1-1-1988

Adaptive arithmetic data compression: An Implementation suitable for noiseless communication channel use

James E. Robinson

Adaptive Arithmetic Data Compression: An Implementation Suitable for Noiseless Communication Channel Use

James E. Robinson
17-August-1988

Rochester Institute of Technology
School of Computer Science and Technology

A Thesis ,
submitted for approval to
The Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Thesis Approved by:

_____
Dr. Donald Kreher (Advisor)

_____
Dr. Peter Lutz

_____
Dr. Peter Anderson

Table of Contents

Abstract

Noiseless data compression can provide important benefits in speed improvements and cost savings to computer communication. To be most effective, the compression process should be off-loaded from any processing CPU and be placed into a communication device. To operate transparently, it also should be adaptable to the data, operate in a single pass, and be able to perform at the communication link's speed. Compression methods are surveyed with emphasis given to how well they meet these criteria. In this thesis, a string matching statistical unit paired with arithmetic coding, is investigated in detail. It is implemented and optimized so that its performance (speed, memory use, and compression ratio) can be evaluated. Finally, the requirements and additional concerns for the implementation of this algorithm into a communication device are addressed.

Chapter 1
Introduction and Background

1.1 Introduction

In the rapidly growing world of computers and computer communication there are several obvious trends. Many systems are distributed in nature requiring both enhanced processing and communication facilities. At the same time that processing speed and memory is getting significantly less expensive, the cost for communicating data is only very gradually reducing in price. This makes the communication channel both a physical and economic bottleneck. The physical problem can be addressed by increasing the actual throughput while the economic one can be improved by reducing the cost/bit ratio. Both of these are benefits of data compression.

The specific type of data compression that fits the communication area is 'noiseless' data compression. This means that upon decompression, the reconstructed data will exactly match the input data, bit for bit. Some methods (often used in image compression) do not have this exact reversibility and are really methods of data compaction. As these would not be suitable for transparent insertion into a communication channel, they will not be surveyed. Another requirement is that the compression method be semantically independent (Reghbati, 1981). Any method that expects a certain specific type of data may not compress other types of data at all. For example, suppose one is using a dictionary substitution compression system that replaces certain words or portions of words by a code symbol. These methods are based on the probabilities of certain strings appearing in the file and are tuned to specific contexts; i.e., English text. If a file of Russian text is passed through the algorithm, very little worthwhile substitution may occur and the resultant file could actually grow in length.

All methods of data compression are apparently based on one main principle: removing redundant information. Computer data is often encoded for storage and transmisslon in a form that contains redundancies. For example, In a text file, 8 bit bytes (which can express 256 different values) are used to store each character even though the standard character set rarely exceeds 70 characters. Also, the language of the text file has certain redundancies around character distributions that can be exploited. There are numerous methods of data compression. Some are general in nature and some are very specific to certain types of data. Because the thrust of this investigation is towards compression in a communication channel, compression methods that are general in nature will be emphasized.

1.2 Compression Techniques

Null suppression is a compression method that replaces runs of repeated blanks with a special character followed by the number of blanks that occur in a row. This provides a savlngs whenever 3 or more blanks are consecutlve. This method ls easy to lmplement, adds very llttle overhead, and can provide signiflcant savlngs when data has long blank strings. IBM's 3780 BISYNC protocol in fact includes this feature and provides enhanced throughput in forms mode applications where significant amounts of white space appears on a screen (Held, 1983).

Run length encoding is another method that is just the general case of null suppression. Rather than just encode strings of nulls, it encodes strings of any repeating character. It usually does this with a flag character, the character being repeated, and the number of characters in the string. The Kermit file transfer protocol uses this method for data compression (Nelson,1986). Other approaches use variable bit length representations for differing length strings of characters based on their statistical occurrence (Golomb, 1966). It provides an improvement over null suppression only when the data includes significant repeating strings of other characters besides blanks. This appears often in lmages but rarely in text (Welch, 1984).

Relative encoding encodes the differences between adjacent source items and sends this rather than the source items themselves. This works very well with telemetry data and digitized images where adjacent values are highly correlated and the magnitude of the differences is very small compared to the source items. It has limited usefulness when applied to character data.

Pattern substitution involves replacing a character or series of characters with another symbol that uses fewer bits to represent the same information. These methods differ in how they decide to divide the source and the way they apply the code words. One way to encode text is to replace common words with 1 or 2 byte codes. When applied with a large statistically generated dictionary, theoretical compression can approach a ratio of 4:1 (Pike, 1981). Dictionary search computation time, the dependency of the method on the how well the text usage of words matches the dictionary, and the strict language dependency all limit this method's usefulness in a general communication channel.

Huffman coding is a way to efficiently implement a dictionary substitution compression method. This method assigns variable bit length codes to input symbols such that the code length in bits approximates $\log_2$(symbol probability) of that input symbol occurring (Huffman, 1952; Tanenbaum, 1981). For example, if a certain symbol occurs one-eighth of the time, it is encoded with a 3 bit code, while a symbol occurring one-one hundreth of the time will

have a 7 bit code. Higher probability symbols need fewer bits than lower probability symbols. It works best with symbol sets that have a skewed distribution (Reghbati, 1981) and needs accurate statistics to compress at all. This usually means a two pass compression technique. The data is first scanned for frequency distributions and then the frequency data is used to encode the data. Huffman coding also requires the frequency tables to be transmitted to the receiver so that the data can be decompressed.

Lempel and Ziv (1977) have devised a single pass adaptive compression method that converts variable length sets of input symbols into fixed-length codes. Welch (1984) has implemented a version of this method which parses the input symbols into strings (which are assigned to code words) with a specific prefix property. That property is, if the string is in the code table, then the string minus the last character is in the table. The input symbols are read one at a time and the longest already coded string piece is parsed off. This piece combined with the next symbol now forms a new string assigned a code word. The code string table begins with code words for each single input symbol. For decompression, the same string table is built as the symbols are translated. Each code value is recursively broken into two parts, the prefix string and the extension character, until the prefix string consists of a single character. Once a code value is translated, an entry in the string code table is made for the newly found extension character and its prefix string. Since the code table is built by both the compression and the decompression algorithm, no explicit code table needs to be sent preceding the actual compressed data.

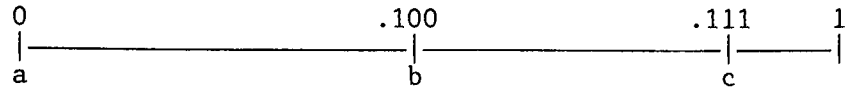
This method is shown be be effective in general data compression, however, the string parsing/searching takes a very large amount of time (Ziv and Lempel, 1978). Rodeh (1981) has shown a linear method for implementation but it takes large amounts of memory and does not compress effectively until a large amount of input data has been processed.

Arithmetic coding is another method of encoding that is based on probabilities of certain symbols occurring. The higher the probability of a character occurring in a message, the fewer bits it takes to encode the character. Arithmetic coding has been described in various sources (Jones, 1981; Langdon, 1984; Langdon and Rissanen, 1982; Rissanen, 1983). Fundamentally it is a coding method that takes a series of input symbols and converts them into one code string. This code string can be thought of as a binary fractional number between 0 and 1. Langdon (1984) provides a good tutorial on the process and the basic principles will be covered here.

For example purposes, a source alphabet of three symbols will be used. Assume that the probability of these three symbols occurring is represented in the following table.

Symbol	Probability	Probability as Binary Fraction	Cumulative Probability	Cumulative as Binary Fraction
a	1/2	.100	0	.000
b	3/8	.011	1/2	.100
c	1/8	.001	7/8	.111

Think of these symbols as being points on the number line from 0 to 1:



Each Interval corresponds to the character at its left edge and the width of each Interval relates to the probability of that character occurring. The encoding process maps a binary fraction to this code space based on magnitude comparison. If a source string begins with an 'a', for example, a value greater than or equal to zero and less than .1 could encode it. For a 'b', a value greater than or equal to .1 and less than .111 could encode it. As a character is processed it picks the encoding interval as its next working interval. The first character can map to an Interval anywhere on the line from 0 to 1 while the second character will only have the interval derived by the first character. This successive subdivision is shown in Figure 1 for the character sequence 'aabaca...'.

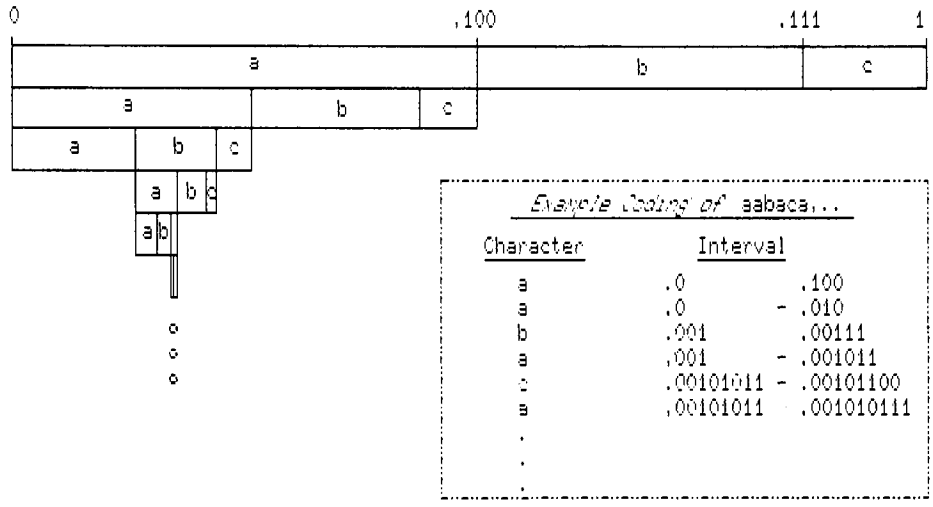


Figure 1 : Successive Subdivisions

The current interval now defined at the code point is from .00101011 to .001010111. Any value within this interval could be interpreted by magnitude comparison as a string starting with 'aabaca'. This successive subdivision continues dividing smaller and smaller intervals, always related to the symbol probabilities. As the probabilities are reflected in the interval widths, the number of bits needed to encode each symbol can very closely approach the entropy of the source symbol (Jones, 1981). Assuming that the probability tables reflect the symbol distribution, many more high probability symbols will occur than low probability symbols. The high probability symbols require the fewest bits to encode and thus cause the greatest compression to occur.

This example has given a pictorial view of arithmetic coding. The actual implementation is done using summing of cumulative probability values and interval widths and is covered in detail in Chapter 2. The basic problem in the algorithm is how to constrain the number of digits of precision that are needed to be carried as the actual interval gets smaller and smaller. What effectively is one huge fraction must be represented by a method of floating point or scaled fixed point arithmetic which only works with the portion needed at the current encoding interval at any one time (Rissanen and Langdon, 1981).

Chapter 2
Thesis Description

2.1 Introduction

The central portion of this thesis is the implementation and optimization of a data compression/decompression method based on arithmetic coding that would be suitable for communication channel use. It is implemented on a general purpose computer (not a specialized communications device) and evaluated for compression efficiency, throughput, and memory use. The results from this evaluation are then used to review important implementation details that need consideration before the compression method could realistically be inserted into a communication device.

In reviewing the literature, Cleary and Witten's (April 1984) adaptive compression method using partial string matching and arithmetic coding seemed the most suitable for investigation. It reports impressive compression statistics, is a single pass general purpose system, and offers areas for significant improvement. The compression system implemented for this project is based on the concepts presented by Cleary.

The data compression system really consists of two independent parts; the coding unit and the statistics unit. The coding unit's input is a character to encode along with a cumulative probability table of the chances of that character occurring. The output of the coding unit is a set of bits that is the encoded character. The statistics unit keeps track of character occurrences and contexts so that cumulative probability tables can be built. Keeping these two portions separate provides two important benefits. It allows flexibility to change the statistics model with no effect on the mechanics of the coding. It also allows different statistics to be presented with each character and thus the capability for a continually adapting model. The data structures and algorithms that make up both the statistics unit and the coding unit will be explored in detail in the following sections.

2.2 Program Implementation

The compression program (ac) and the decompression program (unac) were developed on a VAX computer running VMS. They are written in C and care was taken in only using standard C constructs so that the programs are directly portable at the source level to other systems that implement a 'standard' version of C. They have been successfully run on an HP840 system running HP-UX as well as Unix based Sun workstations.

In describing the operation of this compression / decompression system, the main point of view will be that of the compression program. Algorithms will usually be explained for taking plain text and turning it into compressed text. Decompression involves 'undoing' this operation and uses very similar constructs though with some interesting nuances. Where appropriate, these differences will be highlighted.

2.2.1 Markov Statistics Model

Arithmetic coding provides the greatest compression for a source symbol when it occurs in a context where the symbol is predicted to occur with high probability. Thus to optimize an arithmetic coding scheme with respect to compression, a statistical model with which the next symbol can be predicted with a high degree of certainty from the current source context needs to be developed. This predictive certainty is what gives the real power to any arithmetic coding based compression system (Rissanen and Langdon, 1981). Cleary's (April 1984) method of partial string matching provides an effective and flexible approach to this problem. This method was used for the basis of the statistics unit of this project.

The method involves implementing a variable order Markov model of the source symbols. For each run of the compression package, a maximum order is selected (in the range of 0 to 4). The order defines the maximum context with which each source symbol will be encoded. For example, an order 2 model will encode a source symbol in the context of the 2 preceding symbols, an order 3 model encodes in the context of the 3 preceding symbols, etc. In text compression, higher order models usually give better estimation of source symbols. Given 4 characters in English, for example, one can often predict the next character with a high degree of certainty. Four was chosen as a maximum order for this implementation as orders of five of more can take very large amounts of memory and do not provide statistically significant better performance. Each running of the compression system starts with an 'empty' statistics unit. As the source is read and encoded, the frequency tables for characters and character strings are built. (This 'building as we go' is duplicated at the decompression end and allows the adaptive algorithm to work without the explicit transmission of any statistics.)

This string matching system was enhanced by the concept of partial string matching. In early portions of any compression run, the statistics tables will not be developed enough to produce effective high order Markov predictions to result in much compression. During this time frame, lower order predictions will be used based on the highest order partial string that has been encountered to date. This can be done because of the way that the statistics are saved; that is, the lower order Markov statistics are a subset of the higher order Markov statistics.

Cleary does not give details on the data structure of the statistics unit, just details of an individual element(node). Using this as a starting point, the statistics unit for the compression program in this thesis was designed. The basic piece of information that needs to be saved is 'given this nth order context, how many times has this source symbol been seen before'. If we are using a 0th order (memoryless) model, then the count will just be the number of times that the symbol has occurred in the source to date. If we are using a 1st order model, then the count will indicate how many times a symbol has been seen following the preceding symbol, etc. The counts are organized in a linearized tree of linked lists of symbol nodes. Each node consists of 4 elements:

the symbol

a count of the number of times this symbol has been seen in this context

a pointer to the next symbol node at this same level. (This means another symbol in the same context as the symbol at this node.)

a pointer to the first symbol in the linked list of symbols at the next higher context level.

