N94-35555

2

May 15, 1994

p-16

Maximum-Likelihood Soft-Decision Decoding of Block Codes Using the A* Algorithm

L. Ekroot and S. Dolinar
Communications Systems Research Section

The A algorithm finds the path in a finite depth binary tree that optimizes a function. Here, it is applied to maximum-likelihood soft-decision decoding of block codes where the function optimized over the codewords is the likelihood function of the received sequence given each codeword. The algorithm considers codewords one bit at a time, making use of the most reliable received symbols first and pursuing only the partially expanded codewords that might be maximally likely. A version of the A* algorithm for maximum-likelihood decoding of block codes has been implemented for block codes up to 64 bits in length. The efficiency of this algorithm makes simulations of codes up to length 64 feasible. This article details the implementation currently in use, compares the decoding complexity with that of exhaustive search and Viterbi decoding algorithms, and presents performance curves obtained with this implementation of the A* algorithm for several codes.*

I. Introduction

The A* algorithm is an artificial intelligence algorithm for finding the path in a graph that optimizes a function. Nilsson [1, pp. 72–88] describes the algorithm as a heuristic graph-search procedure and shows that the algorithm terminates in an optimal path. The A* algorithm has been used to implement full maximum-likelihood soft decoding of linear block codes by Han et al. [2–4].

The A* algorithm explores the codewords one bit at a time using the most reliable information first, pursuing the most likely codewords first, and ruling out suboptimal codewords as soon as possible. The details of the A* algorithm are covered in Section II. The implementation discussed here has most of the features recommended in [2] and has allowed code performance simulations for codes up to length 64 in reasonable amounts of time.

The codes considered here are binary linear codes. An (N,K) code has 2^K codewords each of length N bits. The binary symbols are transmitted over a communications channel and corrupted by channel noise. An additive noise channel is one in which each received symbol can be described as the sum of the signal and noise. The deep space channel is accurately described as an additive white Gaussian noise channel, meaning that the additive noise for each symbol is an independent random variable distributed according to a zero-mean Gaussian with variance

σ^2. Therefore, the received symbols, r_i for $i = 1, \ldots, N$, that form the received sequence $\boldsymbol{r}$ are continuously valued and are called soft symbols.

Binary antipodal signaling transmits the bits as signals of equal magnitude, but different signs. For this article, the ith bit, $b_i \in \{0, 1\}$, of the binary codeword $\boldsymbol{b}$ is transmitted as $c_i = (-1)^{b_i}$. That is, a 0 is transmitted as $+1$, and a 1 is transmitted as -1. Thus, the energy in a transmitted codeword $\boldsymbol{c}$ is given by $\sum_{i=1}^{N} c_i^2 = \sum_{i=1}^{N}((-1)^{b_i})^2 = N$ independently of the codeword sent. The signal-to-noise ratio (SNR) is E_b/N_0, where $E_b = N/K$ is the energy per information bit, and $N_0/2 = \sigma^2$ is the white Gaussian noise two-sided spectral density. Thus, the SNR is given by

$$SNR = 10 \log_{10} \left(\frac{N}{2\sigma^2 K} \right) (\text{dB})$$

The hard-limited symbol h_i is the transmitted signal value ± 1 nearest to the received symbol r_i; it is given by

$$h_i = \text{sgn}\,(r_i) = \frac{r_i}{|r_i|}$$

where h_i equals 1 in the zero-probability case of $r_i = 0$. If the received symbols are individually hard-limited before decoding, some information is lost from each symbol. For instance, soft symbols like 0.01 and 1.01 both hard limit to $+1$, yet the received value 1.01 is much more likely to have been transmitted as $+1$ than is the received value 0.01. The decoder performance is improved by approximately 2 dB [5, pp. 404–407] if, instead of the hard-limited symbols, the soft symbols r_i are used for what is then called soft-decision decoding.

A. Maximum-Likelihood Decoding

One soft-decision decoding technique is called maximum-likelihood soft-decision decoding. It decodes a received sequence to the codeword $\boldsymbol{c}^*$ that maximizes the likelihood of the received soft symbols.

It is convenient to think of the codewords, which are length N sequences of ± 1's, and the received sequence $\boldsymbol{r}$ as points in N-dimensional space. Assuming an additive white Gaussian noise channel, the codeword $\boldsymbol{c}^*$ that maximizes the likelihood of the received sequence $\boldsymbol{r}$ is the one that minimizes the Euclidean distance between the received word $\boldsymbol{r}$ and the codeword $\boldsymbol{c}$.

The codeword that is closest to the received word can be found by exhaustively checking all possible codewords, or by cleverly seeking out the one that minimizes the distance. The first technique is called exhaustive search maximum-likelihood decoding. For an (N, K) code, there are 2^K codewords to check, making an exhaustive search prohibitive for most interesting codes. Viterbi decoding the block code on a trellis can give better decoding performance with a fixed number of calculations. Techniques such as the A* algorithm that use a heuristic search to find the minimizing codeword can significantly reduce the calculations needed for decoding, especially at a high SNR. Nilsson [1] shows that the A* algorithm terminates with the optimal path; Han [2] shows how to apply the technique to maximum-likelihood decoding; and this article describes the algorithm as we have implemented it.

B. Linear Codes as Trees

Define $\mathcal{C}$ to be an (N, K) linear code with 2^K length N binary codewords $\boldsymbol{b} \in \mathcal{C}$. The generator matrix G for the code is a $K \times N$ matrix of 0's and 1's that has as rows linearly independent codewords. Given K information bits in a row vector $\boldsymbol{x}$, the corresponding binary codeword $\boldsymbol{b}$ is defined to be $\boldsymbol{b} = \boldsymbol{x}G$. For a systematic code, the K information bits are directly visible in the codewords. For the systematic codes referred to here, the first K columns of G form an identity matrix, and the codewords can be divided into information bits, in the first K positions, and parity bits, in the last $N - K$ positions.

1. Example. Consider the shortened (6,3) Hamming code with the generator matrix

$$G = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

The information sequences and the corresponding binary codewords and transmitted codewords are listed in Table 1.

To apply a heuristic tree search algorithm to the decoding of a block code, the code is thought of as a binary tree with 2^K depth N leaves where each path from the root to a leaf corresponds to a codeword. Figure 1 shows the representation of the (6,3) shortened Hamming code as a tree with solid and dashed edges used to represent 0's and 1's, respectively, or equivalently, transmitted $+1$'s and -1's. A node of level l in the tree is defined by the path $\boldsymbol{p}^l = p_1 p_2, \ldots, p_l$ from the root to that node with binary components p_i. For example, the path $\boldsymbol{p}^2 = 10$ specifies the level 2 node designated by "o" in Fig. 1.

If the codewords of a systematic code live in a binary tree, the tree is full through level $K - 1$, that is, every node branches. This is because any length K sequence of 0's and 1's can be an information sequence. Since the parity bits are determined by the information bits, every level K node has only one descendant path that continues to level N.

II. Algorithm Description

The A* algorithm for soft decoding block codes searches a binary tree for the length N path that minimizes a function. On any given iteration, it uses a heuristic to expand the node that is likely to yield the optimal path and eliminate any nodes that can only have suboptimal descendants. The method by which nodes are selected for expansion or eliminated from consideration uses an underestimate of the function to be minimized, called a heuristic function. The heuristic function at a node must lower bound the function to be minimized for all paths that pass through that node.

