Implementing Associative Coder of Buyanovsky (ACB) data compression
by Sean Michael Lambert

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
In 1994 George Mechislavovich Buyanovsky published a basic description of a new data compression algorithm he called the “Associative Coder of Buyanovsky,” or ACB. The archive program using this idea, which he released in 1996 and updated in 1997, is still one of the best general compression utilities available. Despite this, the ACB algorithm is still barely understood by data compression experts, primarily because Buyanovsky never published a detailed description of it. ACB is a new idea in data compression, merging concepts from existing statistical and dictionary-based algorithms with entirely original ideas. This document presents several variations of the ACB algorithm and the details required to implement a basic version of ACB.

IMPLEMENTING ASSOCIATIVE CODER OF BUYANOVSKY (ACB) DATA COMPRESSION

by

Sean Michael Lambert

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY-BOZEMAN
Bozeman, Montana

April 1999

APPROVAL

of a thesis submitted by

Sean Michael Lambert

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Brendan Mumey — (Signature) — 4/21/99 — Date

Approved for the Department of Computer Science

J. Denbigh Starkey — (Signature) — 4/21/99 — Date

Approved for the College of Graduate Studies

Graduate Dean — (Signature) — 4-22-99 — Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University-Bozeman, I agree that the Library shall make it available to borrowers under rules of this Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Sean Ladd

Date 4/21/99

TABLE OF CONTENTS

TABLE OF CONTENTS — Continued

LIST OF TABLES

Table Page

LIST OF FIGURES

ABSTRACT

In 1994 George Mechislavovich Buyanovsky published a basic description of a new data compression algorithm he called the "Associative Coder of Buyanovsky," or ACB. The archive program using this idea, which he released in 1996 and updated in 1997, is still one of the best general compression utilities available. Despite this, the ACB algorithm is still barely understood by data compression experts, primarily because Buyanovsky never published a detailed description of it. ACB is a new idea in data compression, merging concepts from existing statistical and dictionary-based algorithms with entirely original ideas. This document presents several variations of the ACB algorithm and the details required to implement a basic version of ACB.

CHAPTER 1

INTRODUCTION

Most modern data compression techniques focus on one of two basic concepts. One of these ideas is to statistically model the data, in an attempt to predict the next symbol based on the symbols which have come before. The other idea, which was pioneered by Jacob Ziv and Abraham Lempel in the late 1970's, involves replacing strings of symbols with dictionary references. The Associative Coder of Buyanovsky (ACB) combines portions of each of these concepts.

George Mechislavovich Buyanovsky, first published a general description of his new algorithm in the Russian periodical Monitor in 1994. It went relatively unnoticed until he produced an archive utility based on his idea in 1996. This archive utility topped the Archive Comparison Test (ACT) lists in several categories at its debut, and remains near the top of these lists today. About this time Buyanovsky posted a few articles outlining the technique to the Usenet newsgroup comp.compression. These basic descriptions were vague, however, and language barriers prevented more in-depth discussions on the topic. Buyanovsky's desire to profit from his invention may have also stifled his interest in sharing the details of ACB, although he has not discouraged further research.

The purpose of this study is to bring ACB to the attention of the data compression community by providing an accurate description of the algorithm, some of its variations, and details needed for its implementation. This thesis should be viewed as a point of departure for further research in this area.

CHAPTER 2

COMMON DATA COMPRESSION TECHNIQUES

This chapter provides a brief overview of the common lossless data compression techniques in use today. This overview will help to illustrate the differences and similarities between ACB and well established data compression algorithms.

2.1 Statistical Modeling Methods

Data is usually stored in a computer by assigning each symbol a fixed-size binary code. For example, ASCII code is a standard of representation for symbols such as the letters of the alphabet. Each character is assigned an eight-bit binary code. Statistical models change this paradigm by assigning shorter codes to more-frequently-used symbols and longer codes to less-frequently-used symbols. For example, if the model was English text, an 'e' would be represented with fewer bits than an 'x'.