Pictorially, a data representation of a 2nd order model of a 3 symbol alphabet is shown in Figure 2.

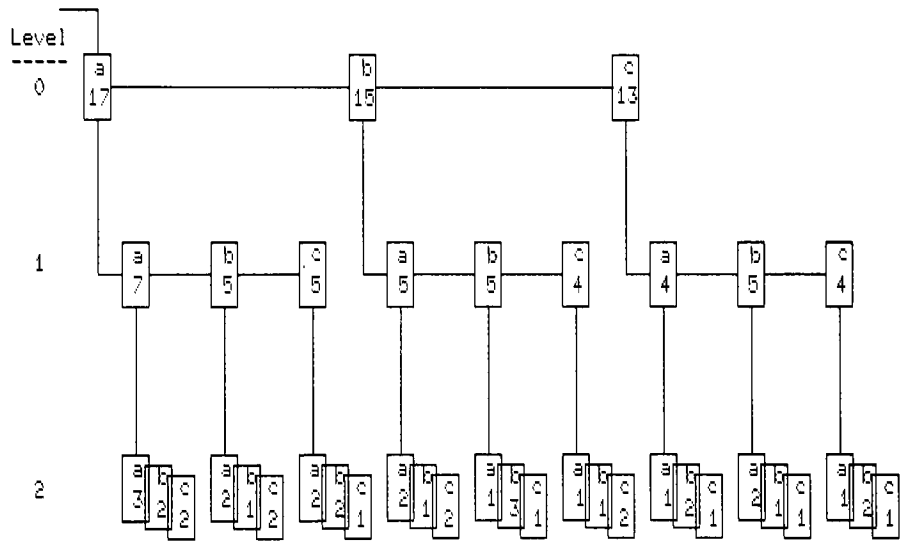


Figure 2 : Linked List Data Structure

Note In any real implementation it is highly unlikely that all source symbols will appear at all levels as in this picture. For example, in English text, it is very rare that 'z' follows 'q'. This makes an actual string tree for text have increasingly smaller linked lists as the level gets larger. It is this fact that allows us to even do Markov models larger than order one in a reasonable amount of memory.

Each character from the input stream follows the basic processing shown in the following psuedo-code segment:

```
{
    order = maximum Markov order of current model;
    for (order >= 0)
      {
          locate the position of this character
            in the context of the current order;
        if (the context exists and the character
                has not yet been encoded)
          {
              figure the cumulative statistics for the
                 character context;
             arithmetic code the character context;
          }
        update the Markov model at this order with the
         character and/or count;
        order = order -1;
       }
  if (the character itself still hasn't been encoded)
     {
        figure the cumulative statistics based on a novel
            character;
        arithmetic code the novel character;
     }
}/*end*/
```

In general terms, during compression, as each source symbol is encountered, the tree is followed starting with the n preceding symbols. If the symbol has been seen before in this context, then the counts of the symbols at that context level will be used to figure cumulative probabilities based on frequencies. If the symbol has not been seen, then the context of n-1 will be checked, etc. , until a context is found in which the character can be predicted. After the character has been encoded, the appropriate counts will be updated in the frequency table. This must be done, *after* encoding since the decompression algorithm needs to duplicate the tree organization.

Besides allowing partial string matching, this data structure provides the necessary organization of linking together all occurrences of symbols at a given context. (For example, the counts of all characters that have occurred following the 2 characters 'qu'.) This provides an easy way to form a cumulative frequency table required by the arithmetic coding algorithm.

A few of the constructs at work in this Markov model could use some additional clarification. Each time a character is processed from the input stream, its occurrence must be 'logged' into the model at each level relating to each of the Markov orders (from 0 to whatever the maximum order is). For example, if we are running an order 2 model and the current character stream is 'This is a'; when the trailing 'a' is added to the model, it will appear in the contexts of 's a', ' a', and 'a'. The step by step building of a Markov search

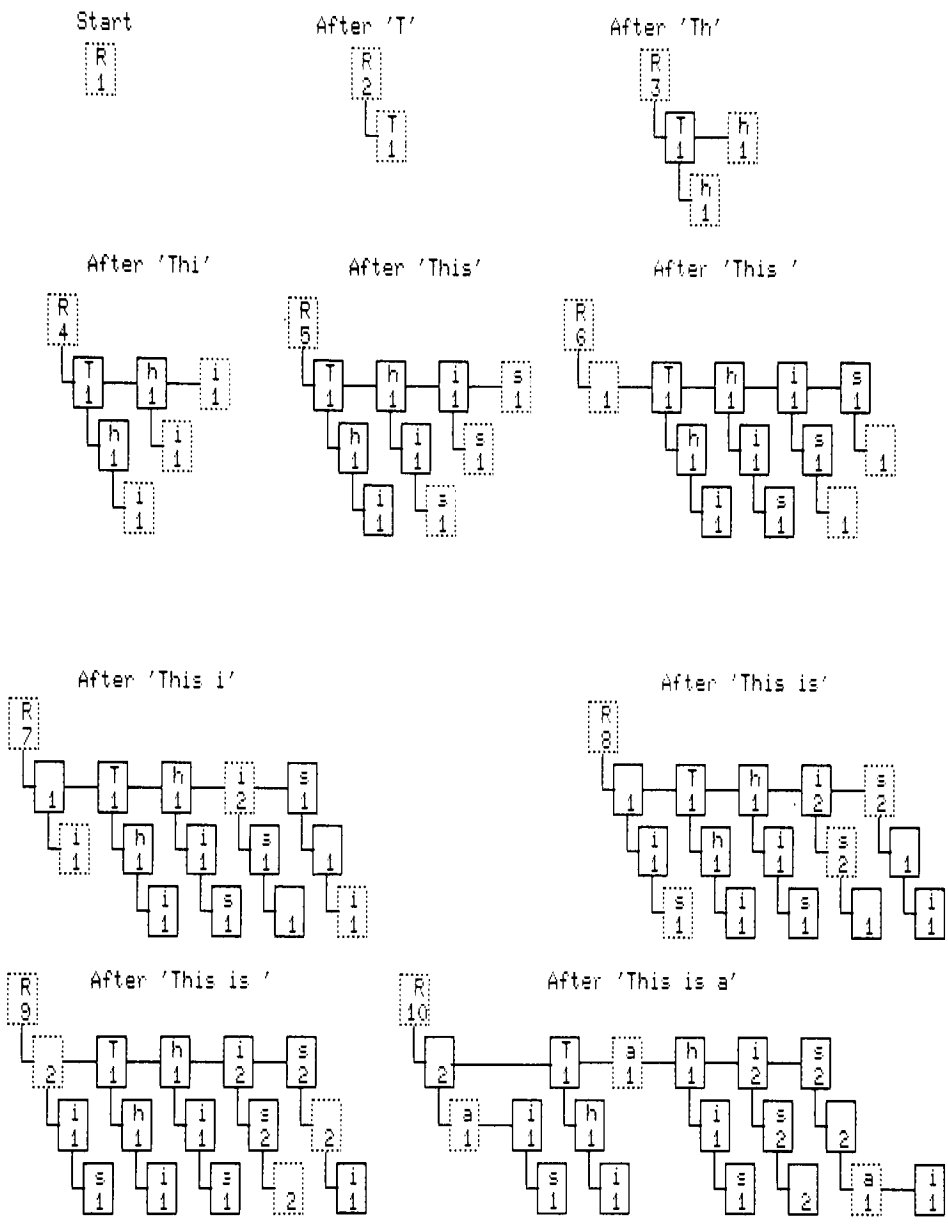


FIGURE 3 : Building of Order 2 Markov Linked List

tree is shown in Figure 3. Here the beginning of a order 2 tree is shown for the initial character sequence of 'This is a'. The root is shown at the top of each sequence and is designated by 'R'. The number in each box indicates the node count that will be used to figure probabilities. The boxes with the dashed borders indicate the boxes that are new (or have had their counts updated) since the last character was added. To minimize the search time needed to find all these contexts, three main constructs are in place.

1) The nodes that represent the characters already found at any given context level are kept In ascending order of character code. This allows the program to not have to search the entire list to determine If the character has been seen before In this context.

2) The current contexts from each character are passed through an array of structures (search_spec in the program) for processing by the subsequent character. This array of structures holds a pointer to the actual character node at each context order as well as a pointer to the node which is at the level above the current level and points to the start of the node level list. This second pointer is called the 'joint' pointer. The 'joint' is the place one would begin when looking for a certain character at the next higher context. Because of the organization of the node tree, the matching pointer of a character at order X is the joint pointer of the next character at order X+1. By passing these contexts from character to character, there never need to be more than (maximum order + 1) node lists searched for any character encoding.

3) The final construct at work in handling the Markov tree search is a special array of pointers to the order 0 level nodes. The order 0 node list has one entry for each character code ever seen in the input data stream. This makes It potentially long (up to 256 elements) and performance analysis of the running program showed a lot of time spent just bumping along this list looking for the character to update its occurrence count. By having this explicit array of pointers, the 0th order character nodes can be accessed much much faster. This set of 0th order pointers is not shown in Figure 3 but the pointers would just point to the top row of nodes.

When the Markov search algorithm looks for a character in a certain context and does not find it (I.e., it has not yet been seen in this context in the input data stream yet) It then needs to back down to the next lower order context and try there. Before It can do that however, it must insure that whatever it does can be interpreted by the decompression algorithm. If when it looks for the character in the original context and there isn't even any context (for example, an order 2 model, '..qui' is the character stream, and there has not yet been an occurrence of 'qu' anything yet) then it can immediately back down to the next lower level. It can do this without inserting anything into the output stream as the decompression routine can recognize the same situation and back down unambiguously. If, however, there already are characters in this context, a mechanism called 'escape' by Cleary (April 1984) needs to be invoked to signal the decompression routine that we're backing down to the next order context to try to compress the character. The 'escape' mechanism can be thought of as sending a special character out in the current context. The method employed in this project is to always bias all the cumulative counts of number of occurrences of certain contexts by one. So, if there have been 19 occurrences of 'qu' in the input stream to date, say that there really were 20. This extra count is assigned to this 'escape' character and always allows its identity to be transmitted. Early on in the message stream, this 'escape' character will be able to be transmitted with minimal overhead as the cumulative counts will not be all that high relative to its count of one. Later on, the 'escape' character's identity will require more bits to transmit but the probability of needing to use it also drops so it balances out.

When a character is not found in a certain context, the 'escape' mechanism is used to back down to the next lower level to try a partial string match at this level (i.e., try to predict the character with the next lower order). One more mechanism that is used to raise the predictive power of the model is the concept of the exclusion list. When the 'escape' mechanism is used, there are a set of characters on the original level that the model could have predicted. Because they have already been entered into the model tree, these same characters are known to exist on the next lower level. If the processing of these characters is excluded when forming the cumulative probabilities of the lower level predictions, the effective probability of the character we want to encode is raised. If on going to the next lower level, the character still is not found, then the additional characters from that level are added to the exclusion list, and an 'escape' occurs again.

If all the levels (from maximum order down to zero) have been traversed and the character still has not been found in the model tree, then the character is a novel character (first occurrence of this character code in the input stream). The algorithm gives equal probability to each character code that is still novel of occurring and encodes it accordingly.

The decompression program builds the Markov tree in the same manner that the compression system does; by including entries for each new character at each order level. It must figure overall cumulative counts at a level before knowing what the character at that level will be. If the match value count that comes out of the arithmetic decoding routine matches the (cumulative count 1) then it is known that this is an 'escape' to the next lower level. (As the 'escape' character was always implicitly allotted one count at the end of each node string.) The back down can then occur immediately without further node string searching.

2.2.2 Arithmetic Coding Method

As previously mentioned, the compression method described in this thesis is based on concepts presented by Cleary (April 1984). While he states that arithmetic coding is used for the coding unit, it is not specified how the arithmetic coding is done. Various algorithms have been presented for doing the coding (Jones, 1981; Langdon, 1984; Langdon and Rissanen, 1981; Langdon and Rissanen, 1982) . After careful review, Jones' (1981) method was chosen to be used in this implementation for a variety of reasons. The other methods have been shown to produce theoretically equivalent compression ratios (Cleary, April 1984), so the choice was mainly based on the following implementation specifics:

> the method works directly on the source symbol frequency counts so no additional calculation step is required in figuring symbol probabilities,
>
> it is a method based on integer arithmetic imitating fixed precision floating point calculations, thereby operating efficiently,
>
> the method extends easily to various source alphabets and
>
> it is easily partitioned from the statistics gathering portion of the compression model so that the separation of these two blocks can be maintained.

Two other aspects that may give it a slight practical advantage in coding efficiency also

make it a good choice.

- It solves the 'carry over' problem in arithmetic coding (which will be discussed in detail later) with an internal register rather than relying on a somewhat less efficient bit stuffing technique employed by Langdon's (1981) method.
- Its subdivision of the code space uses rounding on the end points of the interval rather than on the interval width itself like Jelenik's (1968) and thereby does not waste any of the code space.

While a conceptual view of arithmetic coding has already been covered in chapter 1, a closer look at Jones' actual method is in order. The arithmetic coding algorithm takes an input string of n different symbols and encodes it into a single output sequence made up of m different output symbols. For this data compression project, the general form of the algorithm has been fixed at using a source alphabet where n is an integer less than or equal to 256 (one symbol for each possible value of an 8-bit byte) and the output symbols are just 0 and 1.

The input items that are presented to the coding algorithm for each symbol to be encoded are a set of three integers from which cumulative probability statistics can be represented. An input symbol always exists in the context of other symbols, and they each have a certain probability of occurring represented by how often they have been seen so far in the message. The symbols within each context need to be ordered in some sequence (this implementation just orders by increasing value of input symbol) and the three integer inputs needed for the coding algorithm come from this ordering. The inputs are:

Cum_total	This is just the total count of the number of times all the characters in the context sequence have been seen.
Cum_actual	This is the addition of all the character counts in the context sequence up to and including the character that is to be encoded.
Cum_previous -	This is the addition of all the character counts in the context sequence up to but not including the character that is to be encoded.

(Note: From these values it can be seen that

$$(Cum\_actual - Cum\_previous)/Cum\_total$$

is the actual probability of the character to be encoded of occurring in this context.)