For maximum-likelihood soft-decision decoding, the function that is minimized over all codewords is the Euclidean distance between a codeword and the received word. It is equivalent to minimizing the square of the Euclidean distance:

$$s(\boldsymbol{r}, \boldsymbol{c}) = \sum_{i=1}^{N} (r_i - c_i)^2$$

For the algorithm to find the minimizing codeword, the heuristic function at a node must be less than or equal to the actual squared distance of any full-length path that passes through that node (or equivalently any codeword that is prefixed by $\boldsymbol{p}^l$, the path that defines the node).

The minimum distance over all codewords that begin with the path $\boldsymbol{p}^l$ is lower bounded by the minimum distance over all length N binary sequences that begin with the path $\boldsymbol{p}^l$. This can be made explicit for distance squared by

$$\min_{\{\boldsymbol{b} \in \{0,1\}^N | b_i = p_i,\ i=1,2,\ldots,l\}} s(\boldsymbol{r}, \boldsymbol{c}) \leq$$

$$\min_{\{\boldsymbol{b} \in C | b_i = p_i,\ i=1,2,\ldots,l\}} s(\boldsymbol{r}, \boldsymbol{c})$$

where $c_i = (-1)^{b_i}$. The minimum distance over all length N sequences that begin with the path $\boldsymbol{p}^l$ is achieved by the sequence that begins with $\boldsymbol{p}^l$ and continues with binary symbols consistent with the hard-limited received symbols. The heuristic function is the squared distance from the received sequence to either the codeword defined by the path $\boldsymbol{p}^K$, if the node is at level $l = K$, or the sequence that begins with the path $\boldsymbol{p}^l$ and is completed by symbols consistent with the hard-limited symbols, if $l < K$.

A. Fundamentals of the Algorithm

The A* algorithm keeps an ordered list of possible nodes to operate on. Associated with each node is the path $\boldsymbol{p}^l$ that identifies the node, the value of the heuristic function, and an indicator of whether the node represents a single codeword. The value of the heuristic function determines the order of the list of nodes and, therefore, guides the search through the tree.

When the A* algorithm begins the search, the root of the tree is the only node on the list. The algorithm expands a node if it might yield a codeword with the minimum distance from the soft received symbols, and eliminates from the list nodes that are too far from the received symbols to have the maximum-likelihood codeword as a descendant. The algorithm terminates when the node at the top of the list represents a single codeword. That codeword is the maximum-likelihood codeword.

At each iteration, the node on the top of the list, which has the smallest value of the heuristic function, is expanded. It is taken off the list, and the two possible ways of continuing the path are considered as nodes to put back on the list. Each new node is placed back on the list provided that its heuristic function is not greater than the actual squared distance for a completed codeword. If the node expanded is at level $K - 1$, the two level K children specify codewords, and the heuristic function for each child node is the actual squared distance between the codeword and the received word. These codewords are called candidate codewords. When a node that defines a codeword is placed back on the list, all nodes below it on the list are deleted.

B. Features That Improve the Efficiency

The features described in this section are not necessary to guarantee maximum-likelihood soft-decision decoding, but they improve the efficiency. Sorting the bit positions to take advantage of reliable symbols reduces the number of nodes expanded. Using a simplification of the heuristic function reduces the number of computations during each node expansion. These features have been implemented and make the A* decoder a practical tool for decoding.

1. Sorting by Reliability. If the bit positions corresponding to the more reliable received symbols are expanded first, then the search will be directed more quickly to close candidate codewords. The nearer a symbol is to 0, the less reliable it is because it is almost equally far from both +1 and −1. Similarly, the greater the magnitude of the received symbol, the more reliable that symbol is. To take advantage of the reliable symbols, the received symbols are reordered in descending order of magnitude, and the code symbols are reordered equivalently. Since the A* algorithm defines the code by the $K \times N$ generator matrix G, reordering the code symbols is equivalent to reordering the columns of the generator matrix.

This implementation of the A* algorithm sorts the received symbols by reliability, reorders the columns of the generator matrix in the same way, and then tries to row reduce the generator matrix so that it is systematic. However, if it encounters a column, among the first K columns, that is linearly dependent on previous columns, it moves the offending column and corresponding received symbol to the end, and continues the row reduction.

Typically, the number of nodes expanded while decoding a received word is significantly reduced by sorting the symbols before starting the decoding process. The increase in efficiency from sorting first was found, for the shorter codes like the Golay (24,12) code, to outweigh the cost of sorting and row reducing the generator matrix. For the larger codes, like the quadratic residue (QR) (48,24) code, decoding without sorting was so much more time-consuming that it was not a reasonable option to run comparison tests. Sorting was adopted as a standard feature.

2. Alternative Function to Minimize. Every soft symbol r_i is at least as far away from the codeword symbol c_i as it is from the hard-limited symbol h_i. The squared distance, $s(\boldsymbol{r},\boldsymbol{c})$, can be written as the sum of the square of the distance to the hard-limited symbols, $s(\boldsymbol{r},\boldsymbol{h})$, and an amount $a(\boldsymbol{r},\boldsymbol{c})$ that is nonzero only when there is at least one symbol c_i that does not equal the hard-limited symbol h_i, as follows:

$$s(\boldsymbol{r},\boldsymbol{c}) = \sum_{i=1}^{N}(r_i - c_i)^2$$

$$= \sum_{i=1}^{N}(r_i - h_i + h_i - c_i)^2$$

$$= \sum_{\substack{i=1\\ h_i=c_i}}^{N}(r_i - h_i)^2 + \sum_{\substack{i=1\\ h_i\neq c_i}}^{N}(r_i - h_i + 2h_i)^2$$

$$= \sum_{\substack{i=1\\ h_i=c_i}}^{N}(r_i - h_i)^2$$

$$+ \sum_{\substack{i=1\\ h_i\neq c_i}}^{N}\left[(r_i - h_i)^2 + 4h_i(r_i - h_i) + 4h_i^2\right]$$

$$= \sum_{i=1}^{N}(r_i - h_i)^2 + 4\sum_{\substack{i=1\\ h_i\neq c_i}}^{N} h_i r_i \tag{1a}$$

$$= \sum_{i=1}^{N}(r_i - h_i)^2 + 4\sum_{\substack{i=1\\ h_i\neq c_i}}^{N} |r_i| \tag{1b}$$

$$= s(\boldsymbol{r},\boldsymbol{h}) + 4a(\boldsymbol{r},\boldsymbol{c}) \tag{1c}$$

where Eq. (1a) follows by recombining the summations of like terms, Eq. (1b) uses $h_i r_i = \operatorname{sgn}(r_i) r_i = |r_i|$, and Eq. (1c) introduces an alternative function,

$$a(\boldsymbol{r},\boldsymbol{c}) = \sum_{\substack{i=1\\ \operatorname{sgn}(r_i)\neq c_i}}^{N} |r_i|$$

Since the first term in Eq. (1c) does not depend on the codeword $\boldsymbol{c}$, it is constant over the minimization, and

$$\min_{\boldsymbol{c}} s(\boldsymbol{r},\boldsymbol{c}) = s(\boldsymbol{r},\boldsymbol{h}) + 4\min_{\boldsymbol{c}} a(\boldsymbol{r},\boldsymbol{c})$$

Maximum-likelihood decoding of the received sequence can be done by finding the codeword that minimizes either $s(\boldsymbol{r},\boldsymbol{c})$ or $a(\boldsymbol{r},\boldsymbol{c})$.