2.1.1 Shannon-Fano and Huffman Coding

The Shannon-Fano and Huffman codes are popular choices for variable-size symbol coding. Each provides a heuristic for determining which bit patterns to assign to the symbols of a particular set, given the expected frequency of each symbol.

Compression is then simply a matter of replacing the old fixed-size bit patterns with the new variable-sized bit patterns. The drawback of these methods is that the distribution of the symbol frequencies does not necessarily match the distribution of the bit pattern lengths. This is because the bit pattern lengths must be integral numbers of bits, while the frequencies may be fractional numbers.

2.1.2 Arithmetic Coding

Arithmetic coding improves on the previous methods by allowing a symbol to use a fractional number of bits. This is accomplished by transforming the string of symbols to be compressed into a single real number, which has a very high precision. The number can be any number in a range, and that range is reduced as each symbol is encoded. This is most easily described by an example:

A language has 3 symbols: 'a', 'b', and 'c'. Their probability of appearance has been determined to be 0.5, 0.3, and 0.2 respectively. The output number will be between 0 and 1, so the initial range is (0, 1). As each symbol is encoded, the range is narrowed to the portion of the previous range matching the probability of the symbol's appearance. For this example the first symbol to be encoded is 'a', so the range becomes (0, 0.5). A 'b' would have reduced the range to (0.5, 0.8), and a 'c' would have reduced the range to (0.8, 1). The final encoded number will be in one of these ranges, allowing the decoder to determine what the first character of the string is. The probabilities are scaled to the new range, and these two steps are repeated until the entire string is encoded. The encoder then outputs a number in the final range, using the smallest possible number of

bits. Continuing the example, the next characters are 'b', 'a', and 'c'. The range is narrowed to (0.25, 0.4), (0.25, 0.325), and then (0.31, 0.325).

Arithmetic coders use several tricks to maintain the range without causing the variables used to underflow their precision. As long as the same trick is used by the decoder, the string can be restored. Arithmetic coding also allows the frequency distribution of the symbols to change at any time, as long as the information needed to change the distribution is also available to the decoder. This flexibility allows adaptive models to be constructed, which modify themselves based on the frequency of the symbols in the current string which have already been encoded.

It is also possible to change the frequency distribution based on the current context, or the preceding symbols. For example, in English if the characters 'zo' are the last two characters seen, the next character is very likely to be 'o' and not as likely to be 'e'. A statistical model which considers only the current character is called an order-0 model. A model is called order-*n* if it considers the previous *n* symbols when determining the frequency distribution of the current character.

2.1.3 Prediction by Partial String Matching (PPM)

Higher-order models produce better results than lower-order models, but they require more computation. PPM is a method that uses a higher-order model when possible, and lower-order models otherwise. For example, a particular PPM encoder tries to find an order-3 match for the current context and next character. If one cannot be found in the previously encoded text, an order-2 model is used, and an escape character

is generated to let the decoder know that the model has changed. This subtraction is repeated until the current character can be properly encoded.

2.2 Dictionary Methods

Dictionary methods of data compression substitute substrings of the string to be compressed with fixed-length indices into a dictionary. The indices, of course, must be shorter than the substrings they replace for compression to occur. The differences between the various dictionary techniques mostly have to do with the selection of the substrings which make up the dictionary.

2.2.1 Ziv-Lempel 77 (LZ77)

Many compression algorithms used today are based on the LZ77 algorithm. The dictionary for this method is a sliding window into the context, or a buffer containing the last k symbols. The compressor attempts to find the longest match between the string to be encoded, which is in the look-ahead buffer, and a portion of the sliding window. If a match is found, the compressor outputs a fixed-length index into the sliding window, followed by the length of the match, followed by the first unmatched character. Longer sliding windows and look-ahead buffers increase the compression and the computation required, similar to the way higher-order statistical models achieve better compression but require more computation than lower-order models. Longer sliding windows also require larger indices, which reduce the effectiveness of the substitution. Some compression utilities that use variations of the LZ77 algorithm are *arj*, *lha*, and *zip*.

