Evolutionary Algorithms for Optimal Error-Correcting Codes

Wolfgang Haas
Brock University
500 Glenridge Avenue
St. Catharines, Ontario Canada
905 688 5550
haas.wolfgang@gmail.com

Sheridan Houghten
Brock University
500 Glenridge Avenue
St. Catharines, Ontario Canada
905 688 5550
houghten@brocku.ca

ABSTRACT
The maximum possible number of codewords in a q-ary code of length n and minimum distance d is denoted $A_q(n,d)$. It is a fundamental problem in coding theory to determine this value for given parameters q, n and d. Codes that attain the maximum are said to be optimal. Unfortunately, for many different values of these parameters, the maximum number of codewords is currently unknown: instead we have a known upper bound and a known lower bound for this value.

In this paper, we investigate the use of different evolutionary algorithms for improving lower bounds for given parameters. We relate this problem to the well-known Maximum Clique Problem. We compare the performance of the evolutionary algorithms to Hill Climbing, Beam Search, Simulated Annealing, and greedy methods. We found that the GAs outperformed all other algorithms in general; furthermore, the difference in performance became more significant when considering harder test cases.

Categories and Subject Descriptors
G.2.1 [**Combinatorics**]: *combinatorial algorithms.*

G.2.2 [**Graph Theory**]: *graph algorithms.*

I.2.8 [**Problem Solving, Control Methods and Search**]: *graph and tree search strategies, heuristic methods.*

General Terms
Algorithms, Performance, Design, Theory.

Keywords
Evolutionary Algorithms, Simulated Annealing, Optimal Codes, Maximum Clique Problem.

1. INTRODUCTION
In this section, we briefly introduce the basic concepts from coding theory, only to the extent required to understand our research. There are several excellent books on coding theory for those readers interested in learning more about the subject; these include, for example [9] and [12].

When data is transmitted or stored, errors can occur for a variety of reasons such as noise in the transmission or dirt on the storage media. In the case of binary data, this has the effect of changing the value in a single bit, from 0 to 1 or vice-versa. It is possible to detect and correct such errors if we store data according to a certain format, namely, by using error-correcting codes.

An $(n,M,d)_q$ code is a set of M codewords, each of length n, and in which each symbol is chosen from a possible alphabet of q symbols. Furthermore, if we compare any pair of codewords, they must differ in at least d positions; the value d is referred to as the *minimum distance* of the code and this determines the number of errors that may be corrected. When q=2, the set of possible symbols is {0,1} and the code is called *binary*. Codes with q=3 are called *ternary*; in the general case, the code is called *q-ary*.

There are several (often conflicting) desirable properties in a code. One such property is the number of codewords, as this determines the number of different pieces of information that may be represented by the code. In our research, we consider the problem of finding $(n,M,d)_q$ codes where n, d and q are fixed, and with a large value of M. While the maximum value of M is exactly known for some parameter sets, for many others, it is known only to be within a given range; we consider both types of parameter sets. Tables containing this information for many parameters may be found at [2] for the binary case and [3] for the ternary case.

In this paper we compare several techniques for finding error-correcting codes that have a larger number of codewords than the best currently known for given parameters.

We have several goals in this research. First, we wish to evaluate which techniques are most suitable for finding codes that are optimal. For those parameter sets for which the exact number of codewords in an optimal code is known, we wish to find codes that are optimal, or at least near-optimal. For those codes for which the maximum number of codewords is known only to be within a given range, we would like to eventually improve upon the lower bound for this value.

	0000	0011	1010	1011	1110	1111
0000	0	1	1	1	1	1
0011	1	0	1	0	1	1
0011	1	1	0	0	0	1
1010	1	0	0	0	1	0
1011	1	1	0	1	0	0
1110	1	1	1	0	0	0

Figure 1. Example Compatibility Matrix

2. PROBLEM SET-UP

In this section we discuss several issues relating to the problem at hand. Understanding these issues enables us to define a good strategy for attacking the problem.

