Brock University

Department of Computer Science

Evolutionary Approaches to the Generation of Optimal Error Correcting Codes

D. McCarney, S. Houghten, B.J. Ross
Technical Report # CS-12-02
March 2012

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

Evolutionary Approaches to the Generation of Optimal Error Correcting Codes

Daniel E. McCarney
Carleton University
School of Computer Science
5302 Herzberg Bldg.
1125 Colonel By Drive
Ottawa, Ontario
K1S 5B6, Canada
dmccarney@ccsl.carleton.ca

Sheridan Houghten
Brock University
Dept. of Computer Science
500 Glenridge Ave.
St. Catharines, Ontario
L2S 3A1, Canada
shoughten@brocku.ca

Brian J. Ross
Brock University
Dept. of Computer Science
500 Glenridge Ave.
St. Catharines, Ontario
L2S 3A1, Canada
bross@brocku.ca

ABSTRACT

Error-correcting codes allow for reliable transmission of data over mediums subject to interference. They guarantee detection and recovery from a level of transmission corruption. Larger error-correcting codes increase the amount of error tolerable, which improves system reliability and performance. However, discovering optimal error-correcting codes for different size specifications is equivalent to the NP-Hard problem of determining maximum cliques of a graph.

In this research, three different binary error correcting code problems are considered. Both genetic algorithm and genetic programming are examined for generating optimal error correcting codes for these problems. A new chromosome representation of the GA system is examined, which shows some benefits in certain conditions. The use of GP is novel in this problem domain, and in combination with the Baldwin effect, it is shown to be a promising new approach for code discovery.

1. INTRODUCTION

Error correcting codes are mathematical constructs from the field of coding theory that offer communication error detection and correction. Employing error correcting codes allows for reliable transmission of data subjected to interference. The amount of information that can be expressed, the overhead involved, and the maximum tolerance for error are theoretically proven, and in many cases can be shown to be optimal. A q-ary error correcting code is written $a(n, d)_q$, where q represents the radix, n the length of each codeword and d the error threshold, where $(d-1)/2$ symbols can be detected and corrected. In a q-ary code, faulty data can be repaired to an uncorrupted state by means of a correction process. The ability of an error correcting code to perform this restoration independent of re-transmission helps to minimize overhead while maintaining confidence in accuracy.

The search for larger and more robust codes is computationally difficult. This search is justified, however, as larger codeword sets permit an increased tolerance to error, and improved correction capabilities. The massive number of potential codewords in a large-sized error correcting code requires an exponential number of comparisons in order to maintain the distance requirement of the code while expanding the number of codewords available for use. The search for an optimal code is equivalent to the NP-Hard maximum clique problem.

In this paper, different evolutionary search techniques are explored to test their efficacy in searching for optimal codes of increasing difficulty. In particular, genetic algorithms (GA) and genetic programming(GP) are both investigated. Design decisions related to representations, program languages, tree traversal order, and evolution parameters are explored. The intension is to determine effective search techniques for three target codes under investigation.

It is not a goal of this paper to discover new optimal codes for large, difficult code sets. The search for new codes typically requires CPU-months of dedicated search. It would be premature and distractive to consider new code discovery here, before an understanding of the GA and GP representations are understood in detail. Rather, all our experiments explore codes with known optima. Thorough analyses were done on GA and GP configurations, which required substantial computational time to accomplish. New insights into the effectiveness of GA and GP representations for code discovery will be useful in the future, when computationally-intensive searches for new codes are desired.

Section 2 surveys relevant background literature. Section 3 introduces necessary concepts in coding theory. Experiment designs are described in Section 4, and results are presented and discussed in Section 5. Section 6 summarizes main contributions of this research.

2. LITERATURE SURVEY

Haas and Houghten[4] compare several traditional search algorithms with evolutionary algorithms in the search for improved lower bounds for optimal error correcting codes. They use an indexed representation (Section 4.5.1), as well as Conway's Lexicode algorithm (Section 4.4) to help construct codes.

