Finite State Machine Binary Entropy Coding

Michael J. Gormish and James D. Allen
November 12, 1992
CRC-TR-9238

This work was presented at the poster session of the 1993 Data Compression Conference.
An abstract appears in the proceedings:

Michael J. Gormish and James D. Allen, "Finite State Machine Binary Entropy Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, Mar. 30-Apr. 1, 1993, 449.

Keywords:

Entropy Coding, Data Compression, Image Compression

RICOH California Research Center
2882 Sand Hill Road, Suite 115
Menlo Park, CA 94025
(415) 496-5700, (415) 854-8740 fax
Internet: cip@crc.ricoh.com, gormish@crc.ricoh.com

Finite State Machine Binary Entropy Coding

Michael J. Gormish and James D. Allen

RICOH California Research Center
2882 Sand Hill Road, Suite 115
Menlo Park, CA 94025

Abstract

A method of universal binary entropy coding using simple state machines is presented. We provide a natural rationale for the use of state machines. We discuss the building of state machines which produce uniquely decodable bit streams. Some state machines implement known codes, while others implement finite or even infinite variable length to variable length codes. Once a decodable state machine is found, it is adjusted for maximum compression. Compression results are given for both single context and multiple context coding. All state machines presented have a bounded required lookahead to decode each binary decision. In addition, the machines presented can easily provide joint source channel coding for constrained (e.g., run length limited) channels.

1.0 Introduction

Many current data compression techniques have a separation between modeling and coding. For example both JPEG and JBIG separate image modeling from entropy coding and can use the Q-coder for the entropy coding. Bell, Clearly, and Witten [2] state that this separation is valuable in the compression of text from both pragmatic and conceptual points of view. We will assume the structure of Figure 1, which separates modeling, probability estimation, and coding, without specifying any specific application. We separate modeling from estimation because sometimes the estimation occurs in the model and sometimes in the coder. In addition, it is sometimes possible to separate estimation techniques, particularly solutions to the zero frequency problem, from the model [2].

In this paper we deal with only binary decisions. Clearly, any model which obtains probabilities on some alphabet can break these into binary decisions with associated binary probabilities. The finite state machines we examine perform only the task of bit stream generation. The corresponding bit stream decoder is given the same probability estimates and the bit stream, and produces the sequence of binary outcomes.

The binary coder stage of Figure 1 must have some memory since it is dealing with a binary input and a binary channel. Without memory the only possible operations would be

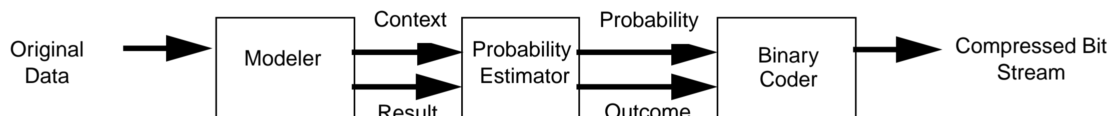

Figure 1 - Binary Encoding System

CRC-TR-9238, Abstract in *Proc. Data Compression Conference*, p. 449, March 1993

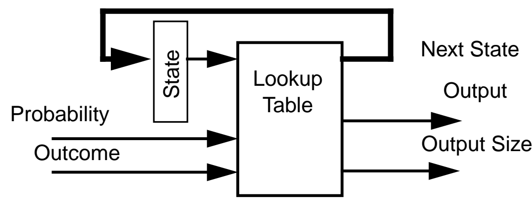

Figure 2 - Binary Coder

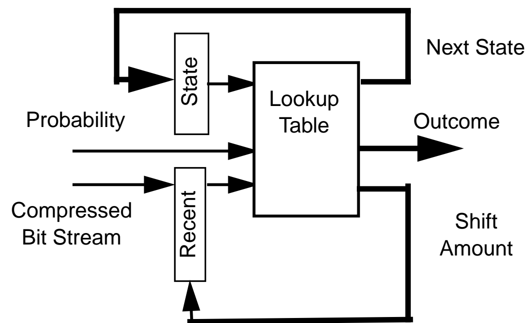

Figure 3 - Binary Decoder

to pass the data uncoded, invert, or expand the data. Figure 2 presents a very general internal description of a possible coder. All the memory has been grouped together and labeled “state.” The rest of the coder is labeled as a lookup table, although it could be combinational logic. Our conceptual point of view prohibits memory in this section since all memory is labeled as state. The Q-coder follows this structure if we consider both registers to be part of the state. Our goal will be to produce encoders with few bits of state.