To map this probability into an output code sequence, a method must be used relate it to a position and width function on the number line from 0 to 1 as described in the arithmetic coding overview. Remember that these positions and widths are used to continually subdivide the code space relative to the probability. Some maximum amount of arithmetic precision must be chosen that can be used to represent any width. This value must be large enough to be able to distinguish between small differences in the cumulative values presented to it yet small enough not to cause arithmetic overflow in the calculations. This value represents bit widths so is a power of 2. This application uses a maximum width value of 21 bits so the *PRECISION* parameter is 21 and it represents a precision of 2 raised to the 21st power or 2097152. Given the initial settings for the empty input string as:

Output_length(in bits)	= 0
Position	= 0
Width	= 2097152 (2^{21})

If e... represents the current input string and x represents the next character to encode then the subsequent values of *Output_length*, *Position*, and *Width* can be recursively represented by:

$$Position(e...x) = (Position(e...) + int(Width(e...) * cum\_previous(x) / cum\_total + .5)) * 2\hat{}roll \quad (1)$$

$$Width(e...x) = (int(Width(e...) * cum\_actual / cum\_total + .5) - (int(Width(e...) * cum\_previous / cum\_total + .5)) * 2\hat{}roll \quad (2)$$

$$Output\_length(e..x) = Output\_length(e...) + roll \quad (3)$$

Where *roll* is the integer which causes *Width* to satisfy the following inequality:

$$2\hat{}PRECISION <= Width(e...x) <= 2\hat{}(PRECISION + 1)$$

A suitable value of *roll* can always be found provided

$$(cum\_actual - cum\_previous) * 2\hat{}PRECISION >= cum\_total$$

when the encoding algorithm is called. For this application, the inequality implies that *cum_total* for any call to the encoding algorithm must be less than 2097152 (because the statistics unit assigns a count value of one to the escape condition). To insure this always happens, a restriction must be put on the file input size limiting it less than 2097152 characters.

Conceptually what is happening with these calculations is as follows. The new interval position on the number line gets 'marked' by scaling the beginning cumulative probability by the current width. The new interval width is calculated by scaling the actual character probability by the current width. Then the new interval width is expanded to the limits of the *PRECISION.* This zooming in (or normalization) on the interval width then also gets reflected in the new position.

These calculations cause the width to stay within a power of 2 of the *PRECISION* value while the position continues to grow in length. Clearly, with arithmetic of fixed precision (32 bit integers in this case), the position value would quickly overflow its storage unless something was done. Jones (1981) devised a very ingenious mechanism to efficiently handle this. Looking at equation **(1),** each new position value consists of adding a new value to the old position value and then multiplying it by a power of 2 (shifting it left). One only needs to keep the most recently added bits (the 21 encompassed by our *PRECISION* parameter) around if the following method is used. The multiplication in equation **(1)** by *2ˆroll,* effectively makes *roll* bits in *Position* be pushed out of the most recent 21 bits saved in *Position.* These bits constitute the encoded output stream at this point. Unfortunately, because of the additive operation in equation **(1)** a subsequent addition could cause a carry operation that could change some of these bits that are ready to go to the output stream. A zero bit in the output stream is a carry 'blocker' and insures that a carry will not extend any farther into any previous bits. This property is exploited by looking at the bits as they become ready to go to the output stream. Rather than output any 'one' bits immediately, they are counted until a 'zero' bit is found. Assuming no carries at this point, the whole string of ones can be output, knowing that the newly found zero bit will block any subsequent carries from affecting them. If there is a carry while 'one' bits are being counted, the algorithm can change them all to zero (the equivalence of a carry rippling through them) and then output them. In this manner *Position* can be kept within *PRECISION*+2 bits and the output stream can be kept consistent.

2.2.3 Example Encoding

For an example of the Input statistics that would be presented to the arithmetic coding algorithm, consider the example text shown previously In Figure 3. The encoded characters, the order in the model at which they are encoded, and the associated statistics values are as follows:

Character	order	cum_previous	cum_actual	cum_total
T	novel	84	85	256
escape	0	1	2	2
h	novel	103	104	255
escape	0	2	3	3
i	novel	103	104	254
escape	0	3	4	4
s	novel	112	113	253
escape	0	4	5	5
' '	novel	32	33	252
i	0	3	4	6
s	1	0	1	2
' '	2	0	1	2
escape	2	1	2	2
escape	1	0	1	1
escape	0	6	7	7
a	novel	95	96	251

The best way to follow the actual calculations being performed to determine *Position* and *Width* is to look at the comments connected with the code listings of module *arith_code*.

The decoding algorithm, runs this whole arithmetic coding operation in reverse. It brings in the encoded data stream up to the limit of *PRECISION* and then figures what *cum_actual* must have been based on the current *cum_total* scaled to the width. This value is then compared to the increasing cumulative totals in the context sequence until a match is found. The *cum_previous* and *cum_actual* values for this character are then used to update the position and width functions (to match the original way it was encoded) and the algorithm moves to the next bits in the input stream. The decoding operation does not need to worry about carries as it is reading the correctly formatted input stream as opposed to trying to write it.

File input to the compression routine and file output from the decompression routine are handled directly using standard C I/O routines; they are just reading and writing the plain text of the files. Input and output of the compressed data, however is handled differently. The compressed data is actually processed a bit at a time rather than a byte at a time. Internal routines read and write data a byte at a time while offering a bit by bit packing or unpacking for the benefit of the arithmetic coding/decoding routines. This forces some special handling to come into play at the end of the data file. When the input data stream ends while compressing a file, there is usually a partly filled byte of data ready for output. It is flushed to the output buffer and then one additional byte is sent to the output buffer indicating the low order byte of the total count of input characters that went into this compressed file. The input routine of the decompression program can then be looking ahead in the file for the last character to know when it has actually completed recreating the original file. This method of decompression termination proved to be more effective than trying to relay which bit in the last actual compressed data byte the file ended on.

2.2.4 Program Enhancements

During the development of the compression and decompression programs, their performance was analyzed to figure out where throughput improvements could be made. As a result, various sections of code were changed or rewritten. Most notable was the optimization of the exclusion list logic and the addition of the special 0th order pointer array. Also, requests for dynamic memory was 'batched'. Rather than ask for enough for a node each time one is needed, enough for a lot of nodes is requested each time more is needed. The program itself then passes out node sized chunks to itself.

The most obvious thing left that would give immediate performance improvement is to remove the subroutine logic and make the programs one big routine. Analysis shows that at least 15% of the CPU time is now spent just setting up and tearing down these linkages. However, in the interest of readability, this step has not been taken. Brief inspection of the generated object code, showed some improvements could be made in rewriting the central loops of the node search and arithmetic coding routines in assembler. In the interest of portability, however, this was not done. Both of these steps would probably be done if this algorithm ever was introduced into an actual hardware communications device.

The memory requirements of a given node were not reduced to their minimum size. One 32 bit integer is used to hold each of the pointers and also the character count at the node. 24 bits would be more than enough for each of these values but the overhead of converting the pointers for C program operation during linked list searching was deemed excessive. This is another improvement that would reduce memory requirements that could be introduced when the program was written in assembler.

Chapter 3
Results

Once the compression program was written and optimized for algorithm performance, it was evaluated for compression performance. The important metrics are its throughput, its memory use, and its compression efficiency. This was investigated for different types of source data and different order Markov models. The basic program (the listings of which are included in this report) was set up without regard to the amount of memory that would be needed to store the model. Large input files (especially with higher order Markov models) can consume large amounts of memory, but the large model also can provide the most effective compression. Evaluation of the compression efficiency and throughput with this mode of operation was used as a baseline against which to compare the algorithm when restrictions on memory use were introduced.

The basic algorithm was run for Markov orders zero through four on eight different types of data files:

1) The source for a short C program fragment
2) The source for a larger C program (the source for the compression program)
3) The source for a large FORTRAN program
4) English Text (this report)
5) An object file (binary format)
6) An executable file (binary format)
7) A screen graphics image (binary format)
8) A random binary file (an already compressed file)

For each of these test runs, data was collected on compression efficiency (input bytes/ output bytes), throughput (input bits/second), and memory used for model storage (number of nodes in Markov model tree; each node takes 13 bytes). The data shown was all collected during the compression operation. In all instances that were examined, the decompression

operation ran as fast (and up to 5% faster) than the compression. The tests were run on a DEC uVAX II system running VMS and the system was otherwise idle. The results appear in the following tables.

Compression Efficiency (Input Bytes/Output Bytes)

Data File	Length (Bytes)	Order 0	1	2	3	4
1)C Program	3073	1.53	1.95	2.11	2.13	2.15
2)C Program	32296	1.80	2.88	3.78	4.16	4.20
3)FORTRAN Pgm	97551	1.75	3.13	4.85	5.73	5.94
4)English Text	50214	1.88	2.64	3.18	3.63	3.65
5)Object File	4236	1.16	1.25	1.29	1.29	1.29
6)Exec. File	51712	1.27	1.40	1.44	1.44	1.44
7)Screen Image	32768	1.53	2.62	2.60	2.55	2.51
8)Random Data	7756	.97	.71	.70	.70	.70

Compression Throughput (Input Bytes/Second)

Data File	Length (Bytes)	Order 0	1	2	3	4
1)C Program	3073	878	1024	1024	1024	768
2)C Program	32296	922	1108	1009	879	807
3)FORTRAN Pgm	97551	985	1108	1096	985	870
4)English Text	50214	929	1024	965	865	772
5)Object File	4236	605	564	498	498	470
6)Exec. File	51712	601	517	457	427	383
7)Screen Image	32768	668	885	744	655	595
8)Random Data	7756	456	235	221	209	204

Nodes Created in Model Tree (13 Bytes/Node)

Data File	Length (Bytes)	Order 0	1	2	3	4
1)C Program	3073	85	693	1863	3375	5117
2)C Program	32296	89	1203	4639	10744	19269
3)FORTRAN Pgm	97551	92	1740	7835	18669	33665
4)English Text	50214	82	1214	5404	14305	28649
5)Object File	4236	214	1935	4671	7888	11355
6)Exec. File	51712	256	10633	34800	66502	102125
7)Screen Image	32768	182	2384	9369	21633	38189
8)Random Data	7756	256	7565	15315	23068	30820

Some interesting information can be found in this set of performance figures. As would be expected, data that has more redundancy (like text and source programs) compresses more efficiently than object or executable file. The files that constitute source for computer programs also compress better than normal English text. This is also to be expected as any language imposes certain redundant structure that the compression algorithm can exploit.

The higher the compression ratio, the lower relative number of nodes (as compared to the input file size) are required to save the model in the Markov tree. This is just the reflection of having more successful 'hits' when making character predictions at the higher order levels.

Files that really do not have much redundancy do not benefit much from higher order models and require large resource allotments to run. The object file and executable file gain little in compression efficiency above the order 1 model and start taking up lots of CPU time and lots of memory with the higher order models. The worst case of this is the compression attempts on the 'random file' which is just an already compressed file. It really has no redundancy and the compression algorithm ends up making it grow in its attempts to find some.

The compression throughput is directly related to how long the linked lists are that need to be searched with each character encoding. For files that compress well, the compression throughput does not drop very fast as one increases the Markov order. This is due to the fact that the higher order linked lists are relatively short. The actual increase in throughput going from the order 0 to the order 1 model is an artifact of the encoding system. At which ever level a character is first found in the Markov model, the cumulative statistics must be figured for it and this means marching along the Markov linked list up to the character in question. This means covering half the list on average. For example, for a Markov model 0, this means covering approximately 45 nodes for file 2) in the tables above. For Markov model 1, it may only require covering 5 nodes and the level 0 statistics update uses the very fast array of order 0 pointers. The end result being that the order 1 model runs significantly faster than the order 0 one.

Overall, the order 3 model seems to give the best performance relative to its memory and CPU requirements.

For comparison purposes, the compression ratios of two other compression techniques available on the VAX computer were benchmarked with the same input data files. A program was written to do run length encoding(rle) for one of the methods. The other method was implemented by writing a program to call the VMS compression utility routines which use a method of Huffman encoding(huff). The results are as follows:

Data File	Length (bytes)	huff effic	huff speed	rle effic	rle speed
1) C Program	3073	.64	2250	1.05	4670
2) C Program	32296	1.12	" "	1.37	" "
3) FORTRAN Pgm	97551	1.26	" "	1.50	" "
4) English Text	50214	1.04	" "	1.15	" "
5) Object File	4236	1.00	" "	1.02	" "
6) Exec. File	51712	1.22	" "	1.18	" "
7) Screen Image	32768	1.41	" "	1.01	" "
8) Random Data	7756	.94	" "	1.00	" "

The speed of both these methods reflects the simple processing involved in their encoding as compared to arithmetic coding with Markov statistics modeling. The Huffman encoding does require two passes through the data however; one to establish a statistical representation of the file and one to actually encode it. This statistical representation is then included in the output file so that the data can be correctly decompressed. The efficiency of

the Huffman encoding using the VMS utility routines is worse than a specially written Huffman encoding application would be. This is due to the way the utilities implement the encoding on a record by record basis as opposed to a file basis. Each record is passed to the utilities and compressed and then written to an output file. This adds overhead bytes specifying record length to every record. While actual data compression approached 2 to 1 in some instances, the overhead of these record lengths along with the statistical representation of the model that needed to be written to the output file, actual end result compression did not even reach 1.5 to 1.

Once the performance of the basic arithmetic encoding compression system was established, there were two main areas that were investigated.

- The early detection of files that do not compress.

  The performance of the model when the use of dynamic memory was restricted.

In comparing the compression efficiency curves as the input data file is processed, the behavior of the 'incompressible file' stands out. The graph in Figure 4 shows the compression efficiency for an order 3 model for four different file types. As a very crude

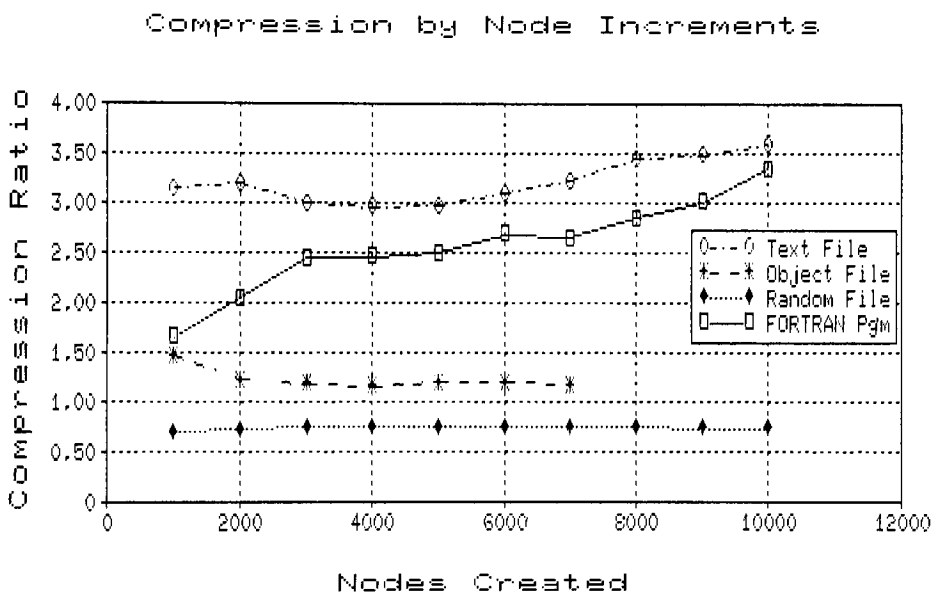


Figure 4: Compression Efficiency

safeguard, the program was updated to look at compression efficiency after the first thousand nodes were added to the model. If the efficiency is less than one (i.e., it is not compressing yet) then it probably will not compress and the program will end without forming an output file. This would be the equivalent of a communication device going into a direct transfer state of passing the data through itself without modification.

