Abstract

Data Compression is the process of removing redundancy from data. Dynamic Markov Compression (DMC), developed by Cormack and Horspool, is a method for performing statistical data compression of a binary source. DMC generates a finite context state model by adaptively generating a Finite State Machine (FSM) that captures symbol frequencies within the source message. Traditionally DMC has operated on a binary alphabet creating a FSM of binary transitions between states. The compression of DMC is comparable with PPM, which is one of the best compression methods in use today. However because DMC is a bit-wise modeller, it is considerably slower than PPM and requires large amounts of memory to achieve this compression.

This thesis extends the bit-wise modeller DMC to a finite character alphabet, which promises to improve compression speed. Considerable attention is given to efficient data structures capable of representing individual states of the FSM. While the performance of Character DMC fails to match bit-wise DMC or PPM, a significant improvement in compression speed can be achieved.

Evolving Character DMC to capture variable length phrases was the next approach taken. Several inherent problems involving locating words within each state have surfaced. Again, the comparable compression with PPM was disappointing, and the compression speed was less than half that of Character DMC.

A final implementation attempted to avoid the large resource requirements of DMC. It achieves this by restricting both the length of the source message, and the amount of memory available for the FSM. Reasonable compression is achieved, at very high speeds. Such an implementation is ideal for network packets, or hard disk block compression.
Acknowledgements

I would like to thank my supervisor, Dr. Tim Bell, who provided me with helpful comments and patiently supervised my research.

The staff of the Computer Science department at Canterbury always helpful in resolving any problems. Thanks to my fellow postgraduate students, room 407 wouldn't have been the same without you.

A special thanks to the numerous companies who provided me with financial assistance, its just a pity I had to work so hard for them to get it!

Finally I would like to express my thanks to family and friends who have helped support me while completing this project, "It's nearly finished Dad!".
Contents

1 Data Compression
  1.1 Data compression in history ................................................................. 2
  1.2 Lossy and lossless compression ............................................................... 2
  1.3 Modelling and coding paradigms .............................................................. 4
  1.4 Static, semistatic and adaptive modelling ................................................. 6
  1.5 Modelling techniques ............................................................................. 7
    1.5.1 Context models .............................................................................. 8
    1.5.2 State models ............................................................................... 9
    1.5.3 Grammar models ......................................................................... 10
    1.5.4 Dynamic markov modelling .......................................................... 11
  1.6 Information theory .................................................................................. 12
  1.7 Coding .................................................................................................... 13
    1.7.1 Arithmetic coding ....................................................................... 14
    1.7.2 Coding novel symbols ................................................................ 15
  1.8 Other compression techniques ............................................................... 16
  1.9 Summary ................................................................................................ 17

2 Introduction to DMC .................................................................................. 18
  2.1 Cloning – the power of DMC ................................................................. 19
    2.1.1 Deciding when to clone a state .................................................... 22
    2.1.2 Original examples ....................................................................... 23
  2.2 Initial state models .................................................................................. 24
  2.3 Self referencing transitions .................................................................... 26
  2.4 Low memory operation .......................................................................... 26
  2.5 DMC – a variable order context modeller .............................................. 28
  2.6 Empirical comparison of compression methods ...................................... 28
  2.7 Previous research involving DMC .......................................................... 29

3 Binary DMC .............................................................................................. 32
  3.1 Algorithm description ......................................................................... 32
  3.2 Representation of the state model .......................................................... 33
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1 Pointer based models</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Array based models</td>
<td>34</td>
</tr>
<tr>
<td>3.2.3 Extending the array model</td>
<td>36</td>
</tr>
<tr>
<td>3.2.4 Practical usage of array models</td>
<td>37</td>
</tr>
<tr>
<td>3.2.5 Allocating arrays in chunks</td>
<td>38</td>
</tr>
<tr>
<td>3.2.6 Consequences for the decoder</td>
<td>39</td>
</tr>
<tr>
<td>3.3 Model regeneration</td>
<td>40</td>
</tr>
<tr>
<td>3.3.1 Model regeneration using a history buffer</td>
<td>42</td>
</tr>
<tr>
<td>3.3.2 Memory recovery performance</td>
<td>43</td>
</tr>
<tr>
<td>3.4 Scaling of transition frequencies</td>
<td>46</td>
</tr>
<tr>
<td>3.4.1 Recovering from frequency overflow</td>
<td>47</td>
</tr>
<tr>
<td>3.5 Initial models</td>
<td>48</td>
</tr>
<tr>
<td>3.6 Cloning variables</td>
<td>49</td>
</tr>
<tr>
<td>3.7 Combining the coder</td>
<td>50</td>
</tr>
<tr>
<td>3.8 Using BDMC</td>
<td>52</td>
</tr>
<tr>
<td>3.9 BDMC evaluation</td>
<td>53</td>
</tr>
<tr>
<td>3.10 Summary</td>
<td>55</td>
</tr>
<tr>
<td>4 Character DMC</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Source alphabets</td>
<td>57</td>
</tr>
<tr>
<td>4.2 Initial implementation considerations</td>
<td>58</td>
</tr>
<tr>
<td>4.2.1 Representing States</td>
<td>58</td>
</tr>
<tr>
<td>4.2.2 Arithmetic coding interface</td>
<td>60</td>
</tr>
<tr>
<td>4.2.3 Preliminary results</td>
<td>61</td>
</tr>
<tr>
<td>4.3 Initial state specifications</td>
<td>62</td>
</tr>
<tr>
<td>4.4 Locating transition symbols</td>
<td>64</td>
</tr>
<tr>
<td>4.4.1 Sequential search</td>
<td>64</td>
</tr>
<tr>
<td>4.4.2 Binary search</td>
<td>64</td>
</tr>
<tr>
<td>4.5 Representing state transitions</td>
<td>65</td>
</tr>
<tr>
<td>4.5.1 Arrays</td>
<td>65</td>
</tr>
<tr>
<td>4.5.2 Dynamically linked lists</td>
<td>66</td>
</tr>
<tr>
<td>4.5.3 Linked arrays</td>
<td>67</td>
</tr>
<tr>
<td>4.5.4 Binary search trees</td>
<td>69</td>
</tr>
<tr>
<td>4.5.5 Cumulative coding trees</td>
<td>69</td>
</tr>
<tr>
<td>4.6 The complete state model</td>
<td>71</td>
</tr>
<tr>
<td>4.6.1 Memory usage</td>
<td>71</td>
</tr>
</tbody>
</table>
Chapter 1

Data Compression

The term data compression refers to the process of reducing the space needed to represent data. Compression of data is an important field of research today because of the large savings in cost that can be gained from using less storage space, or transmission time. Reducing the space required for storing data on a medium increases the amount of data capable of being stored on the medium. Data compression also reduces the cost of transmission across the world’s large communications networks. Without data compression facsimile machines would not be the valuable business tool they are today, as it would take over seven minutes to transmit just one page.

Research into data compression involves developing and analysing methods and algorithms that reduce the storage space required by the data. Data usually has some natural representation that may contain some degree of redundancy for example. Compression methods remove this redundancy by representing the data in a form that is more efficient. The more efficient the form the greater the compression over its natural representation. Compression algorithms translate data from one representation to another and hence perform data compression.

Dynamic Markov Compression is one such method of data compression that achieves excellent results. This thesis investigates several implementations of Dynamic Markov Compression for compressing different types of messages. It begins with an overview of data compression theory and techniques in use today.
1.1 Data compression in history

Data compression has been around for many years. Before the advent of computers compression schemes had to be simple and easily decodable by humans. This meant early compression schemes normally had a one to one relationship between representations. Simple codes replaced individual symbols or phrases within the data. Only recently with the advent of computers has the need and ability arisen to implement efficient methods of performing complicated data compression. Computers offer the ideal platform for developing and testing these methods, with large amounts of data available and fast processors for translating the data between complex representations.

One of the most widely known examples of data compression is used for communications. In 1838 Samuel Morse designed a coding scheme which involved sending a series of dots and dashes over a communication line. The time needed to transmit a single dash is three times longer than a dot, and each letter is separated by a gap the length of a dash. Thus the shortest transmission time is a single dot and its corresponding gap. This code is allocated the letter e, the most common letter in the English language, while the letter q has one of the longest used codes. The average length of a message is less using Morse's Code than a method that allocates codes of equal length to every letter. By designing a coding method based on the uneven frequency of letters Morse reduced the average time needed to transmit messages.

Many natural languages unconsciously perform compression in their attempt to reduce sentence length by reducing the length of commonly used words. Also, in the English language it is common to use such abbreviations as "can't" and "ASAP", for example. The information contained in a sentence using these abbreviations, is practically identical to that of a sentence without them, but requires less space to represent.

1.2 Lossy and lossless compression

When talking about data compression we must be aware of the two distinct types of compression in use today. Compression involves reducing the space needed to represent data and in performing this operation we may, or may not, lose some of the original information. In certain cases such as image or speech compression loss of
Chapter 1. Data Compression

Data is acceptable, but in cases such as text compression we must be able to reproduce the original source message exactly.

When compressing data such as pictures and human speech, it is not necessary to recover exactly the original uncompressed representation. Lossy or inexact compression translates analog signals, of which pictures and speech are fundamentally made up, into approximate digital representations. How close to the original representation the compressed signal is depends on a number of factors, but generally the more approximate the representation the better the compression achieved. The level of approximation determines the clarity of the data when it returned to its original representation.

An excellent example of lossy compression is the JPEG compression method [28] used to compress continuous-tone images. JPEG is capable of compressing image files to many times smaller than their original size. It achieves this by removing some of the detail contained in the image, and the degree to which detail is removed is known as the image quality value. Human perception cannot easily distinguish between images compressed at a quality value of 100 or 80, and perceivable degradation of the image rarely takes place above 60.

When dealing with text compression it is necessary to predict exactly what the next character will be, not just an approximation! Lossless or exact compression removes redundancy from the source message but still allows the exact original to be recovered, by applying the reverse process. Text compression would be of little use if we were unable to return to the original representation. Methods used for exact compression work by predicting the characters within a source message. To determine the next character, a probability distribution containing every possible character is used to assign unique transmission codes for each possible outcome. The creation of these probability distributions is often referred to as modelling the source. If the probability distribution correctly reflects the source message, short codes will be assigned to characters with high probabilities, and compression will be achieved. A technique known as coding converts the supplied probabilities into an optimal representation, how this is achieved is discussed in section 1.7.

The better the model captures the information contained in the source message the better the achievable compression. The methods used to generate models form the major part of research into compression today. Although they are approximate,
models can be used for exact compression as we are not relying on the accuracy of a given model. A model contains predictions of symbols expected in the remaining source, even when these predictions don't exactly match the symbol probabilities compression may be achieved. Of course a model that doesn't reflect the source message well will not provide good compression and in some cases may even expand the original message.

1.3 Modelling and coding paradigms

In 1948 Shannon [33] introduced the field of Information Theory dividing communication into the five components as shown in Figure 1-2. While very general, Shannon's model introduces concepts important to the understanding of data compression. A Transmitter translates a message produced by an Information Source into a signal for transmission over the communication Channel. The channel is the medium over which the signal passes, it may also introduce some transmission noise. Finally the Receiver translates the signal back to the original message which is intended for the Destination.

Shannon's model covers many aspects of communication including error correction, data compression, and cryptography. When dealing with data compression we are aware of these other aspects but assume several restrictions to Shannon's model:

- The source message is a sequence of symbols from a fixed, finite source alphabet.

---

1 Shannon's original paper [33] includes detailed discussion of more general cases, the restrictions given are reproduced from [34].
Chapter 1. Data Compression

• The channel carries a sequence of symbols from a fixed, finite channel alphabet.
• The cost of transmitting each symbol through the channel is identical.
• The channel is noiseless, so there is no need for error correction.

As mentioned, compression relies on a prediction of the next symbol based on a model of the source message. Finding a suitable model is extremely important, and this thesis investigates one such method that is capable of modelling many types of data accurately. Once the model has been generated, the predictions, together with the source message, are used by a coder to produce a signal suitable for transmission across the channel as in Figure 1-3. Transmission might involve simply storing the compressed form of the message on a computer’s disk, in which case the disk can be thought of as the channel.

Figure 1-3 Using a model for compression, notice the similarity with Figure 1-2 Shannon’s model of communication.

Separation of data compression into these two distinct stages – modelling and coding [32] – has been a major advancement in compression over the last two decades. It has enabled better research into the area of generating accurate models for different types of source messages, rather than producing better codes. The modeller supplies a probability distribution which contains the expected probability of every symbol, plus the symbols needing to be transmitted to the coder. The coder then transmits a coded representation of the symbols using the supplied probabilities. This separation has enabled the development of universal coders that accept probabilities from many different modellers.

Conceptually, modelling and coding are completely different activities. While existing modelling methods are still being researched and new ones are emerging, the problem of constructing a perfect coder has been completely solved. Arithmetic
coders have been developed that are as close to optimal as to make practically no difference to the length of the compressed message. Increasing the efficiency of these coders in terms of both execution speed and memory requirements only remains an issue, and will no doubt happen as hardware technology evolves.

Some methods at first do not appear to support this modelling-coding paradigm, but in many cases it can be shown that both operations are intertwined. These methods can either be replaced by a related statistical technique, or broken down into the two fundamental operations, thus provide interesting comparisons with more theoretical methods [4].

1.4 Static, semistatic and adaptive modelling

Given that the coding implementation has already been completed, attention can be given to creating a model that will estimate symbol probabilities producing a coded message of minimum length. Figure 1-3 showed how models are used in practice to produce codes suitable for transmission over a channel. The encoder uses the probability distribution to produce transmittable codes, while the decoder uses the same distribution, translating the codes back to their original representation. It is important that the chosen model is appropriate for the source being compressed, and that both the encoder and decoder use identical distributions produced by identical models. There are three ways of implementing modelling, namely static, semistatic, and adaptive modelling.

A static model is a fixed model, which doesn’t change during compression. Both the encoder and decoder use this fixed model to translate the source message incrementally. Choice of an inappropriate model results in inefficient compression, if the data changes significantly from this fixed model.

A semistatic modeller preprocesses a source message, creating a model appropriate for the message. This overcomes the problems of using a static model, resulting in a model which should be appropriate for any given message. The generated model is transmitted prior to transmission of the coded sequence generated from the

---

2 Arithmetic coding actually produces a coded message arbitrarily close to the length of the entropy. Various factors including message termination, and fixed length precision arithmetic contribute to a fractional overhead.
proportions contained in the model. The decoder, supplied with the correct model, then translates the coded sequence back to its original representation. One disadvantage of using a semistatic technique is the requirement to make two passes over the complete source message, the first to create the model, and the second to actually produce the coded sequence. When transmitting infinitely long source messages this restriction is unacceptable. Another disadvantage is the necessity to transmit the model before the source message can be coded, adding to the overall code length.

Adaptive modelling overcomes these limitations of a semistatic modeller rather elegantly. Initially a single symbol is coded based on a known initial model, and once it is transmitted, the model is updated to reflect this coded symbol. The decoder receives the first coded sequence and determines the transmitted symbol using an identical initial model. Once the symbol has been determined the decoder also updates its model appropriately. Both the encoder and decoder build up identical models that suit the source message well, without the need to explicitly transmit any prior details of the model used. Theoretical comparisons have shown that an adaptive model performs extremely closely, to its static or semistatic counterpart [4], while the static method with an inappropriate model will perform very much worse.

Both static and semistatic modellers are inappropriate for general purpose data compression. Static modellers cannot adapt to unexpected source messages, and semistatic modellers require two passes over a message, making them unsuitable for use over communication channels. Adaptive modellers combine the best of static and semistatic modellers, capable of incrementally transmitting symbols based on a model of previously coded symbols.

1.5 Modelling techniques

Given that the role of a model is to supply probabilities to a coder, we now look at several different classes of modelling techniques used in generating appropriate models. The majority of methods generate probabilities for symbols incrementally as the source message is processed. This enables compression of any data because
Chapter 1. Data Compression

indefinitely long messages can be modelled\(^3\) and the coder can transmit a coded sequence incrementally as the probabilities are supplied. Coding of the message after the complete message has been modelled is infeasible because a large model could be required, and modelling of infinite messages wouldn’t be possible.

1.5.1 Context models

The simplest adaptive model will store frequency counts of each symbol encountered so far in a source message. For every symbol \(k\) of the source alphabet, the symbol frequency \(n(k)\) is used to calculate the probability

\[ P(k) = \frac{n(k)}{N}, \]

where \(N\) is the total of all symbol frequency counts\(^4\). This model is known as a zero order Markov model, or a zero order context model, as it doesn’t use any preceding context to predict future symbols. More sophisticated Markov models base their probabilities on \(n\) preceding symbols, and they are known as \(n\)th-order Markov models. These modellers use \(n\) preceding symbols as a context history to predict the next symbol. The higher the probability that a symbol will occur, the smaller the code required in the transmitted sequence. As the length of the context grows so does the size of the sample need to accurately predict the probabilities of future symbols.

Markov models are currently the most successful class of modelling technique in terms of achievable compression, especially the subclass of variable order context models. Large context models suffer because initially it is difficult to accurately predict symbols based on infrequently used contexts. Variable order models solve this problem by accurately estimating probabilities for the next symbol using some largest appropriate context available. One of the most successful modellers of this type is “Prediction by Partial Matching” [8], or PPM, which stores variable order contexts that have thus far appeared. PPM starts with a reasonably large context and attempts to make a prediction for the next symbol. If unable to predict it, an escape is

\(^3\) Providing the chosen modelling method is capable of capturing the message without running out of computer resources.

\(^4\) Strictly speaking \(N\) is actually the total number of frequency counts plus additional counts for symbols not encountered yet. How these additional counts are calculated is known as the zero-frequency problem and is discussed further in section 1.7.2.
coded and the modeller switches to a lower order context and tries again. The family of PPM modellers consists of several algorithms, each differing in the method of calculating escape probabilities, and resources used. PPM methods have proven to give excellent compression, albeit relatively slowly.

### 1.5.2 State models

State models have the ability to capture regularities in messages that context models cannot. They also offer an alternative and more efficient way to capture variable order contexts. State models consist of a set of $n$ states $[S_1...S_n]$, together with a set of transition edges between these states. Transitions have both an associated alphabet symbol $k$ and transition probability $P_{ij}(k)$ which represents the probability that the current state will move from state $S_i$ to $S_j$. No two transitions from a single state involve the same symbol, so a source message defines a unique path through the state model. The transition probabilities are used by the coder to produce the compressed sequence. State based modelling has the potential to offer better compression and improved execution speed, because only a single transition from one state to another needs to be traversed as each symbol is coded.

Context models are easily implemented using state models, creating states with edge transitions for each source alphabet symbol. Figure 1-4 shows a finite state model that represent an order zero context model for a source alphabet of five symbols $[a,b,c,d,e]$. Order $n$ context models are constructed using $n$ states. If $m$ is the number of symbols in the source alphabet, each state contains $m$ transitions to other states. Variable order context models are represented by variable length paths through the state model. The problem of constructing optimal models for any given source message can be difficult. Practical methods for automatically generating such models are so far unable to take advantage of state modelling's main benefit, that of capturing counting events.
Chapter 1. Data Compression

An example of a message that exhibits counting, that a context model is unable to capture, is shown in Figure 1-5 (a). Such a message is characterised by repetitive symbols, separated by arbitrarily long sequences of other symbols. The example (a) repeats the sequence "some a's followed by a b, more a's and then a c". The finite state model (b) is capable of accurately representing this type of message, while a context model would not capture this counting phenomenon. Unfortunately, as mentioned, no feasible method for automatically generating state models that capture this type of information has yet been found.