The search for optimal error correcting codes is closely related to the search for the maximum clique for a graph.

.

Jagota and Sanchis[7] use greedy heuristic-based search for finding cliques in large graphs with a high node to edge ratio. Also related is Pelillo's survey of heuristic approaches to maximum clique [9]. The heuristics aim to decrease the time required to find optimal codes in difficult parameter sets. Additionally, this paper introduces basic GA techniques as an alternative search methodology. Marchiori [8] examined a GA paired with a heuristic algorithm for optimization purposes. The heuristic used is based on a naive greedy heuristic procedure in which randomly built non-clique subgraphs are corrected to clique status. The use of a vanilla GA paired with a simple heuristic offered promising results.

Using a GA and several modifications, Carter et al. [1] evaluate algorithm performance for finding large cliques in the difficult DIMACS graph test data. They conclude that an unmodified GA will not be suitable for the problem, and that the use of a GA can only be justified by significant improvement over more simple heuristic based search due to the added algorithmic complexity.

GP has also been used for maximal clique discovery. Haynes et. al. [6] use strongly typed GP to detect and enumerate the cliques of a graph structure. Their results demonstrate that GP based solutions can effectively locate the maximum clique, although they are not as suitable for clique enumeration. On the other hand, Soule et. al [11] use a minimalist GP, which uses only the union operator. They show that a complex GP language is not required, and may in fact be a detriment to performance.

3. ERROR CORRECTING CODES

Error correcting codes were conceived by Hamming[5] and others in order to allow for the detection and correction of errors in digital information as a result of transmission error and corruption. The amount of information a given code can express is dependent on the length of each codeword and the alphabet from which the codewords are created. Longer codewords, or a larger radix for each digit of the codeword, results in a greater number of possible codewords. Unfortunately as the number of possible codewords grows, so does the search space for the codewords that share the required relationship with each other. The degree of error tolerance required by a code is defined by its minimum distance requirement. Each codeword must differ from every other codeword in the code by at least the same number of symbols as the minimum distance requirement. In this paper, the distance measure used is Hamming distance – the number of corresponding bits that differ in two binary strings of the same length[5]. If a corrupted codeword that does not match a known codeword is received, it is corrected by replacing it with the closest known codeword.

3.1 Maximum Clique

The construction of binary error correcting codes reduces to the NP-hard problem of finding a maximal clique of a graph. Given a graph G composed of a set of vertices V and a set of edges connecting these vertices E, a clique is a subset of V such that for any node pair x,y in V there exists an edge directly connecting x and y. The maximum clique of G is the clique subset of V with the largest number of nodes. By defining a graph such that V is the set of all possible codewords and E is composed of connections between codewords that meet the minimum distance requirement, then the graph's maximum clique is equivalent to the optimal error correcting code.

3.2 Compatibility Matrix

For any given $A_2(n, d)$ code, processing speed can be improved by pre-computing a static compatibility matrix[4]. A n by n matrix can be constructed such that a true value in the nth row and kth column indicates that codeword n and codeword k meet the minimum distance requirement. Testing the validity of a codeword becomes a simple matrix lookup. With respect to the maximum clique problem, the compatibility matrix defines the graph G. When each codeword defines a vertex in V, then an edge is present in E whenever node x and node y are compatible per the compatibility matrix. The graph for the fictional compatibility matrix shown in 1 is shown in Figure 2.

	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: Compatibility matrix for a $A_2(4, 2)$ code[4]

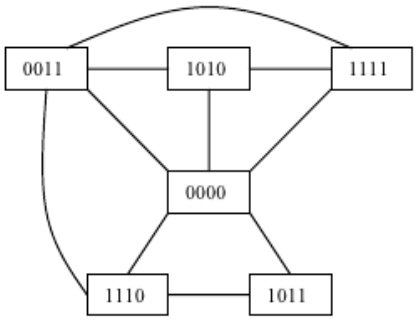

Figure 2: Graph of a $A_2(4, 2)$ code[4]

4. EXPERIMENTS

4.1 Overview

Both GA and GP approaches were applied towards finding codes for the binary (12,6), (13,6) and (17,6) error correcting codes. The GA experiments were inspired by prior research in [4] that used a GA on these problem sets. The GP experiments were new to code word search. Several GP specific variations were explored. It was postulated that a GP system might be ideal for this problem, as it might be less prone to the effects of epistasis. Epistatis is a measure of difficulty imposed by the fitness of a solution being highly dependent on the position of gene values in the chromosome.