Because each term of $a(\boldsymbol{r},\boldsymbol{c})$ is either zero or $|r_i|$, based on a comparison, it is simpler to calculate than $s(\boldsymbol{r},\boldsymbol{c})$, which for each i requires a difference and a square. The alternative function $a(\boldsymbol{r},\boldsymbol{c})$ is used by this implementation of the A* algorithm.

3. Example Revisited. Consider the shortened (6,3) Hamming code and the received sequence $\boldsymbol{r} = (0.05, -1.3,$

1.1, 0.8, −0.25, 0.6). Reordering the received vector by magnitude gives $\boldsymbol{r}' = (-1.3,\ 1.1,\ 0.8,\ 0.6,\ -0.25,\ 0.05)$. The reordered generator matrix in systematic form is

$$G' = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

and a binary codeword $\boldsymbol{b}$ in the original code corresponds to a sorted codeword $\boldsymbol{b}'$. Figure 2 shows the tree explored by the A* algorithm when the sorted code is used. Each node is labeled with the value at that node of the heuristic function using the alternative function. The 3 expanded nodes are each designated by a "o"; the 2 candidate codewords are each designated by a "Δ"; and the 12 edges explored are the edges of the tree that are shown. The nodes with paths 0, 11, and 101 are dropped from the list when the candidate word, with alternative function 0.65, is put on the list. The search promptly terminates because the top node on the list defines a candidate codeword, namely $\boldsymbol{b}' = 100111$. Unshuffling $\boldsymbol{b}'$ gives the maximum-likelihood decoded codeword to be $\boldsymbol{b} = 110011$. For comparison, Fig. 3 shows the larger tree explored by the algorithm when the symbols are not sorted.

C. Verification of the Decoder

The decoding results of the A* algorithm were compared to the results of two exhaustive search decoder implementations. The (24,12) Golay code was used for this test since it has only $2^{12} = 4096$ codewords, making it feasible to get timely results from an exhaustive search decoder. First, the software decoded the received sequences using both A* and exhaustive search, and compared the results internally. Second, a couple hundred received sequences were decoded by both the A* software and an independent exhaustive search decoder written in APL. The results showed that both exhaustive search and A* decoders decoded the same noisy vectors to the same codewords.

The software to implement the A* algorithm has been written in C and run on several Sun platforms. Since integers on these processors are 32 bits long, the software to implement the A* algorithm has been constrained to linear codes with 64 or fewer bits per codeword by using two 32-bit integers for each codeword. Because of this implementation detail, it was important to confirm that the A* software properly decodes codes longer than length 32. Most interesting codes with lengths over 32 bits take a prohibitively long time to decode exhaustively. A test code with length N greater than 32 and one with more than 32 information bits were devised so they could be readily decoded by other means. The code with length greater than 32 was created by repeating the parity bits of the Golay (24,12) code. This formed a (36,12) code that was no more difficult to exhaustively decode than was the Golay (24,12) code. After debugging and testing, the decoder decoded 500 codewords consistent with the exhaustive decoder results. Next, a simple (34,33) code, consisting of 33 information bits and 1 overall parity bit, was tested on 200 noisy received words. This code was selected because a maximum-likelihood decoder is easy to write, and an APL program was used to verify that the 200 test words decoded consistently.

D. Operational Details

To analyze the performance of either the algorithm or a code, data are taken by running the software with different input parameters. For a given run, the software takes as input the generator matrix for the code, the SNR, the seed for the random number generator, and the number of words to decode. It returns the average number of nodes expanded, the average number of candidate codewords, and the number of word errors that occurred. Sometimes a system call from inside the program was used to provide the amount of central processing unit (CPU) time consumed during a run. These decoding runs ranged in size from hundreds to tens of thousands of decoded received sequences. The codes that have been examined include a Bose–Chaudhuri–Hocquenghem (BCH) (63,56) code, a quadratic residue (48,24) code, a Golay (24,12) code, a BCH (31,10) code, and a Reed–Muller (32,16) code. The data from multiple runs are combined carefully to give the results in the following sections.

III. Algorithm Performance

The intricacy of the A* algorithm makes it difficult to describe the number of calculations necessary to decode a received sequence. Possible measures of the number of calculations describe the search size for the A* algorithm and include the number of candidate codewords, the number of expanded nodes, and the number of edges searched in the tree. The number of edges E in the search tree is given by

$$E = 2X + (N - K)C$$

where X is the number of nodes expanded including the root and C is the number of candidate codewords. Because the search size for the A* algorithm varies from one received sequence to the next, the averages of these numbers over many received sequences are used for comparison. Section III.A explains simulation timing results that show that the search size and the time to decode are related linearly. Section III.B shows how the average size of the A* search tree depends on the signal-to-noise ratio. Section III.C introduces other maximum-likelihood decoders that are used for comparisons in Section III.D.

A. Time to Decode Versus Search Size

The average amount of time it takes to decode received sequences reflects both the computational overhead for each sequence decoded and the computations for each part of the search tree. Analyzing the time to decode requires that all of the timing data be taken on the same computer and that the accuracy of the timing data be sufficient to perform comparisons. The system call used to generate the timing information for a run was accurate to within a second, which is too coarse to study data on individual decoded sequences, but sufficient for data on ensembles of decoded sequences.

The relationship between the indicators of search size introduced earlier and decoding time may be observed in the data from many runs for the quadratic residue (48,24) code on a Sparc 10 Model 30 workstation. The average decoding times versus the average numbers of candidate codewords and expanded nodes are shown in Fig. 4, along with a weighted linear fit[1] to the data. Although the data display a small amount of statistical variability, the time to decode displays a nearly linear relationship to the indicators of search size.

The relationship between the average numbers of expanded nodes and candidate codewords for the (48,24) code in Fig. 5 is well described as linear.

B. Search Size Versus SNR

The average size of the tree that the A* algorithm searches is a function of the SNR for the received sequences. For each of the codes studied, the average numbers of candidate codewords, expanded nodes, and search tree edges are shown versus SNR in Figs. 6, 7, and 8, respectively. The total number of sequences decoded for each point in these figures is shown in Table 2.

Not surprisingly, for extremely high SNR, the A* algorithm typically finds only two candidate words, along the way expands K nodes, and therefore has a search tree with $E = 2K + 2(N - K) = 2N$ edges.

For low SNR, the soft symbols are predominantly noise, but the A* algorithm still expands a mere fraction of the nodes in the tree, especially as it bases early decisions on the symbols that contribute most to the final choice. Figures 6 and 7 show that simulation data of the average numbers of candidate words and expanded nodes are almost constant for SNRs below −4 dB. Since the number of edges is calculated from the number of nodes and candidate words, the average number of edges also levels off for low SNRs, as seen in Fig. 8.

The algorithm was also tested for each code with no signal at all, i.e., an SNR of $-\infty$. Table 3 shows that for each code the average number of expanded nodes and the average number of candidate words during A* decoding were comparable to the numbers for SNRs below −4 dB. It also shows the number of words decoded to obtain these averages.