2.2.2 Ziv-Lempel 78 (LZ78)

The LZ78 method uses an actual dictionary, rather than the sliding window that LZ77 uses. This dictionary is built as the encoding proceeds, using only previously encoded symbols. The decoder is then able to build its dictionary in the same way. The encoder collects symbols to be encoded until it has found a substring which is not in the dictionary. Since this substring was built one symbol at a time, there is a substring in the dictionary which matches all but the last character of the substring to be encoded. The compressor outputs the index of the existing dictionary entry followed by the unmatched character. Both the encoder and decoder now add to their dictionaries the substring made from the existing dictionary entry plus the unmatched character. Increasing the size of the LZ78 dictionary has similar consequences to increasing the LZ77 sliding window size. However, unlike LZ77, the LZ78 dictionary will become full. Often LZ78 compressors will switch to a static dictionary at that point, and monitor the compression rate. If the compression rate becomes too low, the entire dictionary is deleted and built up again. This is one of the most popular compression algorithms, and is used by the UNIX compress utility.

CHAPTER 3

ASSOCIATIVE CODER OF BUYANOVSKY (ACB)

There are several variations of the ACB algorithm, written by Buyanovsky and others. In a postscript to one of his letters Buyanovsky writes, "Let me remind [you] again this is a simplified version of the associative coder, and it is fairly easy to invent tens of variations of this scheme[.]" This chapter will illustrate the concept that Buyanovsky feels is the central idea of the associative coder, and will present some variations of his algorithm.

3.1 Some Definitions

The following definitions and Figure 1 will aid in the description of the algorithm variations:

- Let S be the string of symbols to be encoded. Its length is represented by N.
- Let S[*n*] be the nth symbol in S.
- The **context** at S[*n*] is the part of S including and preceding S[*n*], beginning with the most recent symbol and receeding into the past, i.e. {S[*n*], S[*n*-1], S[*n*-2], ..., S[2], S[1]}.
- The **content** at S[*n*] is the part of S following S[*n*], beginning with the next symbol and proceeding into the future, i.e. {S[*n*+1], S[*n*+2], S[*n*+3], ..., S[N-1], S[N]}.

Figure 1. Context and Content.

```
                    <-- context|content -->
S: ...11101000101011101100|10110100100111101011...
                          S[n]^
```

3.2 The Concept

Like LZ77, ACB uses a sliding window dictionary. ACB also outputs a dictionary reference and a length. The dictionary entries, however, are not referenced by fixed-size references into the window. The probability that each index will be chosen is estimated, and the choice is statistically encoded. Thus ACB combines ideas from both statistical modeling and dictionary compression techniques. The statistical concept, which Buyanovsky feels is the root concept of ACB, was unfortunately overlooked by all other references.

3.3 The Dictionary

The dictionary is actually a list of pointers into the sliding window. Buyanovsky specifies that the dictionary be sorted by context. The sliding window is considered to be a circular buffer so that the context and content of all entries are unbounded. Sorting the dictionary by context allows the associative list to be built more efficiently, but does not affect the compression ratio achieved by ACB.

3.4 The Associative List

The dictionary reference which is chosen actually comes from a subset of the dictionary called the **associative list**. The associative list contains all indices from the dictionary whose contexts match the current context by at least k symbols. Furthermore, the probability that each member of the associative list will be chosen is based on the length of this context match. Therefore, if the current context matches a large portion of the dictionary, it is assumed to be likely that the content following the dictionary reference will match the current content. These probabilities are used to arithmetically encode the index of the associative list member with the longest content match with the current content. The length of this match is also encoded, as in LZ77.

3.5 The Difference Bit

When using a binary language, if the associative list is sorted by content then it may be possible to encode much of the content match length using one bit. The **difference bit** is the value of the first bit of the current content which does not match the content of the selected associative list member. This bit will provide more information, however, when the current content is compared to the sorted associative list (Figure 2.)

If the difference bit is a '0', then the current content falls between the encoded index and the index preceding it. If the difference bit is a '1', then the current content falls between the encoded index and the index following it. Zero or more bits will match between the contents of these surrounding associative list members. Those bits

(indicated by 'x' in Figure 2) must also match the current content, so they do not need to be counted when encoding the length of the current content match.

