N89-22347

NOISELESS COMPRESSION USING NON-MARKOV MODELS

Anselm Blumer
Computer Science Department
Tufts University

ABSTRACT

Adaptive data compression techniques can be viewed as consisting of a model specified by a database common to the encoder and decoder, an encoding rule and a rule for updating the model to ensure that the encoder and decoder always agree on the interpretation of the next transmission. The techniques which fit this framework range from run-length coding, to adaptive Huffman and arithmetic coding, to the string-matching techniques of Lempel and Ziv. The compression obtained by arithmetic coding is dependent on the generality of the source model. For many sources, an independent-letter model is clearly insufficient. Unfortunately, a straightforward implementation of a Markov model requires an amount of space exponential in the number of letters remembered. The Directed Acyclic Word Graph (DAWG) can be constructed in time and space proportional to the text encoded, and can be used to estimate the probabilities required for arithmetic coding based on an amount of memory which varies naturally depending on the encoded text. The tail of that portion of the text which has been encoded is the longest suffix that has occurred previously. The frequencies of letters following these previous occurrences can be used to estimate the probability distribution of the next letter. Experimental results indicate that compression is often far better than that obtained using independent-letter models, and sometimes also significantly better than other non-independent techniques.

INTRODUCTION

Adaptive data compression techniques are useful when the statistics of the data are not known in advance or are changing slowly. This paper surveys some well-known adaptive compression techniques and presents two new methods which give improved compression in many cases. All techniques presented are lossless (or noiseless) in that an exact copy of the original data can be obtained by decompressing the output of the compression algorithm. The data is assumed to be a string of characters from an arbitrary alphabet, though many of the ideas presented here could be applied to images or other multidimensional data.

An adaptive compression technique can be viewed as consisting of three parts:

1) A dictionary or database which defines the current state of the model.
2) A coding rule which determines how the encoder transmits the next part of the string and how the decoder interprets the encoder's message.
3) An adaptation rule which determines how the encoder and decoder update the model to reflect the previous transmission.

If the encoder and decoder initialize their models identically, they can maintain identical copies of the model throughout the transmission by using the same adaptation rule.

RUN-LENGTH CODING

The simplest compression technique which fits this framework is probably run-length coding. In this case, the dictionary consists of the single character which forms the current run. The coding rule encodes the number of times this character is repeated, followed by an encoding of the next character. For example, if the encoder's and decoder's dictionaries are initialized to "a", the data string "ccccabbb" could be transmitted as (0,c)(3,a)(0,b)(2,b). Since the initial character in the dictionary is "a", (0,c) is interpreted as no "a"s followed by at least one "c". The "c" then replaces the "a" in both dictionaries. (3,a) then indicates a run of 3 more "c"s followed by an "a". (0,b) indicates that this "a" is followed immediately by a "b", and (2,b) can be interpreted to mean that the end of the string has been reached after two more "b"s, since there would not ordinarily be two consecutive runs of the same character.

LEMPEL-ZIV CODING

A more interesting adaptive compression technique was developed by Lempel and Ziv while investigating a complexity measure for strings(4). In this case, the dictionary consists of that portion of the data string which has already been transmitted. The encoder is allowed to specify any substring of this string by transmitting an index and a length. The encoder parses off the longest prefix of the part of the data which has not yet been transmitted which matches a substring of the previously transmitted data. This is complicated by the fact that the encoder and decoder add letters to their dictionary strings as this matching is being done, allowing the matching substrings to overlap each other. The encoder then transmits the index of the previous substring, the length of the match, and the first character which caused the match to fail. If the data string is "abcabbcabbcb" and "abc" has already been transmitted the encoder's next transmission would be (1,2,b) indicating a match of 2 letters starting at position 1, followed by a "b". The encoder and decoder would then concatenate "abb" to their dictionaries, giving "abcabb". Using the overlap mentioned above, the encoder can now transmit the rest of the string as (3,5,b). If the 5 was replaced by a larger number, this would indicate further repeats of the pattern "cabb", so this technique can be viewed in part as a generalization of run-length coding, where runs are allowed to consist of repeated patterns rather than just repeated single characters.

More recently, Ziv and Lempel have developed another adaptive compression technique based on string matching(7). To avoid the complexity inherent in using both a pointer and a length to refer to any previous substring, the dictionaries for this newer method consist of only those substrings which have been parsed and transmitted by the encoder. The prefix parsed by the decoder must match one of these substrings. The decoder then transmits a code number for that substring and the next character. (Variations of this technique avoid the explicit transmission of the next character by methods such as initializing the dictionaries to contain all single-character strings(5). The adaptation rule then adds the most recently transmitted to the dictionaries. For example, if the first eight characters of "aabaaabaaabb" have been transmitted, the dictionaries consist of the strings "a", "ab", "aa", and "aba", numbered 1, 2, 3, and 4, respectively. The encoder will then parse "aa" and transmit "aab" as (3,b). "aab" is then added to the dictionaries as string 5. The final "b" is then transmitted as (0,b), since it does not match any dictionary string. This technique works quite well in practice, as is evidenced by popularity of the UNIX (TM) "compress" command. One slight drawback is that runs of repeated patterns can no longer be parsed all at once, even when the pattern is only the repeated occurrence of a single character.