There are many possible variations to try in looking at the effects of restricting memory use on the compression program. The main object was to impose memory restrictions without significantly impacting throughput by excessive additional algorithm complexity. Three main methods for doing this were investigated.

The first method tried was one that would just reset the Markov tree back to an empty tree each time a certain node count was reached. This method was easy to implement and adds minimal overhead. An arbitrary node reset limit of 10,000 was chosen and the following performance data was collected (all for an Order 3 model):

Compression Efficiency

File	Bytes	Total Nodes	At 1st Reset	At 2nd Reset	At 3rd Reset	Org Model
FORTRAN Pgm	97551	29553	3.34	4.39	4.94	5.73
English Text	67408	24397	3.61	3.56	3.47	3.63

While compression efficiency has dropped from the original model, it is still remarkably good. What is being lost by restarting the statistics every 10000 nodes, is being partlally gained backed by the fact that the statistics more closely match the section of the file that is being processed. This method provides important benefits when the input file is not of a homogeneous nature.

Another method of memory restriction is just to lock the Markov model when a certain node count is reached; do not add any more nodes and do not update the counts of any existing nodes. This method is also relatively easy to implement and adds only a small amount of overhead to algorithm processing. The basic program was modified to this configuration, the arbitrary node count of 10,000 was again chosen, and the following data was collected:

Compression Efficiency

File	Bytes	Input Character count At 10000 nodes	Final Compression	Org Model
FORTRAN Pgm	97551	18996	3.89	5.73
English Text	67408	31144	3.41	3.63

This method does not perform as well as the 'reset' method especially for the FORTRAN program. In investigating the program, it is found to have a very long comment/ explanation section at the beginning (more like straight English text than source code) before the actual code starts. It is this section of the file that sets the baseline for the model and since the model gets fixed early on, it can not adapt to the later changes in form. This explains the sharp drop in efficiency. The English text is relatively consistent in form throughout and suffers much less when the model is fixed.

A variation on the above method fixes the node growth but continues to update the counts if characters are found in contexts that already exist in the table. This method was somewhat more difficult to implement and begins adding some significant processing overhead to the model (about 30% increase). This overhead is related to the disruption of the organization of the Markov tree when only the node counts are updated of nodes that exist in the table. The 'count' field in each node can no longer be used as the true count of occurrences that exist at the next higher level. Instead, each node list must be followed in its entirety to determine the occurrence count at this level. This effectively doubles the list processing required. The performance of this method with the same 10,000 node limit was as follows:

Compression Efficiency

File	Bytes	Input Character count At 10000 nodes	Final Compression	Org Model
FORTRAN Pgm	97551	18996	3.92	5.73
English Text	67408	31144	3.42	3.63

The results from this method are only marginally better that the strict fixed method and certainly do not warrant the additional processing required.

Chapter 4
Implementation Considerations

The compression algorithm as implemented on the VAX provides important insights as to performance and resource requirements. Any Implementation of this algorithm into a communication device would need to support these requirements in order to provide effective compression. The basic block diagram of how such a device could look is shown in Figure 5. Whether a general purpose microprocessor or specialized VLSI circuitry is used, the performance considerations are the same. The processor must be fast enough to effectively feed the output data stream at the data rate at which it runs and the input and output data rates must be appropriately balanced to provide the most benefit. For example, consider a communications device that attaches to a CPU at one end at 9600 bps and wants to drive a communications line at 4800 bps at the other. The CPU needs to process about 1000 characters per second to keep up with the input stream and needs to provide about 500 characters per second to feed the output stream. Given that the compression ratio of the implemented algorithm is usually greater than two, this input/output ratio will often cause output starvation. While the compression ratio will stay intact, the actual data transmission rate will fall because of idle output cycles.

A better match on input to output data rate would be four to one; e.g. 9600 bps and 2400 bps. More complete advantage will then be taken of the power of the model to compress the data. A general purpose processor with the integer instruction speed of a VAX could handle this input speed of 9600 bps assuming a model of order 3 or less was implemented. Higher input speeds would require faster, or more likely, special purpose processors, that could handle the proportionally greater data rates. The two largest contributors to CPU usage in the algorithm are the arithmetic coding loop itself and the link list searching required for character matching. Implementation of these two aspects in some specialized device would have a tremendous return in effective throughput. (Note: all these speed calculations have been done based on data moving in one direction at a time; i.e.,

half duplex mode. A more realistic general purpose device would need to support full duplex operations. This would require a processing unit to handle each direction of data transfer or have the single processing unit be only able to run at effectively half speed during full duplex operations.)

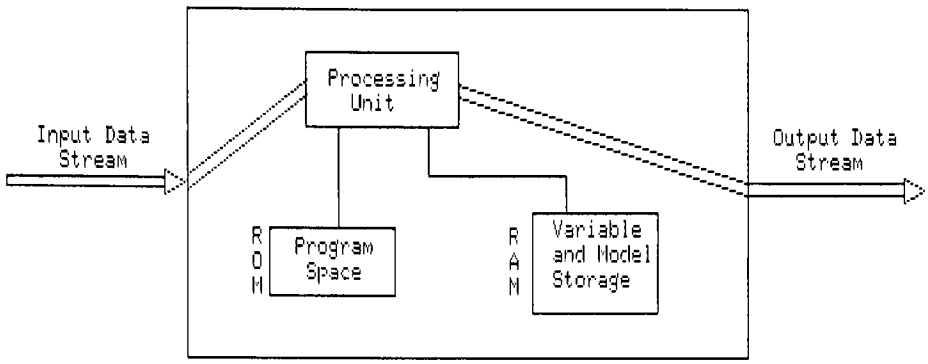


Figure 5: Hardware Block Diagram

The memory requirements of the implementation of this algorithm come in two categories; memory for program storage and memory for model storage. The algorithm itself is relatively compact and should be able to be stored in 8K or less of Read Only Memory (ROM) once it is converted to a stand alone assembler module. The model storage memory requires all uninitialized RAM and as can be seen in the 'Results' section, can require lots of it. The results also show that very good compression can be achieved with models of order 3 and below and with limiting node growth to less than 20,000 nodes. By accepting these restrictions and compressing an individual node accordingly (space needed to hold both pointers and counter is reduced), 256K of RAM is more than enough space to hold a model during transfer.

Other aspects beyond the basic processing/memory requirements also need consideration before the compression algorithm could realistically be implemented into a communication device.

Arithmetic coding requires a noise free path for proper decompression to occur. Because the entire output stream is really just one encoded 'character', an error in any bit will cause the rest of the data stream to be garbled. Communication links are very prone to having errors, so some packet level protocol must be introduced to provide error detection and resynchronization capabilities. Many current modems use a packet protocol to insure error free transmission over standard telephone lines. A protocol like this could be combined with the arithmetic coding compression system to insure proper transmission and subsequent decompression.

In any fixed sized input to variable length output encoding method, the problems of buffer overflow and buffer exhaustion must be addressed (Jelinek, 1968). Buffer overflow will occur if input data arrives at the device faster than it can process and/or retransmit it. Putting a fast enough processor in the device will keep the problem from being caused by processor saturation. There still can be a problem caused by the ratio of input to output data as a result of the compression. Consider the example of a device with a 9600 bps serial input channel, a 2400 bps serial output channel, and a file being compressed at a ratio of 2:1. This means that on the average, to keep the data flowing through the device in a constant stream, an actual output channel speed of 4800 bps would be required. As this is not available, data will accumulate in the device, very quickly causing an overflow condition. This problem can be overcome with the simple addition of either an XON/XOFF protocol or handshaking with the CTS line to throttle the input data stream.

Buffer exhaustion occurs when there is no data to send from the device to the output data stream. This is a normal occurrence on an idle channel but causes a drop in effective data transfer rate if it occurs during an actual transmission. The arithmetic coding algorithm implemented for this project has the potential for causing unnecessary occasional buffer exhaustion because of the way it 'holds on' to parts of the output bit stream to overcome the 'carry over' problem. While this provides the best compression, using a bit stuffing method as suggested by Langdon (1984) may actually give a more effective data transfer rate in a communications device. This method sends data to the output data stream as it is generated and inhibits carry propagation with a zero bit insertion following long strings of sequential ones.

Data compression using arithmetic coding provides an important side benefit of limited data encryption (Cleary and Witten, April 1984). Surely, the compressed file is unintelligible to the casual observer and in some instances may provide all the security that is needed. If more security is needed and the data must also be passed through an encryption algorithm, the compressed data will be encrypted more effectively than the original plaintext (Highland, 1986). This is because the compression has removed the redundancy from the original message stream. There is no set of symbols in the compressed data that directly corresponds to a set in the original (as the encoding probabilities are always changing) so plaintext attacks on the encrypted data will be much more difficult.

Chapter 5
Conclusions

The implementation of the compression/decompression programs based on arithmetic coding and Markov modelling proved to be very successful. They operate effectively on a wide range of data types and provided important insights into the operations and the requirements of the compression method relative to its applicability to communication channel use. Beyond that, the program pair is also an effective file compression utility suitable for general purpose use.

The actual likelihood of this compression method being implemented into a communications device would probably be based on the economics of the situation. It clearly has the performance to be very effective. Its small memory requirements for the actual program are countered by the large requirements in RAM space needed in which to build the MARKOV model. The processing CPU needs to be relatively fast, but it does not need very sophisticated instructions (shifts and following pointer chains comprise the bulk of the processing.) Assuming the necessary volumes, this application lends itself to custom VLSI circuitry for both cost reduction and performance improvement.

This project followed relatively close to expectations and to the plan as specified in the Proposal with the following notable exceptions. The amount of code necessary for the implementation of the project was slightly shorter than anticipated by about 15%. At the same time, the implementation of the Markov context matching proved to be more complex than expected. The actual compression throughput proved to be far in excess of what was hoped for. The results reported by Cleary (April 1984) showed throughputs of 20-100 characters per second (up through models with a Markov order of four). The Proposal stated that program/algorithm optimization would produce sustainable compression at 100 characters per second. Actual results produced compression at rates from 200 1100 characters per second. This miscalculation on expected performance is a combination of overestimating the efficiency of Cleary's implementation and being overly cautious in the efficiency of my own. Whatever, the error was definitely in the right direction!

Chapter 5 Conclusions

The compression method has two bothersome shortcomings. First, if random 8-bit data (like an already compressed file) is passed through it, it does not compress at all; in fact it ends up expanding the data. A couple of methods for detecting and correcting for this were tried but they really did not provide satisfactory solutions. The other shortcoming involves the need for extended precision in a few of the calculations involved with arithmetic coding. Due to the choice of using actual cumulative totals for determining character probability, the precision required in calculations grows with the size of the file being compressed. The current implementation is limited to files up to the size of 2^{21} characters (2,097,152). To expand this, full 64 bit integer math must be implemented, the overhead of cumulative averaging must be introduced, or periodic resetting of the statistics must be employed.

Chapter 6
Future Extensions

The most obvious extension to this project is to actually go ahead and implement the compression/decompression algorithms into micro-processor based hardware devices that could be put into a serial communications channel. Positioning this device between a CPU and a modem or a terminal and a modem could provide substantial throughput gains. Actual integration into a modem would be the most attractive packaging.

Another path for future investigation is to look at various specializations on the basic generic compression algorithm. It currently receives a serial data stream and only exploits serial redundancies. Digital images (on which the generic algorithm has limited compression effectiveness) have two dimensional redundancies. By using a concept of line width and picture element 'neighborhood', much more effective compression would be possible.

The encryption aspects of this model could be investigated in detail. With the possible priming of the node tree as a key, just how effective could this compression algorithm be as an encryption method?

The compression method presented in this project is intolerant of errors in the data stream. A dropped or wrong bit will most likely cause all the following data to be interpreted incorrectly. A method of self-synchronization (along the lines of the work of Ferguson and Rabinowitz (1984) with Huffman Codes) could be developed that added minimal overhead yet still provided synchronization at regular intervals.

Finally, the algorithm as presented, does not compress a data stream that is already 'random'; i.e., trying to compress an already compressed file. In fact, it causes the resultant output file to be longer. An algorithm to optimally detect this condition during compression and back-off on the order of the model could be developed.

Bibliography

Cleary, J.G. and Witten, I.H., "A Comparison of Enumerative and Adaptive Codes," *IEEE Trans. Inform. Theory,* vol. IT-30, no. 2, pp. 306-315, March 1984.

Cleary, J.G. and Witten, I.H., "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Trans. Commun.,* vol.COM-32 , no. 4 ,pp. 396-402, April 1984.

ed. Davisson, L.D. and Gray, R.M., *Data Compression (Bench Mark Papers in Electrical Engineering and Computer Science, V14).* Stroudsburg,Pennsylvania: Dowden,Hutchinson, and Ross,Inc, 1976.

Ferguson, T.J. and Rabinowitz, J.H., "Self-Synchronizing Huffman Codes," *IEEE Trans. Inform. Theory*, vol. IT-30, no. 4, pp. 687-693, July 1984.

Gallager, R.G., "Variations on a Theme by Huffman," *IEEE Trans. Inform. Theory*, vol. IT-24, no. 6, pp.668-674, Nov. 1978.

Golomb, S.W., "Run-Length Encodings," *IEEE Trans. Inform Theory*, pp.399-401, July 1966.

Guazzo, M., "A General Minimum-Redundancy Source-Coding Algorithm," *IEEE Trans. Inform. Theory*, vol. IT-26, no. 1, pp. 15-25, January 1980.

Held, G., *Data Compression.* New York:Wiley Heyden, 1983.

Highland, H.J., "Don't Overlook Data Compression," *Computers and Security,* vol. 5, no. 1, pp. 8-9, March 1986.

Huffman, D., "A Method for the Construction of Minimum Redundancy Codes", *Proc. IRE*, vol. 40, pp.1098-1101, Sept. 1952.

Jelinek, F., "Buffer Overflow in Variable Length Coding of Fixed Rate Sources," *IEEE Trans. Inform. Theory*, IT-14(3), pp. 490-501, 1968.

Bibliography

Jones, C.B., "An Efficient Coding System for Long Source Sequences," *IEEE Trans. Inform. Theory*, vol. IT-27, no. 3, pp. 280-291, May 1981.

Langdon,Jr., G.G., "An Introduction to Arithmetic Coding," *IBM J. Res. Develop.*, vol. 28, no. 2, pp. 135-149, March 1984.

Langdon,Jr., G.G. and Rissanen, J., "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Commun*, vol. COM-29, no. 6, pp. 858-867, June 1981.

Langdon,Jr., G.G. and Rissanen, J., "A Double-Adaptive File Compression Algorithm," *IEEE Trans. Commun.*, vol. COM-31, no. 11, pp. 1253-1255, November 1983.

Langdon,Jr., G.G. and Rissanen, J., "A Simple General Binary Source Code," *IEEE Trans. Inform. Theory*, vol. IT-28, no. 5, pp.800-803, September 1982.

Lynch, T.J., *Data Compression Techniques and Applications*. Lifetime Learning Publications, 1985.

Nelson, B., "Kermit - A Protocol for Painless Micro and Mini File Transfer", *Network Session Notes*, Spring DECUS Symposium, pp. 296-303, April,1986.