If the current content falls before the first associative list entry or after the last entry, there will not be a second content to match against. In these cases there are assumed entries of all '0's and all '1's at the appropriate ends of the list. These entries are not numbered and cannot be matched directly to the current content.

Since the current content is known to match the encoded index, the next bit of the content must also match. The remainder of the content match (indicated by'z' in Figure 2) is then output. However, only the bits which do not match the difference bit need to be counted, since one of those bits is the first bit which does not match.

Figure 2. Use of the Difference Bit and Sorted Associative List.

```
                d=0                                  d=1
i-1  :xxx...xx0yyy...                 i   :xxx...xx0zzz...zz0...
S[n]:xxx...xx1zzz...zzd...            S[n]:xxx...xx0zzz...zzd...
i    :xxx...xx1zzz...zz1...           i+1 :xxx...xx1yyy...
```

3.6 Variations of ACB

The variations of ACB presented in this section are all of the previously published ACB algorithms along with the algorithm developed by the author of this thesis. They have been numbered somewhat arbitrarily by the author of this thesis.

3.6.1 ACB1

This is the algorithm given by Buyanovsky in his 1994 Monitor article. However, it is not the algorithm that the ACB archive utility uses. Buyanovsky states that the newer algorithm is thirty to forty times faster than this one, but shares the same basic premises. ACB1 is the algorithm described above, listed here in detail. The dictionary is considered to be a binary string rather than a character string, so there is a pointer into the sliding window for each bit.

1) Maintain a sliding-window dictionary, sorted by context.
2) Build an associative list containing all dictionary entries matching the current context by at least k bits.
3) Assign each member of the associative list a weight based on the length of the context match with the current context.
4) Sort the associative list by content.
5) Find the associative list entry which has the longest content match with the current content.
6) Statistically encode this index using the weights found above.
7) Encode the difference bit.
8) Encode the match length, modified as described in section 3.5.
9) Update the dictionary, move the current pointer ahead, and repeat.

3.6.2 ACB2

This algorithm is a simplified version of ACB1. It was originally described by Buyanovsky in an article posted to the newsgroup comp.compression on September 18, 1996 (Original Message-ID: <AAX10GoKlE@acb.alma-ata.su>)

ACB2 is exactly the same as ACB1, except that no length is encoded. ACB2 relies on the index and the difference bit alone to imply the match length. The problem with this method is that after a long match, the same dictionary entry will be selected as the most likely to match the next entry. It is apparent that if the same entry is selected

consecutively, it is very likely that the second encoding will match very few or no new bits. For this reason, ACB1 is preferred.

3.6.3 ACB3

This algorithm was posted to the newsgroup comp.compression.research on May 5, 1996, by Leonid A. Broukhis. (Original Message-ID: <4mhjvp$70u@net.auckland.ac.nz>) It is also presented as ACB compression in David Salomon's book Data Compression. This algorithm does not use an associative list, and works with a character-based dictionary.

It is interesting to note that the difference bit and its use were also described by Broukhis, but he did not mention it in any of his examples. This may be because problems arise when mixing character-based and bit-based models (described in section 1.4.) The difference bit's use would have also been problematic in ACB3 because the dictionary is sorted by context and not content.

1) Maintain a sliding-window dictionary, sorted by context.
2) Find the dictionary entry which has the longest context match with the current context.
3) Find the dictionary entry which has the longest content match with the current content.
4) Encode the index of the content match as an offset from the context match. If this number is consistently small then compression can be achieved by statistically encoding this offset.
5) Encode the length of the content match.
6) Encode the first unmatched byte.
7) Update the dictionary, move the current pointer ahead, and repeat.

3.6.4 ACB4

Salomon described a second algorithm in his book Data Compression. It was

presented it as an ACB variation, though it is closer to ACB1 than ACB3 is. This variation uses a character-based dictionary and associative list, but does not use any form of statistical modeling.

The difference bit is explained by Salomon and used in this method, but there is a problem combining a character-based dictionary with the difference bit concept. The problem is that the end of any particular match is very likely to occur in the middle of a byte, rather than at byte boundaries. It is unlikely that the contexts of the dictionary entries would have much meaning in this case.

