Some algorithms for data compression

Anselm Blumer

12 March, 2020

ablumer@cs.tufts.edu

A brief introduction to Shannon information theory

- In 1948, Claude Shannon published "A mathematical theory of communication" in the Bell System Technical Journal

- Important measures of information and communication:
  — Entropy
  — Relative entropy
  — Mutual information
  — Channel capacity

-

-

A brief introduction to Shannon information theory

- In 1948, Claude Shannon published “A mathematical theory of communication” in the Bell System Technical Journal

- Important measures of information and communication:
  — Entropy
  — Relative entropy
  — Mutual information
  — Channel capacity

- Important concepts
  — Data source
  — Communication channel
  — Redundancy

-

A brief introduction to Shannon information theory

- In 1948, Claude Shannon published "A mathematical theory of communication" in the Bell System Technical Journal

- Important measures of information and communication:
  — Entropy
  — Relative entropy
  — Mutual information
  — Channel capacity

- Important concepts
  — Data source
  — Communication channel
  — Redundancy

- Important techniques:
  — Data compression (or source coding)
  — Error detection and correction (or channel coding)

Types of data compression

- Data compression is used for communication
  - When communicating from one location to another, compression can be used to speed up the communication
  -
  -
-
-
-

Types of data compression

- Data compression is used for communication
  - When communicating from one location to another, compression can be used to speed up the communication
  - Storage is communication across time - data compression can be used to make storage more efficient
  -
-
-
-

Types of data compression

- Data compression is used for communication

  — When communicating from one location to another, compression can be used to speed up the communication

  — Storage is communication across time - data compression can be used to make storage more efficient

  — In both cases, data compression can be combined with error detection and correction to make communication more reliable

-
-
- |

Types of data compression

- Data compression is used for communication

  — When communicating from one location to another, compression can be used to speed up the communication

  — Storage is communication across time - data compression can be used to make storage more efficient

  — In both cases, data compression can be combined with error detection and correction to make communication more reliable

- There are two major types of compression:

- Lossless compression, where compression followed by expansion gives back an exact copy of the original
  — Examples include ZIP, GIF, TIFF, MPEG-4 SLS, FLAC, JPEG-LS

Types of data compression

- Data compression is used for communication

  — When communicating from one location to another, compression can be used to speed up the communication

  — Storage is communication across time - data compression can be used to make storage more efficient

  — In both cases, data compression can be combined with error detection and correction to make communication more reliable

- There are two major types of compression:

- Lossless compression, where compression followed by expansion gives back an exact copy of the original
  — Examples include ZIP, GIF, TIFF, MPEG-4 SLS, FLAC, JPEG-LS

- Lossy compression, where better compression is obtained by allowing the decompressed version to differ somewhat from the original
  — Examples include JPEG, MPEG, MP3, AAC, Ogg Vorbis

Text compression

- Text compression is usually lossless
- Text compression methods can take advantage of two types of inefficiency in data representation
- Characters may occur with differing frequencies, so encoding every character using the same number of bits is less efficient than using short encodings for frequent characters and long encodings for infrequent characters
- Characters may not appear independently of each other, so it may be more efficient to have short representations of pairs, triples, or longer sequences of characters

An example of text compression

- An example of the first case: Suppose we would like to store a 1000-character file where the characters are coming from a four letter alphabet, {A, B, C, D} and A occurs 500 times, B occurs 250 times, and C and D each occur 125 times.

- We could represent each character using two bits, for example:

  A ↔ 00 B ↔ 01 C ↔ 10 D ↔ 11

  Storing the file would take 2000 bits

-

An example of text compression

- An example of the first case: Suppose we would like to store a 1000-character file where the characters are coming from a four letter alphabet, {A, B, C, D} and A occurs 500 times, B occurs 250 times, and C and D each occur 125 times.

- We could represent each character using two bits, for example:

  A ↔ 00   B ↔ 01   C ↔ 10   D ↔ 11

  Storing the file would take 2000 bits

- Or we could represent A by one bit, B by two bits, and C and D by three bits each For example

  A ↔ 1   B ↔ 01   C ↔ 001   D ↔ 000

  Now storing the file only takes
  (500 x 1) + (250 x 2) + (125 x 3) + (125 x 3) =
  500 + 500 + 375 + 375 = 1750 bits

An example of text compression

- Summarizing the example from the previous slide:
  — Expressing the frequencies of the letters as probabilities, we have
  P(A) = 0.5   P(B) = 0.25   P(C) = 0.125   P(D) = 0.125
  — To encode, use a lookup table

A	B	C	D
1	01	001	000

- How do we know that we can always get back the original file?