Pechura, M., "File Archival Techniques Using Data Compression," *Comm. of the ACM*, vol. 25, no. 9, pp. 605-609, September 1982.

Pike, J., "Text Compression Using a 4 Bit Coding Scheme," *The Computer Journal*, vol. 24, no. 4, pp. 324-330, 1981.

Reghbati, H.K., "An Overview of Data Compression Techniques," *IEEE Computer*, vol. C-14, pp. 71-75, April 1981.

Rissanen, J., "A Universal Data Compression System," *IEEE Trans. Inform. Theory*, vol. IT-29, no. 5, pp.656-664, September 1983.

Rissanen, J. and Langdon,Jr., G.G., "Universal Modeling and Coding," *IEEE Trans. Inform. Theory*, vol. IT-27, no. 1, pp. 12-23, January 1981.

Rodeh, M., Pratt, V.R., and Even, S., "Linear Algorithm for Data Compression Via String Matching," *J. of the ACM*, vol. 28, no. 1, pp. 16-24, January 1981.

Rubin, F., "Experiments in Text File Compression," *Comm. of the ACM*, vol. 19, no. 11, pp. 617-623, November 1976.

Tanenbaum, A.S., *Computer Networks*. Englewood Cliffs, New Jersey : Prentice-Hall,Inc., 1981.

Welch, T., "A Technique for High-Performance Data Compression," *IEEE Computer*, pp. 8-19, June 1984.

Ziv, J. and Lempel, A., "Compression of Individual Sequences Via Variable-Rate Coding," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 530-536, September 1978.

Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression ," *IEEE Trans. Inform. Theory*, vol. IT-23, no. 3, pp.337-343, May 1977.

Code Listings

The listings for both the compression routine (ac) and the decompression routine (unac) follow.

