Using a Genetic Algorithm to Find Good Linear Error-Correcting Codes

Kelly M. McGuire and Roberta Evans Sabin
Computer Science Department
Loyola College
Baltimore, MD 21210
e-mail: kmcguire@loyola.edu, res@loyola.edu

Keywords: Genetic algorithm, linear error correcting codes

ABSTRACT

A genetic algorithm is used to search for linear binary codes with optimal minimum distance for a fixed length n and dimension k. Several modifications to the algorithm are compared to find an algorithm best suited to this application. The code is parallelized and run on a multi-processor and speedup determined.

INTRODUCTION

Error-correcting codes [4][5] are used in transmitting digital data in such devices as modems, satellite transmitters, and CD players. If we consider information to consist of packets of digital bits, error-correcting codes add redundant bits so that errors that occur in transmission or storage may be detected and corrected. A "good" code is one that keeps the ratio of redundancy bits to data bits low, yet allows the detection and correction of multiple errors. The search for optimal codes is computationally intensive as it frequently requires the generation of a very large number of binary codewords. Here, we examine the use of a genetic algorithm to find good codes and test a parallelized version of the algorithm. A genetic algorithm has previously been successfully applied in a search for very short, ternary (base 3) codes [1][8]. To our knowledge, no attempt has been made to apply a genetic algorithm to binary codes or to parallelize the algorithm for code detection.

THE CODING THEORY PROBLEM

We are seeking good linear binary error-correcting codes. Mathematically, a binary linear code is a vector space over $\mathbf{F}_2^n$ with dimension k. The code consists of 2^k codewords, each a linear combination of the k basis vectors; the basis vectors form a k x n generator matrix, G. The metric n is referred to as the length of the code and k is the code's dimension.

If we consider information to be a stream of bits, viewed as a sequence of discrete packets, each packet containing k bits, a single k-tuple of information may be encoded to an n-bit codeword by multiplying by the generator matrix, G. The efficiency of data transmission is measured by the ratio k/n.

For example, if $k = 2$, there are four possible sequences of information: 00, 01, 10, 11. By multiplying by $G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$, we may encode these 2-bit "infowords" to the 5-bit codewords 00000, 01011, 10101, and 11110. To transmit the two information bits 11, 11110 would be transmitted. The efficiency of this code is 2/5 = .4. In general, the extra bits added to each infoword to form a codeword allow for the detection and correction of one or more errors that may occur during transmission of the data. Error correction occurs in the decoding phase.

After being transmitted, the received binary n-tuple, which may have been damaged in transmission, is decoded by determining its "closest" codeword, *i.e.*, the codeword that differs from the received n-tuple in the fewest number of bits. If a single bit was corrupted ('0' to '1' or '1' to '0'), the correct infoword will result form the decoding process. For example, for the code above, if the codeword 11110 were corrupted and received as 11111, the closest of the four possible codewords, 11110, would be used and decoded to the correct infoword, 11.

The number of errors a code can correct depends upon the minimum distance of the code, d. Minimum distance of a code is the smallest distance between any two distinct codewords, where distance is the number of bits in which the codewords differ. It can be shown that the minimum distance of a binary code is equal to the smallest Hamming weight of any non-zero codeword in the code [4], where Hamming weight is the number of "1" bits in a codeword. Error-correcting capability is directly proportional to minimum distance: a code with minimum distance d can correct $\frac{1}{2}(d-1)$ errors. A linear code is often characterized with the (n, k, d) triple [5], representing its length, dimension, and minimum distance. The sample code above is a (5, 2, 3) code as the non-zero codewords have Hamming weight 3, 3, and 4, and the minimum distance of the code is 3. This code can correct one error.

Attempting to find efficient codes with high error-correcting capability is the goal of coding theory. In practical terms, this means attempting to optimize one of the (n, k, d) parameters in terms of the other two. Here, we search for codes that exceed the minimum distance d of the best known code of a fixed length n and dimension k.