1.5.3 Grammar models

Dispite all their power, state models are unable to capture arbitrary nested symbols, such as used in structured computer languages. Languages such as Pascal and Modula-2 use a grammar to represent their strict syntactic rules, including recursive constructs such as expression parenthesis. State models are only capable of capturing finite levels of these nested constructs. Source messages conforming to this syntax
require a different type of model to accommodate these arbitrary nesting depths. As the message is parsed, according to its grammar, probabilities associated with each production can be used to code the message. The syntax of these formal languages is typically very concise, and grammar models have proven successful in compressing language source text [7]. Natural languages normally don’t contain a high degree of nested grammatical constructs, and are just as well modelled by a state model.

1.5.4 Dynamic markov modelling

Dynamic Markov Compression\(^5\) (DMC) is a state modeller. From the description of context modellers and the fact they rely on Markov chains for their predictions however, one might assume that DMC would belong in this class of modeller. In fact although DMC is a state based modeller, it has been shown that it is only capable of capturing the same information as that of a variable order context model [5].

DMC is an adaptive state modeller that constructs a model of a source message incrementally. Starting from an initial model, DMC adds additional states to reflect the frequencies of symbols contained in the source message. The original authors proposed an implementation that modelled the bits of a source message, thus capable of modelling any arbitrarily source message. Performance tests produce a favourable comparison with PPM in terms of achievable compression (PPM on average outperforms DMC by only about ten percent), however because it uses a bit-wise model DMC is approximately half the speed of PPM and requires large amounts of memory to achieve this compression.

Given the potential power of DMC the original proposal has subsequently been developed in this thesis to provide character – and word-based modellers. This thesis details the steps involved in adapting the original bit based modeller into functional character and word-based modellers. The hope is that these further enhancements will improve on both the speed and memory requirements of DMC.

\(^5\) Traditionally Dynamic Markov Modelling is referred to by the acronym DMC, Cormack and Horspool [10] named the original modelling method they developed Dynamic Markov Modelling. Their implementation however included an arithmetic coder based on Guazzo’s [18] and the modeller and coder combination is referred to as Dynamic Markov Compression.
1.6 Information theory

The field of information theory, founded by Shannon, concerns itself with transmission of information over communication channels, and forms the basis on which modern compression methods exist. Information theory provides answers to the questions, "Is there some limit to the compression that can be achieved for a given source message?", and "How can this limit be determined?"

One of the fundamental ideas of information theory is that of entropy, which is the amount of information or redundancy contained in a message. The more information a message contains, for the receiver, the higher its entropy. Shannon's "noiseless source coding theorem" established a scientific method for measuring the information content of a message based on the probabilities of message symbol choices.

A probability distribution containing the complete set of \( m \) source alphabet symbols is used to calculate the average entropy of the symbols. The entropy measures how much choice is involved, on average, for each symbol \( k \) of the source alphabet. If each symbol has a probability \( P(k) \) of occurring then the entropy \( H \) of all symbols is defined as

\[
H = -\sum_{k=1}^{m} P(k) \log_2 P(k) \text{ bits.}
\]

Using a logarithmic base of two implies the unit of information, or entropy, is bits. In data compression we are more often interested in quantifying the information content of a single choice rather than the average over all choices. The entropy \( H(k) \) of a single symbol \( k \) is defined as

\[
H(k) = -\log_2 P(k) \text{ bits.}
\]

Symbols with low probability are unexpected and therefore contain more information compared with symbols which have a high probability. The entropy \( H \) is simply the weighted average of the information content of individual symbols. A fundamental result in Shannon's original paper was determining that on average the number of bits needed to represent any symbol can never be less than its entropy. This result is significant in data compression, it provides a precise measure of the best expected code length for a source message, with respect to a given model. While it is possible to code a message in fewer bits than its entropy, it can not be done consistently.
Ideally a compression method will code a message to a length equal to its entropy using a model that represents the source exactly. In practice, however, we construct approximate models adaptively and transmit the coded message incrementally. The model estimates probabilities for symbols, and in turn determines the entropy of the symbol. It is important to realise that the entropy of the model is an estimate for the entropy of the source.

The best compression will be achieved when the model represents the source accurately. Once a model generates the symbol probabilities a method is needed to convert these probabilities into bits suitable for transmission across the channel. This is the job of the coder.

1.7 Coding

Given a probability distribution that governs the choice of the next symbol, and the symbol that actually occurs, we need to generate a bit sequence that is equal in length to the symbol's entropy. This problem of coding symbols in this way was one of the first tackled by practitioners in the field of information theory. Two initial approaches that worked well were Shannon–Fano coding [13] and Huffman coding [20]. Both methods generate variable length codes given a probability distribution for a set of symbols.

The Shannon–Fano coding scheme involves ordering all alphabet symbols by decreasing probabilities, and recursively partitioning the list in two such that the sums of probabilities of each new list are approximately equal, assigning each list a single bit value at each recursive level. The average code length of the Shannon–Fano scheme lies within the range $[H \ldots H + 1)$.

Huffman coding produces a set of codes which in general gives greater compression than the equivalent Shannon–Fano codes. The creation of these codes involves recursively, finding the two symbols with the smallest probabilities and replacing them with a set containing both symbols, with their combined probabilities. Once completed, this procedure yields a binary tree. Traversal of the tree from the root determines a code for any symbol. The average code length for the Huffman scheme

---

6 Both Shannon and Fano discovered independently identical coding methods.
lies within the range \([H \ldots H + p + 0.086]\), where \(p\) is the probability of the most likely symbol [16].

Huffman coding achieves the minimum amount of redundancy for a fixed set of variable length codes. The problem with Huffman coding is that it always uses an integral number of bits to represent a single symbol. If the optimal number of bits for a given symbol is 3.5 bits then it will generate a code of either 3 or 4 bits, not 3.5. Blocking techniques must be used to make Huffman coding more effective, by grouping symbols into blocks of \(n\) characters and using them as units for coding. Therefore Huffman coding cannot be considered optimal, but it does provide the best approximation using an integral number of bits, and is suitable for large alphabets or low probabilities.

1.7.1 Arithmetic coding

Huffman's original work dominated the coding world until the early 1980's. Arithmetic coding breaks the restriction of coding with an integral number of bits, thereby providing more efficient coding. It was first introduced by Rissanen [29–31] and Pasco [26], with a practical implementation presented in 1987 by Witten, Neal and Cleary [36].

Arithmetic coding produces a coded sequence incrementally as symbols are provided, the process involves recursively dividing the initial range \([0 \ldots 1)\). Symbols determine which part of the range is narrowed based on their relative position within the probability distribution. The final range can be represented by a unique sequence of bits. The decoder utilises this information to decode the message provided it has an identical probability distribution. Arithmetic coding is well suited to adaptive modelling because each symbol can be supplied together with a unique probability distribution to the coder. Symbols are transmitted incrementally in exactly the number of bits specified by the symbols entropy.

The arithmetic coder used in this thesis is one developed by Witten et al. [36], which provides a simple interface to the coding routines. The three important routines are listed below.

\[
\text{arithmetic_encode(lbnd, hbnd, totl)}
\]

This generates and outputs a code for the symbol specified by the sub-range \([\text{lbnd} \ldots \text{hbnd})\) from the complete range \([0 \ldots \text{totl})\). Each alphabet symbol within
the model is specified by a cumulative range within the probability distribution, often referred to as the symbols *code space*. The code will have a length of

$$-\log_2 \left( \frac{\text{hbnd} - \text{lbnd}}{\text{tot}} \right) \text{bits}$$

**arithmetic_decode_target(totl)**

This is a function that returns a value \( t \), where \( t \) falls within the sub-range of a symbol from the complete range \([0...\text{totl}]\).

**arithmetic_decode(lbnd, hbnd, totl)**

This is complementary to **arithmetic_encode**, used once the symbol is known, to remove the code bits to be ready for determining the next symbol.

It is not necessary to provide the coding routines with the complete probability distribution for all alphabet symbols. Instead just the cumulative range enclosing the symbol to be transmitted is required. This gives the modeller some flexibility in determining a symbol’s probability and doesn’t require that the symbols be in any one particular order. One limitation imposed by the use of 32-bit fixed point integer arithmetic is that it restricts the total of the probability distribution to less than or equal to \( 2^{14} - 1 \), which in practice is not a severe limitation for alphabets with fewer than \( 2^{14} \) symbols.

### 1.7.2 Coding novel symbols

In adaptive modelling, both the encoder and decoder simultaneously build identical models based on symbol frequencies contained in the source. Symbols that have never occurred before, and hence have a frequency count of zero, will, when encountered, be assigned a code of length \( -\log_2 0 \), which is infinite. This is known as the *zero frequency* problem [29]. To overcome this, all symbols need to have some code space reserved for them in the probability distribution by assigning them a non-zero frequency count.

For a small, finite, source alphabet this is normally achieved by ensuring that each symbol has a frequency count above zero, and hence a non-zero probability. For larger alphabets, assigning each symbol a non-zero frequency count can be detrimental to compression because the probability of all novel symbols can overshadow those symbols actually encountered. Resolving this problem requires either the use of frequency scaling, or transmitting an special code to signal a novel
event to the decoder. The former technique gives more emphasis to the symbols encountered by incrementing their frequencies by larger values, and therefore giving them larger relative probabilities. The latter method uses an escape code to signal each novel symbol. The novel symbol is then coded using some additional probability distribution, such as from an equiprobable model. This approach normally has a smaller impact on the probability distribution because only an extra escape symbol is required to be represented, rather than a large number of unseen symbols, but the cost of coding a novel symbol may be higher.

1.8 Other compression techniques

So far we have mentioned compression techniques that involve predicting characters and choosing an appropriate code based on the resulting symbol probability distribution. Often referred to as statistical methods, they are intertwined with the ideas of probabilities and predictions. Another popular technique used for data compression today is known as dictionary compression. In practice a dictionary method can be outperformed by a related statistical technique, and so statistical methods at present provide the best achievable compression [4].

A simple way of reducing the redundancy of a source would be to replace common phrases\(^7\) with some unique identifier. These techniques involve building a phrase dictionary, parsing the source message, and transmitting indexes into the dictionary for matched phrases. To achieve compression, indexes for a phrase are chosen so they are smaller than the phrase they replace.

The most widely known of these dictionary methods form the Ziv-Lempel (LZ\(^8\)) family, performing adaptive dictionary compression using a sliding window over the source message. The sliding window is a history buffer of previously seen phrases, longest match algorithms are used to locate phrases, and just a single pointer into the sliding window needs to be coded. The techniques are outlined in two papers in 1977 [41] and 1978 [42]. Several adaptations have been investigated by numerous authors to produce many variations\(^9\). In practice dictionary techniques from the LZ

---

\(^7\) The term phrase is used to refer to an arbitrary length sequence of source message symbols.

\(^8\) As mentioned in [4, 34] the acronymized reversal of authors initials is a historical mistake.

\(^9\) Bell, Cleary, and Witten [4] includes an excellent section describing the principal variations of the LZ family, and a more up to date description is found in Witten, Moffat, and Bell [35].
family give excellent compression both for natural language and binary files (the best
dictionary method GZIP [35] compresses approximately twenty percent worse than
PPM, and ten percent worse than DMC). The encoding speed depends on the parsing
strategy used, while decoding is extremely fast because it only involves one look up
into the phrase dictionary to output a long series of characters.

1.9 Summary

This chapter presented an overview of data compression techniques in use today, and
the fundamental results of information theory as applied to compression. Dictionary
methods are currently riding a wave of popularity, due to their very high
compression throughput when compared with slower statistical methods that offer
slightly better compression. The motivation behind this thesis is to investigate ways
of improving DMC's heavy resource requirements. Adapting the bit modeller to
model characters potentially offers eight times the original speed performance,
although in reality this may well prove not to be the case. Careful consideration is
also given to the data structures required to represent the more complicated character
model. Further adaptation of the character modeller to model words or phrases
should also see improved performance because long sequences of characters could be
replaced by a single code.

The remaining chapters of this thesis investigate a family of modellers based on
Dynamic Markov Compression. Chapter two explains the DMC algorithm, how the
binary method works, and previous research performed by other authors. In chapter
three a working implementation of the binary DMC method is explained, including
problems with implementation, and comparative results with other compression
techniques. Adapting the DMC method for use with larger symbol alphabets is
discussed in chapter four, with the development of character DMC. Of special
concern is the amount of computing resources required by the new algorithm, and
methods for reducing these are discussed. Further evolution of character DMC to
include modelling of words in the focus of chapter five. In a complete departure from
the traditional resource requirements of the DMC method chapter six discusses an
implementation intended for use in low resource situations. Finally, chapter seven
concludes with a summary of the results contained in this thesis.
Chapter 2

Introduction to DMC

This chapter provides an overview of the original bit-wise DMC algorithm, and specific details of how the algorithm works. The chapter finishes with a survey of other research involving DMC.

The problem of constructing a perfect state model for any given source message has not yet been solved [14, 15]. The only adaptive finite state modeller that has proven to be practical for data compression is Dynamic Markov Compression. DMC generates a finite state machine (FSM) incrementally from symbol frequencies contained in the source. Frequencies stored for each labelled transition edge are used to provide symbol probabilities for the coder. The original DMC algorithm was based on the binary symbol alphabet \{0,1\}, containing two outward transitions from every state.

The state model of figure 2-1 describes a simple binary source. Transitions from each state are labelled \(kn\), where \(k\) represents the symbol, and \(n\) the number of times the transition has been traversed. The probability \(P_s(k)\) that symbol \(k\) will occur in a state \(S\) is determined by the relative frequency of the transition from \(S\), calculated by dividing the frequency count of \(k\) by the total number of transitions made from \(S\). For a model based on a binary source alphabet this gives

\[
P_s(0) = \frac{n_s(0)}{n_s(0) + n_s(1)}, \quad \text{and}
\]

\[
P_s(1) = \frac{n_s(1)}{n_s(0) + n_s(1)}.
\]
Giving each edge transition an initial frequency \( c \) prevents \( P_i(k) \) equaling zero\(^1\), making the model less susceptible to statistical fluctuations while the transition counts are small. A higher value of \( c \) ensures that the model is less affected by wild fluctuations, but it takes longer to adapt to the source message. Experience shows for large files, the value of \( c \) becomes largely irrelevant.

A state modeller that updates frequency counts after each edge transition is traversed will adapt to the source message, based on the popularity of symbols contained in the source. A better method would also alter the structure of the FSM to adapt to the structure of the source message. The strength of DMC is its ability to generate state models that represent the source message reasonably accurately\(^2\). DMC has the ability to create chains of states representing variable length symbol contexts, growing these chains to arbitrary long context lengths. Variable order context models, in comparison, do not usually create contexts longer than some predetermined maximum length, so potentially DMC offers greater flexibility in modelling a source.

### 2.1 Cloning – the power of DMC

Starting with an initial state model, DMC *clones* individual states when their related transitions become sufficiently popular. A heuristic based on the transition frequencies of state symbols determines, before each edge transition is updated, whether a state has become eligible for cloning.

---

\(^1\) Supplying a coder with a probability of zero for a symbol produces an infinitely long coded sequence. See section 1.7.2 for a discussion of the zero-frequency problem.

\(^2\) The accuracy can not be quantified, instead relying on DMC’s performance compared to other compression methods to assert this fact.
A portion of a FSM, capable of being generated by DMC, is shown in figure 2-2. Figure 2-2 (a) contains the state C which we will assume has become eligible for cloning. Two states A and B have transitions into state C, there may be several inward transitions, and all will have the same symbol. Only two transitions ever emanate from a state – one for symbol 0, another for symbol 1 – going from state C to states D and E respectively. Contextual information contained in the source may not be captured due to the fact that both states A and B lead to the same state, C. After cloning state C, the new FSM (figure 2-2 (b)) is capable of capturing more contextual information from future symbols. The relationship between state A and {D, E} will be captured in state C, while any relationship between state B and {D, E} will be captured by state C'. If no correlation exists between these states then the model size and complexity has been increased unnecessarily, and makes the model more susceptible to statistical fluctuations, because C and C' will be used less often. If the new FSM accurately reflects future relationships between symbols, better probabilities will be supplied to the coder, resulting in an improvement in compression.

---

3 The DMC algorithm ensures that a state can not have heterogeneously labelled inward transition edges. The only exception being an initial state that has not yet been cloned.
Chapter 2. Introduction to DMC

Important definitions and data structures, important in the understanding of DMC, are defined in figure 2-3. ALPHABET_SIZE declares a binary symbol alphabet, the symbolStruct represents the information contained in the state about each symbol, and the stateStruct declares a one dimensional array of ALPHABET_SIZE. SymbolStruct contains a pointer to the next state and a frequency count, used in calculating the probability for the symbol. These symbol frequencies also become useful in determining when a state becomes eligible for cloning. The macro TOTAL is useful in calculating the total number of inward and outward transitions from a state. Code describing the cloning procedure is given in figure 2-4. The variable current_state (A in figure 2-2) is a pointer to the current state, next_state (C in figure 2-2) a pointer to the next state, which may be eligible for cloning, and new_state (C' in figure 2-2) a temporary pointer to the new state.

```
#define ALPHABET_SIZE 2  // binary alphabet

typedef struct {
    struct STATE *next;
    short count;
} symboiStruct;

typedef struct STATE {
    symboiStruct sym[ALPHABET_SIZE];
} stateStruct, statePtr;

#define TOTAL(s) (s->sym[0].count + s->sym[1].count)  // total transitions macro
```

Figure 2-3 C code describing the DMC state structures.

```
// CloneState
// Create a new state, connect it to the current FSM, assign new transition counts and
// return pointer to the next state
statePtr CloneState(statePtr current_state, statePtr next_state, short symbol)
{
    double ratio;  // ratio of transition count
    statePtr new_state;  // pointer to the new state

    new_state = AllocateState();

    current_state->sym[symbol].next = new_state;  // connect new state to FSM
    new_state->sym[0].next = next_state->sym[0].next;
    new_state->sym[1].next = next_state->sym[1].next;  // apportion transition counts

    ratio = current_state->sym[symbol].count / TOTAL(next_state);
    next_state[0].count = new_state[0].count = next_state[0].count * ratio;

    return(new_state);
}
```

Figure 2-4 C code describing the DMC cloning operation.
Chapter 2. Introduction to DMC

The cloning operation creates new and connects it to the existing FSM, the operation also apportions the existing frequency counts between the outward edges of next and new. Cloning maintains Kirchoff's Law by ensuring the number of transitions into any state is the same as the number leaving that state\(^4\). The frequency counts for next are proportionally divided between new and next, relative to the symbol’s frequency divided by the total number of transitions in next. By maintaining Kirchoff’s Laws it is becomes unnecessary to retain a total transition count for each state, because the sum of the outward edge transitions equals this inward total. One other side effect of maintaining Kirchoff’s Laws is the simplification of the logic needed to determine the cloning condition for a state.

2.1.1 Deciding when to clone a state

Cloning a state not only increases the complexity of the FSM, but it also increases the number of additional correlations between symbols that could be discovered and used for predictive purposes. It is worth cloning a state whenever that state has been visited a reasonable number of times from each of its predecessors. If both predecessor transitions have been traversed a number of times, cloning will allow DMC to discover additional information about the correlations between the two predecessor states and its children. It is not necessary to clone a state if only one of its predecessors is contributing significantly to the predictions for a symbol because if the state was cloned only one of the new transitions would be used. This desirability of cloning a state when both the transition from current to next state, and the transition from other predecessor states to next are reasonably large, leads to the following cloning criteria. The state next is cloned if and only if “the number of transitions from current to next is greater than \(t_1\), and the number of transitions from all other states other than current to next is greater than \(t_2\)”\(^5\), expressed in C as shown in figure 2-5. The values \(t_1\) and \(t_2\) are referred to as the cloning parameters and have a drastic effect on the performance of the DMC algorithm. Experimental results supporting the best choice for these cloning parameters is detailed in section 3.6.

\(^4\) For the current state these transition counts will differ by one because there is always one more transition into the current state than there are transitions out.