C. Other Maximum-Likelihood Decoders

Many other maximum-likelihood soft-decision decoding algorithms use a fixed number of calculations to decode any received sequence, independent of the SNR.

1. Exhaustive Search and Full Tree Search. An exhaustive search decoder calculates the distance between the received sequence and each codeword in the code individually, and returns the codeword with the minimum distance. For an (N, K) block code, an exhaustive search technique computes the distances from the received sequence to all 2^K codewords. If exhaustive search is cast in terms of a graph with one edge for each bit in each codeword, the number of edges for an exhaustive search is $N2^K$, independent of the SNR.

A slightly more efficient technique to compute the distances to all the codewords is to use the full code tree. Here the squared distance from the received sequence to a codeword, at a leaf, is the sum along the path to that leaf of the squared distances from each received symbol to the symbol associated with each edge. For an (N, K) code, the number of edges in the full tree is $2^{K+1} - 2 + (N - K)2^K$. This technique checks all 2^K leaves but has fewer edges than an exhaustive search.

2. Viterbi Decoding of Block Codes. To apply soft-decision Viterbi decoding to a block code, the code is represented as a trellis. Wolf [6] and Massey [7] introduce a minimal code trellis for block codes. McEliece [8] shows

[1] The number of decoded words in each run was included in the line fitting process to account for the variation in accuracy between data from large and small runs.

a simple technique for constructing the minimal trellis for a given code, and also shows that it is optimal for Viterbi decoding complexity. A Viterbi decoder for a code on a trellis uses a constant number of calculations and comparisons independent of signal-to-noise ratio. The Viterbi decoding complexity for a given trellis for a given code can be measured by the number of edges in the trellis.

An (N, K) code has a minimal trellis that can be constructed from the generator matrix. Different permutations of a code may have different minimal trellises. There are codes, such as cyclic codes, for which the minimal trellis has the most edges compared with the minimal trellises for other permutations of the code. The permutation that gives the most edges in the minimal trellis is the worst permutation of the code. The number of edges in the minimal trellis for the worst permutations is no more than $2^{M+2} - 4 + 2^M(N - 2M)$ where $M = \min\,(K, N - K + 1)$. Other permutations give smaller minimal trellises.

D. Comparisons

The search size for the A* algorithm depends on the received sequence, and the average search size depends on the SNR. The averages found with no signal present are used for comparison with the maximum-likelihood decoders that use a fixed number of calculations.

Table 4 shows the number of edges used for an exhaustive search for each code and for the full code tree, both of which are greater than the average number of edges in the A* search tree shown for the case of no signal in Table 3. The average number of edges in the search tree for the A* algorithm is presented for comparison to the number of edges in the trellis for Viterbi decoders. Table 4 also shows the number of edges in some special trellises for the codes where the numbers are known or bounds have been calculated.

Consider the Golay (24,12) code. An exhaustive search explores $(24)2^{12} = 98,304$ edges. The full tree has $2^{13} - 2 + (12)2^{12} = 57,342$ edges. The number of edges in the minimal trellis for the worst-case permutation of the Golay $(24,12)$ code is $2^{14} - 4 = 16,380$. The number of edges in the minimal trellis for the best permutation of the Golay code is 3580 [8]. By using enhancements on a certain trellis for the Golay code, Forney [9, p. 1183] reduces the number of binary operations of a decoder to 1351. This decoder is mentioned as interesting, but the number of binary operations is not directly comparable to edge counts for the A* algorithm. Note that at low SNRs the A* algorithm on average expands 62 nodes, checks 29 candidate codewords, and has a search tree with 469 edges.

Less is known about the best trellises for the quadratic residue (48,24) code. A worst-case permutation produces a minimal trellis with $2^{26} - 4 = 67,108,860$ edges. A better permutation results in a minimal trellis with 925,692 edges. The A* algorithm search tree has on average $34,429$ edges when no signal is present.

IV. Future Enhancements

There are several elements affecting the efficiency of the software, including the initial computations to set up the search, the size of the search, and the number of computations for each part of the search. With greater understanding of the algorithm come more ideas for improving the software to reduce at least one of these elements. Either using the minimum distance of the code to determine if the search can be successfully terminated before the list is exhausted or improving the heuristic function will reduce the search size. Ideas like reducing the complexity of sorting the received symbols for each decoding will trim the number of initial setup computations, but will increase the search size by an undetermined amount.

A. Escaping When a Candidate Is Definitely Closest

If the angle between the received word and a candidate codeword, as viewed from the origin, is less than half the minimum angle between any two codewords, then that candidate codeword is the closest one to the received word. One of the suggestions in [2] is to calculate the angle between the received word and each candidate codeword as it is found. If this angle indicates that the candidate is the closest codeword, then declare that the decoding is complete, and exit the algorithm.

This feature has not been implemented at this writing, but is expected to reduce the size of the tree searched.

B. Improving the Heuristic Function

Each parity bit is a linear function of a subset of the information bits. If a node in the tree is deep enough to specify all the information bits for a particular parity bit, then any codeword passing through that node will have the same value for that parity bit. The heuristic function could use this parity bit to improve the distance underestimate for all nodes at that depth, and thus reduce the size of the tree searched by the algorithm.

C. Sorting Fewer Symbols

To simplify the sorting of symbols by reliability and the necessary row reductions to the generator matrix, it may be beneficial to sort only the information symbols or to sort only a few of the most reliable symbols.

Sorting only the information symbols in the received word greatly simplifies the production of a systematic generator matrix for the reordered code bits. Since the columns corresponding to information bits are columns of an identity matrix, they are linearly independent, and exchanging rows is all it takes to return the generator matrix to systematic form. Thus, the row reduction portion of the algorithm is simplified.

Another possibility for simplifying sorting is to reorder only the n most reliable linearly independent symbols and not to reorder the rest. In such a design, there are no more than $N!/(N-n)!$ reorderings of the columns, and, hence, this many systematic generator matrices. For small values of n such as 1 or 2, it may be acceptable to store and retrieve systematic generator matrices for each of these reorderings.

V. Code Performance

The A* algorithm that we have implemented has been very useful for simulating code performance. Figure 9 shows the probability of word error versus SNR for the BCH (63,56), Reed–Muller (32,16), BCH (31,10), Golay (24,12), and quadratic residue (48,24) codes. The error bars are one standard deviation of an average of m independent Bernoulli trials. Specifically, the estimated standard deviation is $\sigma = \sqrt{p(1-p)/m}$, where p is the estimate of the probability of word error at a given SNR, and m is the number of decodings done at that SNR.

VI. Conclusions

The application of the A* algorithm to maximum-likelihood soft-decision decoding allows for efficient simulation of code performance. The A* algorithm finds the codeword that maximizes the likelihood of the received word given the codeword. This is equivalent to minimizing either the Euclidean distance between the received word and a codeword or the alternative function described in Section II.B.2. The use of a heuristic function constrains the search to only a subtree of the code's finite binary tree. The heuristic function underestimates the function being minimized in order to ensure that the subtree contains the optimal path.