-

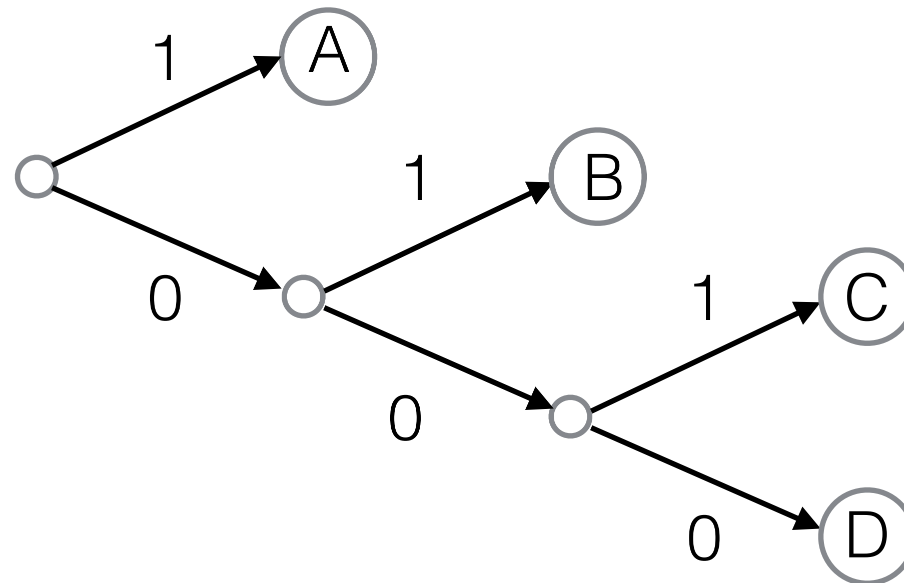
An example of text compression

- Summarizing the example from the previous slide:
  — Expressing the frequencies of the letters as probabilities, we have
  P(A) = 0.5   P(B) = 0.25   P(C) = 0.125   P(D) = 0.125
  — To encode, use a lookup table

A	B	C	D
1	01	001	000

- How do we know that we can always get back the original file?

- Decode using a tree:



An example of text compression

- Summarizing the example from the previous slide:
  — Expressing the frequencies of the letters as probabilities, we have
  $P(A) = 0.5 \quad P(B) = 0.25 \quad P(C) = 0.125 \quad P(D) = 0.125$
  — To encode, use a lookup table

A	B	C	D
1	01	001	000

- How efficient is this? Average number of bits used per letter =

$$(0.5 \times 1) + (0.25 \times 2) + (0.125 \times 3) + (0.125 \times 3) = 1.75$$

- Can we do better? No, according to Shannon, for any uniquely decodable code the entropy gives a lower bound on the average number of bits used per letter

- Under the assumption that the letters are independent, the entropy is

$$\sum P(X) \log (1/P(X)) =$$
$$(0.5 \times \log 2) + (0.25 \times \log 4) + (0.125 \times \log 8) + (0.125 \times \log 8) =$$
$$0.5 + 0.5 + 0.375 + 0.375 = 1.75$$

Generalizing this example to larger alphabets

- To compress a text over a larger alphabet, assuming the characters are independent:
- Determine the frequency of each character
- Use Huffman’s algorithm (see CLRS) to generate an optimal code and decoding tree
- Use table lookup for encoding and the Huffman tree for decoding
- Note that the decoder needs to know the tree in order to start decoding

Text compression when the characters are not independent

- In most cases, the characters in a text are far from independent
- One solution: generate a Huffman tree for pairs of letters (bigrams), triple (trigrams), …, or n-grams
- Problem: the size of the tree grows exponentially in n
- Another solution: scan the text and generate a dictionary of substrings that occur frequently, parse the text into these substrings, and transmit efficient encodings of these substrings

Two well-known dictionary-based compression methods

- In 1977 and 1978, Jacob Ziv and Abraham Lempel published a pair of papers that introduced the best-known dictionary-based text compression methods, LZ77 and LZ78

- LZ77 and LZ78 have each spawned many variants

- LZ77 uses a suffix of the most recently transmitted string (for example, the most recent 1024 characters) as the dictionary, so any substring of this suffix can be transmitted efficiently

- LZ78 builds a tree of recently seen substrings

- In both cases there needs to be a way to transmit characters that have not been seen before