A GENETIC ALGORITHM TO PRODUCE CODES

A genetic algorithm (GA) is an optimization procedure that mimics the behavior of genetic reproduction in biology [3]. A chromosome or "string" is a candidate for optimization. The GA begins with a random initial set or population of

Table 1: Crossover methods

Method Name	Description of crossover site	Example (n=4, k = 3)
CBET	between any of the k codewords that comprise a single chromosome	AAAABBBB|CCCC
CBIT	any of the $n \cdot k$ bits of the chromosome	AAAAB | BBBCCCC
CALL	at the same bit in each of the k codewords	A|AAAB|BBBC|CCC

chromosomes. The fitness of each chromosome is evaluated by a function. The GA seeks to optimize the value of this function over successive populations, thereby mimicking "survival of the fittest." A three step process is used to create each new generation. The first step, reproduction, selects two chromosomes to "mate" with the more fit chromosomes having greater probability of being selected. Then crossover combines the two chosen mates to form two children for the next generation. Finally, mutation may alter all or part of a chromosome: usually mutation occurs with a low probability. The newly formed generation is then used as the initial population and these processes are repeated. If we assume that fit parents produce fit children, over successive generations, the overall fitness of the population increases.

We used several versions of the GA to produce binary linear codes. In every case, we considered a chromosome to be a sequence of k codewords or sequences of bits each of length n. Thus a single chromosome, consisting of $k \cdot n$ bits, generates an entire binary, linear code: the k individual components of a chromosome may be thought of as the basis vectors of the code, or, alternatively, as the k rows of the k x n generating array of the code. A population consists of a number of these generators. It was hoped that the GA would produce successive generations of chromosomes that would exhibit increasing "fitness" over time.

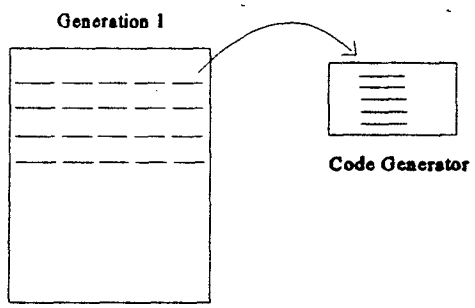


Figure 1

Population Size and Number of Generations

An initial population of 1024 chromosomes was randomly chosen with the requirement that the k codewords for each chromosome be linearly independent. No attempt was was made to ensure that the codes produced by the individual chromosomes were distinct[1]. Each successive generation consisted of 1024 chromosomes, a number which allowed the equal distribution of the population among various numbers of processors (see below). Twenty generations were produced.

[1] The probability that 1000 randomly chosen generators of (32,12) binary linear codes produce distinct codes exceeds .999.

Fitness and Selection Criteria

Pairs of chromosomes are selected for mating based on their "fitness" where an individual chromosome's fitness is the minimum distance of the code it produces. Determining the minimum distance was accomplished by generating the entire code. The total of all minimum distances of all 1024 codes generated by the current population was used to weight a "roulette wheel" [3] which was repeatedly spun to select pairs of chromosomes to be mated.

Crossover

The usual method of crossover is to select a pair of chromosomes, then select a position or site within the chromosomes, cut each member of the pair at the selected site, and paste the head of the first chromosome to the tail of the second and vice versa, producing two "child" chromosomes.

Figure 2: Crossover

In this application, we randomly chose a crossover site for each generation that was used for all pairs of chromosomes mated during the production of that generation. The three crossover methods used are summarized in Table 1.

Two methods were used to pair crossover fragments:

PNEW - fragments paired so that two "new" chromosomes were always produced. When used in conjunction with the CALL crossover method pairing was done in each constituent codeword, *e.g.*

```
A|AAAB|BBBC|CCC   yields   AaaaBbbbCccc
a|aaab|bbbc|ccc            aAAAbBBBcCCC
```

PFIT - fragments with a higher Hamming weight are paired. Here, we used the implicit assumption that a code generator's Hamming weight can be positively correlated with the minimum distance of the code it generates.In all cases, the probability of crossover was 1.0, *i.e.*, all selected pairs are crossed.

Mutation

Mutation is usually accomplished by randomly selecting a bit within a chromosome and randomly altering it. We tested altering an entire one of the k components of a chromosome, and, alternatively, altered a single bit within the sequence of $k \cdot n$ bits. Finding insignificant differences in behavior of the two methods, algorithms A1 through A6 (Table 2) implement the mutation of an entire component, randomly selected from among the k components, with a probability of 0.01.