The size of the tree searched by the A* algorithm, as described by the numbers of nodes expanded, candidate words, and edges, is a good indicator of the complexity for decoding that received sequence. Since the search size depends on the received sequence, the average search size as a function of signal-to-noise ratio is used for comparison. The search tree is smallest for a high SNR, where the algorithm goes straight to the maximum-likelihood codeword, and larger at a low SNR, where the searched portion of the tree is still much smaller than the full code tree. At a low SNR, the average size of the A* search tree is also smaller than the fixed trellis size of a good Viterbi decoder.

For research applications, simulations using this implementation can provide data on code performance, such as word error rate, for comparisons to theoretical results, such as bounds, and for testing other predictors of code performance.

Acknowledgments

The authors would like to thank A. Kiely, R. McEliece, and F. Pollara for many helpful discussions.

References

[1] N. J. Nilsson, *Principles of Artificial Intelligence*, Palo Alto, California: Tioga Publishing Co., 1980.

[2] Y. S. Han, C. R. P. Hartmann, and C.-C. Chen, *Efficient Maximum-Likelihood Soft-Decision Decoding of Linear Block Codes Using Algorithm A**, Technical Report SU-CIS-91-42, School of Computer and Information Science, Syracuse University, Syracuse, New York, December 1991.

[3] Y. S. Han and C. R. P. Hartmann, *Designing Efficient Maximum-Likelihood Soft-Decision Decoding Algorithms for Linear Block Codes Using Algorithm A**, Technical Report SU-CIS-92-10, School of Computer and Information Science, Syracuse University, Syracuse, New York, June 1992.

[4] Y. S. Han, C. R. P. Hartmann, and C.-C. Chen, "Efficient Priority-First Search Maximum-Likelihood Soft-Decision Decoding of Linear Block Codes," *IEEE Transactions on Information Theory*, vol. 39, no. 5, pp. 1514–1523, September 1993.

[5] S. Benedetto, E. Biglieri, and V. Castellani, *Digital Transmission Theory*, Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1987.

[6] J. K. Wolf, "Efficient Maximum Likelihood Decoding of Linear Block Codes Using a Trellis," *IEEE Transactions on Information Theory*, vol. IT-24, no. 1, pp. 76–80, January 1978.

[7] J. L. Massey, "Foundations and Methods of Channel Coding," *Proceedings of the International Conference on Information Theory and Systems*, NTG-Fachberichte vol. 65, Berlin, pp. 148–157, September 18–20,1978.

[8] R. J. McEliece, "The Viterbi Decoding Complexity of Linear Block Codes," to be presented at IEEE ISIT'94, Trondheim, Norway, June 1994.

[9] G. D. Forney, Jr., "Coset Codes — Part II: Binary Lattices and Related Codes," *IEEE Transactions on Information Theory*, vol. IT-34, no. 5, pp. 1152–1187, September 1988.

Table 1. Information sequences and the corresponding binary and transmitted codewords for the (6,3) shortened Hamming code.

Information sequence x	Binary codeword b	Transmitted codeword c
0 0 0	0 0 0 0 0 0	1 1 1 1 1 1
0 0 1	0 0 1 0 1 1	1 1 −1 1 −1 −1
0 1 0	0 1 0 1 0 1	1 −1 1 −1 1 −1
0 1 1	0 1 1 1 1 0	1 −1 −1 −1 −1 1
1 0 0	1 0 0 1 1 0	−1 1 1 −1 −1 1
1 0 1	1 0 1 1 0 1	−1 1 −1 −1 1 −1
1 1 0	1 1 0 0 1 1	−1 −1 1 1 −1 −1
1 1 1	1 1 1 0 0 0	−1 −1 −1 1 1 1

Table 2. The number of codewords decoded and used to generate Figs. 6 through 9.

SNR, dB	BCH (31,10)	QR (48,24)	Golay (24,12)	Reed-Muller (32,16)	BCH (63,56)
−7	40,000	6200	6400	6400	6100
−6	40,000	6200	6400	6400	6100
−5	40,000	6200	6400	6400	5900
−4	40,000	6200	6400	6400	5900
−3	40,000	6200	6400	6400	5900
−2	642,100	47,400	58,900	6400	33,300
−1	923,200	70,600	71,700	19,200	57,200
0	1,204,000	83,500	84,500	32,000	83,700
1	1,483,000	207,900	385,000	279,300	279,600
2	3,362,100	409,800	465,000	359,100	468,900
3	5,014,000	1,042,500	1,102,500	995,000	1,036,000
4	6,212,500	1,571,000	2,113,000	1,200,000	1,256,000
5	7,418,000	2,472,500	6,055,000	1,400,000	1,476,000
6	21,242,000	8,048,000	16,330,000	7,200,000	6,512,000
7	12,400,000	8,337,000	8,400,000	8,400,000	7,547,000

Table 3. The number of codewords decoded at an SNR of negative infinity, and the average numbers of expanded nodes, candidate codewords, and edges in the search tree for the A* algorithm.

A* results at SNR of negative infinity	BCH (31,10)	QR (48,24)	Golay (24,12)	Reed-Muller (32,16)	BCH (63,56)
Number of decoded words	89,400	29,900	90,000	59,600	35,600
Average number of expanded nodes	181.16	2705.54	62.45	244.66	79.59
Average number of candidate codewords	120.96	1209.07	28.65	113.34	12.28
Average number of edges in search tree	2902.56	34,428.71	468.68	2302.77	245.12

Table 4. Indicators of decoding complexity for some maximum-likelihood decoding techniques that use a fixed search size or number of calculations for decoding.

Other decoder sizes for comparison to A*	BCH (31,10)	QR (48,24)	Golay (24,12)	Reed-Muller (32,16)	BCH (63,56)
Number of edges in a minimal trellis for the worst permutation of the code	15,356	67,108,860	16,380	262,140	13,052
Number of edges in a minimal trellis for a better permutation of the code		925,692			
Number of edges in a minimal trellis for the best permutation of the code		$\geq$860,156	3580		
Number of binary operations in a trellis with enhancements for the Golay (24,12) code [9, p. 1183]			1351		
Number of edges in a full code tree	23,550	436,207,614	57,342	1,179,646	6.5×10^{17}
Number of edges in an exhaustive search	31,744	805,306,368	98,304	2,097,152	4.5×10^{18}

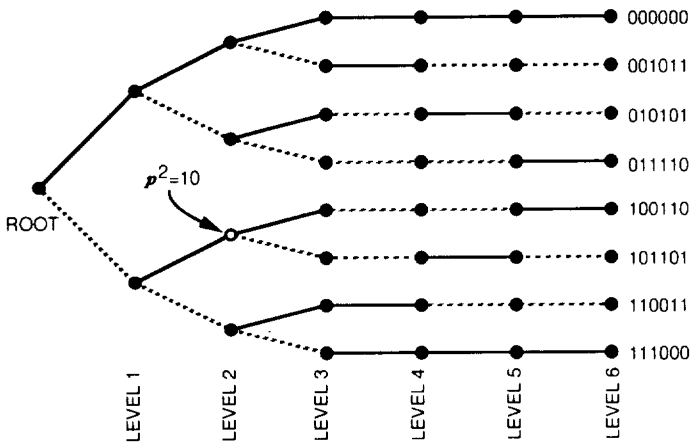


Fig. 1. A binary tree representation of the (6,3) shortened Hamming code.