Note that a block Huffman code for binary inputs could easily be implemented with Figure 2. The state register might keep track of the input symbols until all symbols in a block have been seen; then the machine could output a codeword. We desire a more flexible coder than this because the Huffman code fixes the probabilities, while Figure 1 suggests that the probabilities may change with every symbol. However, below we shall build a more interesting code starting from a Huffman code.

As previously noted, the Q-coder can be implemented as shown in Figure 2. The state register must contain both the base or ‘C’ register and the interval or ‘A’ register. For typical implementations this will be roughly 32 bits. With the addition of bits for the probability estimate and the outcome, the size of the LUT is much too large and combinational logic is necessary. The structure in Figure 2 does not solve the carry over problem, but the state register could be made even larger to handle the bit stuffing methods of [5] or the follow bit counter of [7]. The Q-coder does meet our goal of providing compression of binary decisions with changing probabilities, but we introduced Figure 2 in hopes of using a much smaller state register and LUT.

There are two constraints on the state machine in Figure 2. First, given the bit stream and the same probability estimates presented to the encoder, there must exist a decoder which can recreate the sequence of outcomes. Ideally, the decoder should have a structure similar to the encoder. Second, the length of the compressed bit-stream should be close to the entropy limit given by H(p) when the outcomes presented occur with frequency close to the estimated probability.

In section 2 we discuss one class of state machines which obviously produces uniquely decodable bit streams which can be decoded with a machine similar to the encoder. This class of machines yields codes identical in many cases to those described by Howard and Vitter [4]. Howard and Vitter start with arithmetic codes and represent each interval by a

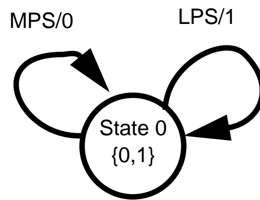

Figure 4 - Uncoded Binary Coder

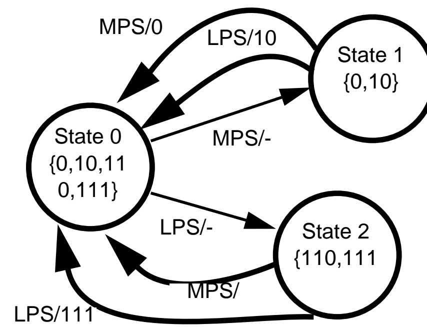

Figure 5 - State Diagram For Huffman Code

state. We believe the conceptual point of view starting from Figures 1 and 2 is useful and is more easily generalized for situations such as source coding for constrained channels.

As is common in the binary arithmetic coding literature, we will use the term "most probable symbol" or mps to refer to the outcome that is estimated to have probability greater than or equal to 0.5. The "least probable symbol" or lps will refer to the other outcome. Thus it is sufficient to consider only probabilities p with $0.5 \leq p \leq 1.0$.

2.0 Uniquely Decodable State Machines

As stated in the introduction, the decoder need only be able to recreate the sequence of outcomes. However, we will consider more closely coupled encoders and decoders. The decoder we will use is shown in Figure 3. It differs only slightly from a typical state machine in that the input is consumed at a variable rate.

The basic operation of the encoder will be to receive a probability and outcome, possibly produce one or more bits of output, and move to a new state. Since the decoder will have the probability estimate and the current state, it is only necessary to be able to tell the difference between the two possible outcomes from the output bit stream. Two transitions will make up a "transition pair" when they start in the same state with the same probability estimate but differ because of the outcome. Figure 4 shows two transitions that are useful when the probability estimate is close to 0.5. Given the output bit stream, it is trivial to determine which transition of the pair occurred. Note that the state machines in Figures 4-7 all have a state 0. This is because these states are all very similar. We will always specify a figure when we refer to a state.