\(^5\) The cloning criteria is reproduced from Cormack and Horspool’s original papers [10, 11] but substituting the cloning parameters originally labelled MIN_CNT1 and MIN_CNT2 with \(t_1\) and \(t_2\) respectively.
These results show that the best performance is gained when the model grows rapidly from its initial FSM.

\[
\text{if ( current-state->sym[symbol].count > T1 \&\& (TOTAL(next_total) - current-state->sym[symbol].count) > T2)}
\]

Figure 2-5  C conditional expression triggering DMC state cloning.

Maintaining Kirchoff's Law when cloning a state simplifies the logic needed to determine how often next has been visited from states other than current. The assumption that the 'total outward transitions always equals the total inward transitions', implies that the total number of transitions of next can be calculated by summing the outward transitions. Subtracting the inward transition frequency from current from this total determines the total number of transitions to the state other than from current.

### 2.1.2 Original examples

The original cloning diagrams described by Cormack and Horspool depict a special case of the cloning operation which rarely occurs. States containing heterogeneously labelled (different) inward symbol transitions only occur in a given initial model, such as in figure 2-7 or 2-8. The cloning operation removes these heterogeneous labels by splitting the transitions between the next and new states. Figure 2-6 is a reproduction of figure 2 of section 4.2 in the original paper depicting the results of cloning these heterogenous labels.

![Diagram](image-url)

Figure 2-6  The example of cloning redrawn from figure 2 of section 4.2 of Cormack and Horspool [11].
2.2 Initial state models

DMC bases its predictions on edge frequencies contained in the FSM. It must be provided with an initial model, from which it clones additional states. Given a simple model, DMC can quickly adapt it to most types of source message. More sophisticated initial models can improve performance when compressing short message sequences, but as the length of the message grows this performance gain is negligible. The simplest FSM for a binary alphabet, capable of capturing all possible message sequences, consists of a single state. Figure 2-7 shows this model, which consists of two heterogenous transitions for the symbols 0 and 1. Once cloning begins the model complexity increases rapidly into a mass of thousands of interconnected states.

![Figure 2-7 Initial one state FSM capable of capturing any message sequence.](image)

While a simple one state FSM is sufficient to model all possible source messages, various other FSM's have been used as initial models in attempts to improve performance. Computer data often has correlations between adjacent bytes or words, rather than bits. Utilizing these correlations in initial models improves compression performance by enabling the model to adapt faster to a suitable source. On the other hand, if an initial model is inappropriate for the source message, results have shown that it still adapts quickly but with some performance loss. A simple FSM for byte orientated data consists of eight states arranged as a chain, each state containing two inward homogeneous transitions, figure 2-8 depicts such a model. A more suitable FSM has 255 states arranged as a binary tree with each leaf transition returning to the root state. This binary tree has a unique path for each possible byte so is capable of capturing order zero character probabilities. A binary tree modelling 4-bit characters is illustrated in figure 2-9.

![Figure 2-8 Simple initial FSM that mimics byte structure of computer data consisting of a "chain" of eight states with non-heterogeneous transitions.](image)
A more complex model known as a braid, or omega network, "because of the way it
in which the transitions interweave when drawn out in two dimensions"[11], provide
the modeller with an even more complex initial model. A braid attempts to model
correlations for the preceding $k-1$ bits, where $k$ represents the depth of the braid
structure. It contains a transition path through the braid for each possible $k$-bit
sequence, back to a unique top level state. It is capable of capturing some order one
context information because it has a top level state for each alphabet symbol. A 3-bit
symbol braid is shown in figure 2-10.

Figure 2-9   An initial model that exhibits correlations between 4-bit characters using a
binary tree.

Figure 2-10   An initial 3-bit symbol braid model, or omega network.
Experience has shown that using such complex initial models will slightly improve the compression obtained for short source messages, while little will be gained for long source messages. Supplying DMC with such complex initial models also goes against the principal of adaptive compression.

2.3 Self referencing transitions

When an initial model contains self referencing transitions, the cloning operation causes these references to be transferred between states. Cloning current when it is also next, causes the self referencing transition to shift from current to new. This a consequence of the new state being assigned to the current transition pointer before the current state transition pointers are copied to the new state. Figure 2-11 depicts the FSM after a single cloning operation of the single state initial model given in figure 2-7. In this case the cloned FSM consists of two states, which represent the finite context strings 0 and 1.

Note that the initial FSM contained heterogeneously labelled transitions but after cloning these no longer exist. The single state of the given initial model is always cloned into two new states without heterogeneously labelled transitions, a consequence of the operation shown in figure 2-6. Once the heterogeneously labelled transitions are removed from the FSM, it will always contain only homogeneously labelled transitions.

![Figure 2-11 Cloning the single state initial model with self referencing transitions results in one of the self references shifting to the new state. The state model shown now represents the context strings 0 and 1.](image)

2.4 Low memory operation

The DMC method starts with an initial state and quickly adapts to any source message using cloning, increasing the size of the FSM. One consequence of this operation is that model memory space is exhausted rapidly. Solutions to get around this problem have been presented by several authors [4, 6, 11, 22, 38].
• Stop the cloning operation and don’t alter the transition frequencies once all available memory is exhausted. This results in the inability to adapt to further changes in the source message.

• Stop the cloning operation as above, but continue to adjust transition frequencies. As long as the remaining source message doesn’t change dramatically this method will offer better compression than the previous method.

• Once the available memory is exhausted discard the FSM and start again using the initial model. As drastic as this method sounds, it is quite effective because of DMC’s ability to very quickly readapt to the remaining source message.

• Once the available memory is exhausted discard the FSM and build an initial model using some number of previous symbols. A less drastic approach than the previous, this allows the model to have a head start in re-adaptation. A buffer containing symbols recently coded provides a recent context for the immediately succeeding symbols.

• Implement some form of “forgetting” of states that have been cloned. That is, if it turns out a cloning operation has not been productive then reduce the cloned states back to their original parent. Unfortunately the extra information required to calculate when a cloning operation has been ineffective uses valuable resources, and by the time it has been determined that a previous cloning operation should be undone the relevant states may have been cloned many more times. This complicates both the uncloning operation and the calculations determining when uncloning should be performed. From examination of the suggested process, the additional memory requirements, and the increased execution time it seems unlikely that additional benefits would be gained.

These memory bounding techniques have been compared in practical experiments, results of which can be found in section 3.3. A special low resource variation of DMC is presented in chapter six, departing from the traditional heavy resource requirements to use a model that requires minuscule amounts of memory.
2.5 DMC – a variable order context modeller

As has been mentioned state models have the ability to capture regularities in a source message that context models cannot. It has been assumed that because DMC has been basing its predictions on a state model it offers the full power of such state models. This was been proven not to be the case by Bell, showing that DMC only generates variable order context models [2].

While this is a surprising result, it does not detract from the fact that DMC has proven to be a high performance compression method. It also offers some explanation as to why in practice DMC’s performance closely follows other variable order context modellers such as PPM. Of course, DMC offers several potential advantages over these context based methods, namely speed and the ability to grow arbitrarily long contexts.

2.6 Empirical comparison of compression methods

A series of test files known collectively as the Calgary corpus is commonly used as a basis for evaluating compression algorithm performance. This corpus first appeared in Bell, Cleary, and Witten [4], and has since been used widely by the compression research community. The corpus consists of fourteen files of nine differing types, including English text, geographic data, black and white bitmaps, executable machine code, and computer language sources. For details of the corpus refer to appendix B of Bell et. al. [4]. These files are used for empirical evaluation and comparison of all compression algorithms developed in this thesis.

One of the most useful measures of the performance of theoretical compression methods is the amount of compression obtained for a source message. In other words the compression is the relative size of the coded sequence compared to the original source message in its uncompressed state, which also represents the amount of redundancy removed from the original source. This measure offers a way of directly comparing the results from different algorithms over the same set of source messages. The Calgary corpus is one such set of reference source messages used in comparisons of lossless text compression algorithms. In text compression it is assumed that eight bits are required to represent each unique character of the alphabet, implying the average number of bits per character is eight uncompressed.
Chapter 2. Introduction to DMC

Methods that code a message with fewer than eight bits per character on average achieve compression.

Another important comparison that can be drawn between compression methods is the relative throughput rates. The ability of a compression method to compress well and quickly may make it more attractive than a method giving similar performance but which takes more time. For the remainder of this thesis, unless stated differently, when talking about the performance of methods it is assumed we are referring to how well the algorithm compresses the data, not how long it takes to achieve this compression. Chapter three concludes with a comparison of an implementation of DMC and other well known methods, including PPM and GZIP.

2.7 Previous research involving DMC

Cormack and Horspool presented their original implementation of Dynamic Markov Modelling in a 1986 departmental research report [10], an identical paper of the same title was published a year later [11]. These original papers included a discussion of using the Guazzo [18] arithmetic coding algorithm to produce coded messages. They introduced the ideas of cloning the state model, effective initial models, and bounded memory readaptation of the model. Comparisons of various values for the two cloning parameters was given, together with the size of the resulting state models. Empirical results obtained from implementing DMC were compared with other well known methods of the time. These results however were theoretical because the authors did not actually implement a working coder or decoder. They instead argue that the achievable compression is equal to the entropy of the message with respect to the model, with arithmetic coding this assumption is of course correct.

Bell and Moffat presented a proof showing that DMC generates variable order contexts rather than finite state models [5]. The authors define Finite Context Automata (FCA), which are shown to be equivalent to variable order context models. The proof shows that DMC with its simplest initial model (figure 2-7) is equivalent to a FCA, and that cloning always produces another FCA. The proof is then extended to include other initial models.

A broad discussion of the current state of data compression is presented by Bell, Witten, and Cleary [3]. Both statistical and dictionary based compression techniques are discussed in detail, along with comparisons of current well known algorithms. A
section on state based statistical modelling details how DMC produces its predictive models. It also mentions suitable initial models useful for compressing text messages.

Elaboration of the content of [3] appears in Bell, Cleary, and Witten [4] and covers DMC in greater detail. Initial models for text compression are explored, and the idea of forgetting the state model when available memory is exhausted is discussed. The lengthy formal proof of [5] is also reproduced.

A concise overview of DMC is presented by Williams in a survey of data compression techniques [34]. The effect of the cloning operation on the state model and details of cloning on states containing self referencing transitions is presented.

An implementation of character based DMC (CDMC) is developed by Lawley [22]. CDMC proved to be a viable approach to data compression, however it couldn’t perform as well as the bit based DMC or the context modeller PPM. It appears that CDMC takes longer to learn new contexts than does PPM. For large source messages, however, CDMC should outperform PPM because of its ability to capture arbitrarily long contexts. In terms of compression speed, CDMC is over twice as fast as DMC and comparable with PPM, and while PPM has been highly optimised CDMC still has a lot of room for improvement. Unfortunately this implementation of CDMC, like its predecessor, suffered from large memory requirements.

DMC is mentioned briefly by Gutman [19] in a survey of data compression techniques.

Compression of languages other than English is mentioned by Yu [40]. In Yu’s experiments bit-wise DMC has achieved good compression of languages that require a 16-bit word to represent characters of their extended alphabets. DMC may therefore offer the ability to serve as a universal compressor for many natural languages. The author used a deterministic model to generate finite messages for which precise message entropies can be calculated. These message are used to analyze the behaviour of DMC and show that its model has not approached the entropy limit for these messages, implying there is still room for improvement.

A method entitled 'Hybrid Dynamic Markov Compression' is introduced by Yu [38] and uses a first order context model, coupled with DMC, to create higher order contexts. Using the context model for early predictions, the author claims improved
compression over traditional DMC when compressing text files of ten kilobytes in size [39].

Current ongoing research by Bunton consists of an implementation of DMC that has the ability to model source alphabets larger than the binary alphabet [6]. The author introduces lazy cloning, which instead of duplicating all transition outputs from a state, stores a pointer to its parent in anticipation of its symbol transition being required. In this way only transitions that are necessary in the cloned state are copied, if needed, resulting in model that uses surprisingly little memory. Building the model in such a way as to enable emulation of the PPM method using a strict state model has provided a new way to analyse the performance of PPM.

Much of the mystery surrounding DMC can be attributed to the fact that so few researchers have analysed its behaviour. While the algorithm itself is relatively easy to understand, the effects it has on the overall structure of the FSM are inherently difficult to visualise.
Chapter 3

Binary DMC

In this chapter the development of a DMC implementation using the information and examples contained in Cormack and Horspool's original paper is discussed [10, 11]. The algorithm, provided with a suitable binary source message, produces a compressed code sequence. While chapter two provided a conceptual outline of DMC, this chapter discusses issues pertinent to its successful implementation. Careful consideration of the data structures required to represent the model efficiently, including the consequences for compression speed, are discussed. Model regeneration, an important part of the original specification, is investigated, and conclusive results as to its usefulness are presented. Other important issues, including choosing an appropriate initial model, determining the ideal cloning thresholds, and implementing the modeller-coder interface successfully, are discussed. Finally, the results obtained from this implementation are compared with other successful compression methods.

3.1 Algorithm description

The Binary DMC (BDMC\(^1\)) implemented by the author, and described in this chapter, was developed initially from the ideas and examples given by Cormack and Horspool. They presented the outline of their algorithm used for cloning elected

\(^{1}\) Several acronyms for Dynamic Markov Compression are introduced, the common acronym DMC indicates that they form a related family of algorithms.
states in an enriched Pascal syntax. The language included additional operators for bit manipulation, namely logical shifts left and right, and bitwise logical "and" and "or". The algorithm developed from this outline evolves into successive DMC variations discussed later in this thesis.

Furthermore, Cormack and Horspool chose not to implement a functioning compressor, instead simulating the algorithm. Their DMC modeller produces a probability distribution suitable for coding source message symbols, and this distribution is used to calculate the expected compression if it was coupled with a suitable coder. The following section examines several implementation issues encountered while developing and integrating BDMC with an arithmetic coder.

### 3.2 Representation of the state model

One of the most important considerations when implementing any algorithm is the careful use of computing resources. BDMC requires large amounts of memory to store a representation of the model, and is computationally expensive during generation of this model. The cloning operation is performed, on average, 1.1 times for every character contained in the source message, so the memory required to store the model grows arbitrarily large very rapidly. The constraints of available primary memory in today's computer systems is definitely an influencing factor in the design of practical algorithms.

The specifications for BDMC's model can be represented by the tuple depicted in figure 3-1. A tuple represents a unique transition from one state of the FSM to another. How this tuple is stored in memory relates directly to the amount of memory required to represent the complete state model.

\[
\text{(current\_state, symbol, next\_state, transition\_count)}
\]

Figure 3-1 Tuple representing stored information for each state transition.

A tuple consists of a symbol, for which a transition from current\_state to next\_state will occur, and a transition\_count, used in calculating the symbols coding probability and to determine when states become eligible for cloning. Calculating a symbol's coding probability requires the total number of transitions from current\_state to be known. By definition, for the binary alphabet this total can be inferred by summing the transition\_counts of two transition tuples. Combining two tuples to form a single state
tuple removes the requirement of storing both current_state and symbol information, as it can be inferred from the state structure. Each state becomes a node of a FSM, interconnected by the next_state transition pointers.

3.2.1 Pointer based models

Encapsulation of states within a node structure implies the use of a pointer based implementation, with nodes containing two pointers to additional nodes of the model, one for each edge transition. Representing the model in this way is the natural encapsulation of the idea that the model contains a number of interconnected states. The major advantage of such an implementation being that individual states can be allocated as the model is generated, ensuring only the memory required is allocated by BDMC.

Use of pointer based structures does have drawbacks, the most obvious being the increased complexity required to handle pointer based objects. States are allocated individually, but the extra complication of dereferencing these pointer based objects has an impact on performance. Allocating many small blocks of data using modern operating system routines can also introduce a significant loss in performance. With many tens of thousands of states, allocating each one individually increases this performance overhead.

Each BDMC states requires a two byte transition_count and a four byte next_state address pointer for each alphabet symbol, a total of twelve bytes for each state. Both the extra complexity and large amount of storage required to represent physical memory addresses of states in the FSM detract from the usefulness of pointer based implementations. Using arrays to implicitly store these state addresses not only decreases the size of each state, but allows multiple states to be allocated simultaneously rather than individually.

3.2.2 Array based models

One of the best uses of memory resources is to store the BDMC model as an indexed array. This allows simple implementation because the complete state model can be stored in a one dimensional array of states, each state consisting of two transitions pointers and two transition counts, one for each alphabet symbol \{0,1\}. Such a representation for any state \(i\) is presented in figure 3-2. The next_state pointers are
simple indexes into the complete state array, and the transition_count variables represent the number of times each transition has been traversed so far. Use of an indexed state array allows fast random access to any state in the model. Index variables using as few as 16 bits, rather than full 32-bit memory addresses, are used to refer to individual states of the FSM. Restricting index variables to only 16 bits implies $2^{16}$ individual states are capable of being contained in the FSM. States are small and compact, implying excellent use of memory resources. An important consideration when using an array to implement the model is that it must be allocated in a contiguous memory block, a restriction imposed by operating systems and high level languages such as Pascal and C, but techniques do exist to circumvent this restriction.

Utilising arrays to store state information removes some of the redundancy associated with strict pointer-based implementations. This allows the size of the state structure to use an optimum amount of memory based on the maximum number of allowable nodes. A limitation imposed by arithmetic coding is that implementations using 32-bit fixed point integer arithmetic restrict the total range of a probability distribution to less than or equal to $2^{14} - 1$ (Max_Frequency). Each state requires a transition counter for each alphabet symbol that can therefore be represented using a 14-bit value. The two unused bits when implementing these counters using two byte values can be included in the array index, effectively quadrupling the maximum number of states. Thus if two bytes plus the additional two bits are used to represent the array index then a maximum of $2^{16}$ or 262,144 states are possible in the model, with each state occupying eight bytes.

An example of the structures required to represent this model are shown in figure 3-3, each stateStruct contains two outward symbolStructs, containing a next_state index and frequency transition_count. In summary, in the array model described

- each state represents two symbol transitions in eight bytes,
- the complete model can contain a maximum of 262,144 states, and
- the complete model requires only 2,048 Kbytes of memory.
Chapter 3. Binary DMC

### Chapter 3. Binary DMC

#### 3.2.3 Extending the array model

To exceed this limit of 262,144 model states, an array index providing a larger ordinal range is required. The number of states that a model can contain when indexed by an unsigned integer of length \( n \) bits is \( 2^n \). Two different methods can be used to increase the range of the array index.

- **Doubling the number of array states** is possible by decreasing the size of the transition counter by a single bit. The advantage of this is that the size of the internal representation of a state does not increase, so additional memory is not required to represent each state. Experimental results have shown that this method, however, decreases the performance of BDMC, in terms of both compression and execution speed. The adverse effect on compression can be attributed to the loss of precision, a result of reducing the maximum value the transition counter can represent. Access to the transition counter is more complex because surplus index bits must be masked out before using the variable, while the extra calculations required to adjust the transition counts so they fall below the maximum value accounts for additional time loss. A standard technique of avoiding counter overflow is discussed in section 3.4.1.