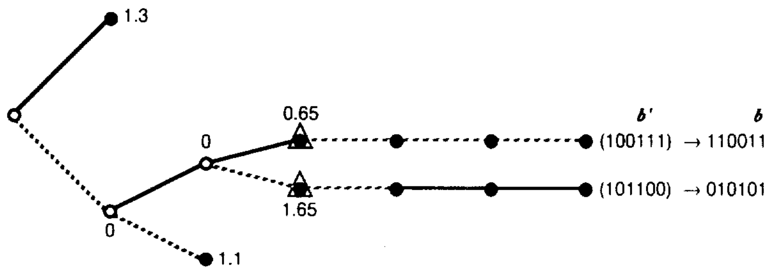


Fig. 2. The tree explored by the A* algorithm when the bit positions are sorted to take advantage of the more reliable symbols first. Expanded nodes are designated by o, and nodes defining candidate codewords are designated by Δ.

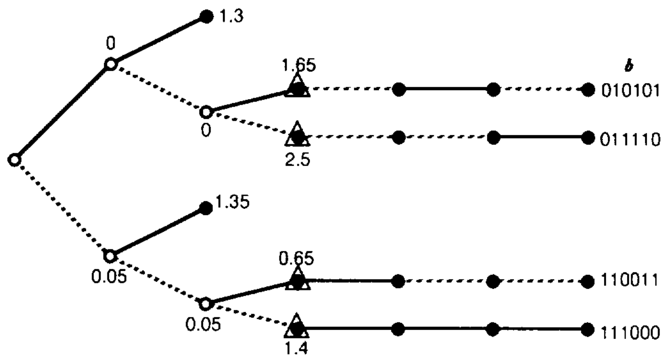
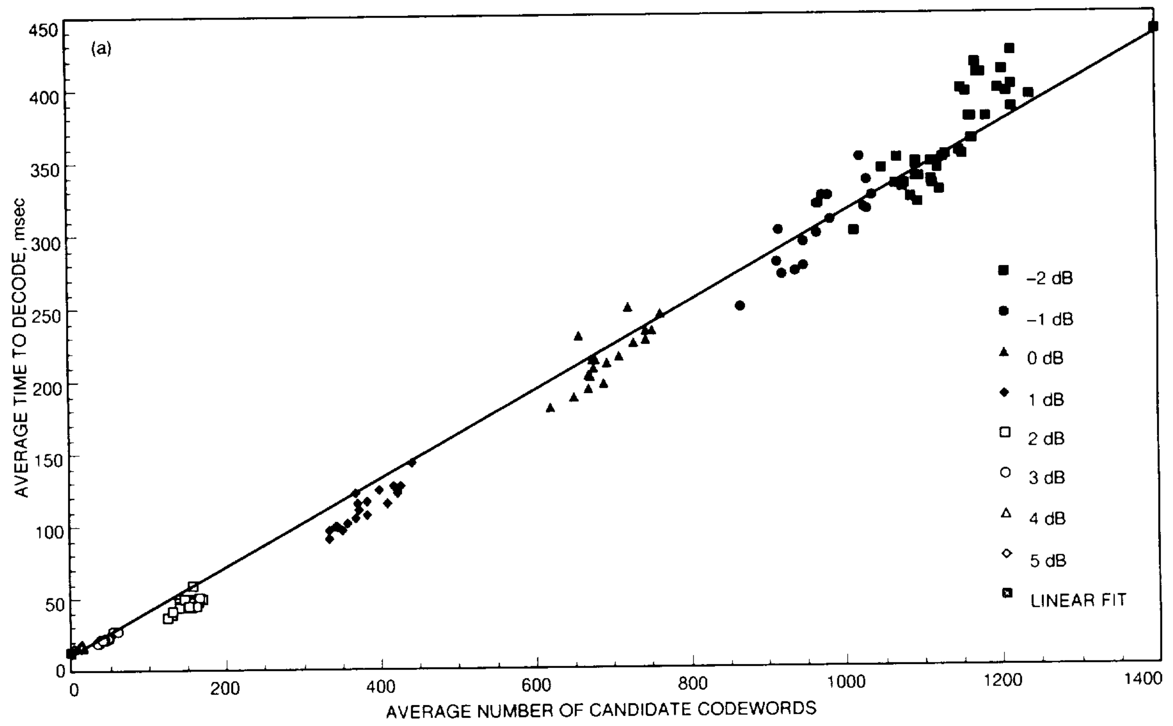


Fig. 3. The tree explored by the A* algorithm when the bit positions are not sorted. Expanded nodes are designated by o, and nodes defining candidate codewords are designated by Δ.



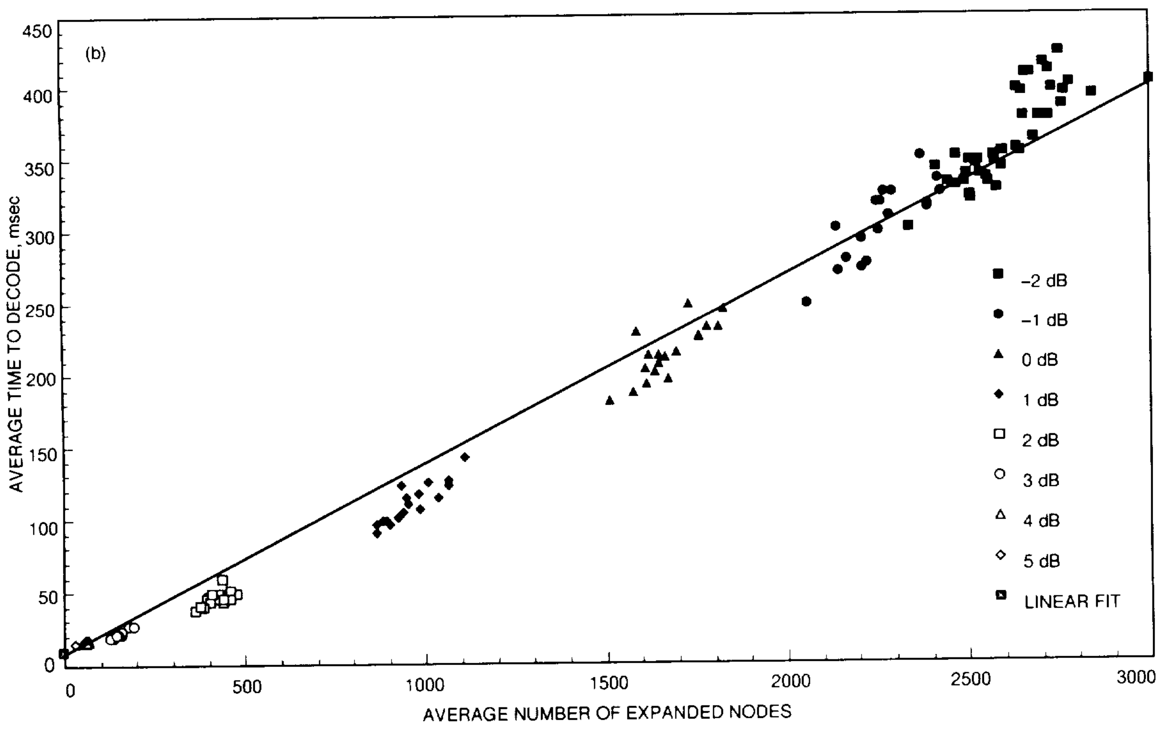


Fig. 4. Scatter plots of average CPU time per decoded word versus (a) average number of candidate codewords per decoded word and (b) average number of expanded nodes per decoded word, for the quadratic residue (48,24) code on a Sparc 10 Model 30 workstation.