4.2 Data Sets

The following codes are considered, in increasing order of difficulty:

1. A_2(12,6): The total number of codewords possible is 2,510. The known optimal code size is 24 codewords.
2. A_2(13,6): There are 5,812 potential codewords. The optimal code size is 32 codewords.
3. A_2(17,6): There are 121,670 codewords possible. The optimal code size is 256 codewords.

The total codewords for the above codes equals the number of nodes in the graph in which a maximal clique is to be found (Section 3.1). Letting N be the total number of code words for a code set, the size of the search space is

$$\sum_{k=1}^{N} \binom{N}{k} \gg 2^N.$$

This combinatorial size prohibits naive exhaustive search.

By using the notion of parity to compute the leading bit, the length of each codeword can be reduced by one bit without loss of generality. Thus, to construct a code for the A_2(12, 6) code, the codewords for the smaller but equivalent A_2(11, 5) code can be considered. This reduces the number of codewords by a half for all cases.

4.3 Validation

Validating codes is required during code construction. Each chromosome produces a list of codeword indexes representing the optimal code created during evolution. Each index corresponds to a codeword denoted in the compatibility matrix. All the codewords marked for inclusion in a chromosome are compared to each other. If the Hamming distance between a new codeword and an existing one is less than the required minimum distance, the new codeword is rejected.

4.4 Conway's Lexicode Algorithm

Conway's Lexicode Algorithm (Algorithm 1) is a greedy algorithm for the creation of minimum edit distance codes[2]. It is used as a specialized local search during some experiments. First, the algorithm sorts all possible codewords in ascending lexicographic order. In the case of a standard Conway Lexicode invocation, an empty code list is initialized to be populated by compatible codes. The implementation used in the GA and GP systems deviates from this pseudocode by pre-initializing the code list either by the valid codewords located in the GA chromosome or by the in-progress code being created by a GP tree. The sorted list of all possible codewords is then scanned linearly, comparing each sorted codeword to every codeword residing in the current list. New valid codes are added to the code list.

Algorithm 1 Conway's Lexicode Algorithm[2]

```
proc GenerateCode(minDist, codewordLength) ≡
  codewords ← lexicographic_sort(codewordLength);
  code ← empty();
  for i ← 0 to length(codewords) step 1 do
      for j ← 0 to size(code) step 1 do
          if hammingDist(codewords[i], codes[j]) ≤ minDist
            then break;
             else add(codes, codeswords[i]);
          fi
      od
  od
  return codes;
.
```

4.5 Genetic Algorithms

4.5.1 Representation

As in [4], a fixed length chromosome is used to represent an evolved code. Genes in the chromosome are indices to specific codewords. The chromosome length is set in each experiment to the known optimal size of the code parameters being considered. For example, a chromosome size of 32 is used for the $(13,6)_2$ code. (Should codes with unknown optima be considered in the future, a maximum would have to be strategically set.) As each index value is dereferenced to a codeword (see below) it is added to the code only if it is valid.

During chromosome processing, a compatibility bitmask is maintained, which represents the codewords compatible with the code being constructed so far. When a new compatible codeword is added, the codewords that are incompatible with it, as seen in the compatibility matrix, are removed (set to *false*) in this bitmask. This reduces the possible remaining choices, until the point where no compatible codewords are left. The code construction is then finished.