- The second method increases the range of the index variable by adding additional bytes to the state structure. This approach allows the greatest scope for large source messages, without affecting compression performance by altering transition count precision. The major drawback is the increase in

```c
#define ALPHABET_SIZE 2 // number of alphabet symbols
#define MAX_STATES 262144 // maximum number of states
#define MAX_FREQUENCY 16383 // maximum frequency for coder

typedef unsigned long state; // declarations of state and counter types

typedef unsigned short counter;

typedef struct {
    state next_state : 18; // 18 bits to represent next_state edge
    counter transition_count : 14; // 14 bits to represent symbol freq.
} symbolStruct;

stateStruct modei[MAX_STATES]; // outward transition for each alphabet symbol

state current_state; // BDMC model array

Figure 3-3 C structures for indexed array representing BDMC transition states.
memory required to represent models. As section 3.9 shows, the model described is capable of compressing most source messages efficiently.

3.2.4 Practical usage of array models

Representing the model using an array offers efficient and practical implementation. Only a single memory allocation is required, compared with several small allocations of a pointer based implementation. In a single process operating system processes are not competing for system resources, but modern operating systems allow processes to execute concurrently. Modelling sources with large contiguous arrays in such an environment is disruptive to other processes because of the waste of valuable computer resources. Memory that BDMC requested might have been better utilised by other or additional processes.

Implementing states within a fixed length array requires that the maximum model size be known before compression commences. This size can either be a constant of the algorithm, calculated from the total amount of memory available, or estimated from known characteristics of the source. These solutions all have several drawbacks.

- If the memory for the model is unavailable then modelling cannot begin.
- If the generated model only occupies a small portion of the total available model space then memory is wasted. Other processes that require memory will not be able to take advantage of this unused memory, or may be unable to begin execution. On the other hand, if the model grows to the maximum model limit, there is no elementary way to increase the size of the model beyond this limit
- If the model is restricted by the available memory, there are consequences for compression performance. Methods to overcome this restriction include ceasing to adjust the model, completely reconstructing the model and resuming coding, or even partial model reconstruction. These techniques are discussed in section 3.3.

One solution to the bounded memory problem that seems practical is to estimate the amount of memory required based on the source message length. Compression methods are typically used in an environment where the length of the source is known, and this approach would be of value when limited memory is available. It is possible for BDMC to determine an appropriate model size if the length of the source message is known. A smaller model is suitable for many files, while an extremely
large model only benefits large files. BDMC would then use roughly the amount of memory it needed to perform its best average compression, leaving any additional memory for other processes. Figure 3-4 shows the maximum model size needed for source messages of varying size. Over a thousand files were encoded with files sizes ranging from 1 to over 500,000 Kbytes. The results show a reasonably linear relationship between file and model size, implying the cloning operation doesn't stop or slow down during compression. Given this linear relationship, it can be used to predict an approximate maximum model size for source messages of varying length. This knowledge could be used in an implementation on a platform where memory resources are in high demand. Of course, when the length of the source message is unknown, BDMC can not offer any help in deciding the model size, and must use some fixed maximum size.

![Figure 3-4 Maximum model sizes for varying source message lengths.](image)

### 3.2.5 Allocating arrays in chunks

An elegant solution to the problems associated with allocating large contiguous arrays would enable the amount of memory used by the modeller to grow adaptively. This would not only benefit other concurrent processes in a multi-process system, but it would also allow the memory usage to grow in steps up to some arbitrary limit. Such a method works by increasing the model's state array each time a new state needs to be cloned and the current model limit has been reached. Implementation of such a scheme can either be handled internally by BDMC, or externally by the operating system's memory management routines. Both techniques have their limitations and drawbacks.

- Allocation of a model using several fixed size chunks offers the greatest flexibility in a multi-process or low memory system. Implementing such a
scheme requires some form of internal memory management to control allocation of chunks of $k$ states. It also requires specialised calls to return the correct state given a states index value. This technique adds to the complexity of BDMC but greatly improves memory handling.

- Modern operating systems allow a chunk of memory to be dynamically extended at run time. In other words, if the size of the array needs to be increased, the system can create a larger contiguous block of memory, copy the values held in the existing array, and return the new larger array. Of course, the major drawback of this method is the fact that a chunk of contiguous memory at least as large as the new array must be available for this method to work. Performed too often this method increases the probability of memory fragmentation – even though there is memory available, it is not in contiguous blocks and so cannot be used for one array.

Both methods provide a convenient way to utilise available memory without disrupting other processes. They enable an initial model size to be allocated, which can be increased if the source message requires a larger model. Small source messages might initial use only the required memory, while large source messages can be modelled with the appropriate amount of memory.

### 3.2.6 Consequences for the decoder

A consequence of adaptive modelling is the important repercussion that both the encoding and decoding operation must generate identical models. The encoding operation works by coding the symbol with respect to the current state of the model using arithmetic\_encode and the state's probability distribution, then updating the model to reflect the occurrence of the symbol just coded. This process continues until no more symbols are left in the source message, when an end of file marker is coded.

The decoder receives a series of coded sequences, which must be correctly translated back to the original symbols for the complete sequence to be decoded correctly. To achieve this a decoder starts with exactly the same initial model, reversing the encoding process by repeatedly obtaining a target value from arithmetic\_decode\_target, which has been supplied with the upper bound of the current state's probability distribution. The value received falls in the cumulative range of a symbol in the states probability distribution, determining the next coded symbol in the sequence.
Calling \texttt{arithmetic\_decode} removes the coded bit sequence ready for the next symbol to be decoded, and the model is updated using exactly the same method as when coded. This complete process is repeated until finally the end of file marker is decoded.

The model is always updated after the encoding or decoding of a symbol, ensuring that both the encoder and decoder adaptively generate identical models. If this wasn't the case the decoder wouldn't be able to extract the next symbol correctly, because the current states wouldn't be identical.

Information about how the model was generated also needs to be available to the decoder. BDMC stores this information in a header block which is transmitted before coding proceeds. This header includes the model's memory limit, which enables the decoder to allocate exactly the required amount of memory, and various other modelling parameters. If this memory isn't available then decoding the message cannot proceed.

### 3.3 Model regeneration

While BDMC works best when large amounts of memory are available, it is possible to compress efficiently using a limited amount of memory. Several alternative methods have been implemented in an effort to compare various memory recovery techniques.

In its simplest form BDMC stops adjusting the state model when its memory is exhausted. This has the advantage of being extremely fast once the memory limit is reached, because the computationally expensive cloning operation no longer occurs, and the modeller simply traverses a single transition as each bit symbol of the source message is read. However, if the remaining source message is not represented accurately by the now static model, or the remaining message changes dramatically from the models representation, poor coding will result.

An improvement on this technique stops adjusting the \textit{structure} of its model, but continues to adapt the model by incrementing symbol frequencies as transitions are traversed. The structure of the model doesn't change because new states cannot be created, but adjusting the transition frequencies does offer some ability to continue adapting to the symbol frequencies in the remaining source. Combined with count
scaling and frequency halving, recently seen symbols are more heavily weighted to improve compression. Details of these compression techniques follow in section 3.4.

An adaptation of this method is the drastic approach of completely replacing the model with the initial model every time the available model memory is exhausted. When the structure of the source message changes rapidly this method produces better results than the static model. This technique has the ability to begin generating a model based on symbols of recent context in the source message. The cumulative compression of BDMC when applied to typical files is examined in figure 3-5, showing that it requires nearly 1,000 source characters before the model settles down and produces intelligent probabilities for the coder. This implies that to be most effective the initial model used to regenerate this model would need to contain over a thousand states. While this method offers the ability to adapt locally to the source message when memory is exhausted, in practice it takes far too long readapting to be effective.

![Figure 3-5](image)

Figure 3-5 Cumulative compression of BDMC for the corpus files book1 and obj2. The horizontal line represents an uncompressed source of eight bits per character.
3.3.1 Model regeneration using a history buffer

In an effort to improve compression on the previous method a further modification is added. The problem of supplying a sufficiently large initial model so that regeneration has a positive effect on compression has been investigated. This method maintains a cyclic buffer of recent message symbols, and is used to generate the new model. Maintaining this buffer not only enables a relatively large model to be captured, but it also supplies an accurate recent context for the succeeding symbols of the source message. This method should therefore offer the ability to adapt locally to the source by regenerating a model that represents the immediately preceding symbols of the source message, reflecting context characteristics of the symbols previously coded. Figure 3-5 would suggest that to be effective an extremely large buffer of previous symbols needs to be fed back into the model each time it is regenerated, making it restrictive in terms of the additional resources required. Using a much smaller buffer can be effective for large object files because of their inherent changing nature.

Just as BDMC creates its FSM from an initial model, each regeneration using the history buffer also must begin from a similar initial model. In practice it has been found that the same conclusions as for initial models apply, section 3.5. Model regeneration using a history buffer from an initial binary tree of depth eight has consistently proven to give the best average compression. In fact, using a single state initial model for regeneration results in worse average compression than regeneration using just a binary tree of depth eight without buffering. It is not until a history buffer of at least 42,000 characters is used that the regeneration method using this initial model outperforms it.

The results of adjusting the size of the rebuild buffer are shown in figure 3-6. All regeneration was performed using a binary tree of depth eight as the initial model. Results were gathered with BDMC operating using 1,024K of memory for both the model and cyclic buffer combined. Several of the corpus files are shown with the compression at various buffer sizes. The corpus files not shown either didn’t use all the available memory, hence model regeneration didn’t occur, or they showed little variation as a result of altering the buffer size. This graph shows how the number of characters stored in the cyclic buffer has an approximate linear relationship to the compression performance for various corpus files as well as for the average compression.
Supplying BDMC with as large a buffer as possible will produce better compression but it requires more memory, takes longer to regenerate the model, and will regenerate more often because less of the remaining source message can be modelled before regeneration is once again required.

![Figure 3-6 Effect of compression by altering the size of the buffer used for model regeneration when memory is exhausted. BDMC used 1,024 Kbytes of memory for these results.](image)

### 3.3.2 Memory recovery performance

The average compression of the corpus for each memory recovery method discussed is shown in table 3-7. The first method was unrestricted in its memory usage. The second and third methods were both restricted to 1,024 Kbytes of model memory. The former stops any adaptation when this limit is reached, while the later continues to allow transition frequency updating and frequency count halving. (See section 3.4 for a discussion of these techniques.) The fourth method regenerates a model using the supplied initial model, in all cases a binary tree with a depth of eight. The fifth and sixth methods both regenerated their models using a cyclic buffer of 30,000 characters. The only difference in these two methods is in the way they allocated
their model and buffer memory. The former always uses a total of 1,024 Kbytes for both model and buffer, while the later allocates additional space for its buffer so required a total of 1,054 Kbytes for both model and buffer. Both methods started regeneration using a binary tree of depth eight.

<table>
<thead>
<tr>
<th>Method</th>
<th>Average Compression (bpc)</th>
<th>Normalised Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlimited memory</td>
<td>2.52</td>
<td>1.00</td>
</tr>
<tr>
<td>no model adaptations</td>
<td>2.88</td>
<td>0.95</td>
</tr>
<tr>
<td>transition freq. updating</td>
<td>2.73</td>
<td>1.00</td>
</tr>
<tr>
<td>regeneration (initial)</td>
<td>2.84</td>
<td>1.01</td>
</tr>
<tr>
<td>regeneration (buffer)</td>
<td>2.76</td>
<td>1.21</td>
</tr>
<tr>
<td>regeneration (buffer extra)</td>
<td>2.76</td>
<td>1.21</td>
</tr>
</tbody>
</table>

Table 3-7 Comparisons of memory recovery methods for BDMC.

It can be seen from these results that BDMC using an unlimited amount of memory performs better than any other method. This is not a surprising result because BDMC is known for its excessive cloning which creates large state models very rapidly. It is interesting to compare the compression achieved by the two methods that cease cloning states when the memory limit is reached. Simply stopping frequency counting resulted in the worst average compression of all the methods tested. Continuing to adjust the transition frequencies offers a significant improvement. It appears that BDMC's model has captured enough structural information to be effective in coding the remaining source without the necessity of regenerating the model. The effect is clearly shown in figure 3-8, where the instantaneous compression closely follows that of the method using unlimited memory. This method performs better than all the model regeneration methods, although by increasing the cyclic buffer size to over 53,000 characters the regeneration methods will give better compression. Stopping frequency counting obviously provides no subsequent way of adapting to the remaining source, resulting in fluctuations as in figure 3-8.

Performance of the three regeneration methods has several explanations. The buffer regeneration methods performed better than simply using an initial model because a
context of previous characters is being supplied. This context provides statistical information about previously encountered symbols and improves the probabilities generated by the modeller. The buffer regeneration methods are approximately twenty percent slower because the modeller must not only update the buffer after every coded character, it must also build a larger model each time regeneration occurs. Leaving less model space available after each regeneration for the remaining source, resulting in an increased number model reclaims. In fact, using the initial model only required ten reclaims, while the other two methods each required sixteen.

![Graph showing comparison of instantaneous compression of corpus file book1 using different methods of adapting BDMC's model during low memory operation.](image)

In summary, the best average compression is obtained using BDMC with an unlimited amount of memory, but it is impractical to assume that this is always available. Methods that restrict the memory usage have been investigated, with the result that halting state cloning but continuing to adjust and halve transition frequencies offers the best compression and throughput performance combination.
3.4 Scaling of transition frequencies

Without transition frequency scaling symbol frequencies would rarely rise above two. This is a consequence of the low threshold values used to trigger BDMC's cloning operation and the use of integers to record these frequencies. The probabilities supplied to the arithmetic coder lose a lot of information that is discarded in rounding errors. These probabilities could all be represented by a small finite set of possible probabilities, reducing compression significantly.

To solve this problem, transition frequencies could be represented using floating point numbers, but this would increase the computation involved because of the required floating point arithmetic. A less computationally expensive approach is to use fixed point integers to simulate floating point numbers. This is accomplished by incrementing the transition frequencies by a scaled count. This value is known as the scaling factor, and gives greater emphasis to symbols that have been seen over those that haven't.

As an example, consider a transition edge for the symbol zero that has never been traversed, while the edge for the symbol one has been traversed twice. Both edges start with an initial frequency count of one. Without scaling their transition frequencies are one and three, yielding the probabilities \( \frac{1}{4} \) and \( \frac{3}{4} \) for the coder respectively. The problem is that even though the transition for the symbol zero has never been used, it has a relatively large probability compared with the symbol one. By incrementing encountered symbol transitions by a scaling factor, for example 32, the frequency for symbol zero remains one, but because the transition belonging to symbol one has been traversed twice, its frequency would now be 65. This yields the probabilities \( \frac{1}{66} \) and \( \frac{65}{66} \) for the coder, giving a much stronger emphasis to the symbol that has been used previously.

Cormack and Horspool do not discuss the dramatic effects of frequency scaling even though they did apply a scaling factor of 1,024. With transition counts represented in 14 bit values, such a large scaling factor causes the transition frequency to overflow rapidly. Because Cormack and Horspool did not use an arithmetic coder but calculated the expected compression using 32-bit transition frequencies, this was not a problem. The dramatic effect caused by scaling can be seen in figure 3-9, and shows the ideal scaling factor for BDMC lies in the range \([20...100]\). All other results in this thesis used a scaling factor of 32.
3.4.1 Recovering from frequency overflow

A consequence of the effect of frequency scaling is the increase in the probability that the total frequency of the probability distribution will exceed the maximum allowable value, Max_Frequency. As mentioned previously arithmetic coders restrict the frequencies to be represented using 14 bit values, which makes it likely they will overflow occasionally as larger scaling values are used. The method often used to recover from this overflow involves halving all symbol frequencies when the total frequency threatens to exceed Max_Frequency, but ensuring that no count is left at zero to avoid the zero frequency problem. Such cases are corrected by explicitly setting the frequency to one. Reducing frequencies by half has little effect on the average code length, and in fact periodic halving of frequency counts can actually improve compression. By halving the frequencies, all symbols keep the same relative frequency, but the effects of scaling dramatically increase the relative probabilities of symbols encountered in the near future. Better compression is achieved because the model is able to adapt to recent symbols, while gradually reducing the relative
probabilities of symbols that were once popular. However Kirchoff's laws are not maintained, a consequence of only halving the frequencies of popular states, but the relative frequencies of symbols do remain approximately identical. Halving a states frequency counts does have a detrimental effect on the cloning criteria, this is soon countered as popular symbols frequencies are traversed. BDMC achieves its frequency halving by applying a simple bitwise right shift as demonstrated in figure 3-10, the prefix increment ensures no frequency count ever reaches zero.

if (TOTAL(current_state) > MAX_FREQUENCY) {
    current_state->sym[0].transition_count = ++current_state->sym[0].transition_count >> 1;
    current_state->sym[1].transition_count = ++current_state->sym[1].transition_count >> 1;
}

Figure 3-10 Halving transition frequencies.

3.5 Initial models

As mentioned in chapter two, BDMC begins cloning states from a given initial model quickly adapting to any source message. Different initial models can have an effect on the obtainable compression of messages. Initial models can either be generated from fixed structures such as binary trees, or generated from a predetermined source message, as is the case used for model regeneration. Several suitable binary structures are described in chapter two. These structures each offer the ability to create binary representations of different depth. Depths that are a power of two have proven to give the best results, and figure 3-11 supports this, showing the resulting compression for various depths.

What is surprising about these results is that no matter what type of initial model is used, as long as its depth is a factor of eight the average compression performance will be adequate, while choosing a model depth that is not a factor of eight results in poorer compression. As long as some form of byte structure is captured in the initial model, BDMC has a chance at adapting to the source. Providing an inappropriate initial model makes it much more difficult for BDMC to recover from the bad start. Adapting from an initial model with no structure actually results in better compression than adapting from an inappropriate initial model.

The pleasant result that all types of initial models give similar average compression shows that is not necessary to provide a complex initial model, such as a braid, as
long as the model is an appropriate one. The best initial model for BDMC is a tree of depth eight, and this initial model is used for all other results shown in this thesis.

3.6 Cloning variables

Perhaps the most influential factor on BDMC’s compression performance are the two cloning thresholds $t_1$ and $t_2$. These determine when a state transition becomes eligible for cloning, consuming resources by creating extra states in the FSM. Cloning therefore should only be performed when it is likely to be productive in the future. Transitions with high frequency counts have, by definition been traversed often and are therefore more likely to be traversed in the future. Consequently they are excellent candidates for cloning, as long as other transitions into the state also have significant frequencies. Appropriate choices for the two thresholds can be determined empirically. Figure 3-12 shows the effect of varying both of these cloning thresholds.
Figure 3-12 Effect on average corpus compression of adjusting the two cloning thresholds.

Low cloning thresholds enable the model to grow rapidly very quickly, while increasing these thresholds causes the model to clone less rapidly and hence slow its adaptation. Values of one and eight for the thresholds $t_1$ and $t_2$ respectively produces the best average corpus compression. This suggests that allowing the model to grow rapidly very quickly enables the model to capture contextual information early.

### 3.7 Combining the coder

Chapter one emphasised the separation of the modeller and coder, giving details on how a modeller interfaces with the arithmetic coder. This chapter has so far focused on implementing BDMC as a method of modelling source messages. One issue that
needs consideration when using BDMC with an arithmetic coder is how to terminate a sequence of binary predictions – specifically, how to signal to the decoder that the last symbol of the message has been decoded.

The coder uses a distribution that predicts the probability that each symbol of a finite alphabet will occur, together with the symbol that actually occurs to produce a minimal length code. It is important that this finite set of possible symbols include all symbols that may occur. For a binary alphabet as used by BDMC this set contains \{0,1\}. When modelling a source, BDMC also needs to take into account the fact that an end of file symbol could be encountered at the end of any byte. Two approaches to resolving this problem are discussed. The first operates at the bit level while the later at the character level.

- The first solution involves adjusting every state’s total transition frequency to enable an end of file to be coded if required. This normally involves implicitly incrementing the total number of transitions from a state by one. Very little compression is actually lost due to this small adjustment. The frequency counts for symbols are normally much larger than one, resulting in a large proportion of the distribution still being allocated to the symbols. When the end of file is actually encountered it can be coded with a probability of \( P_s(eof) \). This operation is only performed once, requiring at most 14 bits to code due to the minimum probability of \( \frac{1}{2^{14}} \) capable of being coded. The equations presented in chapter two require alteration as follows to allow for the eof symbol.

\[
\begin{align*}
P_5(0) &= \frac{n_5(0)}{n_5(0) + n_5(1) + 1} \\
P_5(1) &= \frac{n_5(1)}{n_5(0) + n_5(1) + 1} \\
P_5(eof) &= \frac{1}{n_5(0) + n_5(1) + 1}
\end{align*}
\]

The drawback of this method is that code space needs to be reserved for every bit of the source message to be coded, rather than every character that is coded.

- The second solution involves sending a message to the decoder before every character of the source is coded. The message code enables the decoder to determine whether the next symbol will be a character (not_eof message) or the end of file symbol (eof message). Following each not_eof message the bits of the
character are coded normally as individual binary symbols. This process continues until the end of file is encountered, when the \textit{eof} message is coded. This method has the advantage of coding the end of file at the character level rather than allocating room for it in every state at the bit level. To be successful the \textit{not_eof} message probability supplied to the coder must be as large as possible to reduce the additional code length. This probability is related to \textit{Max\_Frequency}, the maximum frequency that can be supplied to the coder. Coding each \textit{not_eof} message before a character requires 0.0000881 bits, while coding the \textit{eof} message requires 14 bits. This method is simpler to implement than the previous, but does mean that the coding routines are called nine times for every source character. The probabilities for the \textit{not_eof} and \textit{eof} characters are,

\begin{align*}
P(\text{not\_eof}) &= \frac{\text{Max\_Frequency} - 1}{\text{Max\_Frequency}}, \\
P(\text{eof}) &= \frac{1}{\text{Max\_Frequency}}.
\end{align*}

BDMC uses the method involving coding a \textit{not_eof} message before each character, which is both simpler to implement in the encoding and decoding algorithms, and more efficient. Coding the single \textit{eof} message does not greatly affect the compression for most files, as only fourteen bits are required to code it.

### 3.8 Using BDMC

An implementation of BDMC developed by the author has several command line switches that control the various model options discussed in this chapter. Appendix A includes full details of all command options. Without any switches BDMC displays a summary of all available options as follows.