2.1 Precomputation

Every time the program looks at a possible solution, it needs to calculate the distance between each pair of codewords, in order to determine the minimum distance of the code. It is beneficial to pre-compute the distances between all possible codewords to save computation during the run. Note that one can always assume the existence of the all-zero vector in the code since any code not containing the all-zero vector is equivalent to one that does contain it, according to the definition of equivalence of codes.

Hence we generate all possible codewords of length n and distance at least d from the all-zero vector. We store the results in a *compatibility matrix*, a square matrix in which entry $(i,j) = 1$ if codeword i and codeword j meet the minimum distance requirement, and 0 otherwise. It follows from the definition that this matrix is symmetric. Since the program has to find a set of codewords such that the distance between any two is at least d, the corresponding entries in the compatibility matrix must all be equal to 1.

2.2 Maximum Clique Problem

In fact, the problem we are considering is equivalent to the well-known problem of finding the maximum clique in a graph.

A graph $G = (V,E)$ is defined by a set of vertices V and a set of edges E. A clique C is a subset of V such that each vertex in C is connected to all other vertices in C by an edge. The maximum clique problem asks for the largest subset MC of V such that MC is a clique.

To convert an instance of the coding theory problem into an instance of the maximum clique problem, we label the vertices with the possible codewords, and connect two vertices by an edge if they meet the minimum distance requirement.

Example: Suppose that we wish to find the maximum number of codewords in a binary code with length 4 and minimum distance 2. In general, we could consider all binary vectors of length 4. However, in the interests of keeping this example manageable, we shall further suppose that the following vectors are the only possible candidate codewords: {0000, 0011, 1010, 1011, 1110, 1111}.

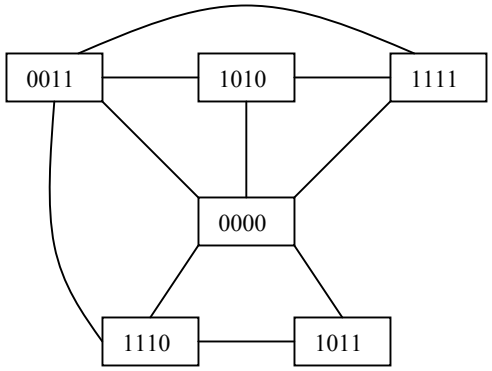

Figure 2. Example Graph for Maximum Clique Problem

This results in the compatibility matrix given in Fig.1, which in turn corresponds to the adjacency matrix of the graph given in Fig.2.

The size of the maximum clique is 4, and the vertices in this maximum clique should be {0000,0011,1010,1111}. Therefore the maximum number of codewords is 4, and the codewords in this optimal code should be {0000,0011,1010,1111}.

2.3 Relationships between Sets of Parameters

Several relationships exist between sets of parameters. We combine the following to reduce the search space for binary codes.

Let C be a binary $(n,M,2r-1)$ code. By adding an overall parity check, we get an $(n+1,M,2r)$ code, thus $A_2(n,2r-1) \le A_2(n+1,2r)$. Note: An overall parity check adds a bit at the end of every codeword. This bit is a 1 if the remaining part of the codeword has an odd number of 1's, and 0 otherwise. As a consequence in the resulting code, every codeword has an even number of 1's and the distance between every pair of codewords is even.

Let C be a binary $(n+1,M,2r)$ code. By deleting any single coordinate, we get an (n,M,d) code with $d \ge 2r-1$. Therefore we have $A_2(n,2r-1) \ge A_2(n+1,2r)$.

Because $A_2(n,2r-1) \le A_2(n+1,2r)$ and $A_2(n,2r-1) \ge A_2(n+1,2r)$, they must be equal. Since a code of length n and minimum distance $2r$-1 has fewer possible codewords than a code of length n+1 and minimum distance $2r$, it reduces the search space while still guaranteeing the same optimal value. For further information, see [9], p.43.

Example: $A_2(10,4) = A_2(9,3) = 40$. There are 848 vectors of length 10 that are at distance at least 4 from the all-zero vector and that thus are candidates for codewords in a (10,40,4) code. Meanwhile, there are only 466 codewords of length 9 that are at distance at least 3 from the all-zero vector. In each case, if attempting to find an optimal code, we need to find 40 compatible codewords; clearly, this would be much simpler in the smaller set.