(i) *Indirect indexing* (I): This representation is from [4]. Each gene is assigned a value between 0 and the number of compatible codewords available. For the first gene processed, its index value corresponds directly to the numeric codeword from the compatibility matrix. As the first codeword, it is always added. Each subsequent gene is treated as an index into the set of *remaining compatible codewords* as defined by the current compatibility mask. A modulus operation is performed on the index, so it references only the current compatible codewords. This continues until no compatible codes are left.

(ii) *Direct modulo indexing* (DM): This is a new indexing scheme. Each gene is assigned a value from 0 to "max integer" (a large integer). As each gene is dereferenced, the codeword indexed to the gene value modulo the total number of codewords is considered for inclusion. In contrast with indirect indexing, the gene value initially dereferences a code word directly. If it turns out that a non-compatible codeword is dereferenced, them a linear search is done from this codeword onwards (modulo maximum limit) a a compatible codeword is located, or all are used up. It is conjectured

Table 1: Base Genetic Algorithm Parameters

Parameter	Value
Population Size	500
Max Generations	100
# Of Runs	20*
Seed	Run start time
Selection	Tournament (size 3)
Elitism	1 Individual
Crossover Chance	85%
Crossover	Two-point
Mutation Chance	15%
Mutation Operator	Random Gene Mutation

that this may reduce epistasis, so that patterns of compatible codewords may be effectively inherited during evolution.

4.5.2 Conway Finish

In [4], two different combinations of the Conway Lexicode Algorithm (Section 4.4) and the GA were explored. One scheme involved running the Conway Lexicode algorithm to generate seed codes that were then extended by codewords the GA chromosomes selected to be included during evolution. The other approach reversed this, by using the GA to evolve seed codes that were extended by the Conway greedy algorithm. They found that seeding the Conway algorithm with a small code evolved by the GA resulted in the best performance. Smaller seed codes between 3 and 9 codewords were most effective.

For the GA implementations tested here, the greedy finish procedure was added as an option. By limiting the fixed size of the chromosome for a Conway Finish, it was possible to evolve seed codes that were then extended using the greedy Conway Algorithm (algorithm 1) during fitness evaluation.

4.5.3 Fitness and GA Parameters

Fitness is defined as the number of compatible codewords included in a generated code. We always try to maximize the number of compatible codewords (i.e., maximal clique).

Common GA parameters are shown in Table 1. All are self-explanatory.

4.6 Genetic Programming

4.6.1 GP Language

Each GP tree denotes a complete code. In contrast with the fixed length GA chromosome, GP trees are variable sized. It is conjectured that this size flexibility, as well as the nature of GP reproduction operations, would minimize the harmful effects of epistasis. The variable length trees may improve the convergence speed of the GP system by avoiding the vestigial genes that occur from a non-optimal code being represented in an over-sized chromosome.

During evaluation of the GP tree, a state object is passed from node to node as the tree is evaluated. This tree state object contained within it an in-progress code list. Function and terminal nodes add codewords to this list when the codewords are deemed compatible. In this way, no invalid codes are created during tree evaluation.

*Only 10 runs were performed for the $(17,6)_2$ code.

Table 2: GP Types

Type	Description
Codeword	A single codeword index
Code	A collection of n codewords
Code or Codeword	Converts codeword to size 1 code

Strongly-typed GP is used (Table 2). A *codeword* is an integer index value that can be dereferenced to a binary codeword, while a *code* is a collection of one or more compatible codewords. The *code or codeword* type is a polymorphic type that converts a codeword to a size 1 code.

Only one terminal is used. A codeword index node is denoted by an ephemeral integer, in the range $[0, n)$ where n is the maximum number of codewords available.

Two basic GP operators are used (along with a Baldwinian State operator described below). (i) *Union Operator*: Inspired by work in .[11], a union operator collects the codes or codewords created by its child nodes. Two versions, with arity 2 and 3, were created, and permit varying shaped trees to be defined. (ii) *Conway Union Operator*: This is a union function that is augmented to perform the Conway algorithm. A *conwayStart* parameter indicates how many codewords should be present in the in-progress code being passed between tree nodes before the Conway algorithm is permitted to extend the code to completion. This parameter establishes the size of the seed code that is to be extended.