Figure 5 shows a state machine with three states. Starting in state 0 there are two possible transitions, neither of which produces any output bits. In fact, if the first bit in the output stream is a 1, either of the two transitions could have occurred. However, by examining the first *two* bits in the output stream it is always possible to tell which transition occurred. If the output stream begins 11, then the lps must have occurred and the decoder can move to state 2; any other two bit sequence indicates the mps occurred and the decoder should move to state 1. From states 1 and 2 it is easy to tell which transition the encoder took by examining the bit stream. We note that this state machine actually implements the

Huffman code in Table 1.

Table 1 - Huffman Code for Figure 5

Input	Output
MPS/MPS	0
MPS/LPS	10
LPS/MPS	110
LPS/LPS	111

Table 2 - Variable to Variable Representation of Figure 6

Input	Output
MPS/MPS	0
MPS/LPS	10
LPS	11

From the Huffman code example we can develop some general rules that allow creation of "prefix state codes." Note that in Table 1, codewords for outcome sequences that begin with a mps are prefixed by 0 or 01, while sequences that begin with a lps always have codewords prefixed by 11. We define the "prefix set" as the set of output bit stream prefixes allowed in a state. The prefix set has been given for each state in Figure 5. The encoder is capable of creating any binary bit stream initially, but after taking the transition to state 1, only binary sequences which begin with 0 or 10 will be permitted. Likewise after the transition to state 2 only binary sequences which begin with 110 or 111 will be permitted. The idea of a set of valid prefixes makes it clear that decoding is possible. If our state machine starts in a state with a set of valid prefixes, in this case {0,10,110,111}, and a transition pair leads to two states where disjoint subsets of the prefixes are valid, then the output stream will be uniquely decodable. The decoder need only examine the output stream and take the transition to the state which allows the prefix that the stream actually begins with. This idea of valid prefix sequences is very similar to arithmetic coding, but it is now intuitively obvious that finite precision will work. Indeed when we simplify our current Huffman code we will end up with a code which was obtained in [1] and [4] by considering limited precision arithmetic coding.

We will make some simplifications to the state machine in Figure 5. First note that the two valid prefixes in state 2 begin with the sequence 11. Instead of having state 2 output only sequences beginning with 110 or 111 we can output the two bits, 11, when the transition is made into state 2. Then the set of valid prefixes in state 2 becomes {0,1}. This clearly has not changed the code at all. Implementation may be easier since at most 2 bits are output for a single transition instead of 3.

Now, note that the prefix set of {0,1} is not really a constraint at all. In fact any binary sequence is now possible starting from state 2. We also note that although we said the prefix set in state 0 was {0,10,110,111}, all possible binary sequences begin with one of these prefixes. Therefore there really is no constraint on binary sequences in state 0. Thus the prefix set for the new state 2 is equivalent to the prefix set for state 0. Instead of having a transition which outputs 11 and enters state 2, we can have a transition which outputs 11 and returns to state 0. Now there are only two states and we have changed the code. The new state machine appears in Figure 6 and the new code appears in Table 2. This new code is a variable length to variable length (V-V) code; the same code appears in [3]. We note