```plaintext
usage bdmc
   -el-d
where: -e encode stdin to stdout
     -d decode stdin to stdout (other switches ignored)
     -c instantaneous compression every nnn characters
     -C cumulative compression every nnn characters
     -g create DAG input file at nnn states [Off]
     -i initial model Single\textbackslash Chain\textbackslash Tree\textbackslash Braid of size nnn [Tree8]
     -m restrict to nnn kaytes of memory [1024K]
     -M restrict to nnn thousand states
     -n restrict to nnn source characters
     -r recover from memory overflow using method n [1]
```
3.9 BDMC evaluation

A summary of the ideal parameterisations for BDMC is shown below. Each of these various parameterisations has been discussed and comparative performance has been measured in a working implementation of BDMC. The preferred parameters are:

- Values of one and eight for the cloning thresholds $t_1$ and $t_2$ respectively (3.6).
- A binary tree of depth eight for the initial model (3.5).
- A transition frequency scaling factor of 32 (3.4).
- No model regeneration, but transition frequency updating and frequency halving if memory is exhausted (3.3.2).

Figure 3-13 shows the compression performance of BDMC measured against two other well known compression techniques. The first is PPMC, regarded as the leading statistical compressor. It uses finite contexts to predict symbol probabilities [25]. The second is GZIP, which is a high performance LZ77 based variant, tuned for both speed and compression, and distributed by the Gnu Free Software Foundation. Figure 3-13 show the compression (in bits per character) for the fourteen files of the Calgary corpus, plus the average compression for each algorithm.

BDMC performs very well compared with the others, and together these three algorithms are among the best compression algorithms known today. BDMC performs particularly well on the files containing text, but poorly on the files containing program source and executable object code. PPMC is the best method overall with the lowest average compression, followed closely by BDMC.

---

2 GZIP was implemented by Mark Adler, Richard Wales, and Jean-Loup Gailly of the Info-ZIP group.
Examination of the throughput of these algorithms shows a considerable difference (figure 3-14). The most dramatic result is that of GZIP's decompression speed, which outperforms the others by a factor of ten. This a result of requiring just a single lookup in a phrase dictionary to output large chunks of characters. BDMC is the slowest of the three at about a third the speed of PPMC.
3.10 Summary

This chapter has examined several important issues arising from the development and implementation of BDMC. These were presented together with factors influencing overall compression such as initial models, frequency scaling, and adjusting the cloning thresholds. While BDMC has the ability to adapt well to any source message, it consumes arbitrarily large amounts of memory to achieve this. In a practical situation BDMC also suffers because of its poor compression speed compared with the fast dictionary methods in use today. The first shortcoming can be overcome by implementing BDMC as described in this chapter using a fixed or staggered memory allocation technique. Once a satisfactory amount of memory is obtained, the cloning must be limited, and updating transition frequencies results in very little compression loss.

Fixing BDMC's slow execution time is a more complicated problem. The algorithm itself is not complicated, several simple operations are performed through each iteration. Even with considerable hand coding using assembly language the significant improvement needed to match the faster methods can not be achieved. The problem is that BDMC operates at the bit level, executing the main loop eight times per source character, compared with other well known methods that operate at the character or phrase level. To overcome this problem DMC needs to be implemented using a larger source alphabet, such as nibbles (4 bit symbols), or bytes (8 bit symbols). Several additional considerations need to be considered when implementing DMC to operate on larger source alphabets, and this is the subject of the following chapter.

Another problem is the execution of the arithmetic coding routines required for every bit transition traversed by the modeller, these routines are computationally expensive. This problem is partly fixed by using a larger source alphabet, but a method that removes the dependency on arithmetic coding routines completely is discussed in chapter six.
Chapter 4

Character DMC

In this chapter DMC is extended to model alphabets consisting of more than two symbols. The previous chapter examined in detail an implementation of DMC modelling binary alphabet symbols. In practice such a method gives excellent compression, but suffers from very slow execution speed, and large memory requirements. Modelling bits rather than characters means that a file of 100 Kbytes requires over 800,000 iterations of the main coding loop. This factor of eight can be eliminated by exploiting the benefits of a larger source alphabet. More bits will be coded during each iteration, reducing the total number of iterations, although each iteration could take longer.

This chapter begins with a short discussion of source alphabets, followed by the details of an initial implementation used to collect state occupancy statistics of a character based model, which is useful information for determining the parameters for a compact and efficient data structure. A discussion of the development of practical data structures and methods for accessing the information contained in these structures follows. Dealing with sparse symbol sets, and coding novel symbols is covered next, followed by model updating routines, model cloning parameters, and methods for improving symbol search speed. Finally, visual representations of the FSM and effectiveness of the model generated are examined, concluding with the overall performance of this new method compared with other compression techniques in use today.
4.1 Source alphabets

In data communications the set of distinct symbols used in a source message is referred to as an alphabet. One of the smallest alphabet consists of the two binary symbols, zero and one. Every possible binary source message can be represented using a sequence of these two symbols. Larger alphabets capable of representing all possible binary source messages consist of $n$-tuples of binary symbols. For example an alphabet consisting of a pair of binary symbols forms the set $\{00, 01, 10, 11\}$.

English text requires a minimal alphabet containing 27 characters, consisting of the letters $\{A...Z\}$, plus the space character. In practice, a much larger alphabet is required, including upper and lower case letters, numerals, and punctuation symbols. English text is normally represented by a standardised code such as ASCII (American Standard Code for Information Interchange [1]). The ASCII code (table 4-1) is defined with seven significant bits capable of representing 128 unique symbols, 95 of which are printable characters, while the remainder are non-printing control characters. During transmission a parity bit is often added, while for reliable media the addition of a zero as the most significant bit facilitates storage in a single byte. No eight-bit ASCII standard has been defined, but many computers utilise the available bit for extending the standard ASCII set. The Macintosh character set [9], for example, contains 256 symbols, including foreign language characters, simple mathematical symbols, and additional punctuation characters. Foreign languages such as Chinese and Arabic contain many more alphabet symbols and therefore must be represented using multi-byte symbol codes.

<table>
<thead>
<tr>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>nul</td>
<td>dle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sof</td>
<td>dcl</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>stx</td>
<td>dc2</td>
<td></td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>etx</td>
<td>dc3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>eot</td>
<td>del</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>enq</td>
<td>nak</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>ack</td>
<td>syn</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>bel</td>
<td>etb</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>bs</td>
<td>can</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>tab</td>
<td>em</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>if</td>
<td>sub</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>vt</td>
<td>esc</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
</tr>
<tr>
<td>ff</td>
<td>fs</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>gs</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>so</td>
<td>rs</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>si</td>
<td>us</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>del</td>
</tr>
</tbody>
</table>

Table 4-1 ASCII code set, groups of letters are non-printing control characters, in particular: cr = carriage return, lf = line feed.
Other forms of data such as executable object code, images, or numeric data, generally have no common alphabet domain. Determining an appropriate alphabet for every kind of data is very difficult, although a suitable compromise is to use an alphabet of \( n \)-tuple binary symbols. Varying \( n \) changes the domain size for the alphabet. For example \( n = 1 \) defines a two character alphabet, \( n = 4 \) a nibble alphabet (four-bit characters, or 16 symbols), and \( n = 8 \) a byte alphabet (eight-bit characters, or 256 symbols). Historically an eight-bit character alphabet is widely used because it represents a single byte, one of the fundamental computer data units. Such an alphabet contains 256 unique eight-bit symbols \{0...255\}, with the advantage that it corresponds to the size of the alphabet used to represent English text. Extending DMC to an eight-bit character alphabet (CDMC) allows modelling of a generic data source without the need for special cases to handle different types of data.

4.2 Initial implementation considerations

This section describes an implementation of CDMC developed to determine the characteristics of states contained in the FSM. This information is useful for developing efficient data structures capable of storing state information. This section begins with a discussion of the structure of each state within the FSM, followed by some considerations required when using an arithmetic coder. Finally, a short discussion of the preliminary results obtained from this implementation is presented.

4.2.1 Representing States

DMC's model consists of a FSM where every state contains a number of symbol transitions to other states contained in the model. Each symbol is a member of the finite source alphabet. In BDMC each state contained two possible transitions, one for each of the source alphabet symbols \{0,1\}. With the expanded source alphabet of CDMC each state must now contain an edge transition for every possible alphabet symbol \{0...255\}. This implies 256 state transition pointers, and 256 transition frequency counts in each state, figure 4-2. Such an implementation will require a large amount of model memory. For example an implementation using dynamically allocated contiguous arrays, with each transition counter requiring 14 bits, and transition pointers represented by 18 bit array indexes, will require 1 Kbyte of memory for each state. A maximum of 262,144 possible states requiring 256 Mbytes of memory could be used. Clearly such an implementation will be of little use in a
practical situation, although it does enable practical evaluation of more suitable data structures capable of representing a CDMC model.

Allocating these states dynamically as the model grows increases the state structure size because the next_state transition pointers must be 32-bit memory addresses. Each state now requires 3 Kbytes of memory, although this approach is a more flexible implementation for research. Generating models for the corpus files using such an implementation of CDMC results in memory requirements as high as 180 Mbytes (for approximately 60,000 states).

BDMC’s cloning operation consists of duplicating the next_state to be visited and reassigning outward transition frequencies for the existing and newly created state. These new frequencies are in proportion to the frequency of the original inward transitions. The cloning of BDMC states involves duplicating two outward state pointers, and recalculating values for four transition frequencies. With the increase in size of the state structure of CDMC comes the additional complexity of performing this fundamental cloning operation. In this case each state requires 256 symbol transitions to be duplicated and 512 frequencies to be recalculated using floating point arithmetic, or integer approximations.

One method for decreasing the large number of transition symbol duplications and floating point calculations during state cloning is currently being researched by Bunton [6]. Rather that duplicating all outward transitions when a state is cloned, lazy cloning delays duplicating the transitions until they are required in the cloned state. This is achieved by creating an escape path to the original state, and utilising this path whenever an unseen symbol is encountered in the cloned state. These paths are traversed until the appropriate symbol is found in a parent state, the symbol is then copied to the cloned state. Therefore only symbols actually required will be copied to the cloned state. Lazy cloning does not, however, generate an identical predictive distribution as the generic cloning operation. Not only may the symbol transition of the parent state be traversed many times before it is copied, but the ratio of transition frequencies is not maintained when reassigning these counts. The focus of this thesis has been to duplicate the traditional cloning of DMC for larger source alphabets.
4.2.2 Arithmetic coding interface

Other factors also need to be considered as a consequence of each state containing 256 transition symbols. The arithmetic coding interface requires a cumulative symbol probability to be supplied, together with the total outward transition frequency from the state. BDMC’s method of calculating the total frequency involved summing just two outward transition frequencies. This operation applied to a CDMC state results in the summation of 256 transition frequencies, for every source character coded, which is prohibitively slow. Storing a total transition frequency count in each state not only provides an efficient method of retrieving the frequency total, but it also facilitates transition frequency overflow checking.

Another consequence of supplying a probability range to the arithmetic coder is that some order must be imposed on the symbols in each state. This is because when decoding a message the arithmetic coding routines return a value that falls within the range supplied during encoding. If the order of the symbols is different during the encoding and decoding operations there is no way of determining the correct symbol to decode and the operation will fail. The simplest and most intuitive symbol ordering for the 8-bit alphabet is by linear byte ordinate. This means that a single array access can be used to locate the transition frequency for alphabet symbol \( k \), but \( k - 1 \) accesses are required to calculate the cumulative total of frequency transition for symbols before \( k \) in state \( S \). The cumulative frequency before symbol \( k \) is represented by

\[
C_s(k) = \begin{cases} 
0 & k = 0 \\
\sum_{i=0}^{k} n_s(i), & k = 1, \\
\sum_{i=0}^{k-1} n_s(i), & k \geq 2 
\end{cases}
\]

Several alternative methods for decreasing the significant overhead of calculating this cumulative total are discussed later in section 4.10. Figure 4-3 shows how this calculation can be performed. Decoding a target value \( t \) provided by the arithmetic decoding routine \( \text{arithmetic
decode
target} \) involves determining the symbol whose cumulative range encloses it, so that \( t \) falls in the range

\[
C_s(k) \leq t < C_s(k) + n_s(k).
\]

Figure 4-4 gives code that locates this frequency range by stepping, in order, through each symbol of the alphabet, maintaining a cumulative transition frequency count, and locating the required frequency range for any given \( t \).
Chapter 4. Character DMC

4.2.3 Preliminary results

An initial implementation of CDMC provided promising results both in terms of achievable compression and execution speed. However the major benefit of such an implementation is in determining the number of unique symbol transitions utilised in each state so that more efficient data structures can be designed.

Figure 4-5 displays a summary of the average symbol transition occupancy statistics of CDMC states for all files of the Calgary corpus. From these results assumptions about the requirements for a data structure capable of representing CDMC states can be formed. These results showed that on average a maximum of 150 of a possible 256 transition symbols are represented in a state. Very few states contained such a large number of transitions – in fact, over 99% of all states contained fewer than 50 symbol transitions. This fact is useful in determining an upper limit on the number of transitions capable of being represented in each state. On average 90% of the states contain 30 or fewer symbol transitions, implying that most of the time states would be able to contain only 30 transition symbols and experimental comparisons justifying this statement.
4.3 Initial state specifications

With this result that only a small proportion of symbol transitions are used from any CDMC state, together with several additional assumptions and facts outlined below, a suitable data structure can be determined as follows.

- Each state has several outward transitions leading to other states in the FSM implying next_state transition pointers are either array indexes representing individual states or full 32 bit memory addresses of state structures.
- Outward transitions have an associated symbol contained in the finite source alphabet, and all outward transition symbols are unique.
- Initial experimental results have shown that the number of symbols represented in each state is relatively small compared to the actual source alphabet size, that is, there is a sparse transition alphabet for individual states. Figure 4-5 suggests...
that as few as 50 transitions from an alphabet of 256 possible symbols could be represented in each state.

- There is little correlation between the alphabet symbols represented in each individual state, in other words, the distribution of symbols in each state is distinct. Therefore no assumption can be made about which symbols will never be encountered, and no symbols can be excluded from the source alphabet.

A data structure capable of representing multiple occurrences of a tuple containing \((\text{symbol}, \text{next} \_\text{state}, \text{transition} \_\text{count})\) members is required in each state. This structure must also facilitate the following operations needed to supply statistics for the arithmetic coder together with some standard operations augmenting them.

- `get\_count(k)`: return the current frequency count of symbol \(k\).
- `inc\_count(k)`: increment the frequency count of symbol \(k\).
- `get\_cumulative\_count(k)`: return the sum of the frequency counts of symbols that precede \(k\), providing suitable probabilities for the coder.
- `add\_symbol(k)`: insert the symbol \(k\) in the states list of symbols.
- `get\_symbol(t)`: given a value \(t\) return the symbol whose cumulative range encloses it.

This requires a structure that provides access to individual symbol information, together with an appropriate order to enable calculation of cumulative ranges for the coder.

Determining the cumulative range for any symbol \(k\) can be achieved using one of two distinct methods. The simplest method involves summing all the symbol transition frequencies before symbol \(k\) for some ordering. The advantage is no additional space is required for cumulative frequencies, and incrementing symbol frequencies involves a single counter update. The second and more complicated method involves storing a cumulative range for each symbol rather than its transition frequency. Each cumulative range is represented by a lower and upper bound, where the upper bound can be inferred from symbol \(k+1\)'s lower bound. Determining the cumulative range of any symbol therefore requires only two lookups, but at the expense of complicated frequency updating procedures. To update a single transition symbol \(k\), not only does the cumulative range for \(k\) require updating, but all symbols greater than \(k\) also require their cumulative ranges to be updated.
4.4 Locating transition symbols

No matter what data structure is used, some consideration must be given to the method used to locate symbols within a state. The various data structures used in representing a sparse set of alphabet symbols ultimately determine which methods are appropriate. Furthermore, a method that requires few additions or alterations to the data structure is essential. Analysis of search methods involving the number of symbol comparisons is useful in deciding which search method will perform better. The following discussion assumes that a state contains a number of transition symbols. Details on how these symbols are added to a state is left till section 4.8.1.

4.4.1 Sequential search

The simplest approach to searching any list of \( n \) transition symbols involves a sequential search, scanning the list from the top until either the desired symbol is found or the end of the list is reached. The average number of comparisons made during a successful search is \( \frac{1}{2}(n+1) \), while an unsuccessful search makes exactly \( n \) symbol comparisons.

Sequential search has the advantage of not requiring the symbols to be ordered within the list, although maintaining an ordered list of symbols will reduce the average number of comparisons required for an unsuccessful search.

Through each iteration a sequential search requires two comparisons, one compares the transition symbols, the other checks to see if the end of the list has been reached. This can be partly overcome by adding a sentinel symbol to the end of the list, ensuring the symbol will always be found and eliminating the need to perform the second comparison. When the search terminates it will have been successful if the symbol located is not the last item in the list, and unsuccessful if the sentinel item was the one located.

4.4.2 Binary search

Sequential search is efficient for short lists, but a disaster for long ones. One of the best methods for locating symbols within an ordered list is to locate the centre of the list, then restrict further comparisons to just one half of the list, recursively reducing the size of the search list by half for every comparison until a match is found. In only twenty comparisons this method will locate any symbol in a list of about a million
symbols. For an alphabet containing 256 symbols a maximum of eight comparisons would be required compared with 256 for a linear search.

Some restrictions apply to binary search. First, it requires that the items within the list be of a scalar type, and second, the list must already be completely ordered. This ordering property is required because each binary comparison must determine if the symbol occurs before or after the symbol located at the centre of the list. For a list containing $n$ ordered symbols, $\log_2 n + 1$ comparisons are required for both a successful and an unsuccessful search.

### 4.5 Representing state transitions

Several structures capable of representing the FSM required for BDMC were examined in chapter three. The problem with CDMC is slightly different because it is not a single structure being allocated to store the complete FSM, but individual structures to represent each state.

For the following discussion a slot is defined as a space available for storing and retrieving information about a single transition between two states. The advantages and disadvantages of various data structures, namely arrays, linked lists, binary search trees, and cumulative coding trees are examined in the following sections.

#### 4.5.1 Arrays

The benefits of using contiguous arrays for storage of structures was discussed extensively in chapter three. They are simple to implement, allow random element access, and they are fast. Each transition symbol structure must contain a *symbol*, *transition_count*, and *next_state* pointer, the state symbol array consists of $k$ structures in a contiguous block each capable of being accessed individually. The problem is how to manage the allocation of these structures within each state so as not to waste memory.

A simple solution is to allocate each state with a predefined number of transition symbol slots. However this offers poor memory usage when all slots are not required in a state, and some states may exceed the number of symbols initially allocated. Determining the number of symbols to include in each state requires trade-offs between memory wastage and the effect this can have on the average compression.
A more subtle solution determines the number of slots required in each state when it is created. The cloning operation duplicates an existing state, allocating room for a known number of existing slots. Using this information while the model is being developed allows the total possible number of symbols within a state to be estimated. If the cloning operation requires $k$ new slots to be created then creating an additional $n$ slots, during this operation, allows for additional symbols to be added to the state if needed.

Other solutions rely on the popularity of state transitions to determine when a state requires additional symbol slots. When a state has run out of available slots they can be created by reallocating the existing state structure. This enables states to grow in size from an initial number of available slots to arbitrarily close to the required number of slots. This number will not be exactly right because increasing the size of the state by just a single slot will have a big effect on execution speed, while increasing the number of slots by a larger value will waste slots if they are never used. Experimental comparisons would determine the best value for this increase in slot space, balancing trade-offs between execution speed and memory requirements. This method relies heavily on the operating system providing a memory reallocation or recovery system that can reuse released memory efficiently.

The advantage of using arrays containing explicit symbols rather than implicitly indexed arrays ceases once a state contains over 128 transition symbols, but initial experiments indicated that this should occur rarely.

### 4.5.2 Dynamically linked lists

The power of linked lists is that they allow structures to be dynamically allocated as required, and to be referenced by existing structures or variables. Structures are allocated and can be returned for reuse from a global pool of available memory.

The operations on lists required for CDMC include inserting items in the list, removing items from a list, and moving items from one place to another. All these operations can be implemented using either singly or doubly linked lists. These operations facilitate the use of linked lists to store individual transition symbols for each state. Such a structure is shown in figure 4-6, which depicts a single state containing three symbol transitions to other states of the FSM. A state header contains a $\text{transition\_total}$ count along with a $\text{transition\_symbols}$ link to the first
transition symbol. Each transition structure consists of a `next_symbol` link to the following transition symbol together with the required `symbol`, `next_state`, and `transition_count` values. Access to singly-linked lists is sequential requiring the use of a sequential search rather than a binary one for locating transition symbols.

A linked list offers excellent flexibility in allocating slots for transition symbols but introduces several problems while achieving this flexibility. The cloning operation of CDMC involves duplicating all transition symbol slots when a new state is created. Clearly an implementation using linked lists requires a large number of memory allocations to duplicate all the transitions of a state.

Every symbol transition requires an additional link to the following symbol transition. With small symbol transition structures such as required by CDMC the overhead of these links is large compared with the overall size of the structure. Reducing this overhead can be achieved by using a combination of data structures that take advantage of the best properties of both arrays and linked lists.

4.5.3 Linked arrays

The benefits gained in using linked lists of transition symbols within each state are overshadowed by the problems outlined above. Clearly the large number of memory allocations is a hindrance both in terms of compression speed, and the additional complexity required in maintaining the linked lists. A compromise involves creating a structure that offers the benefits of both arrays and linked lists. Each state is a single block of memory containing a header and an array of several transition slots. The state header contains a link to further arrays of transition symbols if additional slots are required within the state. These arrays can be allocated as required and offer the ability to expand the number of transition symbols within any state dynamically.

Figure 4-7 depicts such an arrangement of a state header (SH) containing a number of transition symbol slots (SS). The state header contains `num_symbols` whose value determines the bounds of the symbol slot array, `filled_symbols` which holds the number of currently used slots and the familiar frequency total `transition_total`. The
header also contains an array of length `num_symbols`, each element being a single symbol slot. Once the header's slots are exhausted additional slots can be added by allocating state slot headers (SSH in figure 4-7), linked together by the `next` pointers contained in both the state header and state slot header. Once again each slot header contains a fixed length array of $m$ symbol slots, with a value `filled_symbols` the current number of slots in use.

![State structure containing linked arrays for transition symbols.](image)

Determining how many transition symbols to allocate when a state is cloned becomes a trade-off between execution speed and memory utilisation. When a state containing $k$ symbols is cloned, at least $k$ slots must be allocated in the new state. An addition $n$ slots are also allocated to facilitate a possible increase in the number of state transition symbols. When $n$ is small the slots fill quickly and additional slot headers will be required. When $n$ is large, slots may never be utilised and hence memory will be wasted. Once the initial state header contains no more empty slots $m$ additional slots are allocated in state slot headers and attached using the `next` link. The size of $m$ also has an effect on the execution speed and memory usage, again because of the overheads in allocation of additional slots, and the proportion of space occupied by the header information. Determining the best values for $n$ and $m$ is achieved through experimentation.

This method has two subtle side effects. The first occurs when cloning a state containing a number of linked slot headers. During the cloning operation, CDMC concatenates each of the slot header arrays to form a single state header array containing all of the states symbol slots, offering better utilisation of model memory.
The second occurs when creating an initial state containing a large number of symbol slots. This is done because the initial state is likely to contain a large proportion of the symbols from the alphabet. When this initial state is cloned the new state will contain room for $k + n$ slots, where $k$ is the number of currently occupied slots, rather than the total number of slot allocated in the initial state. This allows the initial state to grow without additional slot headers being allocated, and not forcing new states to contain a large number of symbol slots.

4.5.4 Binary search trees

The previous data structures are all capable of representing lists of transition symbols within a state, but they have one weak feature. They are all sequential lists, requiring a linear search to locate a transition symbol. As noted earlier, a faster method of locating symbols involves a binary search, which compares the symbol in the centre of the list and restricts the remaining comparisons to the first or second half of the list. A suitable structure to facilitate a dynamic binary search of transition symbols is a binary search tree (BST).

While a BST enables efficient searching of ordered lists, it does not easily facilitate calculating the cumulative frequencies for symbols. In other words once a symbol is located all the symbols preceding it still need to be accessed to calculate $C_s(k)$.

4.5.5 Cumulative coding trees

A special implementation of binary trees is capable of calculating cumulative symbol frequencies with only $\log_2 n$ accesses [4, 24]. The cumulative frequency for any symbol is obtained by summing the counts of selected ancestors in the tree. To achieve this factor of $\log_2 n$ accesses, a balanced tree must be maintained otherwise degeneration into a long stranded tree can cause performance no better than a linear list.

Construction of the tree requires that the symbols with a high frequency occur close to the root, as shown in the example in figure 4-8 (a). Each node of the resulting tree (b) contains not only a frequency, but a left_count which is the sum of the frequencies of its left sub-tree. For example the root node has a frequency of 15, and a left_count of 22 ($7 + 3 + 12$). The $C_s(k)$ of a symbol is calculated by traversing the tree from the root to the symbol, accumulating left_counts and symbol frequencies whenever a right
branch is taken. As an example, to find symbol b's cumulative range from the root the first left branch contributes nothing, while the second right branch contributes \( 7 + 12 \). The resulting range is \([19 \ldots 22]\). The remaining symbols \( C_s(k) \)'s shown in (a) were calculated using the same method.

A similar procedure is used to determine a symbol given any target value within its range. Comparing the target with the value accumulated as above ascertains whether to stop at, traverse left, or traverse right from a node.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( n_s(k) )</th>
<th>( C_s(k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>a</td>
<td>9</td>
<td>38</td>
</tr>
<tr>
<td>e</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>37</td>
</tr>
</tbody>
</table>

Figure 4-8  Building a cumulative coding tree from frequency ordered symbols.

Incrementing a symbol's frequency involves not only updating the node's frequency, but traversing up the tree incrementing \( \text{left\_counts} \) for appropriate nodes. To ensure the ordering property of the tree, the target node is moved left and up the tree as far as possible before incrementing it. After updating, the symbol's frequency will at worst be equal the next ordered symbol. Applying frequency count scaling further complicates this method because the reordering must occur after the symbol has been incrementing requiring a complete rebuild of the tree including recalculation of \( \text{left\_counts} \).

While such a data structure seems to offer the benefit of efficient cumulative frequency calculations, it requires additional memory to achieve this. Each node of the binary tree requires links to two child nodes, which is expensive if full memory addresses are used. Binary trees can be stored efficiently within an array [21, for example]. Because the tree must be maintained in frequency order, to enable direct access to nodes an additional index must be kept to convert symbols to tree positions. With a sparse distribution of symbols this also means either a linear or binary search of symbols must be performed. These overheads make this method extremely fast.
but require additional data structures to achieve this speed. Making it an unsuitable structure for representing CDMC symbol transitions, because we are attempting to reduce the amount of memory required by each state.

4.6 The complete state model

The final implementation of CDMC uses master linked arrays of state indexes to access individual states of the FSM. States therefore need only contain a 16-bit index to distinguish any state of the model, where otherwise they would require 32-bit memory addresses. As numerous cloned states contain pointers to the same state a significant amount of memory becomes available for further states.

A state address array contains a variable number of 32-bit addresses for individual states. For example if each array comprises 5,000 state addresses, then it will require 19.54 Kbytes of memory. All address arrays are linked together using a doubly linked list, to facilitate forwards and backwards searching for any state. To discover a state’s address from its index the correct address array is located, and a single access determines the states physical memory address. A 16-bit index is used to reduce the size of each next state pointer required in each symbol slot. The complete FSM is capable of maintaining 65,536 possible states. The operation of determining a state’s physical address is only performed once through each iteration of updateModel\(^1\), which is not a large loss in efficiency. Retaining a pointer to the last accessed address array improves search hits because linked states of the model are more likely to occur close together.

The final state structure uses a state header with additional linked state slot headers. These structures offer excellent flexibility and memory utilisation in comparison to the other methods described. This is important consideration when the number of states contained in the FSM can reach 60,000 for source messages of only 1 Mbyte.

4.6.1 Memory usage

CDMC can operate in as little as 128 Kbytes of model memory, although increasing the available memory improves compression considerably. The amount of memory

\(^1\) Refer to Figure 4-17 for an outline of the CDMC algorithm.
in use by a model again has a fairly linear relationship to size of the source message being compressed, as did BDMC. Figure 4-9 depicts various memory configurations and the resulting number of states contained in the model, showing a clear linear relationship. The average compression dips steeply initially as extra memory is made available to the model, but levels off once a suitable limit is reached. Adjusting the memory parameters $m$ and $n$ changes the amount of wastage and overheads required to maintain the model. Table 4-10 shows various results for several values of $m$ and $n$. The amount of memory wasted by such implementations is shown by Unused, while the memory required to maintain the model is shown by Overhead. The relative compression speed, and an example of the number of states represented by such a models, are also shown. Changing these parameters ($m$ and $n$) has some effect on the overall compression when a constant amount of model memory is used. As a larger percentage of memory is wasted, or used in maintaining the model, fewer states are able to be represented in the FSM. This results in a slight reduction in the compression obtained. These results were gathered for the file book1 of the corpus, allowing 2,048 Kbytes of model memory.

![Figure 4-9](image-url)  
**Figure 4-9**  
CDMC Compression for varying model sizes.
Comparing these results for the same file with an implementation using simple linked lists of transition symbols gives some interesting results. Using the same amount of memory, 2,048 Kbytes, the linked list implementation was only capable of representing 3,079 states because of the increased overhead associated with each symbol slot. The memory required to maintain the linked lists of transition symbols for each states was as high as 33 percent of the total 2,048 Kbytes. The overall decrease in states resulted in approximately two percent poorer compression, and was over three times slower.

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Unused (%)</th>
<th>Overhead (%)</th>
<th>Normalised Throughput</th>
<th>Total States</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.01</td>
<td>5.25</td>
<td>1.00</td>
<td>6073</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0.62</td>
<td>5.12</td>
<td>1.02</td>
<td>6043</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5.29</td>
<td>4.70</td>
<td>0.95</td>
<td>5805</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>5.31</td>
<td>4.70</td>
<td>0.96</td>
<td>5803</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>5.48</td>
<td>4.65</td>
<td>1.15</td>
<td>5797</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>18.11</td>
<td>4.20</td>
<td>1.27</td>
<td>5211</td>
</tr>
</tbody>
</table>

Table 4-10 Adjusting the memory parameters affects memory wastage.

## 4.7 Sparse symbol sets within states

The natural initial model for CDMC consists of a single state containing all 256 characters with circular transitions to the same state. Cloning this initial state, however, will always produce new states containing 256 transitions, clearly not the desired effect. Therefore the initial model for CDMC must contain no initial transitions, adding them to each state as required. This has several important ramifications, including deciding how to code a symbol that has not yet been encountered in any state.

A simple solution to account for the symbols not present in a state is to implicitly store a frequency count of one for each symbol. The initial state wouldn’t contain any symbols, but would have an initial transition_total of 257, frequencies for 256 alphabet symbols and an end of file marker. Such a method produces very poor compression due to the relatively large probabilities given to unseen symbols. Even with large scaling factors applied to encountered symbols, the combined frequencies of all unseen symbols will normally exceed two hundred. The following sections examines methods for coding these unseen symbols, as well as details of the compete coding routines.
4.7.1 Escape symbol probabilities

A better solution relies on the fact that the initial state contains a large proportion of the alphabet symbols because during the process of building the model it is visited a large number of times in comparison to other states. A special transition symbol in each state instructs the decoder to use the initial state, which is more likely to contain the symbol and hence produce better coding frequencies – in fact very similar to those of an adaptive order zero model. Such a transition is known as an escape and is assigned some small frequency count, and therefore has less of an impact on the probability distribution than a large number of individual unseen symbols. When the decoder encounters the escape symbol, it then attempts to decode a symbol from the initial state. Three different methods were considered to determine the probability for the escape symbol.

- A fixed transition frequency \( x \) for the escape symbol yields the probability

\[
P_s(\text{escape}) = \frac{x}{N(S) + x},
\]

where \( x \) is traditionally equal to one, and \( N(s) \) is the total of transition frequencies for all symbols in state \( S \). As symbols are added to a state and \( N(s) \) increases, the escape probability becomes relatively small compared with the symbols in the state.

- Treating the escape symbol as another alphabet symbol and adjusting its transition frequency as it is traversed, this yields the probability

\[
P_s(\text{escape}) = \frac{n_a(\text{escape})}{N(S) + n_a(\text{escape})}.
\]

This method allows the frequency of the escape symbol to increase initially, reflecting the fact that it will be used often, while the state contains few symbols. As symbols are added, and the escape is coded less frequently, its relative transition frequency will be reduced.

- Starting with a large initial transition frequency and decreasing the escape frequency as each new symbol is added to the state. This reflects the fact that as additional symbols are added to a state the probability of another unseen symbol occurring is reduced. The probability calculation for \( P_s(\text{escape}) \) is identical to the previous method. The escapes transition frequency is decre-
mented rather than incremented as each additional symbol is added, checking of course it never reaches zero.

Storing an explicit escape frequency, \textit{esc\_transition}, in the state header facilitates coding of the escape using any one of these three methods as shown in figure 4-11. The second and third methods each involve adjusting the \textit{esc\_transition}, the former each time the escape is coded, the later whenever new symbols are added to the state. The effect of these three different escape methods on the average corpus compression is shown in figure 4-12. The second method, which allows the escape frequency to adapt based on how often it is used, provides the best average compression.