In Baldwinian evolution, a local search is used at the time of fitness evaluation[12]. Rather than encoding beneficial results obtained by the local search back into the chromosome, only the fitness value is updated. In this way, the fitness landscape changes while the evolution model remains Darwinian. A *Baldwinian Tree State* object is introduced. Each time a codeword is added to the code list by a GP Function node or terminal, the Baldwinian Tree State creates a new in-progress code list by cloning the current code list and adding the new codeword to it. Then the cloned codelist is extended as far as possible using the Conway Lexicode Algorithm. During fitness evaluation, all of the code lists created in a tree are considered, and the largest found is used as the fitness value for the individual. The actual tree is not altered. At the end of a run, however, the code list found with Baldwinian search is produced as the solution.

4.6.2 Tree Processing Order

The traversal order of a GP tree may impact the code developed. For instance, a tree traversed in a depth-first order will result in the leaf nodes from the leftmost branch be given highest priority. This may be detrimental to the effectiveness of reproductive search. To address this, both depth-first and breadth-first traversal were considered. It was conjectured that breadth-first traversal would allow for greater flexibility in the placement of codeword terminal nodes. Placing codewords higher in the tree structure would result in earlier inclusion and radically different compatibility options as a result.

4.6.3 GP Parameters and Experiment Configurations

The GP parameters used are in Table 3, and Tree depth

Table 3: Base Genetic Programming Parameters

Parameter	Value
Population Size	2000
Max Generations	100
# Of Runs	20*
Tree Initialization	Koza Half Builder
Grow Chance	50%
Node Select	10% Terminals, 90% Non-terminals
Selection	Tournament
Tournament Size	3
Elitism	1 Individual
Crossover Chance	60%
Tree Mutation Chance	10%
ERC Mutation Chance	30%

Table 4: GP Tree Depth Limits

Code	Limit	Value
$(12,6)_2$	Half Tree Max Depth	5
$(12,6)_2$	Crossover Max Depth	8
$(12,6)_2$	Mutation Max Depth	8
$(13,6)_2$	Half Tree Max Depth	5
$(13,6)_2$	Crossover Max Depth	10
$(13,6)_2$	Mutation Max Depth	10
$(17,6)_2$	Half Tree Max Depth	8
$(17,6)_2$	Crossover Max Depth	12
$(17,6)_2$	Mutation Max Depth	12

limits are in Table 4.

5. RESULTS

5.1 GA Results

5.1.1 $(12,6)_2$ Results

Based on the results in Table 5, the $(12,6)_2$ code posed little difficulty to either GA configuration. Changing indexing strategies resulted in no performance increase.

5.1.2 $(13,6)_2$ Results

The $(13,6)_2$ code proved challenging for the GA. As seen in Table 6, no single GA configuration was able to evolve an optimal 32 codeword code. The largest evolved code for any configuration was size 27 and the main difference between configurations was the frequency with which codes of this size could be evolved.

T-tests show that a statistically significant difference in performance was noted ($p < .05$) when the GA chromosome used the direct modulo indexing scheme (#2) as opposed to indirect indexing (#1), with the DM representation generating better solutions in most runs. The performance graphs of the configurations further substantiate the merit of the direct modulo indexing scheme.

Upon introduction of the Conway Finish algorithm to the GA configurations the benefits added by the switch to direct modulo indexing are lessened. Comparing configuration #3 (Indirect indexing, Conway Finish) with configuration #4 (Direct modulo indexing, Conway Finish) the difference in indexing is no longer statistically significant ($p > .05$). Both configurations were able to achieve codes of size 27 on the majority of runs. The Conway Finish benefits both indexing models equally, producing the best results seen for the GA $(13,6)_2$ configurations.

Unsurprisingly, a t-test shows that the difference between the random search configuration (#5) and a representative GA configuration (#3) is statistically significant ($p < 0.05$).

*Only 10 runs were performed for the $(17,6)_2$ code.