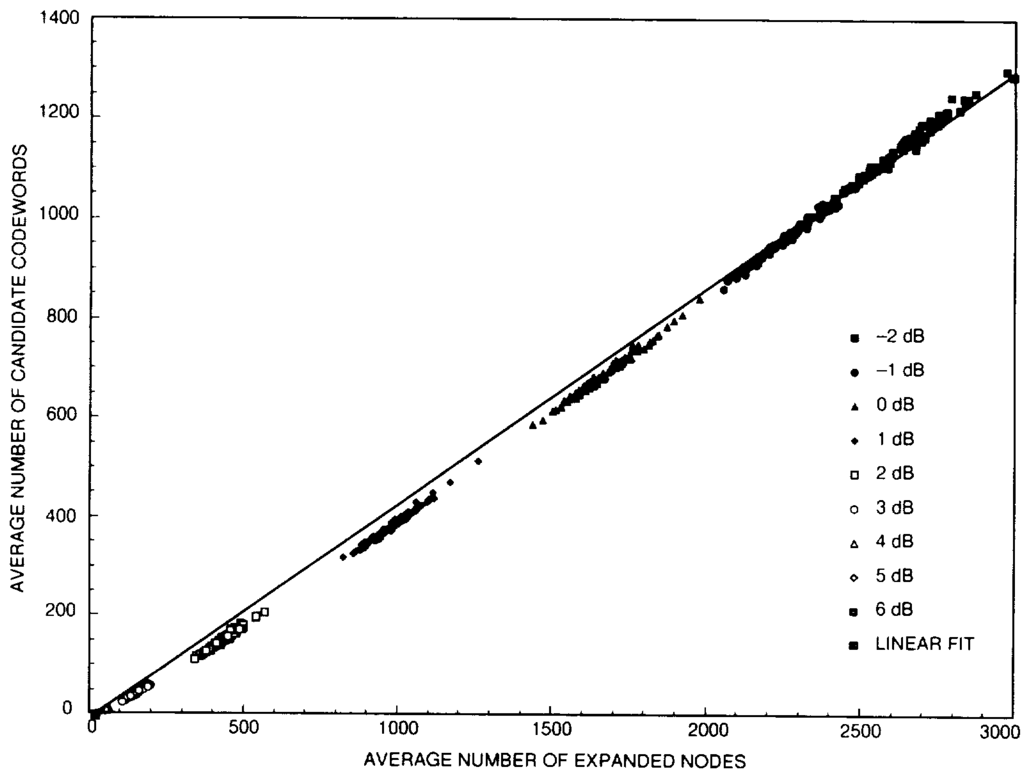


Fig. 5. A scatter plot of the average number of candidate codewords versus the average number of expanded nodes for the quadratic residue (48,24) code, shown with a weighted linear fit.

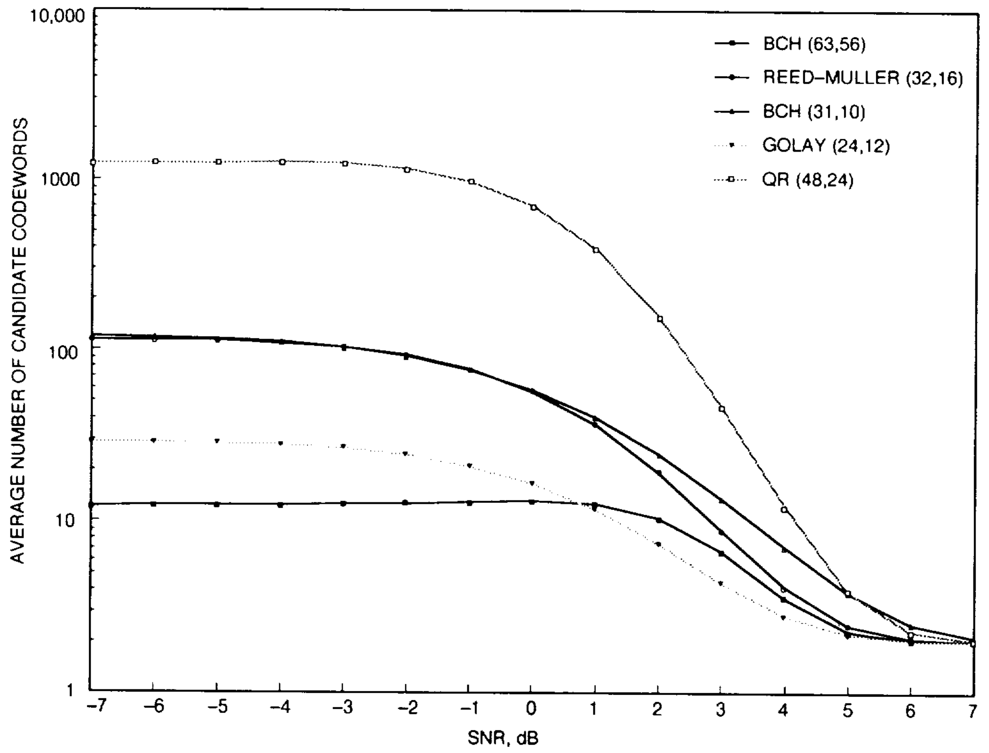


Fig. 6. The average number of candidate codewords for several codes as a function of SNR.

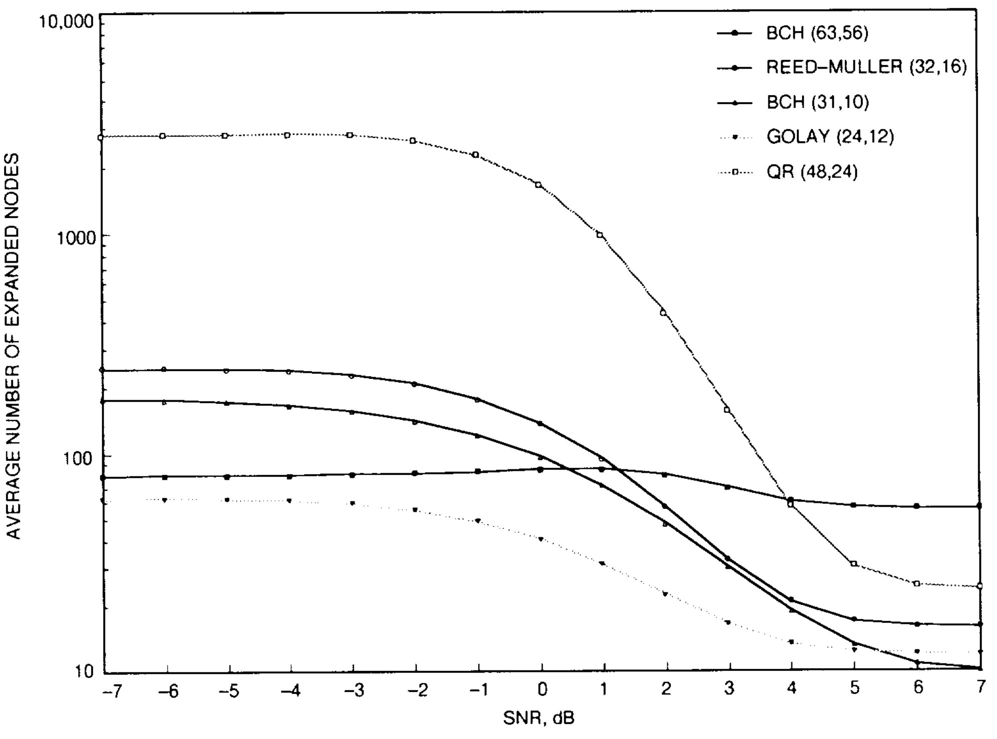


Fig. 7. The average number of expanded nodes for several codes as a function of SNR.