1) Maintain a sliding-window dictionary, sorted by context.
2) Build an associative list containing all dictionary entries matching the current context by at least k bits.
3) Sort the associative list by content.
4) Find the associative list entry which has the longest content match with the current content.
5) Encode the index of the content match as a fixed-length index.
6) Encode the difference bit.
7) Encode the match length, modified as described in section 3.5.
8) Update the dictionary, move the current pointer ahead, and repeat.

3.6.5 ACB5

This algorithm was developed by the author of this thesis, and is the algorithm discussed in detail in chapter 4. It is much like ACB1, but uses a character-based dictionary. The associative list is a fixed size, listing the best m context matches, rather than all context matches of length k.

1) Maintain a sliding-window dictionary, sorted by context. Implemented dictionary size was 1024 bytes.
2) Build an associative list containing the m dictionary entries which most closely match the current context. Implemented associative list had 256 entries.
3) Assign each member of the associative list a weight based on the length of the

context match with the current context.
4) Sort the associative list by content.
5) Find the associative list entry which has the longest content match with the current content.
6) Statistically encode this index using the weights found above.
7) Encode the difference bit.
8) Encode the match length in whole bytes, modified as described in section 3.5.
9) Encode the first unmatched byte.
10) Update the dictionary, move the current pointer ahead, and repeat.

3.7 Example Using ACB5

To better illustrate how ACB works, a short example of ACB5 follows. Each iteration of the algorithm produces a token to be arithmetically encoded. The token contains an associative list index *i*, a difference bit *d*, a match length *l*, and an unmatched character *c*. This example will show how the token (*i*, *d*, *l*, *c*) is generated, but will not show the encoding of the token. The encoding process is explained in detail in chapter 4.

Figure 3 shows the sliding window which is used in the example. The current context ends with the 'i' in "willson.", which means the current content is "llson.". The length of the dictionary is 23 characters in this example.

Figure 3. Example Sliding Window and Current Content.

Figure 4 shows the dictionary sorted by context. Remember that the context is read from right to left in this example. Since the sliding window is kept in a circular buffer, the content and context can be imagined to repeat indefinitely. The contexts and contents of the dictionary entries in Figure 4 have been repeated for one entire buffer length to illustrate this. The ten dictionary entries which most closely match the current context are selected to become the associative list. These ten entries are marked with asterisks, and the current content/context is marked S.

Figure 4. Example Dictionary Sorted by Context.

	<-- context	content -->	
1	illwillstillkilljillwib	illwillstillkilljillwib	
2	llwillstillkilljillwibi	llwillstillkilljillwibi	*
3	llwibillwillstillkillji	llwibillwillstillkillji	*
4	lljillwibillwillstillki	lljillwibillwillstillki	*
5	llkilljillwibillwillsti	llkilljillwibillwillsti	*
6	llstillkilljillwibillwi	llstillkilljillwibillwi	*
7	billwillstillkilljillwi	billwillstillkilljillwi	S
8	illwibillwillstillkillj	illwibillwillstillkillj	*
9	illjillwibillwillstillk	illjillwibillwillstillk	*
10	lwillstillkilljillwibil	lwillstillkilljillwibil	*
11	lwibillwillstillkilljil	lwibillwillstillkilljil	*
12	ljillwibillwillstillkil	ljillwibillwillstillkil	*
13	lkilljillwibillwillstil	lkilljillwibillwillstil	*
14	lstillkilljillwibillwil	lstillkilljillwibillwil	
15	willstillkilljillwibill	willstillkilljillwibill	
16	wibillwillstillkilljill	wibillwillstillkilljill	
17	jillwibillwillstillkill	jillwibillwillstillkill	
18	killjillwibillwillstill	killjillwibillwillstill	
19	stillkilljillwibillwill	stillkilljillwibillwill	
20	tillkilljillwibillwills	tillkilljillwibillwills	
21	illkilljillwibillwillst	illkilljillwibillwillst	
22	illstillkilljillwibillw	illstillkilljillwibillw	
23	ibillwillstillkilljillw	ibillwillstillkilljillw	