Table 2: Algorithm Summary

Algorithm	Crossover Site	Pairing Method
A1	CBET	PNEW
A2	CBET	PFIT
A3	CBIT	PNEW
A4	CBIT	PFIT
A5	CALL	PNEW
A6	CALL	PFIT

Scaling

A problem encountered by some genetic algorithms is premature convergence, *i.e.*, converging to a less than optimal value in early generations. Fitness scaling is one way to overcome this problem [3]. In scaling, a new function is created to re-fit the fitness values of the population. The two criteria for this scaling function f are: 1) $f(avg) = avg$ and 2) $f(M) = c \cdot avg$ where c is the scaling factor and avg and M are the average fitness and maximum fitness of the pre-scaled population.

With fitness identified as the minimum distance of the generated code (see Fitness above), both linear and quadratic functions were developed and tested with scaling factors of 3, 5, and 7. Linear scaling was implemented with the function

$$Fitness_{new} = avg/(M - avg) \cdot [c(Fitness_{old} - avg) + M - Fitness_{old}].$$

Quadratic scaling was implemented with the function

$$Fitness_{new} = avg/(M^2 - avg^2) \cdot [(c-1) \cdot (Fitness_{old}^2 - avg^2)] + avg.$$

Parallel Nature of the Algorithm

The generation of a single linear code, a process needed to determine fitness, can be parallelized, since each codeword in a code is a unique linear combination of basis vectors [6]. We parallelized code generation when using a machine with vector architecture, the Cray C90. But the GA is intrinsically parallel in structure: the same procedure is performed independently for each member of the population. With the availability of p processors, we divided the work of generating the next generation into p equal parts. All processors use the same initial population. Each processor uses the same algorithm (reproduction, crossover, and mutation) to produce $1/p$ of the next generation, then each processor determines the fitness of each chromosome in its portion of the new population by generating an entire linear code. No parallelization was attempted in the generation of a single linear code. The results, code generators and fitness measures of each generator, are broadcast to all other processors, and the procedure is repeated for each generation (Figure 3).