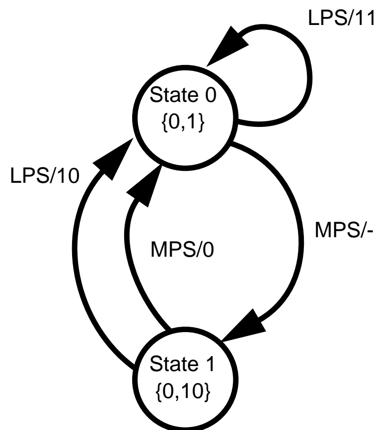

Figure 6 - Variable Length To Variable Length Code

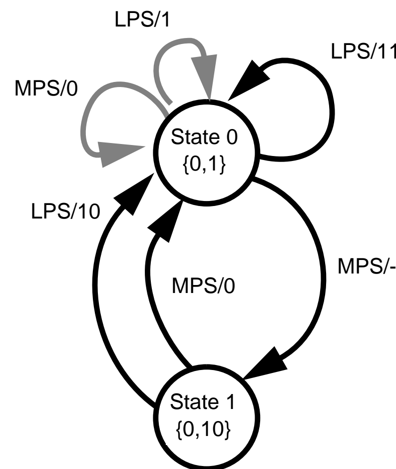

Figure 7 - Multiple Transition Code

that this new V-V code is better than the Huffman code we started with when the most probable symbol probability p satisfies $\theta < p < 1$, where $\theta = 1/\phi \approx 0.618$, and ϕ is the golden ratio. For $p < \theta$, the uncoded code from Figure 4 is better than either the Huffman code of Figure 5 or this V-V code. This can be seen in Figure 8.

So far we have dealt with only one input probability rather than allowing the input probability to change with each symbol. Now we note that the prefix set for state 0 of Figure 4 is {0,1} the same prefix set we could assign to state 0 of Figure 6. We noted that for $p < \theta$ it is better to use uncoded bits. Therefore we combine the machines from Figures 4 and 6 to create Figure 7. When $p < \theta$ and the encoder is in state 0 the transitions from Figure 4 (light gray) are used, otherwise the transitions from Figure 6 (black) are used.

In a general state coder there will be several pairs of transitions leaving each state. The transition pair used to encode an outcome will depend on the probability estimate and the current state, but not on the outcome. Thus we will need to determine several thresholds θ_i to compare the probability estimate with. Since the decoder has access to the same probability estimates, it can determine which transition pair was used without reference to the bit stream. Then, because the two transitions in a pair lead to disjoint subsets of the current prefix set, the decoder can determine which transition of the pair occurred by examining the actual prefix of the bit stream produced. For a simple lookup table, the probability estimate input bits will be chosen to differentiate between the transition pairs rather than being uniformly spaced.

In the case of Figure 7, one bit suffices to specify state, one bit specifies the probability estimate p, and one bit indicates the outcome (mps or lps). The bit specifying p is ignored when the encoder is in state 1. The lookup table needs only 3 input bits. The output consists of one bit for the next state, two bits for possible output, and two bits to indicate how many of the output bits are valid (0, 1, or 2). Coding can be implemented with a 40-bit lookup table, one bit of state, and an output shift register!

The corresponding decoder can examine two input bits, one bit of probability, and one bit of state to produce one bit for the mps/lps declaration, one bit for the next state, and two bits to indicate how many input bits should be consumed. A 64-bit lookup table, one bit of state, and an input shift register allow decoding.

The results from section 4 will show that the two-state coder we have described does not compress data close to the entropy limit for probabilities greater than 0.8. To do better we must have more states.

In order to make larger state machines we could try to improve a larger Huffman, Tunstall, or arithmetic code, but there is an easier way. The prefix set can be written in an equivalent form and this will allow the set to be divided in more ways. Consider the unconstrained prefix set {0,1}. There is only one way to divide this into two disjoint nonempty subsets. However, the prefix set {0,10,11} is equivalent to {0,1} in terms of the constraints it places on binary sequences (none), but it can be divided in more ways. The division into {0} and {10,11} occurs in Figure 4 (by realizing {10,11} is equivalent to outputting a 1 and returning), while the division into {0,10} and {11} occurs in Figure 6. The state machine in Figure 7 can be expanded by writing the state 0 prefix set as {000,001,010,...,111}. This set can be divided as in Figure 4 by realizing that {000,...,011} is equivalent to {0}. The set can be divided as in Figure 6 by realizing {110,111} is equivalent to {11} and {000,...,101} is equivalent to {0,10}.

The set {000,001,010,...,111} can be divided in additional ways, for example, into {111} and {000,...,110}. There is no need to make a new state for {111}; the three bits may simply be output and the transition back to state 0 made. There does need to be a new state for the prefix set {000,...,110}. Once this state is created, there are two possible ways to transition out of it into already existing states. First, the prefix set can be split into {110} and {000,...,101}={0,10}. Second, the prefix set can be split into {000,...,011}=0+{0,1} and {100,...,110}=1+{0,10}, where we have used digit+set to indicate an output followed by a transition to a new state. This expansion will improve compression for higher probabilities p, but it requires more states and more output bit lines.