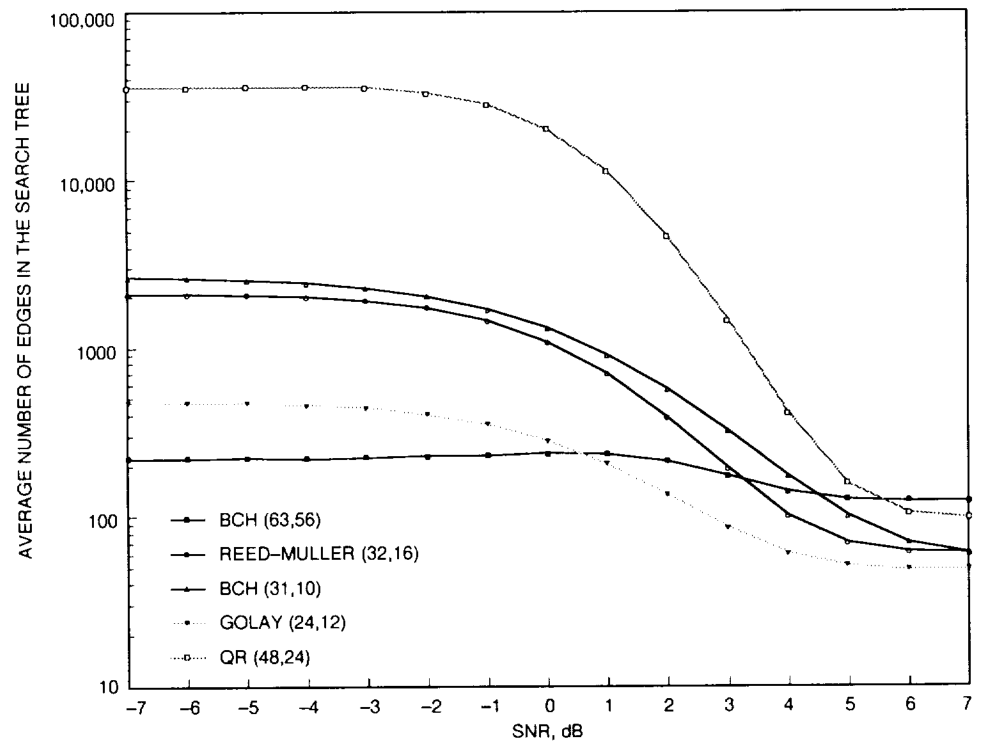


Fig. 8. The average number of edges in the search tree for several codes as a function of SNR.

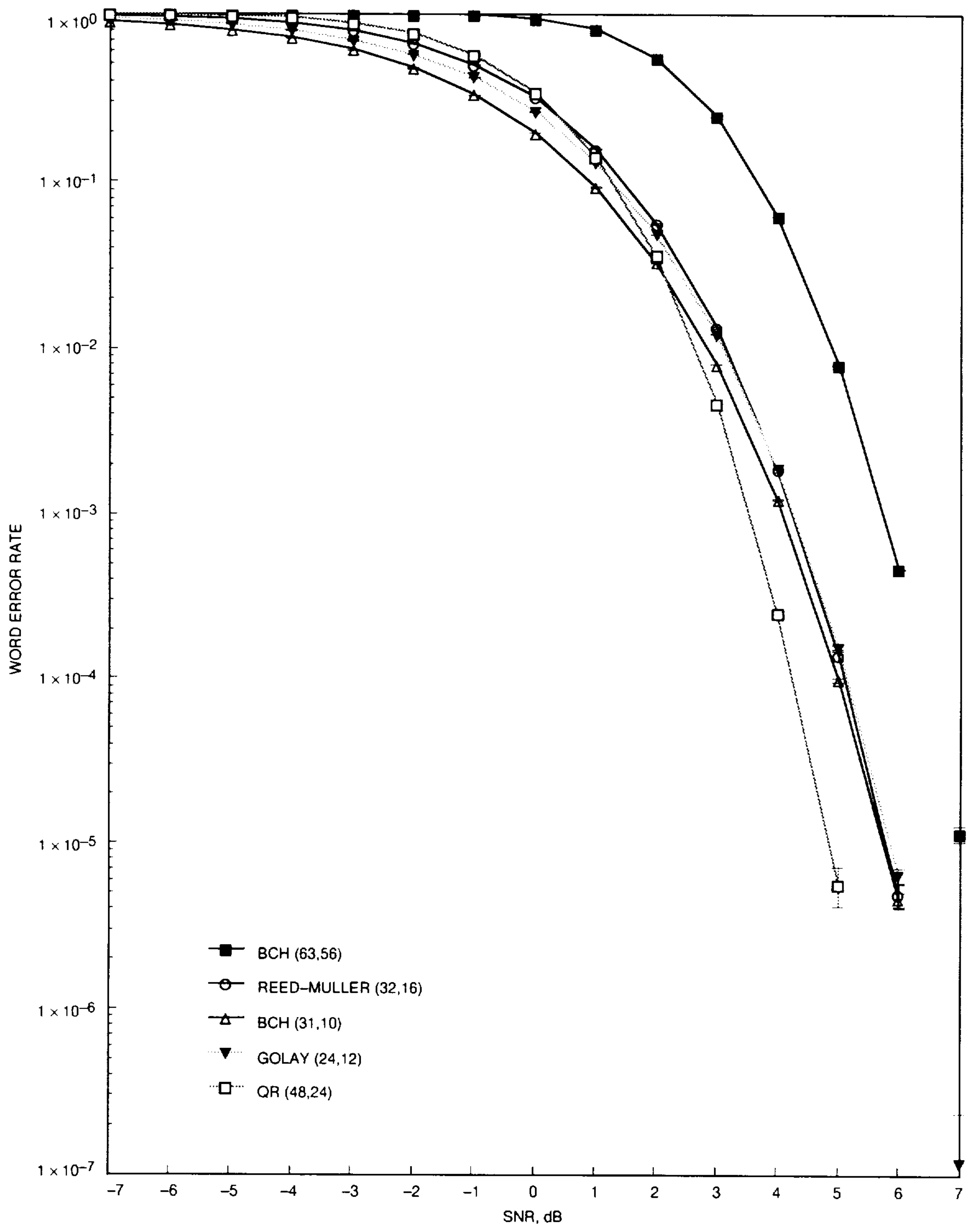


Fig. 9. Probability of word error versus SNR for the BCH (63,56), Reed–Muller (32,16), BCH (31,10), Golay (24,12), and quadratic residue (48,24) codes. The error bars are $\pm\sigma$, one standard deviation.