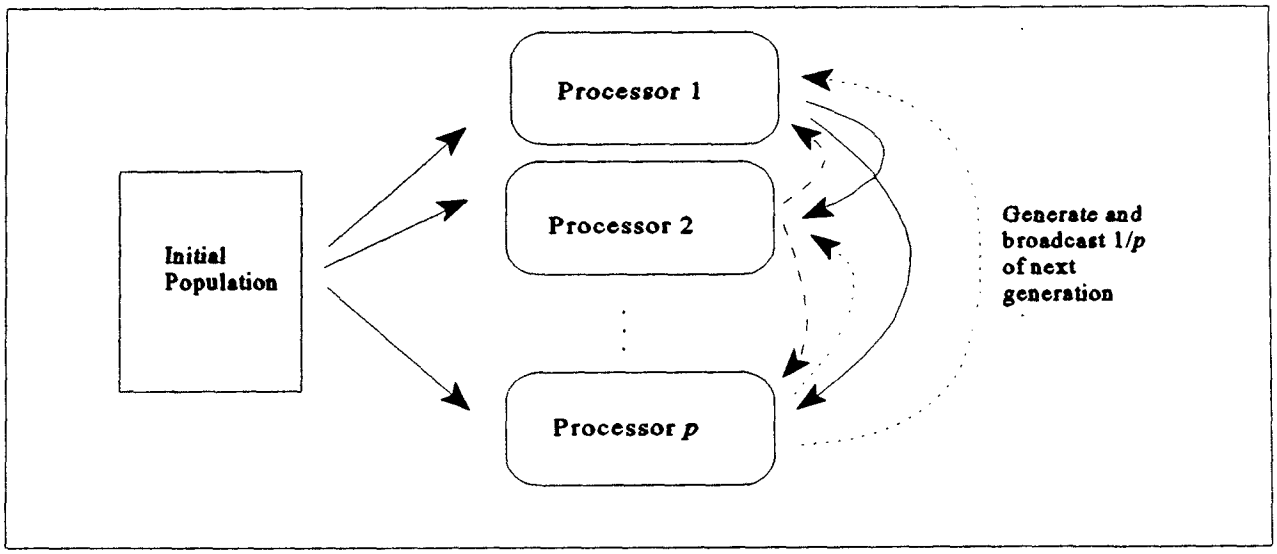


Figure 3: Parallelization of algorithm

EXPERIMENTAL SETTING

Software

All of the algorithms were implemented in ANSI C for test execution on a Unix workstation, then ported to the supercomputers, the Cray C90 and T3D. Chromosomes were represented as sequences of k unsigned integers and their binary representations utilized. In implementations on the Cray machines, IEOR (bitwise exclusive or), a predefined Cray function, was used to perform addition of binary codewords, needed in linear code generation. Another Cray extension, Popcnt, which returns the number of ones in the bit representation of its argument, was used to determine the weight of the codewords. Programs executed on the T3D utilized Parallel Virtual Machine (pvm) routines for communication among processors.

Hardware

The Cray Y-MP C90 at the Pittsburgh Supercomputing Center has 16 vector processing units, each of which is capable of performing a billion calculations per second with a peak aggregate speed of 16 Gflops. The capacity of the central memory on the C90 is 4 GBytes.

The Cray T3D at the Pittsburgh Supercomputing Center (PSC) is a scalable parallel supercomputer with 512 DEC Alpha processors and has a theoretical peak speed of 76.8 Gflops. The topology of the T3D is a three dimensional torus and the memory is logically shared and physically distributed with 64 MB per processor. Programs were run in batch mode on the PSC T3D, submitted from the C90.

Parallelization

Implicit vectorization was used on the C90 by compilation with the command

cc -h aggress,bl,inline report=fv file

A listing file verified that vectorization was accomplished.

On the T3D, a parallel version of the GA (as outlined above) was implemented. Each of p processors computed $1/p$ of the new population. The "master" processor (PE0) created an initial population and generated the linear code of each chromosome of that generation. It then sent the initial population along with the minimum distance of the code generated by each chromosome in that population to each of the other processors. Each processor then used the same algorithm for reproduction, crossover, and mutation to produce $1/p$ of the next generation and generated the entire linear code corresponding to each chromosome. The new population members and their measure of fitness, *i.e.*, the minimum distance of the linear code generated by the chromosome, were broadcast via the pvm_bcast function to all other processors. The procedure was repeated for each generation.

The performance of the GA was determined by the total fitness of the population. A weight spectrum illustrated the fitness of each member of the population, as well as the best code achieved. The effectiveness of parallelization was measured by the timing on the T3D. A timing function, rtclock, was called at the beginning and end of each program and the difference in times recorded.

RESULTS

As stated earlier, the aim of this work was to discover a method of uncovering new, "best" linear error-correcting codes constrained by the computationally intensive nature of code production which is $O(2^k)$ for a dimension k code. (In all algorithms presented here, the production of each generation of a length 32, dimension 10 code required in excess of $2^{10} \cdot 10 \cdot 1000$ integer operations.) We choose to investigate codes of length 32 and 64 as these lengths utilized all bits in the internal representations of C unsigned integers. Codes with dimensions 5, 10, 12, and 14 were produced. After initial trials, a population size of 1024 was chosen with 20 generations studied. We were disappointed to find no new "best" codes (Table 3); however we were heartened by several patterns that emerged. Current minimum distances of best codes are found in [2].

Table 3: Summary of codes found

(n,k)	Highest d_{min} known	Highest d_{min} possible	Highest d_{min} found by GA
(32,5)	16	16	14
(32,10)	12	12	9
(32,12)	10	10	7
(32,14)	8	10	7
(64,12)	34	37	21
(64,14)	32	36	19

Besides examining the best minimum distance achieved in each generation, we measured "overall fitness" as the sum of the minimum distances of the linear codes corresponding to the chromosomes of a generation. The algorithms of Table 2 were each run with 64 processors on the T3D. Of the six algorithms, A2 and A4, utilizing PFIT pairing and crossover at only one bit per chromosome consistently produced the most fit twentieth generation. Figure 4 displays a representative set of data gathered in generating (32,12) codes on the T3D utilizing 16 processors.

We also examined the distributions of individual chromosomes' fitness, *i.e.*, numbers of codes with each minimum distance generated by the chromosomes of a generation. Figure 5 displays A2's production of increasingly more fit generations of (32,12) codes.