Both [1] and [4] provide rules for automatically creating states and transition pairs as long as the states correspond to intervals. These rules essentially start with a prefix set of all binary sequences of length N and divide off one end of the set to form new states and at most an N-bit output. The rules allow transition pairs to be created for several of these new states.

The concept of prefix sets and reduced precision arithmetic codes are very similar. We note the carry problem of arithmetic coding never occured, because each state is broken into a set of valid prefixes. Solving the carry over problem has not been cost free. We have given up the possibilities of representing the set {01,10} as bit plus follow and a return to the unconstrained state. In arithmetic coding, with the use of follow bits it is conceivable that no bits will be output until the entire message has been seen, and then either 01111111... or 1000000000... will be output. In an N-state finite machine coder one bit must be output for every N binary inputs. This limits the maximum compression to N:1,

but it also bounds delay. For the decoder there is a maximum number of bits M which must be examined to determine the transitions made. Thus, if a decoder is tied to an encoder over an instantaneous channel, the decoder can always be no more than NM binary decisions behind the encoder.

All the divisions of the prefix set shown so far have been into lexicographically contiguous parts that are equivalent to the interval states in [4]. Now consider a situation were there are three possible channel symbols, namely 1, 2, and 3, of three different costs, namely 1, 2, and 3. The unconstrained prefix set is {1,2,3}. Given a binary decision this set can be divided in three ways {1} and {2,3}, {2} and {1,3}, and {3} and {1,2}. Because the symbols have different costs, each of these divisions is different and is appropriate for different probability estimates. The state with prefix set {1,3} clearly is not an interval. The techniques we describe are still applicable in this situation. Namely, it is easy to prove decodability, and it is not too difficult to assign probability estimates to transition pairs to provide compression.

3.0 Setting Transition Probabilities

The idea of prefix sets for finite state machines makes proof of decodability trivial. For compression it is necessary to choose between transition pairs depending on the probability estimate. As discussed there is some threshold probability where two transition pairs have an equal cost in terms of the number of bits output, above this threshold one transition pair is used, below the threshold a different pair is used. The correct threshold depends on the distribution of the probability estimates, but the correct thresholds are not a function solely of the bits output on each transition.

Consider the state machine in Figure 7. There are two transition pairs leaving state 0. For some probability estimates the gray transitions should be used, for other probability estimates the black transitions should be used. The cost of the gray transition pair is clear. Regardless of the outcome, mps or lps, one bit will be output and the state machine will remain in state 0. The cost of the black transition pair is more involved. If the lps occurs two bits are output and the state machine remains in state 0. If the mps occurs 0 bits are output, but the new state is state one. There is a cost associated with state 1 due to the prefix set. Only bit sequences beginning with 0 or 10 are allowed from this state, which is 3/4 of the possible bit sequences from state 0. Thus there is a cost of at least $-\log_2 3/4$ associated with being in state 1 compared with the starting state 0. This is the cost of the black mps transition. With probability p the mps will occur and a cost of $-\log_2 3/4$ incurred. With probability 1-p the lps transition will be taken and a cost of 2 incurred. The expected cost is:

$$(1-p)*2 + p * (-\log_2 3/4) = 2 - p * \log_2 3.$$

To chose between gray and black transition pairs, we set this cost equal to one and solve for p. We obtain $p = 1/\log_2 3 \approx 0.631$. This suggests that for probability estimates above 0.631 the black transition pair should be used while for smaller estimates the uncoded gray transition pair should be used. This differs from the θ computed in section 2 because

in section 2 we computed the bit rate assuming the probability p was slowly changing.

Unfortunately, there is an additional cost in each state which has not be considered. The uncoded gray transition pair attains the entropy for p = 0.5. The black transition pair leaving state 0 in Figure 7 attains the entropy for p = 0.75. At other values of p we are using the probability “closest” to the estimated probability. The a theoretical cost of using an ideal code for probability q on a source of probability p is given by the divergence, D(p||q). If there are many ideal probabilities q available, i.e., many transition pairs, the divergence will be small for any p. If there is only one, however, the cost will be large for some probabilities.