3. RELATED WORK

In this section we briefly review several papers that consider the use of evolutionary algorithms to solve the Maximum Clique Problem. Many of the algorithms have been run on DIMACS graphs in order to allow for a good comparison between the

different methods. Since finding optimal error-correcting codes can be reduced to finding a maximum clique, all these strategies are also applicable to our problem. However, when finding error-correcting codes we deal with the problem at a much larger scale. For example, in considering binary error-correcting codes, we see that the smallest parameter values for which the maximum number of codewords is currently unknown are {n=17, d=4}. There are 130,238 possible codewords for this case; in comparison, the largest DIMACS graph has approximately 6,000 vertices.

Bui and Eppley [4] created a hybrid genetic algorithm that uses a local optimizer during each generation. Their fitness value was based on the number of vertices selected by the chromosome and a measure of how close this set was to being a clique. At the time of publication (1995) their genetic algorithm tied the best known results.

Murthy and Parthasarthy [11] used a similar approach without any local optimizing strategy. However, their test cases were small graphs. Since our graphs are very large, it is difficult to say whether this approach will perform well.

Carter and Park [5] implemented two different genetic algorithms for the maximum clique problem: a simple genetic algorithm with a binary string chromosome and a multi-phased annealed approach. The latter very slightly outperformed the simple one and had a tendency of getting stuck in local optima. Later, in [6], they reported that Simulated Annealing worked much better than genetic algorithms, because crossover didn't work for the problem at all. They found that running the algorithm without crossover gave better results.

Jagota and Sanchis [8] used Neural Nets to find the maximum clique of a graph. To have a higher chance of getting larger cliques, they pre-sorted all vertices by their degree and found that doing this actually resulted in worse performance.

Marchiori [10]] obtained very good results by allowing non-cliques to remain in the population. During each generation, local optimization techniques extracted a clique from the selected set of vertices and then tried to maximize the size by adding more vertices.

Finally, Ashlock, Guo and Qui [1] use a genetic algorithm to develop an error-correcting code for DNA constructs using the edit distance as a similarity measure. Each chromosome contained 3 selected codewords and Conway's Lexicode Algorithm (see section 4.4) was applied until the maximal number of codewords was reached.

4. THE ALGORITHMS

In our research we compare the performance of Hill Climbing, Beam Search, Simulated Annealing, greedy methods and several varieties of genetic algorithms. In this section we briefly summarize the parameters used for each algorithm, as well as other technical information.

Hill Climbing, Beam Search and Simulated Annealing all use the same representation for a candidate solution: a sequence of integers, where the first integer represents the first codeword to be chosen, the second integer represents the second codeword to be chosen, etc. However, in order to keep the search space at a minimum, we have to make sure that all candidate solutions are actually cliques. We achieve this by treating all integers as offsets.

4.1 Hill Climbing

We evaluate the performance of Hill climbing using 100 iterations.

4.2 Beam Search

We evaluate the performance of Beam Search with a Beam Size of 5 and using 100 iterations.

4.3 Simulated Annealing

Simulated annealing has been used very successfully in many different types of combinatorial optimization problems and should perform reasonably well for finding optimal error-correcting codes. We evaluate the performance of Simulated Annealing using the parameters given in Table 1.

Table 1. Parameters for Simulated Annealing

Initial Temperature c_0	2
Number of Transitions for each value of c	100
Rule for changing c	$c_{k+1} = c_k * 0.8$
Termination Criterion	No Change of Best for 1 Iteration

4.4 Conway's Lexicode Algorithm

Conway's Lexicode algorithm, described in [7], is a greedy algorithm that can be used for creating an error-correcting code with given minimum distance. We first sort the candidate codewords lexicographically and initialize an empty set *C* of codewords. We then consider, in turn, each candidate in sorted order, and add it to *C* if the distance to all codewords already in *C* is at least *d.*

It should be noted that in the binary case, a linear code is generated when the words are considered in lexicographic order. For many parameter sets, the best-known code is linear; hence by using this algorithm we may reach the lower bound on the maximum number of codewords, but we will be unable to improve upon it.