```c
void encodeESC (shPtr sh_state) {
    // transition_total includes the esc_transition frequency
    arithmetic_encode(0, sh_state->esc_transition, sh_state->transition_total);

    switch (esc_method) {
        case ESC_FIXED: break; // esc_transition never changes
        case ESC_DYNAMIC:
            sh_state->esc_transition += scale;  // esc_transition frequency needs to be incremented
            sh_state->trans_total += scale;
            checkOverFlow(sh_state); break;
        case ESC_REVERSE: break; // esc_transition is adjusted when symbols are added
            // see updateModel()
    }
}
```

Figure 4-11 Coding an escape symbol from a state.

![Figure 4-11 Coding an escape symbol from a state.](image)

Figure 4-12 Empirical comparison of the three different escape methods.

### 4.7.2 Probability of unseen symbols

There is still a problem if the symbol to be coded doesn't exist in the initial state. In this case another escape symbol is coded to indicate the symbol has not yet been seen. The modeller then explicitly codes the symbol using a equiprobable model. CDMC is
capable of employing two different methods for determining the probability of unseen symbols.

- Using a fixed probability for every alphabet symbol, the probability of symbol $k$ is
  $$P(k) = \frac{1}{\text{Alphabet Size} + 1}.$$  
- Using a dynamic probability which changes as more symbols are added to the initial state, in this case the probability of symbol $k$ is
  $$P(k) = \frac{1}{\text{Alphabet Size} - U(\text{initial}) + 1},$$  

where $U(S)$ is the number of transition symbols currently contained in state $S$.

Both of these methods implicitly add one to the denominator of their probability calculation to enable the end of file symbol to be coded as a special case of an unseen symbol. In the worst case an unseen symbol will be coded with two consecutive escapes, followed by the unseen probability, requiring just over 36 bits. Experimental results have shown however on average only 28.40 bits are required for each unseen symbol coded.

Normally symbols are only ever added to the initial state when they are encountered while in that state so the same unseen symbol may be coded many times before it is eventually added to the initial state. An improvement to this method could be realised by adding an adaptive zero order model to the initial state, maintaining adaptive frequency counts of the complete source in parallel to the normal method of adding symbols to the initial state. Rather than coding two escapes, a single escape is coded, then the zero order adaptive model is used to generate symbol probabilities. On the surface this method seems to overcome the problem, although in practice it offers compression improvements of less than one percent.

### 4.7.3 Coding symbols

The complete symbol encoding method is outlined in figure 4-13. If the given symbol is contained in $sh_{\text{current}}$ it is coded using its current $transition_{\text{count}}$, otherwise an escape to the initial state is coded. If the initial state contains symbol it is coded using the initial states $transition_{\text{count}}$ for symbol, otherwise an escape to the equiprobable
model is coded². Finally the equiprobable model is used to code the unseen symbol. The complementary decodeSymbol function is simple to deduce by tracing the coding path and implementing the complementary operations.