In short there is a cost of entering state 1 of Figure 7 due to the prefix set, but there is an additional cost due to the fact that only one transition pair is available to leave state 1. Thus if the next probability happens to be far from the ideal probability of the single transition pair in state 1, there is an added cost. This cost must be balanced by the advantage of using the black transition pair rather than the gray transition pair in state 0. Unfortunately, this cost depends on the next probability which we have assumed the coder does not know. We may know the distribution of probability estimates, in which case it is possible to calculate different values of θ. For example, if the probability estimates are uniformly distributed the correct value of $\theta = 0.644$. This is larger than the p = 0.631 calculated earlier because we are considering an additional cost of state 1. For values of $p > \theta$ this additional cost is offset by the gain of using the black transition pair.

The correct threshold probabilities can be found by computing state occupation probabilities as a function of the transition thresholds, θ_i, and the distribution on p. Then using the distribution on p and the state occupation probabilities the expected number of bits per outcome can be calculated. This can be minimized with respect to the transition threshold values using calculus. This may be difficult for large state machines.

An iterative approach is much simpler. We can start with the cost of transitions due to bits output and prefix set cost. From this we can compute estimates of the thresholds. Now, given a probability distribution on p, the probability estimate, it is possible to compute the expected cost of being in a state. This cost is simply the sum of the bits output and state change costs weighted by the probability that each transition will be taken. This new cost can be used in place of the state cost due solely to prefix set size and the process repeated. When the threshold probabilities are changing by a small enough amount the process is terminated and the thresholds are fixed. Another advantage of the iterative method is that it can be performed without an explicit distribution of probability estimates. For example, probability estimates from running the JBIG compressor with some probability estimator can be saved to a file and this file used to set the transition probabilities for a finite state machine entropy coder.

4.0 Simulation Results

For a single context we can assume that the probability of the mps is changing slowly. In this case we can compare the bit rate of a specific state machine with the entropy of the

mps probability, H(p). A simple test was performed by generating one milllion binary decisions with a fixed probability and coding the outcomes with different binary coders. The two-state state machine described previously was used along with a 16-state machine which used three bits for the probability estimate. In addition the QM-coder from the JPEG standard was given the same bit stream. In Figure 8, the actual bits/decision used by three coders minus the entropy is plotted. For probabilities close to one half the 16-state machine always performs better than the QM-coder and even the two-state machine performs better at some probabilities. At high probabilities the finite state machines are limited by the maximum compression acheivable and the QM-coder has a lower bit rate. In many applications, this will not be a major drawback, and the conceptual and implementation simplicity of the state machines may be more important.

It is possible to have codes that do well for a single context, i.e., one slowly changing probability, but perform poorly when the probability changes radically with each decision. To test performance with changing probabilities, we replaced the QM-coder of the JPEG standard with a finite state machine and a simple probability estimator. Using the example quantization matrix but scaling to provide a variety of compression/quality points we tested a number of images. The quality is the same for images compressed with the default Huffman code, the QM-coder or our state machine since this part of the algorithm is lossless. A plot of the number of bits saved by using optimized Huffman codes, finite state machines, and the QM-coder is plotted in Figure 9 for the "barbara" image. The 16-state machine performs almost as well as the QM-coder for all scale factors tested, with the bit rate ranging from 0.5 bits/pixel to 2.0 bits/pixel. Note also, the 16-state machine clearly outperformed the optimized Huffman code which is a two pass algorithm whereas both the finite state machine and the QM-coder were using one pass algorithms.

5.0 Conclusions

We have presented a general method of binary entropy coding using finite state machines. This method can be used to implement Huffman coding, Tunstall coding, variable length to variable length coding, and even arithmetic coding. We have demonstrated one large class of prefix set codes which are obviously uniquely decodable and easy to implement. The lookup table structure provides simplicity in both hardware and software. The speed of operation is simply the state machine cycle time plus shifting time. The method is an alternative to the well known Q-coder and allows any probability estimation method desired. The Q-coder will definitely perform better at extremely skewed probabilities, while this method will work better for probabilities close to 0.5.