4.5 Randomized Greedy Algorithm

This is a variation of Conway's Lexicode Algorithm, in which the codewords are considered in a random order. By doing so, we may find non-linear codes and thus improve upon the lower bound. This approach is used in [4] to initialize the population.

4.6 Genetic Algorithms

We evaluate the performance of several varieties of genetic algorithms. We use the parameters summarized in Table 2 for all of these varieties.

4.6.1 Indirect

The chromosome representation of this approach is the same as the one used for Hill Climbing, Beam Search and Simulated

Annealing. The chromosome size must be large enough to be able to represent the largest set of codewords possible. Consequently, the number of integers in the chromosome equals the upper bound of the particular code. As a result, the execution speed mostly depends on the currently known upper bound and this algorithm might not be feasible for a set of parameters when the maximal number of codewords of the corresponding error-correcting code is very large.

Note that while a direct representation is possible, we chose not to use it since applying either crossover or mutation almost always creates "illegal" chromosomes that must then be corrected. As a result, the most important aspect of the GA is in fact the method chosen to correct illegal chromosomes. We decided to choose a representation that would not allow this to happen, to ensure that we are focusing on the GAs themselves.

4.6.2 Indirect Seed – Lexicode Finish

This approach combines the indirect chromosome representation with Conway's Lexicode algorithm as seen in [1]. The chromosome representation is the same as in the indirect approach, however, its size is limited by the parameter *seed_size*. Any compatible codewords that have not been selected by the chromosome itself will be chosen by Conway's algorithm. The advantage of this approach is that it should be possible to attack even very large codes, since most of the codewords are selected by a greedy algorithm.

4.6.3 Lexicode Seed – Indirect Finish

This strategy also combines the indirect chromosome representation with Conway's Lexicode algorithm. The main difference to the last approach is that it works in the opposite direction: Conway's algorithm is used to pre-select a set of codewords and the chromosome defines the codewords selected after this. To make this work, we must stop Conway's algorithm before it has selected all the codewords, and hence the parameter *Conway_limit* specifies when this is done.

4.6.4 Indirect Seed – Randomized Greedy Finish

This approach combines the indirect chromosome representation with the randomized greedy algorithm (see section 4.5) and works very much like 4.6.2. The only difference is that the randomized greedy technique is used to select the remaining codewords rather than Conway's algorithm. Similarly, the chromosome size is limited by the parameter *seed_size* as in 4.6.2.

Table 2. Parameters for Genetic Algorithms

Population Size	500
Number of Generations	100
Number of Runs	10
Selection Strategy	Tournament of Size 3
Crossover Probability	85%
Mutation Probability	15%

5. THE TEST CASES

One of the advantages of translating the problem of finding an optimal error-correcting code into the maximum clique problem is that it works the same way regardless of alphabet size q. The reason is that the compatibility matrix is always a binary matrix.

We evaluated the performance of the algorithms on different parameter sets for both binary and ternary codes. It is important to note that the different test cases might generate graphs with very different properties. For example, even though the number of candidate codewords (i.e. vertices) might be similar, the number compatible with each other (i.e. edges) could be very different. Consequently, the performance of the algorithms may vary across test cases.

We summarize the binary test cases in Table 3. The first two cases are relatively small and thus most of our algorithms should perform fairly well. In comparison, the other two cases are difficult, and in fact are cases for which the exact number of codewords in an optimal code is unknown. These latter two cases will thus provide an interesting comparison of which algorithms are most useful when trying to improve on the best codes currently known.

Table 3. Binary Test Cases

n	d	#Codewords in Optimal Code	#Candidate Codewords for (n,d)	#Candidate Codewords for (n-1,d-1)
12	6	24	2 510	1 486
13	6	32	5 812	3 302
17	6	256-340	121 670	63 019
17	4	2720-3276	130 238	65 399

We summarize the ternary test cases in Table 4. The test cases were chosen with the same goal as for the binary case: namely, to provide a comparison of the performance of the algorithms on cases that range in difficulty from easy to hard. In particular, the first test case should be considered easy while the last two should be considered very hard.

