigned to the most frequent letters in English.

This principle is easy show formally.

- Let $a, b \in \Sigma$ be two symbols that violate the principle, i.e., satisfy $P(a) < P(b)$ and $|C(a)| < |C(b)|$.
- We can correct the violation by swapping the codes of a and b. The resulting the change in the average code length is

$$\begin{aligned} & P(a)(|C(b)| - |C(a)|) + P(b)(|C(a)| - |C(b)|) \\ & = -(P(b) - P(a))(|C(b)| - |C(a)|) < 0 \end{aligned}$$

- Thus we can reduce the average code length for any code that violates the principle.

An optimal code can be computed with the Huffman algorithm.

Algorithm 1.6: Huffman
Input: Probability distribution P over $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$
Output: Optimal binary prefix code C for Σ

for $s \in \Sigma$ *do* $C(s) \leftarrow \epsilon$
$\mathcal{T} \leftarrow \{\{s_1\}, \{s_1\}, \ldots, \{s_\sigma\}\}$
while $|\mathcal{T}| > 1$ *do*
    Let $A \in \mathcal{T}$ be the set that minimizes $P(A) = \sum_{s \in A} P(s)$
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{A\}$
    for $s \in A$ *do* $C(s) \leftarrow 1C(s)$
    Let $B \in \mathcal{T}$ be the set that minimizes $P(B) = \sum_{s \in B} P(s)$
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{B\}$
    for $s \in B$ *do* $C(s) \leftarrow 0C(s)$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{A \cup B\}$
return C

Example 1.7: Huffman algorithm

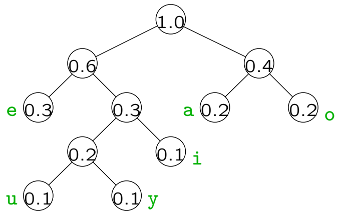
$P(\{\mathtt{e}\}) = 0.3$	$P(\{\mathtt{e}\}) = 0.3$	$P(\{\mathtt{e}\}) = 0.3$	$P(\{\mathtt{a},\mathtt{o}\}) = 0.4$
$P(\{\mathtt{a}\}) = 0.2$	$P(\{\mathtt{a}\}) = 0.2$	$P(\{\mathtt{i},\mathtt{u},\mathtt{y}\}) = 0.3$	$P(\{\mathtt{e}\}) = 0.3$
$P(\{\mathtt{o}\}) = 0.2$	$P(\{\mathtt{o}\}) = 0.2$	$P(\{\mathtt{a}\}) = 0.2$	$P(\{\mathtt{i},\mathtt{u},\mathtt{y}\}) = 0.3$
$P(\{\mathtt{i}\}) = 0.1$	$P(\{\mathtt{u},\mathtt{y}\}) = 0.2$	$P(\{\mathtt{o}\}) = 0.2$	
$P(\{\mathtt{u}\}) = 0.1$	$P(\{\mathtt{i}\}) = 0.1$		
$P(\{\mathtt{y}\}) = 0.1$			

$P(\{\mathtt{e},\mathtt{i},\mathtt{u},\mathtt{y}\}) = 0.6$ $P(\{\mathtt{e},\mathtt{a},\mathtt{o},\mathtt{i},\mathtt{u},\mathtt{y}\}) = 1.0$
$P(\{\mathtt{a},\mathtt{o}\}) = 0.4$

Huffman code

$C(\mathtt{a}) = 10$
$C(\mathtt{e}) = 00$
$C(\mathtt{i}) = 011$
$C(\mathtt{o}) = 11$
$C(\mathtt{u}) = 0100$
$C(\mathtt{y}) = 0101$

Huffman tree



- The collection of sets $\mathcal{T}$ can be implement using a priority queue so that the minimum elements are found quickly. The total number of operations on $\mathcal{T}$ is $\mathcal{O}(\sigma)$ and each operation needs $\mathcal{O}(\log \sigma)$ time. Thus the time complexity is $\mathcal{O}(\sigma \log \sigma)$.