Code Listings

```
/*
**-------------------------------------------------------------
**------------------- Program ac.c ----------------------------
**-------------------------------------------------------------
**
**  ABSTRACT:
**
**      This program uses Markov Modeling and Arithmetic Coding to implement
**     a single pass data compression program. This program module performs
**     the compression while a companion module (unac.c) does the
**     decompression.
**
**  AUTHORS:
**
**      Jim Robinson        061-42-0880
**
**     The concept for the MARKOV partial string matching comes from
**     a paper by J.G. Cleary and I.H. Witten:
**     "Data Compression Using Adaptive Coding and Partial String Matching"
**     IEEE Trans. On Comm., Vol. COM-32, No. 4, April 1984
**
**     The algorithm for the arithmetic coding comes from a paper by
**     C.B. Jones:
**     "An Efficient Coding System for Long Source Sequences"
**     IEEE Trans. On Info. Theory, Vol. IT-27, No. 3, May 1981
**
**
**  CREATION DATE:      09/29/87
**
**  MODIFICATION HISTORY:
**  ---------------------
**    1.0    10/28/87 - Original Version works!!!
**    1.1    11/04/87 - Optimized use of exclusion[] - 30% gain in performance
**    1.2    11/06/87 - Added order0[] pointer array - 14% gain
**    1.3    11/10/87 - Added count field to exclusion array - 10% gain
**                    - Removed inp_char field from search_node structure - 6%
**    1.4    11/11/87 - Added completion logic to flush last few bits to output
**                      file
**--
**/
/*
**-----------------------------
**------  Include Files -------
**-----------------------------
*/
#include <stdio.h>
#include "ac_def.h"
```

```
/*
**-----------------------------------------------------------------
** The following variables have been chosen to be global to reduce
** parameter passing logic in the program.
**-----------------------------------------------------------------
*/

struct
{
    intoutput_chars;              /* # of Chars to Output file  */
    intnodes_created;             /* # of Char nodes created    */

}       pgm_stats = { 0,0 };

FILE    *ofileptr, *ifileptr;
```

```
/*
**-----------------------------------------------------
**      main(argc,argv)    - Main line function for ac.c
**-----------------------------------------------------
**
**  FORMAL PARAMETERS:
**
**      "ac" should be called with the input and output file specs
**     passed on the comand line.
**
**            ac input.dat output.ac
**
**     The input file name is required. If no output file name is
**     specified then 'input.ac' is used.
**
**-------------------------------------------------------------------
**/
/*----------------*/
  main(argc, argv)
/*----------------*/

int    argc;
char   *argv[];

{
    int    max_order         = 0;
    int    this_char;
    int    loop;
    unsigned char      count_char;
    struct  ch_node     root = {NULL,1,NULL,NULL};
    struct search_node search_spec[MARKOV+2];
    static  struct  ch_node *order_0[256];
    struct  cumulative cums_dummy;

/*-----------------------------------------------------------*/
/*-------- Start of executable ------------------------------*/
/*-----------------------------------------------------------*/
    search_spec[0].match_ptr = &root;
/*
**-----------------------------------------------------------
**  We'll prime the model with the first MARKOV-1 characters
**-----------------------------------------------------------
*/
    if (file_check(argc,argv))
       {
           while (max_order < MARKOV)
              {
                  if ((this_char = fgetc(ifileptr)) != EOF)
                      encode(max_order,search_spec,this_char,order_0);
```

```
                max_order++;
            }
/*
**-------------------------------------------------------------
** This is the main loop of the program now. We'll encode chars
** for the full extent of the MARKOV order until EOF is found.
**-------------------------------------------------------------
*/
          while ((this_char  = fgetc(ifileptr)) != EOF)
              encode(max_order,search_spec,this_char,order_0);
/*
**-------------------------------------------------------------
** We've processed the last character from the input file. All
** that's left to do is to flush out the remaining bits still
** held in the internal registers, close out the files, and
** display some statistics.
**-------------------------------------------------------------
*/
          /*-----------------------------------------------------*/
          /* This causes arith_code to flush its registers       */
          /*-----------------------------------------------------*/
          cums_dummy.total = 0;
          arith_code(&cums_dummy);

          /*-----------------------------------------------------*/
          /* We now must flush the 'send_out' shift register.    */
          /* Pack any unused bits with the '2nd' code.           */
          /* Send out additional bits until the last partial byte */
          /* goes out or until we can decide there was no partial */
          /* byte to go out.                                     */
          /*-----------------------------------------------------*/
          max_order = pgm_stats.output_chars;
          this_char = 2;
          for (loop=0; (loop < 7) && (pgm_stats.output_chars == max_order);
               loop++)
             send_out(this_char);

          /*-------------------------------------------------------*/
          /* Last byte in file will be low byte of total char count  */
          /*-------------------------------------------------------*/
          count_char = (unsigned char)root.count;
          fputc(count_char,ofileptr);
          pgm_stats.output_chars++;

          /*------------------*/
          /* Close and done   */
          /*------------------*/
          fclose(ifileptr);
          fclose(ofileptr);
```

```
            display_stats(&root);
        }
} /*main*/
```

```
/*
**++
**-------------------------------------------------------------------------
**     encode(order,search_spec,this_char,order_0)  - encode function
**-------------------------------------------------------------------------
**
**      This function is called to encode a single character.
**
**  FORMAL PARAMETERS:
**
**      encode(order,search_spec,this_char)
**
**         order       - max MARKOV order to search for
**         search_spec - array of structures defining search
**         this_char   - current character
**         order_0     - special pointer array to order_0 pointers
**
**  IMPLICIT OUTPUTS:
**
**      ofileptr  -  output chars will be written to file
**     pgm_stats -  character counts will be updated
**
**-------------------------------------------------------------------------
**/
/*-----------------------------------------*/
  encode(order,search_spec,this_char,order_0)
/*-----------------------------------------*/

int               order;
struct search_node  search_spec[];
int               this_char;
struct ch_node    *order_0[];
{
    static long       exclusion[257];  /* One for each 8 bit code    */
                                       /* and the last one for count.*/
    static long       ex_val;
    static struct      cumulative cums;
    intencode_switch = FALSE;
    intescape        = TRUE;


    /*-------------------------------------------------------------*/
    /*-- Update the exclusion value and clear the count        --*/
    /*-------------------------------------------------------------*/
    ex_val++;
    exclusion[256] = 0;
    /*-------------------------------------------------------------*/
    /*-- Make one loop for each level of the MARKOV order, going --*/
    /*-- from high to low looking for decreasing length string   --*/
```

```
    /*-- matches.                                                    --*/
    /*-------------------------------------------------------------*/
    for (;  order >= 0;  order--)
        {
           locate (order,search_spec,this_char,&escape,order_0);
           if ((!encode_switch)&&(search_spec[order+1].match_ptr != NULL))
              {
                  cum_stat(search_spec,order,exclusion,ex_val,&cums,escape);
                  arith_code(&cums);
                  if (!escape) encode_switch = TRUE;
              }
           update_tree_stats(search_spec,order,this_char,escape,order_0);
       }
    if (!encode_switch)
        {
           novel_char(&cums,this_char);
           arith_code(&cums);
       }
    search_spec[0].match_ptr->count++;
}/*encode*/
```

```
/*
**-------------------------------------------------------------------------
**     locate(order,search_spec,this_char,escape,order_0) - locate a char
**-------------------------------------------------------------------------
**  This routine looks through the node tree to find the location of
**  character in a particular context (MARKOV order). Search_spec is
**  used retroactively to get a head start on each partial string
**  match and to reduce search time.
**
**  FORMAL PARAMETERS:
**
**      order       - current MARKOV Order to base search on
**     search_spec - array of structures defining context
**     this_char    - character code to match
**     escape      - return flag to indicate whether code was found
**                     TRUE means character not found - need Escape
**                     FALSE means character found    - no Escape
**     order_0     - special array of pointers to the oth order nodes
**
**  IMPLICIT INPUTS:
**
**      The whole node tree defined in dynamic memory
**
**  IMPLICIT OUTPUTS:
**
**  search_spec[order+1] will contain the information of interest from
**     locate. The joint_ptr will just have search_spec[order]'s match_ptr.
**     The match_ptr will contain one of three things:
**     1) a ptr to the matching node if found.
**     2) A pointer to the node before where the matching node would be if
**        other codes are defined on this level but this_char isn't.
**     3) NULL if no other characters are even defined on that level.
**-------------------------------------------------------------------------
**/
/*-------------------------------------------------------*/
  locate(order, search_spec, this_char, escape, order_0)
/*-------------------------------------------------------*/
int    order;
struct search_node    search_spec[];
int    this_char;
int    *escape;
struct ch_node*order_0[];
{
    int    index;
    struct  ch_node    *work_ptr , *last_ptr;

    /*-------------------------------------------------------*/
    /* Everything is bumped by one since we start at 1       */
    /*-------------------------------------------------------*/
```

```
    index = order+1;
    *escape= TRUE;

    /*-------------------------------------------------------*/
    /* Get partial context from search_spec of this char     */
    /*-------------------------------------------------------*/
    search_spec[index].joint_ptr = search_spec[order].match_ptr;
    work_ptr = search_spec[index].joint_ptr->next_level;
    if (work_ptr == NULL)
       search_spec[index].match_ptr = NULL;
    else
       /*-------------------------------------------------------*/
       /* If this is order 0 and the character exists then we   */
       /* have a quick path to it via the order_0 array.        */
       /*-------------------------------------------------------*/
      {
       if ((order == 0) && (order_0[this_char] != NULL))
           {
               search_spec[index].match_ptr = order_0[this_char];
               *escape = FALSE;
           }
       else
           {
              for (last_ptr = search_spec[index].joint_ptr;
                   (work_ptr->code < this_char) &&
                  (work_ptr->next_node != NULL) ;
                  work_ptr = work_ptr->next_node)
                  last_ptr = work_ptr;

              search_spec[index].match_ptr = work_ptr;
              if (work_ptr->code == this_char)
                  *escape = FALSE;
              else
                  if (work_ptr->code > this_char)
                     search_spec[index].match_ptr = last_ptr;
           }
      }
}/*locate*/
```

```
/*
**-----------------------------------------------------------------
**     cum_stat(search_spec,order,exclusion,ex_val,&cums,escape)
**-----------------------------------------------------------------
**  FUNCTIONAL DESCRIPTION:
**
**      This routine generates the cumulative statistics that feed the
**     arithmetic coding algorithm. It generates the statistics based
**     on the current order, excluding characters from the count that
**     are in the exclusion list and have appeared in higher order
**     unsuccessful encodings. If this encoding is for an escape
**     character, then it will be give 1 probability value at the end
**     of the node list.
**
**  FORMAL PARAMETERS:
**
**      search_spec - array of context structures
**     order       - current order we are addressing
**     exclusion    - array of exclusion characters
**     ex_val       - matching value to use in exclsuion array
**     cums        - strauctures to return cumulative stats in
**     escape      - switch saying whether this will be generating an
**                   escape sequence.
**
**  IMPLICIT INPUTS:
**
**      Node tree formed in Dynamic memory.
**-----------------------------------------------------------------.
**/
/*-------------------------------------------------------------*/
  cum_stat(search_spec, order, exclusion, ex_val, cums, escape)
/*-------------------------------------------------------------*/
struct        search_node search_spec[];
int           order;
long          exclusion[];
long          ex_val;
struct        cumulative  *cums;
int           escape;
{
    int             index;
    int             running_total;
    int             running_cum;
    struct  ch_node    *work_ptr;
    register long      excl_work;
    /*-------------------------------------------------------------*/
    /* Initialize offset into search_spec, the starting total count*/
    /* of occurances at this level, and the pointer to the start of*/
    /* the node list.                                              */
    /*-------------------------------------------------------------*/
```

```
index         = order + 1;
running_total = search_spec[index].joint_ptr->count;
work_ptr      = search_spec[index].joint_ptr->next_level;

/*-------------------------------------------------------------*/
/* If this is generating a real character encoding (and not an */
/* escape encoding) then we don't need to continue to update   */
/* the exclusion list. We'll just use what's there to improve  */
/* our probability estimates and form stats from that.         */
/*-------------------------------------------------------------*/
if (!escape)
   {
       /*-----------------------------------------*/
       /* Get current exclusion count             */
       /*-----------------------------------------*/
       excl_work = exclusion[256];

       /*-------------------------------------------------------------*/
       /* Remove each count from total of characters that are on      */
       /* exclusion list and add in each count of characters that     */
       /* are not to running_cum. (up to matching character)        */
       /*-------------------------------------------------------------*/
       for (running_cum = 0;  work_ptr != search_spec[index].match_ptr;
                              work_ptr = work_ptr->next_node)
          {
              if (excl_work == 0)
                 running_cum += work_ptr->count;
              else
                 {
                     if (exclusion[work_ptr->code]==ex_val)
                        {
                            running_total -= work_ptr->count;
                            excl_work--;
                        }
                     else
                        running_cum += work_ptr->count;
                 }
          }
       cums->prev    = running_cum;
       cums->actual   = running_cum + work_ptr->count;
       /*-------------------------------------------------------------*/
       /* Now remove any counts of characters found before that come */
       /* after the character being encoded.                          */
       /*-------------------------------------------------------------*/
       for (;  (work_ptr != NULL) && (excl_work != 0);
              work_ptr = work_ptr->next_node)
          if (exclusion[work_ptr->code] == ex_val)
              {
                 running_total -= work_ptr ->count;
```

```
                    excl_work--;
                }
            cums->total = running_total;
        }
    else
        /*-------------------------------------------------------------*/
        /* Were encoding an escape. This means we'll adjust the total  */
        /* count by the counts found by the exclusion list, and update */
        /* the exclusion list with any new characters.                 */
        /*-------------------------------------------------------------*/
        {
            for (;  work_ptr != NULL;  work_ptr=work_ptr->next_node)
               {
                   if (exclusion[work_ptr->code]==ex_val)
                      running_total -= work_ptr->count;
                   else
                      {
                          exclusion[work_ptr->code] = ex_val;
                          exclusion[256]++;
                      }
               }
            cums->total  = running_total;
            cums->actual = running_total;
            cums->prev   = running_total-1;
        }
}/*cum_stat*/
```

```
/*
**-------------------------------------------------------------
**     arith_code(&cums)     - Arithmetic Code a Character
**-------------------------------------------------------------
**  This routine does arithmetic coding of a character based
**  on cumulative probability. This module follows the alogorithm
**  proposed by C.B. Jones mentioned in the program heading.
**
**  FORMAL PARAMETERS:
**
**       cums - Structure Containing the cumulative statistics for
**              the Character to be encoded.
**
**  IMPLICIT OUTPUTS:
**
**     Values are passed to send_out for output to file.
**-------------------------------------------------------------
**/
/*-----------------*/
  arith_code(cums)
/*-----------------*/

struct cumulative      *cums;

{
    /*---------------------------------------------*/
    /*-- Variables for Arithmetic Coding Proper ---*/
    /*---------------------------------------------*/
    static longcode_size      = 2;
    static longprecision_power = PRECISION;
    static longwidth_func     = PRECISION;
    static longposition_func  = 0;
    static longrun_length     = 0;
    static longcode_subscript = 0;
    static double  work1;
    register long      scale;
    register long      digit_out;
    register long      code_sub_run;
    register long      partial;

    /*-------------------------------------------------------------*/
    /*-- If cums->total is not 0 then process as normal character-*/
    /*-------------------------------------------------------------*/
    if (cums->total != 0)
       {
           /*-------------------------------------------------------*/
           /* Current scale width of the new interval is just the   */
           /* cumulative count of 'occurances' for this call.       */
           /*-------------------------------------------------------*/
```

```
scale        = cums->total;

/*-------------------------------------------------------------*/
/* Calculate lower edge of new interval. Done in double        */
/* precision for needed precision in numerator.               */
/*-------------------------------------------------------------*/
work1         =(2*((double)width_func)*cums->prev + scale)/
               (2*scale);

/*-------------------------------------------------------------*/
/* Resultant quantity fits nicely back into long int           */
/* for calulation of new position value.                       */
/*-------------------------------------------------------------*/
partial      = (long)work1;
position_func = position_func + partial;

/*-------------------------------------------------------------*/
/* Calculate width of new interval. Done in double             */
/* precision for needed precision in numerator.               */
/*-------------------------------------------------------------*/
work1        = ((2*(double)width_func*cums->actual + scale)/
                (2*scale)) - partial;
width_func    = (long)work1;

/*-------------------------------------------------------------*/
/* Expand the new width out to the PRECISION of the encoding */
/* 'catching' the bits that roll off the end so they can be  */
/* output as the encoded bit stream.                           */
/*-------------------------------------------------------------*/
while (width_func < precision_power)
   {
       /*-----------------------------------------------------*/
       /* Get what's fallen of the end                        */
       /*-----------------------------------------------------*/
       digit_out    = position_func / precision_power + 1;

       /*-----------------------------------------------------*/
       /* Shift the width and position left                  */
       /*-----------------------------------------------------*/
       position_func = (position_func % precision_power)
                       * code_size;
       width_func    = width_func * code_size;

       /*-----------------------------------------------------*/
       /* If what fell off represents a binary '1' then we'll*/
       /* just count those up until it chanages.             */
       /*-----------------------------------------------------*/
       if (digit_out == code_size)
          run_length++;
```

```
            else
               {
                   /*-------------------------------------------*/
                   /* If what fell off represents a binary '10'  */
                   /* then we're having a carry operation and    */
                   /* need to flip the states of the code_sub    */
                   /* and code_sub_run.                          */
                   /*-------------------------------------------*/
                   if (digit_out > code_size)
                      {
                          code_subscript++;
                          code_sub_run = 1;
                          digit_out    -= code_size;
                      }
                   else
                      code_sub_run = code_size;
                   /*-------------------------------------------*/
                   /* Send out the old position holder bit      */
                   /*-------------------------------------------*/
                   send_out ( code_subscript );

                   /*-------------------------------------------*/
                   /* Switch to the new position holder bit     */
                   /*-------------------------------------------*/
                   code_subscript = digit_out;

                   /*-------------------------------------------*/
                   /* Send out the run of alike bits            */
                   /*-------------------------------------------*/
                   for (; run_length > 0 ; run_length--)
                      send_out ( code_sub_run );
               }
         }
   }
else
   /*-------------------------------------------------------------*/
   /*-- Special Handling to Dump Partial Functions After Last --*/
    /*-- Character Has been processed.                        --*/
   /*-------------------------------------------------------------*/
   {
       digit_out = position_func / precision_power + 1;
       if (digit_out > code_size)
          {
              code_subscript++;
              code_sub_run = 1;
              digit_out -= code_size;
          }
       else
          code_sub_run = code_size;
```

```
            send_out(code_subscript);
            for (;  run_length > 0;  run_length--)
               send_out(code_sub_run);
            send_out(digit_out);
         }
}/*arith_code*/
```

```
/*
**-------------------------------------------------------------
**         send_out(code) - pass a bit off for output
**-------------------------------------------------------------
**
**      This function is used to add a bit into the output stream
**     and block the bits for character output.
**
**  FORMAL PARAMETERS:
**
**      code - 1 means 1st code (0) and 2 means 2nd code (1)
**
**  IMPLICIT OUTPUTS:
**
**      Output goes to output file defined by ofileptr.
**
**     Global output character count pgm_stats.output_chars will be
**     incremented.
**
**-------------------------------------------------------------------
**/
/*--------------*/