Table 4. Ternary Test Cases

n	d	#Codewords in Optimal Code	#Candidate Codewords for (n,d)
5	3	18	192
6	3	38	656
7	3	99-111	2 088
8	3	243-333	6 432

Table 5. Summary of Results for all Algorithms

		$A_2(12,6)$	$A_2(13,6)$	$A_2(17,6)$	$A_2(17,4)$	$A_3(5,3)$	$A_3(6,3)$	$A_3(7,3)$	$A_3(8,3)$
Best known		**24**	**32**	**256**	**2720**	**18**	**38**	**99**	**243**
Hill climbing	Best	16(1/10)	22(1/10)	134(1/10)	1721(1/10)	14(9/10)	33(2/10)	76(1/10)	183(1/10)
	Ave.	14.0	20.7	130.6	1706.4	13.9	31.6	74.0	180.7
Beam search	Best	19(1/10)	23(1/10)	136(3/10)	1728(1/10)	18(2/10)	34(2/10)	78(1/10)	186(1/10)
	Ave.	15.7	21.6	133.8	1716.0	15.0	33.0	76.1	182.8
Simulated Annealing	Best	16(6/10)	22(7/10)	136(1/10)	1729(1/10)	18(3/10)	34(1/10)	79(1/10)	185(1/10)
	Ave.	15.5	21.7	133.4	1719.3	15.8	32.7	77.3	183.4
Conway Lexicode	Best	16	16	256	2048	15	32	76	200
Randomized Greedy	Best	13(2/10)	20(1/10)	129(1/10)	1693(1/10)	13(2/10)	30(2/10)	71(1/10)	176(1/10)
	Ave.	11.8	18.6	124.9	1678.5	11.9	28.4	68.4	170.4
GA: Indirect	Best	24(5/10)	26(1/10)	140(1/10)	N/A	18(10/10)	36(1/10)	83(1/10)	194(1/10)
	Ave.	21.1	23.6	137.3	N/A	18.0	34.6	80.7	189.8
GA: Indirect Seed - Lexicode Finish	Best	24(10/10)	32 (2/10)	256(7/10)	2238(1/5)	18(10/10)	36(10/10)	88(2/10)	219(1/10)
	Ave.	24.0	28.8	253.9	2214.8	18.0	36.0	87.2	216.1
GA: Lexicode Seed – Indirect Finish	Best	19(7/10)	26(10/10)	256(10/10)	N/A	15(10/10)	36(9/10)	84(2/10)	206(3/10)
	Ave.	18.1	26.0	256.0	N/A	15.0	35.9	83.0	205.0
GA: Indirect Seed – Randomized Greedy Finish	Best	24(8/10)	24(4/10)	139(1/10)	N/A	18(9/10)	34(5/10)	80(2/10)	192(1/10)
	Ave.	23.0	23.4	133.9	N/A	17.7	33.5	78.8	188.3

6. RESULTS

The results for all of the above test cases, and for all of the described algorithms, are summarized in Table 5. In this table, the shaded entries highlight the best overall results found for each test case. Furthermore, the numbers in brackets identify the number of times a best result was obtained, out of the total number of runs. Thus, for example, we see that the Indirect GA obtained a best result of 24 for $A_2(12,6)$, five times out of ten runs; furthermore, this was one of three algorithms that obtained the best overall result for that case.

Notice that the genetic algorithms outperformed all of the other algorithms in general. This is particularly noticeable when we consider the difficult test cases, in which the Indirect Seed – Lexicode Finish GA significantly outperformed all other algorithms. It is interesting to note that Simulated Annealing (often considered by many to be the most suitable choice for combinatorial problems of this type) only achieved the best overall result in the easiest case.

Since this strategy requires a parameter *seed_size* (see section 4.6.2), it is interesting to compare the performance of this strategy with different values for this parameter.

From the easy test cases we deduced that the smaller the value of this parameter, the better the algorithm performed. Even for the hardest case, $A_2(17,4)$, where 65,399 codewords result in a currently best known code with 2,720 codewords, values between 6 and 9 worked best. These numbers are very small compared to the large set of codewords and as a consequence most of the words are actually selected by the greedy algorithm.