The LZ77 approach

- LZ77 divides the sequence into the *search buffer*, a suffix of the already encoded sequence, and the *lookahead buffer*, a prefix of the next part of the sequence
- The dictionary consists of all substrings of the search buffer plus all single characters
- In order to encode a single character that doesn't occur in the search buffer, prefix it with a flag bit, say 0
- Substrings of the search buffer will be prefixed with a 1 bit, followed by an index into the search buffer and a length
- This approach is due to Storer and Szymanski (LZSS)

An LZSS example

- As a simple example, suppose the alphabet is {A, B, C, D} and the length of the search buffer is 8. In realistic implementations, these values would be much larger.
  Suppose we use the two bit encodings of characters:
  A ↔ 00   B ↔ 01   C ↔ 10   D ↔ 11

- To encode ABCABDBCA
  — The first three characters are all new, so they are encoded (000)(001)(010)
  — Now searchbuf = ABC, lookahead = ABDBCA
  — The longest prefix of the lookahead buffer that matches a substring of the search buffer is AB, so encode AB as (1,2) = (1001010)
  — Now searchbuf = ABCAB, lookahead = DBCA
  — D isn't in the search buffer, so encode it as (011)
  — Now searchbuf = ABCABD, lookahead = BCA
  — BCA is in the search buffer, so encode it as (2,3) = (1010011)

- Putting this together, ABCABDBCA ↔ 00000101010010100111010011

A special case for LZSS

- In some cases, the encoded string can extend beyond the boundary of the search buffer

- Suppose the string to be encoded is ABABABC

- The first two characters are encoded as A->(000), B->(001) as before

- The next AB could be encoded as (1,2)->(1001010)

- In fact, we can encode ABAB as (1,4)->(1001100) even though the length of the search buffer is only 2. Once the AB in the search buffer has been copied, the new copy can itself be copied.

- This generalizes run-length coding for repeated patterns as well as repeated characters.

The LZ78 approach

- Two drawbacks of LZ77:
  — encoding a short substring with two integers can be inefficient
  — fast search of every every substring of the search buffer requires a sophisticated algorithm
- LZ78 uses a tree (or a *trie*) instead of LZ77's search buffer to store the dictionary
- Matching the lookahead buffer to the tree gives the longest prefix that is already in the tree, so this prefix can be encoded as a pointer into the tree
- The original LZ78 algorithm also encoded the next character (the first character that didn't match) explicitly
- Terry Welch pointed out that if the tree is initialized to contain all letters in the alphabet then the next character doesn't need to be encoded explicitly since it is the first character of the next encoded string
- This approach, known as LZW, is the most popular variant of LZ78

An LZW example

- Again suppose the alphabet is {A, B, C, D} and that these occupy positions 1-4 in the dictionary tree

- To encode ABCABDBCABCABCB
  — The first three letters are encoded as 1, 2, 3 and entries 5↔AB, 6↔BC, 7↔CA are added to the dictionary tree
  — Since AB is now in the dictionary, it can be encoded as 5 and 8↔ABD is added
  — The single letter D is encoded as 4 and 9↔DB is added
  — BC, AB, CA, and BC are encoded as 6, 5, 7, 6 and 10↔BCA, 11↔ABC, 12↔CAB are added
  — The final B is encoded as 2, allowing 13↔BCB to be added to the dictionary tree

An LZW example

- In one special case the decoder needs to use an entry that isn't yet in the tree
- For example, when encoding ABABABA the encoder starts with 1,2 for A,B and adds 5↔AB, 6↔BA to the dictionary
- The second AB is then encoded as 5 and 7↔ABA is added to the dictionary
- The third ABA can now be encoded as 7
- Decoding is straightforward until reaching the 7. The decoder doesn't have a complete entry for 7, but knows that it must begin with AB. When entry 5 was added to the dictionary, the next letter was A, so entry 7 must be ABA

References

- Khalid Sayood, Introduction to Data Compression, 4th ed. (2012) Morgan Kaufman

- David Salomon, Data Compression: The Complete Reference, 3rd ed. (2004) Springer

- Thomas M. Cover and Joy A. Thomas, Elements of Information Theory, 2nd. ed. (2006) Wiley

- Cormen, Leiserson, Rivest, and Stein, Introduction to Algorithms, 3rd ed. (2009) MIT Press

- Claude Elwood Shannon, Collected Papers, edited by Sloane and Wyner (1993) IEEE Press