Both linear and quadratic scaling significantly improved the overall fitness of the population. Experimentation showed that quadratic scaling with a factor of 7 was most advantageous. Figure 6 displays the effect of quadratic scaling with scaling factor = 7 on algorithm A2's generation of (32,12) codes.

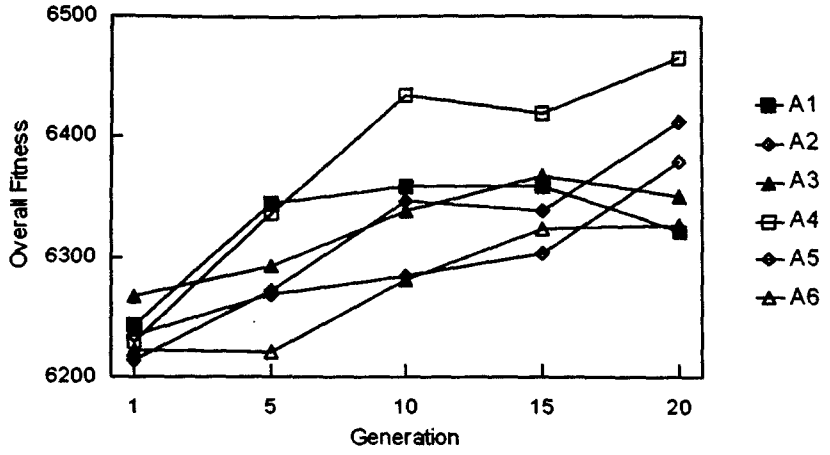
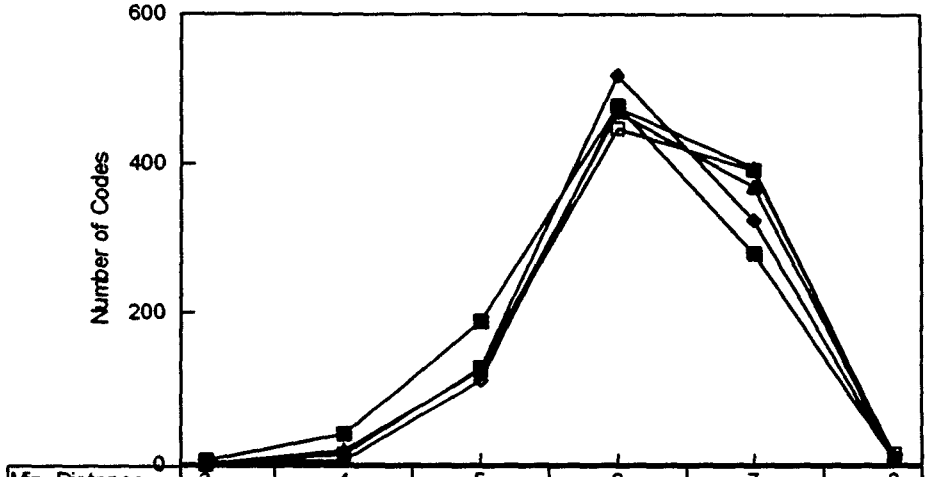


Figure 4: Overall fitness of generations of (32,12) codes



Min. Distance	3	4	5	6	7	8
■ Initial Pop.	6	41	189	476	280	8
◆ Gen. 5	2	16	131	516	326	9
▲ Gen. 10	2	19	128	469	370	12
⊟ Gen. 15	1	16	131	445	392	15
◆ Gen. 20	3	6	114	473	393	11

Figure 5: Numbers of codes with each minimum distance in successive generations of (32,12) codes using algorithm A2