References

[1] J. D. Allen, "Entropy Coding with the B-Coder, a Tutorial," Ricoh California Research Center Technical Report CRC-9138, 1991.

[2] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*, Englewood Cliffs: Prentice Hall, 1990.

[3] P. G. Howard and J. S. Vitter, "Parallel Lossless Image Compression Using Huffman and Arithmetic Coding," in *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah,

pp. 299-308, 1992.

[4] P. G. Howard and J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Image and Text Compression*, J. A. Storer, ed., Norwell, MA: Kluwer Academic Publishers, pp. 85-112, 1992.

[5] G. G. Langdon and J. Rissanen, "Compression of Black-White Images with Arithmetic Coding," *IEEE Trans. Comm.*, COM-29, pp. 858-867, 1981.

[6] G. G. Langdon and J. Rissanen, "A Simple General Binary Source Code," *IEEE Trans. Info. Theory*, IT-28, pp. 800-803, 1982.

[7] I. H. Witten, R. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Comm. of the ACM*, 30 (6), pp. 520-540, June 1987.

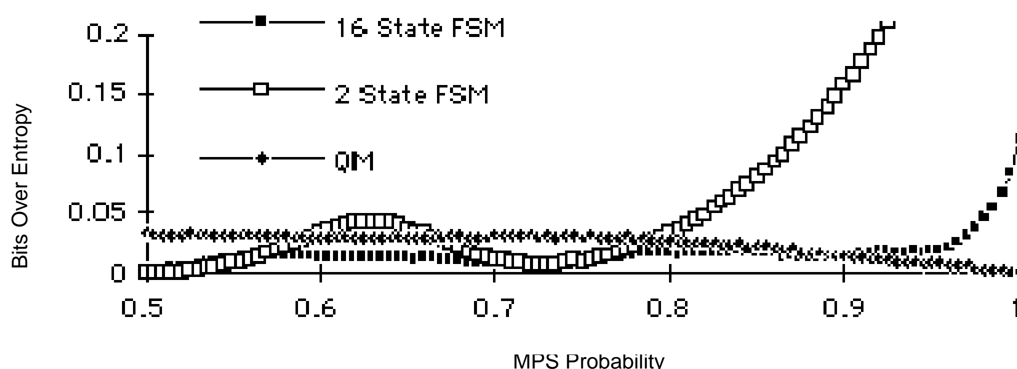

Figure 8 - Fixed Probabilioss

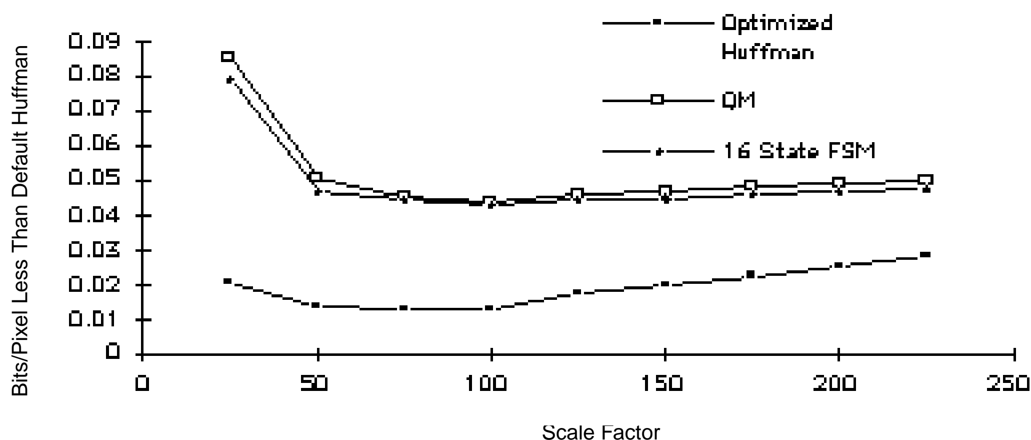

Figure 9 - FSM in JPEG