A GENERALIZATION OF RUN-LENGTH CODING

Another way to avoid transmitting both a pointer and a length is to use an implicit pointer which can be computed by both the encoder and decoder(1). This can be done using a data structure known as the Directed Acyclic Word Graph (DAWG), a data structure which stores information about all substrings of a string in space proportional to the length of the string and can be built in time proportional to this length (linear space and time)(2,3). The linear-time construction algorithm always maintains a pointer to the longest suffix that has occurred elsewhere in the string. This suffix is known as the "tail". The previous occurrence of the tail can be used as the implicit pointer mentioned above. The encoder locates this previous occurrence (which is uniquely defined by the DAWG construction algorithm) and predicts that the next character to be transmitted will be the same as the character which follows this previous occurrence. If this prediction is correct, the DAWG is then updated by adding this character and another prediction is made based on the new tail. This process continues until a prediction fails, at which point the encoder transmits a count of the number of correct predictions and the actual character for the first prediction which failed. For example, if the data string is "aababaabaa" and the first five characters have already been transmitted, then the tail is "ab" and the prediction is that the sixth character is "a". This is correct, so the "a" is concatenated to the string and the DAWG is updated. The tail is now "aba", which predicts a "b". This is incorrect, so the encoder's transmission is (1,a). The next predictions are "b", "a", and "b", of which the first two are correct, so the next transmission is (2,a). With this technique any sufficiently long repeated pattern can be parsed in a single transmission, so it will act like a generalization of run-length coding in these cases.

ARITHMETIC CODING WITH DAWG-BASED MODELS

Although experimental results with this last technique were encouraging, it can be shown to be asymptotically nonoptimal for a wide variety of data sources. Fortunately, the information available from the DAWG can be used in combination with arithmetic coding(6), resulting in a technique which is both asymptotically optimal and gives good results in the cases where the above technique worked well. Arithmetic coding, like Huffman coding, is based on using estimates of the probabilities for the next character to encode that character. An adaptive arithmetic encoder will revise these estimates as each letter is transmitted. If the characters are statistically independent, a table of frequencies for each character will provide these estimates. In most cases, characters are strongly dependent, so a more sophisticated model is needed. The most natural next step is to use a Markov model of order m, where the probabilities of the next character depend on the previous m characters. There are two problems with this: the amount of space needed to store the probability estimates is exponential in m, and the ideal m may change as the characteristics of the data change. Both of these problems can be solved by using a DAWG to store the frequency counts and using the previous occurrence of the tail to provide the appropriate amount of context. Using just the frequency counts from the DAWG gave good compression in many cases, but worked very poorly when the data contained long runs. The reason for this is that in a long run there will be relatively few previous occurrences of the tail, so the next character is not predicted with great certainty. When the probabilities were modified to take into account the length of the tail, the performance improved greatly in these cases.

EXPERIMENTAL RESULTS

Filetype	Filesize	ARITH	LZW	GRL	DAWGARITH
Commands	347	253	200	184	142
C Program	1729	1160	840	603	515
Object Code	3890	2585	2096	2544	1759
Load Module	49152	29.47	23544	27774	18168
Font File	16384	906	478	384	263
This Report	15945	9528	7810	10982	8809

Numbers are number of bytes before and after compression.
ARITH is the adaptive arithmetic coding algorithm from[6].
LZW is the UNIX (TM) "compress" command[5].
GRL is the generalization of run-length coding from[1].
DAWGARITH is the last technique described above.

REFERENCES

1) Blumer, "A Generalization of Run-length Coding," Presented at the IEEE International Symposium on Information Theory, June 1985, Brighton, England.

2) Blumer, Blumer, Ehrenfeucht, Haussler, Chen and Seiferas, "The Smallest Automaton Recognizing the Subwords of a Text," Theoretical Computer Science, (40) 1985, pp. 31-55.

3) Blumer, Blumer, Haussler, McConnell and Ehrenfeucht, "Complete Inverted Files for Efficient Text Retrieval and Analysis," JACM, July 1987, pp. 578-595.

4) Lempel, Abraham and Jacob Ziv, "On The Complexity of Finite Sequences," IEEE Transactions on Information Theory, IT-22, no. 1, Jan. 1976, pp. 75-81.

