Compression: Other Lossless Compression Algorithms

Greg Plaxton
Theory in Programming Practice, Spring 2005
Department of Computer Science
University of Texas at Austin

LZ78 (Lempel-Ziv)

- The encoder and decoder each maintain a "dictionary" containing certain words seen previously
  - Initially the dictionary contains only the empty string (in practice it is often initialized to the set of single-symbol words)
  - The algorithm maintains the invariant that the encoder and decoder dictionaries are the same (except the decoder dictionary can lag behind by a word)
  - The encoder communicates a dictionary entry to the decoder by sending an integer index into the dictionary
  - If the dictionary becomes full, a common strategy is to evict the LRU entry

LZ78: Outline of a Single Iteration

- Suppose the encoder has consumed some prefix of the input sequence
- The encoder now considers successively longer prefixes of the remaining input until it finds the first prefix αx such that α is a word in the dictionary and αx is not a word in the dictionary
- The word αx is added to the dictionary of the encoder
- The word αx is communicated to the decoder by transmitting the index i of α and the symbol x
- The decoder uses its dictionary to map i to α, and then adds the word αx to its dictionary

LZ78: Dictionary Data Structure

- It is common to implement the dictionary as a trie
  - If the set of symbols is, e.g., the 256 possible bytes, then each node of the trie might have an array of length 256 to store its children
  - While fast (linear time), this implementation is somewhat inefficient in terms of space
  - A trick that can achieve a good space-time tradeoff is to store the children of a trie node in a linked list until the number of children is sufficiently large (say 10 or so), and then switch to an array
  - Alternatively, the children of a trie node could be stored in a hash table
- The integers used to represent dictionary entries are indices into an array of pointers into the trie

LZ Algorithms

- Quite a few variations of LZ77 and LZ78 have been proposed
- The LZ algorithms are popular because they run in a single pass, provide good compression, are easy to code, and run quickly
- Used in popular compression utilities such as compress, gzip, and WinZip

Arithmetic Coding

- Assume an i.i.d. source with alphabet A and where the ith symbol in A has associated probability p_i, $1 \leq i \leq n = |A|$
- Map each input string to a subinterval of the real interval $[0, 1]$
  - Chop up the unit interval based on the first symbol of the string, with the ith symbol assigned to the subinterval

$$[\sum_{1 \leq j < i} p_j, \sum_{1 \leq j \leq i} p_j]$$

  - Recursively construct the mapping within each subinterval to handle strings of length 2, then 3, et cetera
- The encoder specifies the real interval corresponding to the next fixed-size block of symbols to be sent

Arithmetic Coding: Specifying a Particular Interval

- To specify an interval, the encoder sends a (variable length) bit string that is itself interpreted as a subinterval of $[0, 1]$
  - For example, 010 is interpreted as the interval containing all reals with binary expansion of the form $.010***\ldots$ where the $*$'s represent don't cares (0 or 1)
  - Thus 010 corresponds to $[1/4, 3/8)$, 0 corresponds to $[0, 1/2)$, 11 corresponds to $[3/4, 1)$, et cetera
- Once the decoder has received a bit string that is entirely contained within an interal corresponding to a particular block, it outputs that block and proceeds to the next iteration

Arithmetic Coding: An Example

- Consider $A = \{a, b\}$ where the probability associated with a is close to 1, e.g., 0.99
  - The entropy per symbol is close to zero, so a direct application of Huffman coding performs poorly
  - Even with a block size of 50, arithmetic coding communicates the all-a's block using only a single bit since $0.99^{50} > 1/2$

Run-Length Coding

- Another technique that is useful for dealing with certain low-entropy sources
- The basic idea is to encode a run of length k the same symbol a as the pair (a, k)
- The resulting sequence of pairs are then typically coded using some other technique, e.g., Huffman coding
- Example: FAX protocols
  - Run-length coding converts document to alternating runs of white and black pixels
  - Run lengths are encoded using a fixed Huffman code that works well on typical documents
  - A long run such as 500 might be coded by passing Huffman codes for $128+$, $128+$, $128+$, $64+$, 52

Move-To-Front Coding

- A good technique for dealing with sources where the output favors certain symbols for a while, then favors another set of symbols, et cetera
- Keep the symbols in a list
- When a symbol is transmitted, move it to the head of the list
- Transmit a symbol by indicating its current position (index) in the list
- The hope is that we will mostly be sending small indices

Move-To-Front Coding: Compressing the Index Sequence

- The sequence of indices can be compressed using another method such as Huffman coding
- An easy alternative (though perhaps unlikely to give the best performance) is to encode each k-bit index using $2k - 1$ bits as follows
  - Assume the lowest index is 1; thus $k > 0$
  - Send $(k - 1)$ 0's followed by the k-bit index
  - The decoder counts the leading zeros to determine k, then decodes the k-bit index

Prediction by Partial Matching

- This is essentially the approach that Shannon used in his experiments with English text discussed in an earlier lecture
- The idea is to maintain, for each string α of some fixed length k, the conditional probability distribution for the symbol that follows the string α
- The encoder specifies the next symbol using some appropriate code, e.g., a Huffman code for the given probability distribution
- Shannon showed that for a wide class of discrete Markov sources, the performance of this technique approaches the entropy lower bound for k sufficiently large
  - But in practice we cannot afford to use a value of k that is very large since the number of separate probability distributions to maintain is $|A|^k$

Burrows-Wheeler Transform

- A relatively recent (1994) technique
- A number of compression algorithms have been proposed that make use of the Burrows-Wheeler transform in combination with other techniques such as arithmetic coding, run-length coding, and move-to-front coding
- The bzip utility is such an algorithm
  - Outperforms gzip and other LZ-based algorithms

Burrows-Wheeler Transform: Abstract View

- Take the next block of symbols to be encoded
- Construct n strings corresponding to all rotations of the block, numbering then from 0 (say)
- Sort the resulting n strings
- Given this sorted list of strings, transmit the index of the first string and the sequence of last symbols
- Symbols with a similar context in the original string are now grouped together, so this sequence can be compressed using other methods
- A nontrivial insight is that the information transmitted is sufficient for decoding

Burrows-Wheeler Transform: Implementation

- The preceding high-level description seems to imply that quadratic space is needed
- In fact, each of the n rotations of the original string can be represented by a pointer into the original string
- A standard sorting utility can be used, but each comparison could be costly in the worst case (e.g., if all of the symbols in the block are the same)
- Better worst-case guarantees can be achieved using algorithms specifically designed for suffix sorting