```c
void encodeSymbol( short symbol, shPtr sh_current, ssPtr *ss_symbol ) {
    short low, before;  // cumulative freq and in-order slot count respectively
    if ( findSymInState( symbol, &ss_symbol, sh_current, &low, &before ) == ESCAPE ) {
        encodeESC( sh_current );  // symbol not found - code escape to initial state
        if ( sh_current == sh_initial )
            encodeESC( sh_initial );  // symbol not found - code escape to -1 order model
        encodeUnseen( symbol, before );
        // symbol unseen - code using -1 prob. distribution
    } else {
        // symbol exists in initial state - code using symbol frequency
        arithmetic_encode( low, low + ss_symbol->transition_count, sh_initial->transition_total );
        // return symbol not found in sh_current
    }
}
```

Figure 4-13 Coding any alphabet symbol from the current state

4.8 Updating the model

The process of updating the model is independent of the coding phase. Once the encoding or decoding has been completed, updating the model is identical in both cases, not only simplifying the algorithm, but ensuring that both the encoder and decoder again have an identical model, prior to coding of the next symbol. The procedure of updating CDMC's model can be separated into two distinct operations, each mutually exclusive. The operation involves either adding a new symbol to a state if it does not exist, or updating an existing symbol and cloning a state if necessary, these operations are discussed in the following sections.

² A single escape is only ever coded for the special case when sh_current happens to be the initial state.
4.8.1 Adding transition symbols

How transition symbols not present in a state are actually added has yet to be elaborated on. Space is made available for the additional symbol in the state either by adding it to an existing symbol slot, or alternatively by allocating an additional state slot header if no spare slots exist. Determining where the symbol's \texttt{next\_state} transition should point is the next critical decision. The logical choice would seem to be the initial state, not only because it contains a large number of symbols, but also because the symbol previously "escaped" to this state when unable to be located. An improvement however, found through experimentation has been to set the \texttt{next\_state} of the new symbol to the \texttt{next\_state} of the identical symbol contained in the initial state, if it exists. Such a modification has improved compression performance on average by just over four percent. The symbol is also given a suitable initial transition count. Figure 4-15 shows the slight effect on the average corpus compression, of varying this initial frequency. A initial transition count of 32 was used in all other experiments.

4.8.2 CDMC state cloning

The threshold conditions used in BDMC to determine when a state becomes eligible for cloned, were extended to support the larger alphabet of CDMC. For BDMC, cloning thresholds of one and eight for parameters \(t_1\) and \(t_2\) respectively were found to produce the best average compression. Experimental results (figure 4-14) for CDMC have shown a larger \(t_2\) threshold, ensuring other symbol transitions are also popular, gives better compression. Values of 2 and 128 for parameters \(t_1\) and \(t_2\) respectively achieve the best average compression.

Once a state becomes eligible for cloning, the operation itself is relatively simple, involving the duplication of all existing symbol slots and reassigning symbol transition frequencies. A new state is created containing enough slots to duplicate all the symbols contained in the target state, plus \(n\) additional slots for future growth. All existing transition symbol information is transferred to the new state, and the transition frequencies distributed, between the cloned and new states, based on the popularity of the symbol transition.
Chapter 4. Character DMC

4.8.3 Transition frequency scaling

The dramatic effects of transition frequency scaling were detailed in section 3.4. This section showed the effects of incrementing transition frequencies by a large integer value, enabling the frequencies to be represented by fixed point integers. Scaling counts in this way gives a greater emphasis to symbols that have been seen often over those that haven’t been encountered. Results shown in figure 4-15 have determined the ideal range for the scaling factor of CDMC is in the range [30…100].
4.8.4 Symbol forgetting

Another subtle variation, which takes into account the fact that some symbols added to a state never improve compression, involves removing infrequently used transitions. When attempting to add additional symbols, rather than create a new slot header, an existing symbol can be removed from the state header and the new one inserted in its place. Two conditions must be met before a symbol can be removed. The first and most important the symbol’s next_state transition must lead to the initial state. Removing a transition to a state does not guarantee that another transition still exists to that state, hence leaving the state within the FSM dangling. The second condition specifies a frequency a transition must reach for removal to occur. This prevents aggressive symbol swapping before a transition has any chance of cloning. This operation only occurs once the appropriate state header has filled all the available symbol slots, and therefore will not occur during the initial phases of building the model as long as \( n \) is significantly large.
Using this method, a reduction in the unused memory has been noted, while overall performance given an inappropriate parameter will degrade compression by as much as thirty percent. Tests have shown that a model with the ability to adapt and grow, by always being able to add additional symbols to a state, will always outperform the restricted model just described.

An outline of the complete `updateModel` procedure is shown in figure 4-16. If the symbol does not exist in the target state it is added, otherwise if the cloning conditions are met a clone of the target is created. Finally the transition frequencies for the symbol and state are incremented. Notice the use of `resolveSAAIndex`, which determines the physical address of a state given an index into the list of state address arrays.

```c
shPtr updateModel( shPtr sh_current, short symbol, ssPtr ss_symbol ) {
    shPtr sh_next, sh_new;
    index new_index;
    double ratio;
    if( ss_symbol == NULL) return addSymToState( sh_current, symbol );
    if ( ss_symbol->transition_count >= t1 && sh_next->transition_total >= t2 + ss_symbol->transition_count ) {
        ss_new = createNewState( queryFilledSlots( sh_next ) + n, &new_index );
        ratio = ss_symbol->transition_count / sh_next->transition_total;
        duplicateSymbols( ratio, ss_next, ss_new );
        ss_symbol->next_state = new_index;
        sh_next = sh_new;
        ss_symbol->transition_count += scale;
        sh_current->transition_total += scale;
        return next_state;
    } else return addSymToState( sh_current, symbol );
}
```

Figure 4-16  CDMC Model Updating.

### 4.9 Algorithm outline

The complete `encodeFile` procedure is shown in figure 4-17, and follows from the modelling-coding paradigm. As each symbol is received from the source it is coded using the current statistics contained in the model, and the model is then updated to reflect the occurrence of the symbol. Finally, the end of file is detected and coded.
Chapter 4. Character DMC

The complementary `decodeFile` procedure is very similar, using the current statistics contained in the model and a target value to determine the coded symbol. Once known, the symbol is used to update the model to reflect its occurrence, and this process is repeated until the coded end of file is received.