At this point we do not know whether this limits the search space in such a way that this approach cannot actually get to the global maximum for hard test cases. It is possible that larger values for this parameter actually perform better when running

this algorithm for a very large number of generations, because the search space is not as restricted.

Since $A_2(17,4)$ is a very difficult case with a very large search space, it is computationally very expensive to run the GAs. As a result, we chose to run only the most promising GA, namely the Indirect Seed – Lexicode Finish GA.

7. CONCLUSION AND FURTHER WORK

Although we have not been able to find an error-correcting code that contains more codewords than the currently best known, we have been successful in showing that genetic algorithms can be applied to this particular problem. In fact, when you consider all the test cases we have experimented with, our genetic algorithms performed better than all of the three standard search techniques.

Since we have only tied current best known codes, our goal remains to find an error-correcting code with a larger set of codewords than the currently best known. It is possible that one of the algorithms described in this project is already able to do that for certain parameters values. It would be interesting to see how the different GAs perform on different parameter values.

We decided to use an indirect chromosome representation for all our genetic algorithms, because this guarantees that all our chromosomes represent cliques. Most of the papers described in section 3 use a binary representation and apply different techniques to handle chromosomes that do not form a clique. A different idea to deal with this issue is to create a multi-objective genetic algorithm, where one objective is to have a large set of vertices, while the other objective is for that set to be a clique. At the end of the run we would most likely get a lot of candidate solutions which are not cliques, but since the vertices in these solutions are just a small subset of all possible codewords, it should be relatively easy to extract the maximum cliques.

8. ACKNOWLEDGMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, as well as a Brock Undergraduate Student Research Award.

The authors gratefully acknowledge the valuable comments and assistance provided by Prof. Dan Ashlock of the University of Guelph as well as by Prof. Brian Ross of Brock University.

9. REFERENCES

[1] Ashlock, D., Guo, L. and Qui, F., Greedy Closure Genetic Algorithms, In *Proceedings of the 2002 Congress on Evolutionary Computation*, 1296-1301.

[2] Brouwer, A.E., Small binary codes: Table of general binary codes, web-site at: http://www.win.tue.nl/~aeb/codes/binary-1.html

[3] Brouwer, A.E., Small binary codes: Table of general ternary codes, web-site at: http://www.win.tue.nl/~aeb/codes/ternary-1.html

[4] Bui. T. and Eppley, P., A Hybrid Genetic Algorithm for the Maximum Clique Problem, In *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA6)*, July 1995, 478-484

[5] Carter, R. and Park, K., How good are genetic algorithms at finding large cliques: an experimental study, Boston University Technical Report BU-CS-93-015, 1993.

[6] Carter, R. and Park, K., On the Effectiveness of Genetic Search in Combinatorial Optimization, In *Proceedings of the 10th ACM Symposium on Applied Computing,*1995, 329-336.

[7] Conway, J.H. and Sloane, N.J.A., Lexicographic codes: Error correcting codes from game theory, *IEEE Transactions on Information Theory*, Vol. IT-32, No.3 (1986), 337-348.

[8] Jagota, A. and Sanchis, L., Adaptive, Restart, Randomized Greedy Heuristics for Maximum Clique, *Journal of Heuristics* 7 (2001), 565-585.

[9] MacWilliams, F.J. and Sloane, N.J.A., *The Theory of Error-Correcting Codes*, North-Holland Publishing Company, Amsterdam, 1977.

[10] Marchiori, E., A Simple Heuristic Based Genetic Algorithm for the Maximum Clique Problem, In *Proceedings of the 1998 ACM Symposium on Applied Computing*, 366-373.

[11] Murthy, A.S. and Parthasarathy, G. and Sastry, V.U.K., Clique Finding – A Genetic Approach, In *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, 1994, 18-21.

[12] Pless, V. *Introduction to the Theory of Error-Correcting Codes*, 3rd edition, John Wiley & Sons, New York, 1998.