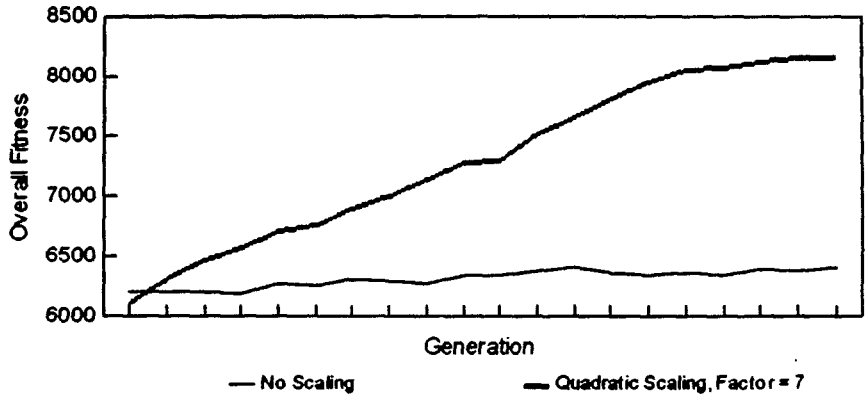


Figure 6: Effect of quadratic scaling on (32,12) codes

Not surprisingly, the parallel version of the algorithm did not display "perfect" speedup. The cost of interprocessor communication and data broadcasting as well as the uniprocessor nature of the generation of the initial population (by PE0) adversely effected speedup. Experimental results (Figure 7) were in keeping with expectations.

DISCUSSION

We found that a GA did produce successively more fit generations of code generators. But in all cases we found the algorithm to hit a "wall" at a particular minimum distance, below the best known minimum distance for the given length and dimension. This behavior may be attributable to one or more of the following factors:

- Minimum distance may be too coarse a measure of fitness. A more refined measure might involve the number of codewords displaying the minimum distance, awarding to those codes with fewest lightweight codewords a higher fitness. Alternatively, the entire distribution of weights in the code generated by the chromosome could be involved, rewarding a chromosome in proportion to the number of one bits in the code generated by the chromosome.
- The number of codes with "good" minimum distances is extremely small and thus very difficult to find. Given an environment with sufficient computing resources, it may be beneficial to simply replicate the experiment with larger and more generations or simply repeat the experiment many times over.
- Linear codes without any additional algebraic structure may not be the best source of new best codes. Many existing best codes exhibit an additional algebraic structure, *e.g.*, BCH codes are cyclic in structure [5].

Future work may be directed at compensating for these deficiencies. In particular, it may be profitable to examine quasi-cyclic codes [7] which have the attractive property that generators may be cut and pasted at certain sites (as in crossover) and yet retain their algebraic structure.

REFERENCES

[1] Aarts, Emile H.L. (1993) Local Search in Coding Theory, *Discrete Mathematics*, **106/107**, pp. 11-18.

[2] Brouwer, A.E. a table of best known linear codes maintained at Eindhoven Univ. of Tech., Eindhoven, The Netherland, accessible at http://www.cwi.nl/htbin/aeb/lincodbd/2/24/8.

[3] Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA.

[4] Hill, Raymond (1986). *A First Course in Coding Theory*, Clarendon Press, Oxford.

[5] MacWilliams, F.J, and N.J.A. Sloane (1977). *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam.

[6] Sabin, R.E. (1995). On Generating Large Binary Linear Error-Correcting Codes, *Journal of Combinatorial Mathematics and Combinatorial Computing*, **19**, pp. 245-258.

[7] Sabin, R.E. (1994) On Row-Cyclic Codes with Algebraic Structure, *Designs, Codes, and Cryptography*, **4**, pp. 145-155.

[8] Vaessens, R.J.M., E.H.L. Aarts, and J.H. vanLint (1993) Genetic algorithms in coding theory--a table for $A_3(n,d)$. *Discrete Applied Mathematics*, **45**, pp. 71-87.

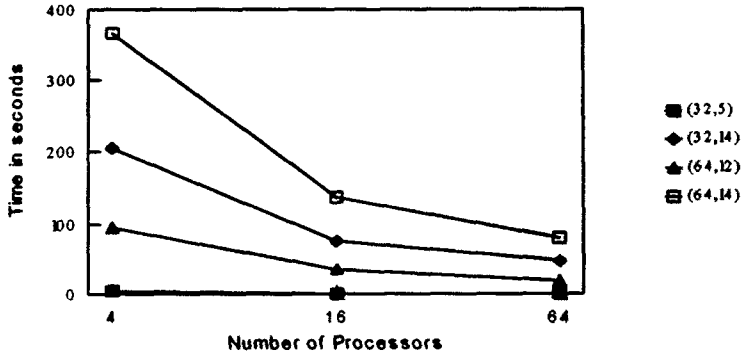


Figure 7: Speedup due to parallelization