```c
void encodeFile ( FILE *source ) {
    shPtr sh_current;
    ssPtr ss_symbol;
    short symbol = getc( source );
    sh_current = sh_initial = initiaiModel( ); // initial state of FSM
    while ( symbol != EOF ) {
        encodeSymbol( symbol, sh_current, &ss_symbol ); // encode symbol using current model
        sh_current = updateModel( symbol, sh_current, ss_symbol ); // update model to reflect symbol
        symbol = getc( source ); // grab the next source symbol
    }
    encodeEOF( sh_current ); // encode the EOF symbol
}
```

Figure 4-17 CDMC encoding algorithm.

### 4.10 Improving search times

Initially symbols were added to a state in the order they appeared in the source, common sense would suggest, and profile analysis of the algorithm confirms, that a large proportion of time is spent searching the symbols of a state attempting to locate a match. As discussed, the nature of arithmetic coding requires that a linear search be used, enabling a symbol’s cumulative range to be calculated. Simple strategies can be employed to facilitate more efficient search times, these are discussed in the following sections.

#### 4.10.1 Symbol ordering

Ordering symbols within a state enables a binary search of state symbols to be made, but implementation is complex because all additional state slot arrays must be taken into account. Once a symbol is located, the cumulative frequency still needs to be calculated, by summing all preceding state symbols. Nevertheless, a binary search will provide an efficient way of determining if a symbol actually exists within a state.

Another efficient but unrelated method of determining whether a symbol is contained in a state is to store a bit array in the state header. When a symbol is added to a state its corresponding bit is set in the bit array. Determining if a symbol exists within a
state is simply a matter of checking the correct bit, then performing a search if the bit is set.

Even without using a binary search or a bit array, by ordering symbols within each state unsuccessful searches on average will terminate before every element of the list is compared. Using a linear search from the start of the symbol list, comparisons against each symbol can determine whether the symbol has been located, the search should continue, or if the symbol doesn't exist in this state.

Unfortunately such schemes have proved to be ineffective because very few unsuccessful searches occur—indeed, less than one percent of all searches are unsuccessful.

### 4.10.2 Move to front

To help speed up the linear search, whenever a symbol is located within a state it is swapped with the symbol located in the first element. If the symbol is popular a subsequent search should then locate it faster. Variations on this method could swap the symbol into some other position, say halfway up the symbol list. This method is practical because only a single swap needs to be performed for each symbol access. However problems can occur if the state contains a small number of very popular symbols and several with low frequencies. Excessive swapping of symbols between the front and back of the list will occur. Results show this method only results in a five percent improvement in speed.

### 4.10.3 Frequency ordering

A better method takes account of individual symbol frequencies. Symbols with a high frequency have, by definition, been accessed more often. Keeping these symbols near the front of the symbol list will improve the average symbol search time because popular symbols will be located quickly, compared with unpopular symbols which take longer to locate.

The method of symbol ordering used by CDMC involves reordering the state’s list of symbols whenever a symbol is coded from it. This is achieved by reversing up the symbol list, from the located symbol, comparing frequency counts. Rather than moving every symbol as the reverse search is done, just the final symbol is swapped with the original. This means the list is not kept in strict order, but will, over time, converge to an ordered list. New symbols added have a low frequency count, and
therefore can simply be added to the tail of the list. This technique improves CDMC compression time by approximately 35% over the unordered list.

4.11 Examining the FSM

We have seen visually how the cloning operation affects locally the part of the FSM it acts on, but we have yet to examine the model itself as it is generated by this process. Originally this was approached both in the hope that it would provide some insight into exactly how the cloning operation affects the model, and to verify the cloning operation was working correctly. CDMC was modified to generate suitable input for DAG [17], a program that creates visual directed acyclic graphs.

Visual output from various stages during model generation is shown in figure 4-18. Note the dramatic number of transitions between states. Dotted lines represent transitions between states for multiple symbols, while unique symbol transitions are represented by solid lines. Variable length contexts are generated rapidly from the initial state to outer states of the FSM. By the time 100 states are contained in the FSM, the model has become a mass of intertwined states and transitions.

An additional illustration of how effective the model is in representing text can be gained by randomly generating text from a stored model. Generation of a large model from the corpus file book1 enables the statistics gathered to be used for generating random text. Moving from state to state, based on transition probabilities, and outputting the appropriate symbol, gives a feel for how successful the model is at modelling English text. An excerpt of text generated from this model follows.

Breat fand. won't sice ouest was inder y>1ttlow what dayKoor flaying 'y cly brn - lookinioor of the oy nual, al Ly y4 button fing m She was the lot halse shny is were vone iout-ling sped the the p+en'orgood-he would leatherburld.uch fland and tely what was call by her suriel took the dmer ended catsheba was say s I had with a gost her. The said, which unce alone naren, and with without gtes of her rrivent to pusinesn't,' said Gabriel. Creamed nh her hair. She and its with him weed -sualf that had movid him, surl Bring :

for the r her band again,piliat ervs the lope appeared

The modeller has been able to pick up quite a few short words, and grammar constructs such as beginning sentences with capital letters, intermingled with a great deal of gibberish. Similar comparisons performed by Bell, Cleary, and Witten [4] confirm that this text resembles that generated by an order two model, suggesting
Figure 4-18 Visual representation of the model created by CDMC.
that CDMC has been able to capture information of this order. However a closer look at the text reveals the fact that some longer contexts are being used to generate words, for example CDMC has generated the words 'said Gabriel.', which appears numerous times in the text, something an order two model would have been unlikely to do. This result confirms that DMC does generate variable length contexts.

The text above can hardly be construed as an excerpt of English text, and despite the sophisticated methods used in text compression today, modelling techniques certainly have room for improvement.

4.12 Using CDMC

The implementation of CDMC developed by the author has several command line switches that control the various techniques discussed in the chapter. Appendix A includes full details of all command options. Without any switches CDMC displays a summary of all available options as follows.

usage cdmc [-e -d]

where:
- `e` encode stdin to stdout
- `d` decode stdin to stdout (other switches ignored)
- `c` initial transition counts are nnn [32]
- `f` output frequencies of state character transitions [Off]
- `g` create DAG input file after nnnn states [Off]
- `h` remove slots only after nnn transitions [2]
- `i` expand states by nnn slots when cloning [5]
- `j` additional state slot arrays contain nnn slots [10]
- `k` instantaneous compression every nnn characters
- `m` restrict to nnnn kbytes of memory [2048k]
- `n` malloc memory in nnnn byte chunks [1024]
- `p` probabilities for zero-order using method n [1]
- `q` probabilities for escape using method n [2]
- `r` initial escape transition counts are nnn [9224]
- `s` scale escape transitions counts by nnn [32]
- `g` scale transition counts by nnn [64]
- `t` set threshold #1 to nnnn [2]
- `T` set threshold #2 to nnnn [128]
- `x` exit when memory limit reached [Off]

4.13 CDMC evaluation

A summary of the ideal parameterisations of CDMC is shown below. Each of these various options has been implemented and experimental results verify these choices.

- Values of 2 and 128, for the two cloning variables $t_1$ and $t_2$ respectively (4.8.2).
- A transition frequency scaling factor of 64 (4.8.3).
- Initial frequency counts for symbols added to a state are 32 (4.8.1).
- Calculation of escape probabilities using method two (4.7.1).

Figure 4-19 shows the compression performance of CDMC compared with three other algorithms which are discussed in section 3.9. For large files containing English text, CDMC performs better than GZIP, but fails to do as well as BDMC or PPMC. GZIP does very well in the files containing a lot of repetition, namely progc, progl, progp, and trans. This can be attributed to the fact that dictionary methods are capable of replacing complete phrases or common words in the text with small coded sequences. CDMC performs worse than BDMC on these files even though it is modelling characters. It also performs poorly on the object files obj1 and obj2.

![Figure 4-19 Overall performance of CDMC.](image)

While CDMC's performance in terms of achievable compression is a disappointment, one goal of implementing it was in hope of improving compression throughput. Figure 4-20 demonstrates the dramatic differences in throughput of modern compression methods. Statistical methods are overshadowed by their dictionary...
counterparts, which display remarkable speed at compressing all types of sources. The promising result of this graph is the compression throughput achieved by CDMC compared with PPMC. While the latter has been fine-tuned to deliver maximum performance, the former is ripe for optimisation. With efficient data structures in place, increasing the efficiency of the code could well improve on this figure again.

![Figure 4-20 Normalised throughput of CDMC.](image)

Examining the instantaneous compression of BDMC, CDMC, and PPMC sheds some insight as to where CDMC performs poorly, in terms of achievable compression. In figure 4-21, both BDMC and CDMC take longer initially before they produce good results. Even though PPMC rebuilds its model during compression, it very rapidly overtakes both other methods before it needs to rebuild its model again. The gap between CDMC and BDMC appears to be constant, but both will converge with PPMC as the length of the source grows. This is most probably because of DMC’s ability to represent arbitrarily long contexts, while PPMC is restricted to a maximum context length.

An explanation as to why PPMC outperforms CDMC, which theoretically similar, may lie in the way CDMC increases its contexts. While PPMC adds to its set of contexts after every symbol is coded, CDMC must see a particular symbol a number of times before cloning increases the context length. This is supported by the result that the best compression is achieved when the cloning thresholds are small, increasing the amount of cloning occurring during generation of the model.
Chapter 4. Character DMC

4.14 Summary

Evolving the original ideas of Cormack and Horspool to use larger symbol alphabets has been the focus of this chapter. Implementing an efficient data structure for representing states has been a major goal. Comparisons of techniques and parameters affecting the average compression have also been discussed. While CDMC has proved to give good compression for all types of sources, it does not perform as well as its cousin BDMC. In terms of compression speed, the promise of improved performance with the translation to a larger source alphabet has become a reality. CDMC has a similar throughput as PPMC, and this could easily be improved upon by optimisation of the code.

Dictionary methods have had great success replacing complete phrases with short code sequences. A further avenue of research for CDMC is the addition of modelling words in the FSM. Chapter five details the results obtained by extending CDMC to include edge transitions containing complete phrases.
Chapter 5

Word DMC

Chapter four presented an implementation of character DMC, which utilises a larger source alphabet to improve compression speed. Comparatively though this method still fell short of the more attractive dictionary method GZIP both in terms of average compression and execution speed. Dictionary modellers are typically capable of compressing multiple characters, or phrases, into a single code sequence. Compression speed is dramatically affected by the type of method used to locate previous phrases, and decompression speed is extremely fast because only lookups into a phrase dictionary are required.

Attempting to take advantage of coding multiple characters simultaneously, as dictionary coders can, is the focus of this chapter. In adapting CDMC to model words, an improvement in compression should be realised. This new method is known as Word DMC (WDMC). Because DMC is a statistical based compressor, novel events must be capable of being coded. Simply adapting CDMC to model only words would not be feasible, because the source alphabet would consist of every known word, and some unknown ones! Additionally with such a scheme, each state could contain many thousand of outward transitions, but in practice only a subset of the word alphabet would be present in each state.

Starting from some initial model, WDMC must be capable of coding novel events in each state. To achieve this WDMC must rely on a lower order model to code novel words, and using a character modeller seems the natural choice. WDMC in fact has been adapted from CDMC.
5.1 How WDMC works

Evolving CDMC to contain word transitions as well as character transitions involves adapting the coding and decoding routines to recognise when to code a word. The algorithm works by replacing a series of character transitions with a single word transition to the identical finishing state. So instead of coding a series of consecutive characters, only a single code is required. Figure 5-1 shows this procedure, the transition for the word the replacing the sequence of transitions for t, h, and e.

![Figure 5-1 Replacing multiple character transitions with a word transition.](image)

Execution of WDMC involves splitting the source message into words or phrases. That is a simple operation that involves scanning the source message for symbols signifying the end of a word. WDMC considers an end of word symbol to be anything not contained in the set \{a...z,A...Z\}, or if no suitable marker is encountered a maximum length of ten characters is enforced. Once the next word has been determined it is located in the current state. If it located then its transition frequency is used to produce a code sequence. If the word does not exist in the state, a reference to the state is stored, then each individual character of the word is coded using the CDMC frequency statistics. Finally the word is added to the saved state with a transition to the final state.

Word transitions are considered to have exactly the same characteristics as their character counterparts. Some form of word order within a state must be maintained to facilitate cumulative frequency calculations. The probability distribution of symbols and words within a state is now broken in three distinct parts,

\[
\begin{align*}
[0...n_{5}(esc)] & \quad \text{the escape probability,} \\
[n_{5}(esc)...n_{5}(k)) & \quad k \text{ character probabilities, and} \\
[n_{5}(k)...n_{5}(w)) & \quad w \text{ word probabilities.}
\end{align*}
\]
Separate transition totals within the state header enable the word transitions to be searched, and cumulative frequencies calculated, without the need to first sum all the character frequencies. Similarly, the decoder using the same information can easily determine whether the next coded sequence is an escape, a character, or a word. By comparing the decoded target value $t$ with the three known ranges, this method of decoding is actually faster than the encoding process.

### 5.1.1 Representing words

Changes to the state model include the use of a `word_total` within the state header, which stores the total frequency of all outward word transitions, and `words` which contains a link to a word slot header. A word slot header is similar to a symbol slot header (section 4.5.3), containing a number of word transition slots. Words are stored in each state together with their frequency, which are again used to determine their appropriate probabilities, and a `next_state` index.

### 5.2 Updating the model

The process of updating the model when a word is found simply involves updating the word's transition frequency, and checking for transition frequency overflows. The cloning conditions have not been altered and are therefore still only triggered when character transitions are taken. A new word is added to a state once all the individual characters of the word have been coded, plus the symbol signifying the end of word. This ensures that the decoder builds up an identical word incrementally, and when the end of word marker is decoded, the word can be added in an identical manner to the encoder. New words are stored in a state at the end of an existing word slot header, or if required in one newly created. They are given an initial frequency and point to the state found by traversing the individual characters of the word.

The cloning operation differs slightly in that it must also duplicate a number of word transitions as well as character transitions. Each word slot contains a pointer to the recorded word, and rather than duplicate the complete word just the pointer is duplicated. Transition frequencies are redistributed in exactly the same way as in CDMC.
5.2.1 Memory usage

A dramatic decrease of around 70% in the number of states represented in the FSM was noted when using word transitions. Unfortunately this also results in an overall decrease in the compression of WDMC compared with CDMC. Increasing the memory allows WDMC to represent more states within the FSM, but improvements in compression have not been noted.

One method to reduce the large memory requirements of the new states, a result of cloning states containing a large number of word transitions, is to restrict the number of word transitions that are duplicated. This technique should also improve throughput performance because states no longer contain such a large number of words, and word searches will be faster. Relative compression and throughput of example restrictions on word transition duplication is shown in table 5-2. The throughput represents how fast the algorithm can compress, and \textit{scale} refers to the scaling factor described in section 4.8.3. By restricting the number of word transitions that are cloned, not only does it achieve slightly better compression, but the compression throughput is more than twice as fast.

<table>
<thead>
<tr>
<th>Description</th>
<th>Normalised Compression</th>
<th>Normalised Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>No words duplicated</td>
<td>0.96</td>
<td>0.44</td>
</tr>
<tr>
<td>Word frequency &gt; \textit{scale}</td>
<td>0.94</td>
<td>0.48</td>
</tr>
<tr>
<td>Word frequency &gt; \textit{scale} x 2</td>
<td>0.94</td>
<td>0.44</td>
</tr>
<tr>
<td>Word frequency &gt; \textit{scale} x 3</td>
<td>0.95</td>
<td>0.46</td>
</tr>
<tr>
<td>All words duplicated</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 5-2 Restrictions to word transition cloning.

5.3 Improving the word search

Similar techniques adopted for the characters contained in a state can be used to also increase the speed of each word search. Maintaining the list of words in decreasing frequency order will improve successful search times. Unfortunately, word searches of states are more likely to be unsuccessful – experimental results show approximately 60% of searches are unsuccessful, compared with character searches where less than one percent were unsuccessful.
Applying a hashing function to each word and using this value to perform word comparisons would increase the search speed because only a single integer comparison would need to be performed. This method not only requires additional space to store each word's hash value, but some form of collision detection would need to be employed. Once a hash match is found, further checking against the original word would confirm the correct match, an unsuccessful match implying that further checking is required. Not only do the original words need to be stored, but also their hashed value. A suitable hashing function can be found in McKenzie, Harries, and Bell [23].

5.4 WDMC evaluation

The compression performance of WDMC in comparison to the other methods developed in this thesis, is shown in figure 5-3. While it is disappointing that the average compression of CDMC has not been improved upon, the results show that WDMC has improved compression of the source code files. Throughput of WDMC, however, has been halved overall, because of the large number of searches required in locating word transitions within each state. Improving on this throughput requires some faster method of determining if a word is located within a state. Figure 5-4 shows a portion of the FSM generated by WDMC, depicting several word transitions between states.

Despite the best intentions, improvements made to WDMC to capture word frequencies have failed to improve on the average compression of CDMC. It still performs worst than BDMC, and compression speed has also degraded significantly from CDMC. Additional memory is required to store word transitions within each state, but this can be significantly reduced by restricting the words that are duplicated during the cloning operation. Overall about 30% less states are contained in the FSM when using word transitions, but compression remains practically identical. Future research into WDMC needs to concentrate on applying methods to facilitate faster searching of transition words within a state, to improve throughput. Applying the cloning technique to word transitions may also have some effect on the compression obtainable, and remains unresearched.
Chapter 5. Word DMC

Figure 5-3 Overall performance of WDMC.

Figure 5-4 Example of word transitions generated by WDMC.
Chapter 6

Minusculc DMC

For several reasons DMC typically requires large amounts of computing resources to achieve its significant compression, both in terms of model memory and processing time. Furthermore, there are several situations in computing where relatively small, fixed length packets or blocks are transmitted between entities, and would benefit from some form of compression. This includes compressing packets on a network \cite{27} and blocks on a hard disk. Such situations require that the compression use minimal computing resources, appearing transparent to the end user. Conventional DMC handles such circumstances rather badly. The performance suffers when small amounts of memory are used to represent the model, and because of the dependency on arithmetic coding, it is computationally expensive.

This chapter discusses an implementation of DMC developed to examine the tradeoffs made when very small amounts of memory are used by the model, and a much simpler coding method is used instead of arithmetic coding.

6.1 Resource requirements

Compression of short fixed length messages, typically in the two to sixteen Kbyte range, requires a method with low resource requirements so that compression results in unnoticeable throughput performance degradation. For example compressing and transferring data packets across a network may take less time than transmission of the uncompressed packets. In such situations, data compression is highly desirable, as long as it is fast enough.
The contraints on such a method can be summarized as follows

- Input and output buffers each consist of \( n \) bytes.
- Memory for the model is restricted to \( m \) bytes.
- Fast compression is required, prohibiting the use of arithmetic coding.

### 6.2 Algorithm specifications

While Minuscule DMC (MDMC) operates at the character level, as does CDMC, this is where the similarities end. The limited amount of model memory makes the excessive cloning of states inappropriate in MDMC's case. The approach taken by the author consists of creating a simple first order statistical model of the source, then adaptively increasing the context lengths by applying the cloning operation.

#### 6.2.1 Zero order model

Three types of modellers exist that are capable of capturing zero order contexts, namely static, semistatic, and adaptive modelling. A static model is inappropriate in MDMC's case because little is known about the input data. While a semistatic model decreases the complexity required and provides a way of transmitting unseen symbols. It does however necessitate that the model be added to the beginning of the output buffer. Experimental results have shown that eight bit counts are capable of keeping accurate transition counts for the zero order model, therefore only 256 bytes are required to represent the semistatic model. The benefit of using a semistatic model is that the statistics gathered accurately represent the input source and allow an efficient output code to be created based on symbol frequencies contained in the input buffer. An adaptive model adds no explicit additional bits to the output buffer, but requires additional complexity to regularly redistribute coding sequences based on changing symbol popularities.

A semistatic zero order model can either be based on the complete input buffer, or a subset of it. Preprocessing the complete source ensures that accurate statistics are gathered, with little overhead for small input buffers. Scanning only a subset implies the need to adjust frequencies for unseen symbols, they are assigned a fixed frequency count.
Once the relative frequencies of all source symbols has been obtained they are ordered to produce efficient coding tables, without resorting to arithmetic coding. Adaptive modelling requires this step to be performed regularly to ensure the code used matches symbol frequencies. To provide efficient coding various schemes can be implemented, including Huffman coding trees [20], variable length integer codings such as Elias's γ code [12], or even simple two-level static codes\textsuperscript{1}. In the experiments described here the entropy of the model is calculated and used to estimate the achievable compression of MDMC, although in practice it would probably not be realized if one of these simple coding methods is used.

### 6.2.2 Increasing the contexts

A zero order context model will on average compress a source to approximately 55% of its original size. Increasing the lengths of the contexts will significantly improve this. DMC creates contexts of variable length by cloning popular transitions to create additional states, reflecting the characteristics of the input buffer.

Traditionally DMC clones states from a given initial FSM. In DDMC's case, no initial FSM exists. It would also be highly desirable to include the statistics gathered for the zero order model and include them some way in the FSM. This is achieved by attaching states to individual symbols of the zero order array based on their popularity. When a symbol is coded from the zero order array a new state is attached if the symbol has proved to be significantly popular. This new state can ultimately be cloned again and again using the familiar cloning operation.

With the limited amount of memory it is very desirable for cloned states to be as small as possible. This is achieved by restricting the number of symbol transitions that can exist within a state, denoted by $\ell$, to

$$\ell = 2^n - 1,$$

where $n$ is an integer and $n > 0$.

\textsuperscript{1} A simple coding scheme in which the most common characters are coded using say 4 bits, while the remaining are coded using 12 bits. On average a small saving is made [37].
6.2.3 Coding symbols

Before the FSM becomes large most coding is performed using the zero order model. Coding of symbols from this zero order array simply uses the symbol's frequency to determine its probability

\[ P_{\text{root}}(k) = \frac{n_{\text{root}}(k)}{N(\text{root}).} \]

This formula calculates the probability regardless of whether an adaptive or semi-adaptive method is used to calculate symbol frequencies.

Coding characters from a state is made simpler by the restriction previously imposed on \( \ell \). This enables short binary sequences to be used to represent individual symbols. To be decodable, the length of the sequence must be known beforehand. \( \ell + 1 \) symbols can be coded using \( \log_\ell (\ell + 1) \) bits. An additional escape code is required to inform the decoder that the symbol doesn't exist in the state and to use the root state probabilities. A symbol can be coded with probability

\[ P_s(k) = \frac{1}{\ell + 1}, \]

so each symbol is coded using an identical number of bits. Improving this using shorter length codes for popular symbols requires additional computation to maintain the symbols in descending order, with little benefit when \( \ell \) is small.

6.3 Model structure

The model can be separated into two distinct parts, the zero order array and the FSM of cloned MDMC states. Both are implemented using implicit arrays entailing the use of array indexes rather than memory addresses. The root array consumes

\[ \text{root\_size} = \text{Alphabet\_Size} \times (\text{sizeof(transition\_count)} + \text{sizeof(next\_state)}) \]

bytes, and the FSM uses the remaining \( \text{fsm\_size} = m - \text{root\_size} \) bytes. The FSM is divided into an array of fixed length state structures, each state contains \( \ell \) symbol transitions consisting of

\[ \text{state\_symbol\_size} = \text{sizeof(symbol)} + \text{sizeof(transition\_count)} + \text{sizeof(next\_state)} \]

bytes, so each state structure requires
state_size = (ℓ × state_symbol_size) + sizeof(others_count) bytes. Figure 6-1 (a) depicts a representation of the complete model consisting of the zero order array, and the FSM array of \( i \) states where,

\[
i = \floor*{\frac{fsm\_size}{state\_size}} - 1.
\]

Ultimately \( i \) determines sizeof(next_state) because \( \lceil \log_2 i \rceil \) bits are required to uniquely index the \( i \) states.

![Diagram](image)

Figure 6-1 (a) MDMC model consisting of a zero order array and FSM array. (b) State structure containing \( \ell \) transition symbols.

### 6.3.1 Some examples

Table 6-2 displays some examples of the number of states possible for various values of the model parameters \( m \) (maximum model memory) and \( \ell \) (symbols transitions within a state). Experimental results have shown model sizes approximately half the size of the input buffer utilize memory most efficiently.

<table>
<thead>
<tr>
<th>( m )</th>
<th>( \ell )</th>
<th>( i )</th>
<th>( \lceil \log_2 i \rceil )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>3</td>
<td>398</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>199</td>
<td>8</td>
</tr>
<tr>
<td>8192</td>
<td>3</td>
<td>853</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>384</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6-2 Examples of model suitable configurations.
6.4 Model generation

States are added to the root array by determining if a symbol is popular enough to justify having additional contexts stored. Parameter $t_3$ is the threshold value above which states are attached to the appropriate zero order symbol. This threshold prevents infrequent symbols taking valuable state resources. Each zero order symbol then builds its own independant variable order context tree.

Additional states are added by the familiar cloning operation, which copies all existing state symbols, and redistributes transition frequencies. Symbols are added to a state if an empty slot is available. They are given an initial transition frequency, and their next_state transition becomes the zero order array.

6.4.1 Symbol forgetting

To prevent the model from becoming stagnant and indifferent to changes in the source a further refinement to manipulate state symbols is added. Whenever a symbol is encountered which cannot be added to the current state, an others_count value is incremented. Once this counter becomes significantly large, at least 25% symbols with low transition frequencies, that haven’t cloned further states are removed. This provides recent symbols with a chance to join the state and increase the context if their transition proves popular. Parameter $t_4$, the threshold for the others_count, determines when these symbols should be removed from a state.

6.5 MDMC performance

To evaluate the performance of MDMC, the files of the corpus were split into individual 8,192 byte files. The average compression of each corpus file can then be calculated by averaging the obtainable compression of each eight Kbyte file. Comparing MDMC’s compression performance determines the most efficient values for parameters $t_1$...$t_4$. Results have shown no one set of parameters suits all variations of $m$ and $\ell$, so models must be fined tuned individually for best results. The results shown in figure 6-3 were obtained with model parameters,

\[ m = 4,096; \ell = 3; t_1 = 2; t_2 = 1; t_3 = 7; t_4 = 2. \]
They compare the MDMC results against a simple zero order model of the source, and compression obtained using BDMC. Better compression is gained by MDMC over the zero order model because of the longer contexts being generated. BDMC, as expected, performs much better than both other methods. However it took BDMC almost eight times as long, and required as much as 120 Kbytes of memory to represent the model. The unexpected outcome is parameter $t_4 = 2$, the threshold for adding contexts to the zero order model. This is explained by examining the symbol frequency distributions contained in typical sources. Figure 6-4 shows symbol distributions for two types of sources, text, and object code. For a text source few symbols display large frequencies, while the binary source has several very common symbols, with the remaining spread over a large distribution of low frequencies. The low threshold of $t_4$ enables longer contexts to be generated for all prevalent symbols.
Chapter 6. Driver Level DMC

6.6 Summary

An efficient adaptation of DMC has been developed for use in situations where the use of computing resources must be minimal. Applications might include disk driver compression, peer-to-peer network compression at the packet level, or high-speed hardware compression for use in modems or facsimilies. Typically these applications are restricted both in terms of available memory, and processing time. This new method uses DMC’s traditional state cloning mechanism to create an adaptive variable order context model for any source, while using very small amounts of memory. Compression has proved to be better than the simple zero order model, but not as good as that achieved by BDMC. While the speed of the algorithm has shown considerable promise.
Chapter 7

Conclusions

This thesis has surveyed previous research involving Dynamic Markov Compression (DMC) and explored previously unresearched applications of the original concept. An overview of statistical data compression has been presented, and followed by a summary of the original DMC algorithm designed by Cormack and Horspool. Binary DMC has proven to produce excellent compression, comparable with the best known algorithms today. One of its weakest features however is its large memory requirements because it rapidly generates large finite state models as the source message is coded. Evolving DMC to use larger source alphabets promised improved compression speed, and lower memory requirements. Development and analysis of such methods forms the major part of this thesis.

The first such method, Character DMC, transformed the tradition bit modeller DMC into a byte alphabet modeller. Implementation issues pertinent to reducing Binary DMC's considerable memory resource requirements were of high priority, as well as improving compression speed. The performance of Character DMC in terms of compression was disappointing because it failed to capture as much information as its binary counterpart. One promising result was the increased throughput achievable by this implementation, which could no doubt be improved on with additional time and care.

The additions made to produce Word DMC, attempting to capture phrase information within source messages, has also produced disappointing results. The average compression has not increased, and the time taken to compress files has
doubled over the Character DMC implementation. Research in the future of word based DMC would need to overcome the inherent problems of searching a number of words within a state, and look at altering the cloning conditions to include word transition information.

Typically DMC has required both large amounts of memory, and processing power to achieve good compression but with low throughput. Minuscule DMC overcame these problems with several restrictions to the generation of its state model, increasing the throughput to an acceptable speed, while still providing reasonable compression.

Conceptually DMC consists of an elementary algorithm, which in practice is simple to implement. It achieves good compression comparable with some of the best known methods in use today. To achieve this large amounts of memory are required, and the overall compression speed is comparably slow. But in practice is has proved difficult to increase the throughput of DMC without compromising some of its compression.
Appendix A

Algorithm Command Options

All the algorithms developed in this thesis accept a stream of bytes from stdin and send their output to stdout. This appendix contains complete descriptions of all switches available in the command line for the four algorithms implemented.

A.1 BDMC

-e encode the standard input stream, the coded message is sent to the standard output stream, errors are sent to standard error.

-d decoded the standard input stream, the original message is sent to the standard output stream, errors are sent to standard error. Any additional switches are ignored because the header of the coded sequence contains the decoding switches.

-c creates the file BDMC_ic and outputs the instantaneous compression of BDMC over each specified source interval. The instantaneous compression is a measure of the average compression for the interval \((j, k]\), where \(j\) was the last coded character for the previous interval and \(k\) was the last coded character. eg. -c5000

-C creates the file BDMC_cc and outputs the cumulative compression of BDMC over a specified source interval. The cumulative compression is a measure of the average compression of the message for the interval \([0, k]\) source characters, where \(k\) was the last coded character. eg. -C5000
Appendix A. Algorithm Command Options

-g creates the file BDMC_dag and outputs a representation of the model, when the specified number of states has been cloned, suitable for input into DAG [17]. eg. -g1000 outputs a DAG representation after 1000 states, -g outputs a DAG representation once coding is complete.

-i specifies an initial model for BDMC to create before modelling begins, this initial model is also used for memory regeneration. A depth can be specified for several of the models. eg. -iS creates a model with a single state, -iT8 creates a binary tree of depth 8.

-m restricts the amount of memory to be used for BDMC’s model to the specified number of bytes. This value is rounded to the nearest 64 Kbyte. eg. -m2048

-M restricts the number of states for BDMC’s model to the specified number (in thousands). eg. -M10 allows only 10,000 states in the model.

-n specifies the number of source characters to be modelled before regeneration occurs. This amount includes any characters modelled during regeneration, see also -r. eg. -n1000 causes the model to be regenerated every 1,000 source characters.

-r specifies the memory recovery method to use, one of the following
  -r0 stop cloning and model adaptations when limit reached
  -r1 stop cloning but continue transition frequency updating
  -r2 model regeneration using only the initial model
  -r3 model regeneration using a cyclic history buffer, buffer allocated in models heap, see also -R.
  -r4 model regeneration using a cyclic history buffer, buffer allocated separately, see also -R.

-R specifies the number of characters to store in the cyclic buffer when using model regeneration, see also -r. eg. -R30000

-s specifies the scaling factor to use when updating transition frequencies. eg. -s64 scales frequencies by 64.

-t specifies the cloning threshold $t_1$ for state cloning, eg. -t1

-T specifies the cloning threshold $t_2$ for state cloning, eg. -T8
Appendix A. Algorithm Command Options

-x forces compression to halt if the model exceeds the available memory.
-v specifies verbose output, which simply display the compression achieved in bits/character.

A.2 CDMC

-e refer to BDMC.
-d refer to BDMC.
-c refer to BDMC.
-f generates a concise report containing population statistics for all states contained in the model, including a frequency distribution for the number of symbols in each state.
-g refer to BDMC.
-h specifies the threshold value for which state symbol slots are removed from the state. This switch is ignored if additional state slot arrays are able to be created, see also -j. eg. -h2
-i specifies the number of symbol slots to add to a state when cloned. eg. -i5
-j specifies the size of any additional state slot arrays required in each state, after the all state slots are full. eg. -j10 adds ten additional slots at a time, -j0 enables removal of slots from the state.
-k specifies the initial transition frequency for symbols added to a state. eg. -k32
-m refer to BDMC.
-n enables CDMC own internal memory allocating routines which allocate large chunks of memory, then split them up as required. This method is slightly faster than using the traditional system calls. eg. -n1024
-p specifies the method to use for generating zero order probabilities
   -p1 simple 1/ALPHABET_SIZE probability
   -p2 similar to -p1 but excludes symbols previously encountered
Appendix A. Algorithm Command Options

-q specifies the method to use for escape probabilities
-\( q_1 \) escape probability is equal to one divided by the number of symbols within the state.
-\( q_2 \) escape probability is calculated using an escape frequency and is updated, similar to a symbol, whenever the escape transition is taken.
-\( q_3 \) escape frequency starts at some initial value and is decremented whenever the escape transitions is taken, see also \(-r\) and \(-R\).

-r specifies the initial escape frequency for method \(-q_3\), see also \(-R\). e.g. \(-r\ 1000\)

-R specifies the scaling factor to use when decrementing escape frequencies for method \(-q_3\), see also \(-r\). e.g. \(-R\ 32\)

-s refer to BDMC.
-t refer to BDMC.
-T refer to BDMC
-x refer to BDMC.
-v refer to BDMC.

A.3 WDMC

WDMC adds only a single command switch to the CMDC command list.

-w enables use of word transitions between states
A.4 MDMC

MDMC accepts a list of command line values, rather than switches, and outputs the expected compression for each file contained in its file list. Each value is defined below in the order required, and no value may be left out.

- **semi** specifies whether to use a semistatic model to generate order zero probabilities. (0 - use adaptive, 1 - use semistatic).
- **verbose** specifies verbose display (0 - normal, 1 - verbose).
- **k** specifies the number of slots in each state.
- **model** specifies the amount of memory to use for the complete model.
- **size** specifies the size (in bytes) of each state structure.
- **clone1** specifies the cloning threshold $t_1$.
- **clone2** specifies the cloning threshold $t_2$.
- **others** specifies the threshold for removing symbols from a state.
- **2nd_order** specifies the threshold for adding variable order contexts to the zero order model.
- **[filename]** a number of files to compress, if none specified then use stdin
References


7 R. Cameron, Source encoding using syntactic information source models. LCCR Technical Report 86-7, Simon Fraser University, Canada (1986).


References


37 E. Young, MacWrite file formats. *Wheels for the Mind*, 34(1985)


39 T.L. Yu, Personal Communication, Chinese University of Hong Kong (August 1993).