  send_out(code)
/*--------------*/
int    code;
{
    static unsigned char   out_bits = 0;
    static int            out_count= 0;

    /*-------------------------------------------------------------*/
    /*-- Roll the bits left until a char is full and then output */
    /*-------------------------------------------------------------*/
    out_bits <<= 1;
    out_bits  |= code - 1;
    out_count++;
    if (out_count == 8)
       {
           out_count = 0;
           fputc(out_bits,ofileptr);
           pgm_stats.output_chars++;
       }
}/*send_out*/
```

```
/*
**-------------------------------------------------------------------
**     update_tree_stats (search_spec,order,this_char,escape,order_0)
**-------------------------------------------------------------------
**
**      This function is used to update the context of the MARKOV
**     tree after a character has been processed. It will just
**     update the count if the context has occurred before. If
**     the context is novel, a new node will be atached to the
**     tree in the proper spot.
**
**  FORMAL PARAMETERS:
**
**      search_spec - array of structures deining the context
**     order       - current position in search_spec
**     this_char    - current character
**     escape       - flag that if true means character is novel
**     order_0     - specail array of pointers to order 0 nodes
**
**  SIDE EFFECTS:
**
**      If when trying for more dynamic memory, a request fails, this
**     module will abort the program.
**
**     Global Variable 'pgm_stats.nodes_created' will be incremeneted
**     for each node added to the tree.
**-------------------------------------------------------------------
**/
/*-------------------------------------------------------------*/
  update_tree_stats(search_spec,order,this_char,escape,order_0)
/*-------------------------------------------------------------*/
struct search_node     search_spec[];
int                   order;
int                   this_char;
int                   escape;
struct ch_node        *order_0[];
{
    static     char   *mem_ptr;
    static     int    mem_count;
    struct     ch_node *new_node;

    /*-------------------------------------------------------------*/
    /* If character not in novel situation just increment count.     */
    /*-------------------------------------------------------------*/
    if (!escape)
       search_spec[order+1].match_ptr->count++;
    else
    {
       /*------------------------------*/
```

```
/* Get memory for new node first */
/*-----------------------------*/
if (mem_count == 0)
    {
       mem_ptr = malloc(MEM_CHUNK * sizeof(struct ch_node));
       if (mem_ptr == NULL)
           {
              printf("ran out of memory...");
              exit();
           }
       mem_count = MEM_CHUNK;
    }
new_node = (struct ch_node *) mem_ptr;
mem_ptr += sizeof(struct ch_node);
mem_count--;
pgm_stats.nodes_created++;
/*-------------------------------------------------------*/
/* Attach new_node to tree and initialize its values     */
/*-------------------------------------------------------*/
new_node->code       = this_char;
new_node->count      = 1;
new_node->next_level = NULL;

/*-------------------------------------------------------*/
/* If this is an order 0 insertion - must enter node     */
/* address in order_0.                                   */
/*-------------------------------------------------------*/
if (order == 0)
    order_0[this_char] = new_node;

/*-------------------------------------------------------*/
/* If this situation is unique, then we need to add this */
 /* character to next level.                              */
/*-------------------------------------------------------*/
if (search_spec[order+1].match_ptr == NULL)
    {
       new_node->next_node = NULL;
       search_spec[order+1].joint_ptr->next_level= new_node;
       search_spec[order+1].match_ptr            = new_node;
    }
/*-------------------------------------------------------*/
/* If other characters already exist on this level then  */
/* we can just add the character into the chain.         */
/*-------------------------------------------------------*/
else
    if (search_spec[order+1].joint_ptr
           != search_spec[order+1].match_ptr)
       {
           new_node->next_node  =
```

```
                        search_spec[order+1].match_ptr->next_node;
                    search_spec[order+1].match_ptr->next_node = new_node;
                    search_spec[order+1].match_ptr            = new_node;
                }
            else
                /*-------------------------------------------------------*/
                /* Special case where we're adding character to start  */
                /* of existing node chain.                               */
                /*-------------------------------------------------------*/
                {
                    new_node->next_node  =
                          search_spec[order+1].match_ptr->next_level;
                    search_spec[order+1].joint_ptr->next_level= new_node;
                    search_spec[order+1].match_ptr            = new_node;
                }
    }
}/*update_tree_stats*/
```

```
/*
**-----------------------------------------------------------------------
**     novel_char(cums, this_char)
**-----------------------------------------------------------------------
**
**  This routine is called when a completely novel character is found.
**  It generates cumulative statistics based equal probability of each
**  not yet found character of occurring.
**
**  FORMAL PARAMETERS:
**
**      cums       - structure to put cumulative statistics into
**     this_char   - the character to encode
**
**-----------------------------------------------------------------------
**/
/*---------------------------------------*/
  novel_char(cums, this_char)
/*---------------------------------------*/
struct cumulative  *cums;
int    this_char;
{
    static  unsigned char       novel[257];  /* One byte for each char code */
                                             /* and last one for count      */
    int                     subscript;
    int                     found;

    /*-------------------------------------------------------------*/
    /* Count Number of characters ahead of this one we've already  */
    /* seen before.                                                */
    /*-------------------------------------------------------------*/
    found = 0;
    for (subscript=0;  subscript < this_char;  subscript++)
       if(novel[subscript] == '1') found++;

    /*-------------------------------------------------------------*/
    /* Set this character as found                                 */
    /*-------------------------------------------------------------*/
    novel[subscript] = '1';

    /*-------------------------------------------------------------*/
    /* Set cumulative stats and increment total count              */
    /*-------------------------------------------------------------*/
    cums->prev = subscript - found;
    cums->actual = cums->prev + 1;
    cums->total = 256 - novel[256];
    novel[256]++;
}/*novel_char*/
```

```
/*
**++
**-------------------------------------------------------------
**--   file_check(argc,argv) - Validate/open files
**-------------------------------------------------------------
**
**      This function is used to validate and open the input and output
**     files for the program.
**
**  FORMAL PARAMETERS:
**
**     argc - count of items on the command line
**     argv - array of ptrs to character strings from command line
**
**  OUTPUTS
**
**      If an error occurs while opening the files an error message will
**     go to stderr.
**
**  RETURN VALUES
**
**      TRUE  means input and output files were opened successfully
**     FALSE means input and output files were not opened successfully
**            and the program should end.
**
**-------------------------------------------------------------------------
**/
/*---------------------*/
  file_check(argc,argv)
/*---------------------*/

int    argc;
char   *argv[];
{
    intstatus = FALSE;
    int index = 0;
    char out_file[80];
/*
**-------------------------------------------------------
** Choose processing based on # of command line arguements
**-------------------------------------------------------
*/
    switch (argc)
    {

    case 1:
       {
           printf("ac:No File Specifications");
           break;
```

```
        }
    case 2:
        {
            ifileptr = fopen( argv[1] , "r");
            if (ifileptr == NULL)
               perror("ac:Input File");
            else
               {
                   for (;
                        (*argv[1] != '.') && (*argv[1] !='\0');
                        *argv[1]++,index++)
                      out_file[index] = *argv[1];
                   out_file[index]   = '\0';
                   strcat ( out_file , ".ac");
                   ofileptr = fopen(out_file, "w");
                   if (ofileptr == NULL)
                       perror("ac:Output File");
                   else status = TRUE;
               }
            break;
        }
    case 3:
        {
            ifileptr = fopen( argv[1] , "r");
            if (ifileptr == NULL)
               perror("ac:Input File");
            else
               {
                   ofileptr = fopen(argv[2], "w");
                   if (ofileptr == NULL)
                       perror("ac:Output File");
                   else status = TRUE;
               }
            break;
        }
    default:
        printf("ac:Too many command line arguments");
    }
    return (status);
} /*file_check*/
```

```
/*
**-------------------------------------------------------------
**        display_stats(root)     - Display summary statistics
**-------------------------------------------------------------
**
**      display_stats is used to display some overall program statistics
**     at the end of a compression run.
**
**  FORMAL PARAMETERS:
**
**      root - structure of root node
**
**  IMPLICIT INPUTS:
**
**      The statistics have been collected in the global structure
**                          pgm_stats.
**
**
**-------------------------------------------------------------
**/
display_stats(root)
struct ch_node       *root;
{
    printf("*********************************\n");
    printf("  Input Character Count = %d\n",root->count-1);
    printf("  Output Character Count= %d\n",pgm_stats.output_chars);
    printf("  Compression Ratio     = %.2f\n",
           (float)(root->count-1) / pgm_stats.output_chars);
    printf("  Node count            = %d\n",pgm_stats.nodes_created);
    printf("*********************************\n");
}/*display_stats*/
```

```
/*
**    ---------------------------
**    Include File   -  ac_def.h
**    ---------------------------
**
**    This include file contains common constant and structure definitions
**    for module 'ac.c' .
**
*/

#define       TRUE   1
#define       FALSE  0
#define       NULL   0
#define       MARKOV 2              /*Maximum Markov Order Model is
                                      allowed to grow to            */
#define       MEM_CHUNK 1000        /*Number of nodes to allocate at
                                      once when a piece of dynamic
                                      memory is requested           */

/*-------------------------------------------------------------------------*/
/* PRECISION indicates the precision of the arithmetic encoding algorithm.  */
/* This number must be a power of 2 and is used to scale encoding intervals.*/
/* In general, this number ,must be greater than the number of characters   */
/* in the file to be encoded. The current encoding algorithm uses math      */
/* with a calculation limit of about 15 decimal digits. This means for      */
/* this application, PRECISION can not be made larger than 2097192 without  */
/* risk of undetected arithmetic overflow. The two lines that risk the      */
/* overflow appear in routine 'arith_code' and are highlighted in the       */
/* comments.                                                                */
/*-------------------------------------------------------------------------*/
#define       PRECISION 2097152      /* This is 2^21 */

struct ch_node                      /* Character node of MARKOV tree*/
   {
       unsigned char  code;
       int           count;
       struct ch_node*next_node;
       struct ch_node*next_level;
   };

struct search_node                   /*Search node of pointers*/
   {
       struct ch_node*joint_ptr;
       struct ch_node*match_ptr;
   };

struct cumulative                    /*Hold Cumulative AC stats here*/
```

```
{
    int     total;
    int     prev;
    int     actual;
};
```

```
/*
**-------------------------------------------------------------
**------------------- Program unac.c ------------------------
**-------------------------------------------------------------
**
**  ABSTRACT:
**
**      This program uses Markov Modeling and Arithmetic Coding to implement
**     a single pass data decompression program. This program module performs
**     the decompression of data that has been compressed by the companion
**     module ac.c.
**
**  AUTHORS:
**
**      Jim Robinson        061-42-0880
**
**     The concept for the MARKOV partial string matching comes from
**     a paper by J.G. Cleary and I.H. Witten:
**     "Data Compression Using Adaptive Coding and Partial String Matching"
**     IEEE Trans. On Comm., Vol. COM-32, No. 4, April 1984
**
**     The algorithm for the arithmetic coding comes from a paper by
**     C.B. Jones:
**     "An Efficient Coding System for Long Source Sequences"
**     IEEE Trans. On Info. Theory, Vol. IT-27, No. 3, May 1981
**
**
**  CREATION DATE:      09/29/87
**
**  MODIFICATION HISTORY:
**  ---------------------
**  1.0     10/28/87    - Original Version
**  1.1     11/04/87    - Optimized use of exclusion[]
**  1.2     11/09/87    - Added order_0[] pointer array
**  1.3     11/10/87    - added count filed to exclusion[] and removed
**                        out_char field from search_node structure
**  1.4     11/12/87    - Mods to terminate decoding based on trailing count
**--
**/
/*
**-----------------------------
**------  Include Files -------
**-----------------------------
*/
#include <stdio.h>
#include "unac_def.h"

/*
```

```
**-------------------------------------------------------------
** The following variables have been chosen to be global to reduce
** parameter passing logic in the program.
**-------------------------------------------------------------
*/

struct
{
    intinput_chars;              /* # of Chars read from input file */
    intnodes_created;            /* # of Char nodes created    */

}       pgm_stats = { 0,0 };

FILE    *ofileptr, *ifileptr;

int hold_bits;                   /*Input character buffer words*/
int hold_bits1;

int stop_mask = 0x1;              /*Match mask for stopping decoding*/
```

```
/*
**-------------------------------------------------------
**      main(argc,argv)    - Main line function for unac.c
**-------------------------------------------------------
**
**  FORMAL PARAMETERS:
**
**      "unac" should be called with the input and output file specs
**     passed on the comand line.
**
**          unac input.ac output.dat
**
**     The input file name is required. If no output file name is
**     specified then 'input.dat' is used.
**
**-------------------------------------------------------------------
**/
/*----------------*/
  main(argc, argv)
/*----------------*/

int    argc;
char   *argv[];

{
    int                     max_order         = 0;
    int                     this_char;
    struct  ch_node           root = {NULL,1,NULL,NULL};
    struct search_node        search_spec[MARKOV+2];
    static  struct  ch_node    *order_0[256];


/*-----------------------------------------*/
/*---- Start of executable      -----------*/
/*-----------------------------------------*/
    search_spec[0].match_ptr = &root;
/*
**-------------------------------------------------------------
**  We'll prime the model with the first MARKOV-1 characters
**-------------------------------------------------------------
*/
    if (file_check(argc,argv))
       {
           while ((stop_mask != (0xFF & root.count)) && (max_order < MARKOV)
              {
                  decode(max_order,search_spec,&this_char,order_0);
                  send_out(this_char);
                  max_order++;
              }
```

```
/*
**-------------------------------------------------------------
** This is the main loop of the program now. We'll decode chars
** for the full extent of the MARKOV order until EOF is found.
**-------------------------------------------------------------
*/
          while (stop_mask != (0xFF & root.count))
             {
                 decode(max_order,search_spec,&this_char,order_0);
                 send_out(this_char);
             }
/*
**-------------------------------------------------------------
** Now We're done. Close out the files and display some summary
** information.
**-------------------------------------------------------------
*/
          fclose(ifileptr);
          fclose(ofileptr);
          display_stats(&root);
       }
} /*main*/
```

```
/*
**++
**--------------------------------------------------------------------
**     decode(order,search_spec,&this_char,order_0)  - decode function
**--------------------------------------------------------------------
**
**      This function is called to decode a single character.
**
**  FORMAL PARAMETERS:
**
**      decode(order,search_spec,&this_char)
**
**         order       - max MARKOV order to search for
**         search_spec - array of structures defining search
**         this_char   - current character to return
**         order_0     - special pointer array to order_0 nodes
**
**  IMPLICIT INPUTS:
**
**     Bits come in from input file (ifileptr)
**
**  IMPLICIT OUTPUTS:
**
**     pgm_stats -  character counts will be updated
**
**--------------------------------------------------------------------
**/
/*-----------------------------------------*/
  decode(order,search_spec,this_char,order_0)
/*-----------------------------------------*/

int               order;
struct search_node  search_spec[];
int               *this_char;
struct ch_node    *order_0[];
{
    static long       exclusion[257];  /* One for each char code and  */
                                       /* the last one for a count    */
    static long       ex_val;
    intcum_total;
    intescape        = TRUE;
    int loop;
    int org_order;

    /*-------------------------------------------------------------*/
    /*-- First, bump back up the ptrs in search_spec              */
    /*-- in preparation of the next character.                   */
    /*-------------------------------------------------------------*/
    for (loop = order+1;  loop > 0;  loop--)
```

```
        {
            search_spec[loop].joint_ptr = search_spec[loop-1].match_ptr;
            search_spec[loop].match_ptr = search_spec[loop].joint_ptr;
        }
    /*-----------------------------------------------------------------*/
    /*-- Bump the exclusion value and clear the count          --*/
    /*-----------------------------------------------------------------*/
    ex_val++;
    exclusion[256] = 0;

    /*-----------------------------------------------------------------*/
    /*-- Make one loop for each level of the MARKOV order, going --*/
    /*-- from high to low looking for decreasing length string   --*/
    /*-- matches until the character is decoded.                 --*/
    /*-----------------------------------------------------------------*/
    for ( org_order = order  ; (order >= 0) && (escape);  order--)
        {
            if (search_spec[order+1].joint_ptr->next_level != NULL)
                {
                    cum_stat(search_spec,order,exclusion,ex_val,&cum_total);
                    unarith_code(search_spec,order,&cum_total,
                                exclusion,ex_val,&escape,this_char);
                }
        }

    /*-----------------------------------------------------------------------*/
    /*-- If character still is not decoded - process as a novel character -*/
    /*-----------------------------------------------------------------------*/
    if (escape)
        {
            unarith_code(search_spec,order,&cum_total,
                        exclusion,ex_val,&escape,this_char);
        }

    /*-----------------------------------------------------------------------*/
    /*-- Now update the tree by inserting the new character into the lists.*/
    /*-----------------------------------------------------------------------*/
    for(;org_order >= 0 ;org_order--)
        update_tree_stats(search_spec,org_order,this_char,order_0);
    search_spec[0].match_ptr->count++;
}/*decode*/
```

```
/*
**-----------------------------------------------------------------------
**     cum_stat(search_spec,order,exclusion,ex_val,&cum_total)
**-----------------------------------------------------------------------
**  FUNCTIONAL DESCRIPTION:
**
**      This routine figures the cumulative total value that feeds the
**     arithmetic coding algorithm. It calculates the value based
**     on the current order, excluding characters from the count that
**     are in the exclusion list and have appeared in higher order
**     unsuccessful encodings.
**
**  FORMAL PARAMETERS:
**
**      search_spec - array of context structures
**     order       - current order we are addressing
**     exclusion    - array of exclusion characters
**     ex_val       - exclusion indicator value
**     cum_total    - cumulative total of counts on this level
**
**  IMPLICIT INPUTS:
**
**      Node tree formed in Dynamic memory.
**-----------------------------------------------------------------------
**/
/*-------------------------------------------------------------*/
  cum_stat(search_spec, order, exclusion, ex_val, cum_total)
/*-------------------------------------------------------------*/
struct search_node search_spec[];
int    order;
long   exclusion[];
long   ex_val;
int    *cum_total;

{
    int              index;
    struct     ch_node*work_ptr;
    register   long   excl_work;

    /*-------------------------------------------------------------*/
    /* Initialize offset into search_spec, the starting total count*/
    /* of occurances at this level, and the pointer to the start of*/
    /* the node list.                                              */
    /*-------------------------------------------------------------*/
    index         = order + 1;
    *cum_total    = search_spec[index].joint_ptr->count;
    work_ptr      = search_spec[index].joint_ptr->next_level;
    excl_work      = exclusion[256];

    /*-------------------------------------------------------------*/
```

```
    /* If a character in the node list appears in the exclusion    */
    /* list then we will remove its character counts from the total*/
    /*------------------------------------------------------------*/
    for (;  (work_ptr != NULL) && (excl_work != 0);
            work_ptr = work_ptr->next_node)
       {
           if (exclusion[work_ptr->code]==ex_val)
              {
                  *cum_total -= work_ptr->count;
                  excl_work--;
              }
       }
}/*cum_stat*/
```

```
/*
**--------------------------------------------------------------------
**     unarith_code(search_spec,order,&cum_total,exclusion,ex_val,
**                  &escape,this_char)
**--------------------------------------------------------------------
**  FUNCTIONAL DESCRIPTION:
**
**      This routine does the actual decoding of a character. Based on
**     the cumulative probability total and the node list minus the
**     exclusion characters, unarith_code determines which character
**     has been specified. If a total value equals the cumulative
**     probability then the encoding denotes an escape character. If this
**     routine is called with order = -1, then special handling comes into
**     play for novel characters.
**
**  FORMAL PARAMETERS:
**
**      search_spec[]   - array of structures defining context of search
**     order          - current MARKOV order of encoding
**     cum_total       - cumulative total value for this encoding
**     exclusion[]     - exclusion list of characters
**     ex_val         - exclusion indicator value
**     escape         - return value to indicate when decoding successful
**     this_char       - value to return decoded character in
**
**  IMPLICIT INPUTS:
**
**      The node tree built in dynamic memory.
**
**
**--
**/
/*-------------------------------------------------------*/