5) Welch, T.A., "A Technique for High-Performance Data Compression," Computer, 17, no. 6, June 1984, pp. 8-19.

6) Witten, Ian H., Radford M. Neal, and John G. Cleary, "Arithmetic Coding for Data Compression," Communications of the ACM, 30, no. 6, June 1987, pp. 520-540.

7) Ziv, Jacob and Abraham Lempel, "Compression of Individual Sequences via Variable-rate Coding," IEEE Transactions on Information Theory, IT-24, no. 5, Sept. 1978, pp. 530-535.

APPENDIX: THE DAWG CONSTRUCTION ALGORITHM

The DAWG is a Partial Deterministic Finite Automaton (PDFA) which recognizes the set of all substrings of a word. If the word has length n, the DAWG will have less than 2n nodes and 3n edges. It can be built online in linear time using some auxiliary edges (one per node) called suffix pointers. Each node corresponds to the class of strings labeling the paths from the source to that node. The longest such path is called primary. The destination of the suffix pointer of a node is identified by removing the first letter from the label of the shortest path from the source to that node. Thus the suffix pointer of the sink node can be used to identify the longest suffix which occurs somewhere else in the string (the "tail").

The pseudo-C code below updates a DAWG for a word, w to a DAWG for wa. It assumes that wa is stored in a global buffer, and that "index" is the position of the letter a in this buffer. Each node contains:

position:	a buffer index pointing to the letter labeling edges to that node
depth :	the length of the primary path from the source to that node
edges :	a linked list of outgoing edges, and
suffix :	the suffix pointer for that node

"source" and "currentsink" are global variables. The following auxiliary procedures are needed:

allocnode(position, depth)	allocates and returns a pointer to a new node
allocedge(node, edgelist)	adds to "edgelist" an edge which points to "node"
findedge (node, letter)	returns a pointer to an outgoing edge from "node" labeled "letter", or NIL if no such edge exists

```
     unlist    ( edgelist, node ) removes an edge pointing to "node"
                  from "edgelist"

update( ch, index )
char ch;                /* ch is the character pointed to by index */
unsigned index;     /* index points into the text buffer       */ {
/* make a new node, "newsink", and make an edge to
/* this node from "currentsink", the old sink.  */
newsink = allocnode( index, index+1 );
edges( currentsink ) = allocedge( newsink, edges( currentsink ) );

suffixnode = source; /* the default value */

/* follow chain of suffix pointers from currentsink */
for (currentnode = suffix( currentsink ); currentnode ISNT NIL;
                  currentnode = suffix( currentnode ) ) {

   /*  no  edge  with  this  character,  so  make  a  secondary  edge  to
"newsink" */    if ((Edge = findedge( currentnode, ch )) IS NIL)
      edges( currentnode ) =
         allocedge( newsink, edges( currentnode ) );

   /* a secondary edge labelled "ch", so split */
   else if ((depth(currentnode)+1) ISNT depth(node(Edge))) {

      /* Make Edge into a primary edge to a new node */
      childnode = node ( Edge );
      newchildnode = allocnode( index, depth(currentnode)+1 );
      edges( currentnode ) = unlist( edges( currentnode ), childnode
);       edges( currentnode ) =
         allocedge(  newchildnode, edges( currentnode ) );

      /* Give a copy of each of childnode's edges to newchildnode */
    for (Edge = edges( childnode ); Edge ISNT NIL; Edge = next( Edge
))          edges( newchildnode ) =
            allocedge( node( Edge ), edges( newchildnode ) );
       /* Set the suffix pointer of newchildnode to that of childnode,
      /* and reset childnode's to point to newchildnode     */
       suffix( newchildnode ) = suffix( childnode );
       suffix( childnode ) = newchildnode;

       /* follow chain of suffix pointers, changing secondary edges
   /* which point to childnode to point to newchildnode      */
for (currentnode = suffix( currentnode ); currentnode ISNT NIL;
      currentnode = suffix( currentnode )) {
          if ((Edge = findedge( currentnode, ch )) IS NIL) break;
    if ((depth(currentnode)+1) IS depth(node(Edge))) break;
edges( currentnode ) = unlist( edges( currentnode ), node( Edge ) );
       edges( currentnode ) =
            allocedge( newchildnode, edges( currentnode ) );
       }
       suffixnode = newchildnode;
       break;
    }
    /* otherwise, it's primary, so set suffixnode and break out */
else {
       suffixnode = node( Edge );
       break;
    }
}
/* set the suffix of newsink to be the node the above "for" loop found
*/ suffix( newsink ) = suffixnode;
currentsink = newsink;
}
*7*
```