- The resulting code is often called *Huffman code*. The time complexity of constructing the code (lines (6) and (9)) is linear in the total length of the codes.

- The code can be represented as a trie of $\mathcal{O}(\sigma)$ nodes known as the *Huffman tree*. When constructing the tree, lines (6) and (9) can be replaced with a constant time operation that creates a new node and adds two trees as its children. Thus the trie representation can be constructed $\mathcal{O}(\sigma)$ time.

Let us prove the optimality of the Huffman code.

Lemma 1.8: Let $\Sigma = \{s_1, s_2, \ldots, s_{\sigma-1}, s_\sigma\}$ be an alphabet with a probability distribution P satisfying $P(s_1) \geq P(s_2) \geq \cdots \geq P(s_{\sigma-1}) \geq P(s_\sigma) > 0$. Then there exists an optimal binary prefix code C satisfying

- No code $C(s_i)$ is longer than $C(s_\sigma)$.
- $C(s_{\sigma-1})$ is the same as $C(s_\sigma)$ except that the last bit is different.

Proof. Let C be an optimal binary prefix code that does not satisfy the conditions. We show that it can be modified to satisfy the conditions.

- If some code $C(s_i)$ is longer than $C(s_\sigma)$, we can swap the codes $C(s_i)$ and $C(s_\sigma)$ without increasing the average code length.
- W.l.o.g., assume that $C(s_\sigma) = B1$ for some bit string B. If there were no symbol with a code $B0$, then we could set $C(s_\sigma) = B$ without violating the prefix condition. This would reduce the average code length, which contradicts the assumption that C is optimal.
- Thus $C(s_j) = B0$ for some $s_j \neq s_\sigma$. If $s_j \neq s_{\sigma-1}$, we can swap the codes $C(s_j)$ and $C(s_{\sigma-1})$ without increasing the average code length.

□

Lemma 1.9: Let $\Sigma = \{s_1, s_2, \ldots, s_{\sigma-1}, s_\sigma\}$ be an alphabet with a probability distribution P satisfying $P(s_1) \geq P(s_2) \geq \cdots \geq P(s_{\sigma-1}) \geq P(s_\sigma) > 0$.
Let $\Sigma' = \{s_1, s_2, \ldots, s_{\sigma-2}, s'\}$ with a probability distribution P' such that $P'(s') = P(s_{\sigma-1}) + P(s_\sigma)$ and $P'(s) = P(s)$ for other s.
Let C' be an optimal binary prefix code for Σ'. Let C be a code for Σ such that $C(s_{\sigma-1}) = C'(s')0$, $C(s_\sigma) = C'(s')1$, and $C(s) = C'(s)$ for other s.
Then C is an optimal binary prefix code for Σ.

Proof. C is clearly a binary prefix code. Suppose C is not optimal, i.e., there exists a code D with a smaller average code length.

- By Lemma 1.8, we can assume that $D(s_{\sigma-1}) = B0$, $D(s_\sigma) = B1$ for some bit string B. Thus we can construct a prefix code D' for Σ' by setting $D'(s') = B$ and $D'(s) = D(s)$ for other s.
- Let L, L', K and K' be the average code lengths of the codes C, C', D and D', respectively. Then
$$L - L' = P'(s') = K - K'$$
and thus
$$K' = L' - (L - K) < L'$$
which contradicts C' being optimal.

□

Theorem 1.10: The Huffman algorithm computes an optimal binary prefix code.

Proof. Each iteration of the main loop of the Huffman algorithm performs an operation that is essentially identical to the operation in Lemma 1.9: remove the two symbols/sets with the lowest probability and add a new symbol/set representing the combination of the removed symbols/sets. Lemma 1.9 shows that each iteration computes the optimal code provided that the following iterations compute the optimal code. Since the code in the last iteration is trivially optimal, the algorithm computes the optimal code.

□