  unarith_code(search_spec, order, cum_total, exclusion,
               ex_val, escape, this_char)
/*-------------------------------------------------------*/
struct search_node    search_spec[];
int           order;
long          exclusion[];
long          ex_val;
int           *cum_total;
int           *escape;
int           *this_char;
{
    static  long    code_size      = 2;
    static  long    position_func   = 0;
    static  long    width_func     = 1;
    static  long    precision_power = PRECISION;
    static  double  work1;
```

```
register long   max_value;
register long   partial;
int           code;
struct  ch_node *work_ptr;
long           running_cum;

/*-------------------------------------------------------------*/
/* First roll in the number of new bits from the input stream  */
/* to match the precision of the encoding.                     */
/*-------------------------------------------------------------*/
while (width_func < precision_power)
   {
       send_in(&code);
       position_func = position_func * code_size + code - 1;
       width_func *= code_size;
   }
/*-------------------------------------------------------------*/
/* Now if order is >= 0 we will do a normal attempt at decoding*/
/* Otherwise, this is a novel character and will use special   */
/* handling.                                                   */
/*-------------------------------------------------------------*/
if (order >= 0)
   {
       /*-----------------------------------------------------*/
       /* First figure cumulative probability value to search*/
       /* for.                                                */
       /*-----------------------------------------------------*/
       work1 = ((double)(*cum_total)*(2*position_func+1)-1)/(2*width_func);
       max_value = (long)work1;
       /*-----------------------------------------------------*/
       /* If this cumulative probability is > cum_total then */
       /* no need to go any further as this indicates an     */
       /* escape. Just put the characters at this level in   */
       /* the exclusion list.                                 */
       /*-----------------------------------------------------*/
       work_ptr = search_spec[order+1].joint_ptr->next_level;
       if (max_value+1 >= *cum_total)
          {
              for(;work_ptr != NULL ;work_ptr = work_ptr->next_node)
                 {
                     if (exclusion[work_ptr->code] != ex_val)
                        {
                            exclusion[work_ptr->code] = ex_val;
                            exclusion[256]++;
                        }
                 }
              work1 = (2*(double)width_func*(*cum_total-1) +
                        (*cum_total))/(2*(*cum_total));
              partial = (long)work1;
```

```
                position_func -= partial;
                work1 = ((2*(double)width_func*(*cum_total)+(*cum_total))/
                            (2*(*cum_total))) - partial;
                width_func = (long)work1;
            }
        else
            /*-------------------------------------------------------*/
            /* Character will be decoded on this level.              */
            /*-------------------------------------------------------*/
            {
                /*-------------------------------------------------*/
                /* Must now march down the node list throwing out  */
                /* exclusion characters and summing up the rest    */
                /* until max_value is exceeded.                   */
                /*-------------------------------------------------*/
                running_cum = 0;
                while (running_cum <= max_value)
                   {
                       for (;  exclusion[work_ptr->code] == ex_val;
                            work_ptr = work_ptr->next_node)
                          ;
                       running_cum += work_ptr->count;
                       if (running_cum <= max_value)
                          work_ptr = work_ptr->next_node;
                   }
                /*-------------------------------------------------*/
                /* We've found the character                       */
                /*-------------------------------------------------*/
                *this_char = work_ptr ->code;
                search_spec[order+1].match_ptr = work_ptr;
                *escape = FALSE;

                work1 = (2*(double)width_func*(running_cum-work_ptr->count)
                          + (*cum_total))/(2*(*cum_total));
                partial = (long)work1;
                position_func -= partial;
                work1 = ((2*(double)width_func*running_cum+(*cum_total))/
                            (2*(*cum_total))) - partial;
                width_func = (long)work1;
            }
    }
else
    /*-----------------------------------------------------------*/
    /* Novel Character decoding comes to here                    */
    /*-----------------------------------------------------------*/
    {
        static  unsigned char    novel[257];
        int                   subscript;

        /*-------------------------------------------------------*/
```

```
            /* Here we need to count characters in the novel list to    */
            /* match our max total.                                     */
            /*----------------------------------------------------------*/
            /*----------------------------------------------------------*/
            /* Total for encoding was just # of characters not seen before*/
            /*----------------------------------------------------------*/
            *cum_total = 256 - novel[256];
            work1 = ((double)(*cum_total)*(2*position_func+1)-1)/(2*width_func);
            max_value = (long)work1;

            /*----------------------------------------------------------*/
            /* Now run through rest of list up to max_value             */
            /*----------------------------------------------------------*/
            running_cum = 0;
            for(subscript=0; running_cum <= max_value; subscript++)
                if (novel[subscript] != '1')
                   running_cum++;
            subscript--;
            /*----------------------------------------------------------*/
            /* Found the character - get it and set no longer novel     */
            /*----------------------------------------------------------*/
            *this_char     = subscript;
            novel[subscript]= '1';
            novel[256]++;

            work1 = (2*(double)width_func*(running_cum-1) +
                           (*cum_total))/(2*(*cum_total));
            partial = (long)work1;
            position_func -= partial;
            work1 = ((2*(double)width_func*running_cum + (*cum_total))/
                        (2*(*cum_total))) - partial;
            width_func = (long)work1;

        }
}/*unarith_code*/
```

```
/*
**-------------------------------------------------------
**         send_in(code) - get a bit for output
**-------------------------------------------------------
**
**      This function is used to bring a bit in from the input stream.
**
**  FORMAL PARAMETERS:
**
**      code - 1 means 1st code (0) and 2 means 2nd code (1)
**
**  IMPLICIT OUTPUTS:
**
**     Global intput character count pgm_stats.input_chars will be
**     incremented.
**
**-------------------------------------------------------------------
**/
/*--------------*/
  send_in(code)
/*--------------*/
int    *code;
{
    static unsigned char   in_bits = 0;
    static int           in_count= 0;

    /*-----------------------------------------------------------*/
    /*-- If no bits left in current byte, we must get the next    */
    /*   byte from the input stream. The input stream is bufferred*/
    /*   through 2 internal holding words so that we can be       */
    /*   looking ahead to the end of file. The last character     */
    /*   written to the file is really the low byte of the total  */
    /*   number of characters that were encoded. We must be sure  */
    /*   to get this character out and use it as the stop mask and*/
    /*   not decode it as part of the normal input stream.        */
    /*-----------------------------------------------------------*/
    if (in_count == 0)
       {
           in_count = 8;
           in_bits  = hold_bits;
           hold_bits= hold_bits1;
           /*-----------------------------------------------------------*/
           /* If we've already found the stop_mask just pad the next char*/
           /*-----------------------------------------------------------*/
           if (stop_mask < 0x100)
              hold_bits1= 0xFF;
           else
              {
                  hold_bits1= fgetc(ifileptr);
```

```
                if (feof(ifileptr))
                   {
                       /*-----------------------------------------*/
                       /* EOF found - get last good char read as the*/
                       /* stop mask.                              */
                       /*-----------------------------------------*/
                       stop_mask = hold_bits;
                       hold_bits = 0xFF;
                   }
                else
                    pgm_stats.input_chars++;

             }
       }

    /*-----------------------------------------------------------*/
    /* The bits were loaded in from right to left so we must     */
    /* check the left most bit first and roll to the left.       */
    /*-----------------------------------------------------------*/
    if (in_bits & 128)
       *code = 2;
    else
       *code = 1;
    in_bits <<= 1;
    in_count--;
}/*send_in*/
```

```
/*
**-------------------------------------------------------
**         send_out(this_code) - write a char for output
**-------------------------------------------------------
**
**      This function is used to output characterstо the output stream.
**
**  FORMAL PARAMETERS:
**
**       this_char - character code for output
**
**-----------------------------------------------------------------
**/
/*-------------------*/
  send_out(this_char)
/*-------------------*/
unsigned char   this_char;
{
    fputc(this_char,ofileptr);
}/*send_out*/
```

```
/*
**-------------------------------------------------------------
**     update_tree_stats (search_spec,order,this_char,order_0)
**-------------------------------------------------------------
**
**      This function is used to update the context of the MARKOV
**     tree after a character has been processed. It will just
**     update the count if the context has occurred before. If
**     the context is novel, a new node will be atached to the
**     tree in the proper spot.
**  FORMAL PARAMETERS:
**
**      search_spec - array of structures deining the context
**     order       - current position in search_spec
**     this_char    - current character
**     order_0     - special array of pointers to order_0 nodes
**
**  SIDE EFFECTS:
**
**      If when trying for more dynamic memory, a request fails, this
**     module will abort the program.
**
**     Global Variable 'pgm_stats.nodes_created' will be incremeneted
**     for each node added to the tree.
**-------------------------------------------------------------------
**/
/*-----------------------------------------------------*/
  update_tree_stats(search_spec,order,this_char,order_0)
/*-----------------------------------------------------*/
struct search_node     search_spec[];
int                   order;
int                   *this_char;
struct ch_node        *order_0[];
{
    static     char   *mem_ptr;
    static     int    mem_count;
    struct     ch_node *new_node, *work_ptr, *last_ptr;
    int       index;

    index = order + 1;

    /*-------------------------------------------------------------*/
    /* If node already located, just increment count.              */
    /*-------------------------------------------------------------*/
    if ((search_spec[index].joint_ptr->next_level != NULL) &&
       (search_spec[index].joint_ptr != search_spec[index].match_ptr))
       search_spec[order+1].match_ptr->count++;
    else
    {
```

```
/*-------------------------------------------------------*/
/* If this situation is unique, then we need to add this */
 /* character to next level.                              */
/*-------------------------------------------------------*/
if (search_spec[index].joint_ptr->next_level == NULL)
    {
       get_node(&new_node,this_char);
       new_node->next_node = NULL;
       search_spec[index].joint_ptr->next_level= new_node;
       search_spec[index].match_ptr            = new_node;
       if (order == 0)
           order_0[*this_char] = new_node;
    }
/*-------------------------------------------------------*/
/* If other characters already exist on this level then  */
/* we need to figure out if this_char exists on this     */
/* level. If it does we just increment its count, other- */
/* wise we insert a new node in the chain.             */
/*-------------------------------------------------------*/
else
    {
       work_ptr = search_spec[index].joint_ptr->next_level;

       /*-----------------------------------------------------*/
       /* If this is order 0 and order_0 has a node pointer  */
       /* for this character then we can get the node address*/
       /* directly out of order_0 and save lots of search    */
       /* time. Otherwise march along the chain looking.     */
       /*-----------------------------------------------------*/
       if ((order==0) && (order_0[*this_char] != NULL))
           work_ptr = order_0[*this_char];
       else
           for (last_ptr = search_spec[index].joint_ptr;
                (work_ptr->code < *this_char) &&
               (work_ptr->next_node != NULL);
                work_ptr = work_ptr->next_node)
               last_ptr = work_ptr;


       /*-------------------------------------------------*/
       /*- Found the character - just inc the count      -*/
       /*-------------------------------------------------*/
       if (work_ptr->code == *this_char)
           {
              work_ptr->count++;
              search_spec[index].match_ptr = work_ptr;
           }
       else
           {
```

```
                    /*-------------------------------------------------*/
                    /* Character doesn't exists - so we need a new node*/
                    /*-------------------------------------------------*/
                    get_node(&new_node,this_char);
                    if (order == 0)
                       order_0[*this_char] = new_node;
                    /*-------------------------------------------------*/
                    /* Handle special case of adding after last node*/
                    /* on the chain.                                 */
                    /*-------------------------------------------------*/
                    if (work_ptr->code < *this_char)
                        {
                            last_ptr = work_ptr;
                            work_ptr = NULL;
                        }
                    /*-------------------------------------------------*/
                    /* Put the character node in the chain           */
                    /*-------------------------------------------------*/
                    if (search_spec[index].joint_ptr != last_ptr)
                       {
                           new_node->next_node       = work_ptr;
                           last_ptr->next_node       = new_node;
                           search_spec[index].match_ptr    = new_node;
                       }
                    else
                    /*-------------------------------------------------------*/
                    /* Special case where we're adding character to start  */
                    /* of existing node chain.                               */
                    /*-------------------------------------------------------*/
                       {
                           new_node->next_node                     = work_ptr;
                           search_spec[index].joint_ptr->next_level= new_node;
                           search_spec[index].match_ptr            = new_node;
                       }
                    }
            }
        }
}/*update_tree_stats*/
```

```
/*
**-------------------------------------------------------------------------
**     get_node(&new_node,this_char)  - get a new node
**-------------------------------------------------------------------------
**
**      This module gets a new node of memory from the dynamic memory
**     pool. If there is no more memory, this routine causes the
**     program to abort.
**
**  FORMAL PARAMETERS:
**
**      new_node - pointer to return address of new node in
**     this_char- value to inset into code field of new node
**
**
**  SIDE EFFECTS:
**
**      If not enough memory - program aborts.
**
**     pgm_stats.nodes_created is incremented for each node granted
**
**--
**/
/*---------------------------*/
  get_node(new_node,this_char)
/*---------------------------*/
int                *new_node;
int                *this_char;

{
    static  char     *mem_ptr;
    static  int     mem_count;

    /*-------------------------------*/
    /* Get memory for new node first */
    /*-------------------------------*/
    if (mem_count == 0)
        {
           mem_ptr = malloc(MEM_CHUNK * sizeof(struct ch_node));
           if (mem_ptr == NULL)
              {
                  printf("ran out of memory...");
                  exit();
              }
           mem_count = MEM_CHUNK;
       }

    *new_node = (struct ch_node *) mem_ptr;
    mem_ptr += sizeof(struct ch_node);
```

```
    mem_count--;
    pgm_stats.nodes_created++;

    /*-------------------------------------------------------*/
    /* Initialize the node's values                          */
    /*-------------------------------------------------------*/
    (*new_node)->code       = *this_char;
    (*new_node)->count      = 1;
    (*new_node)->next_level = NULL;
}/*get_node*/
```

```
/*
**++
**--------------------------------------------------------------
**--   file_check(argc,argv) - Validate/open files
**--------------------------------------------------------------
**
**      This function is used to validate and open the input and output
**     files for the program.
**
**  FORMAL PARAMETERS:
**
**     argc - count of items on the command line
**     argv - array of ptrs to character strings from command line
**
**  OUTPUTS
**
**      If an error occurs while opening the files an error message will
**     go to stderr.
**
**  RETURN VALUES
**
**      TRUE  means input and output files were opened successfully
**     FALSE means input and output files were not opened successfully
**             and the program should end.
**
**--------------------------------------------------------------------------
**/
/*---------------------*/
  file_check(argc,argv)
/*---------------------*/

int    argc;
char   *argv[];
{
    intstatus = FALSE;
    int index = 0;
    char out_file[80];
    int dummy;
/*
**-----------------------------------------------------------
** Choose processing based on # of command line arguements
**-----------------------------------------------------------
*/
    switch (argc)
    {

    case 1:
       {
           printf("ac:No File Specifications");
```

```
        break;
    }
case 2:
    {
        ifileptr = fopen( argv[1] , "r");
        if (ifileptr == NULL)
           perror("ac:Input File");
        else
           {
               for (;
                    (*argv[1] != '.') && (*argv[1] !='\0');
                    *argv[1]++,index++)
                  out_file[index] = *argv[1];
               out_file[index]   = '\0';
               strcat ( out_file , ".dat");
               ofileptr = fopen(out_file, "w");
               if (ofileptr == NULL)
                   perror("ac:Output File");
               else
                  status = TRUE;
           }
        break;
    }
case 3:
    {
        ifileptr = fopen( argv[1] , "r");
        if (ifileptr == NULL)
           perror("ac:Input File");
        else
           {
               ofileptr = fopen(argv[2], "w");
               if (ofileptr == NULL)
                   perror("ac:Output File");
               else
                   status = TRUE;
           }
        break;
    }
default:
    printf("ac:Too many command line arguments");
}
if (status)
    {
        hold_bits = fgetc(ifileptr);
        pgm_stats.input_chars++;
        if (!feof(ifileptr))
           {
               hold_bits1 = fgetc(ifileptr);
               pgm_stats.input_chars++;
```

```
                if (!feof(ifileptr))
                   {
                       stop_mask = 0x100;
                       /*----------------------------------------*/
                       /*- Dump 1st 2 bits - artifact of encoding */
                       /*----------------------------------------*/
                       send_in(&dummy);
                       send_in(&dummy);
                   }
             }
       }
    return (status);
} /*file_check*/
```

```
/*
**-------------------------------------------------------------------
**         display_stats(root)     - Display summary statistics
**-------------------------------------------------------------------
**
**      display_stats is used to display some overall program statistics
**     at the end of a compression run.
**
**  FORMAL PARAMETERS:
**
**      root - structure of root node
**
**  IMPLICIT INPUTS:
**
**      The statistics have been collected in the global structure
**                          pgm_stats.
**
**
**-------------------------------------------------------------------
**/
display_stats(root)
struct ch_node       *root;
{
    printf("*********************************\n");
    printf("  Input Character Count = %d\n",pgm_stats.input_chars);
    printf("  Output Character Count= %d\n",root->count - 1);
    printf("  Compression Ratio     = %.2f\n",
           (float)(root->count - 1) / (pgm_stats.input_chars));
    printf("  Node count            = %d\n",pgm_stats.nodes_created);
    printf("*********************************\n");
}/*display_stats*/
```

```
/*
**    ----------------------------
**    Include File   -  unac_def.h
**    ----------------------------
**
**    This include file contains common constant and structure definitions
**    for  module 'unac.c' .
**
*/

#define       TRUE   1
#define       FALSE  0
#define       NULL   0
#define       MARKOV 2              /*Maximum Markov Order Model is
                                      allowed to grow to           */
#define       MEM_CHUNK 1000        /*Number of nodes to allocate at
                                      once when a piece of dynamic
                                      memory is requested          */
/*-------------------------------------------------------------------------*/
/* PRECISION indicates the precision of the arithmetic decoding algorithm.  */
/* This number must be a power of 2 and is used to scale encoding intervals.*/
/* This number must match the PRECISION parameter used by the encoder.      */
/* In general, this number ,must be greater than the number of characters   */
/* in the file to be encoded. The current encoding algorithm uses math      */
/* with a calculation limit of about 15 decimal digits. This means for      */
/* this application, PRECISION can not be made larger than 2097192 without  */
/* risk of undetected arithmetic overflow. The seven lines that risk the    */
/* overflow appear in routine 'unarith_code' and are highlighted in the     */
/* comments.                                                                */
/*-------------------------------------------------------------------------*/
#define       PRECISION 2097152      /* This is 2^21 */


struct ch_node                      /* Character node of MARKOV tree*/
   {
       unsigned char  code;
       int           count;
       struct ch_node*next_node;
       struct ch_node*next_level;
   };

struct search_node                   /*Search node of pointers*/
   {
       struct ch_node*joint_ptr;
       struct ch_node*match_ptr;
   };
```