Table 5: GA $(12,6)_2$ Results. Optimal=24. Total 20 runs. (I: indexing, DM: direct modulo)

ID	Indexing	Best (#runs)	Average
1	I	24 (16)	22
2	DM	24 (16)	22

5.1.3 $(17,6)_2$ Results

From Table 7, without the use of the Conway Finish, neither the direct modulo or indirect representations were able to evolve optimal codes. However, #2 outperformed #1 on more runs ($p < 0.05$), showing that direct modulo indexing seemed more suitable. For the two configurations using the Conway Finish (#3 and #4), optimal and near optimal results were achieved. The difference between indexing schemes in these cases was not significant ($p > 0.05$).

Table 6: GA $(13,6)_2$ Results. Optimal=32. Total 20 runs. ID5 uses random search. (CS=Conway Start)

ID	Indexing	CS	Best (#runs)	Avg
1	I	-	23 (3)	22.1
2	DM	-	25 (20)	25
3	I	√	27 (20)	27
4	DM	√	27 (19)	26.9
5	DM	-	24 (5)	23.2

Table 7: GA $(17,6)_2$ Results. Optimal=256. Total 10 runs.

ID	Indexing	CS	Best (#runs)	Avg
1	I	-	135 (2)	133.5
2	DM	-	142 (5)	141.3
3	I	√	256 (1)	252.5
4	DM	√	255 (5)	253.8

5.2 GP Results

5.2.1 $(12,6)_2$ Results

Based on the results from Table 8 all of the GP configurations were able to achieve the optimal code size of 24. GP configurations $(12,6)_2$ #1 and #2 were especially well suited to the problem, achieving the optimal code size on 19 out of 20 runs.

5.2.2 $(13,6)_2$ Results

From Table 9, no single GP configuration was able to evolve a code equal to the optimal of 32. However, in comparison to the GA approaches, several GP configurations (#3, 7, 11 and 12) were able to come close, evolving codes of size 31. Generally, the GP configurations produced codes of a large size (27 and up).

The difference between configurations that were processed in depth-first order versus breadth-first order was statistically insignificant.

There was statistically significant difference ($p < .05$) in performance between the arity 2 union configuration (#2) compared to the arity 3 union configuration (#4) when traversed in a depth-first order. This was not found to be the case when breadth first order was used.

As seen in [4], the Conway start parameter value had great influence on the fitness achieved. Using the start value of 3 as in configuration #7 had the best result, achieving a fitness of 31 in 1/20 runs and an average fitness of 26.85 across all runs. Conway start values of 2 (#5) and 4 (#9) were also explored, obtaining codes of size 27 in 7/20 runs and 14/10 runs respectively. The difference between arity 2 and 3 union operators was found to be significant when paired with the Conway union for two of the three Conway start parameters.

The best performance encountered was a result of the Baldwinian tree state coupled with a Conway start of 2 (#12). This configuration was able to achieve codes of size 31 on 3/20 runs with an average code size of 27.6. The power of this configuration was found to stem from the Baldwinian tree state and not the Conway union nodes, as a similar configuration lacking the Conway Union node (#11) was found to have statistically insignificant difference in fitness ($p > .05$).

5.2.3 $(17,6)_2$ Results

Looking at Table 10, the use of the breadth first or depth first traversal (#1, 2) had minimal effect ($p > .05$).

Three Conway start values were explored for the $(17,6)_2$ GP configurations. The most successful of these configurations was configuration #3, with a Conway start parameter value of 10. These results were found statistically significant when compared to the 20 start value ($p < .05$) and the 30 start value ($p < .05$) indicating that it was the primary influence on the improved fitness of configuration #3 compared to #4 and #5.

The best results encountered for the $(17,6)_2$ code were obtained by configurations using the Baldwinian tree state option. Configuration #6 used the Baldwinian tree state with the arity 2 union to evolve an optimal code of size 256 on 1/10 runs, with an average fitness of 255.1 across all 10 runs. Configuration #7 was identical save the allowance of the Conway union node in addition to the standard arity 2 union. The power of configuration #6 and #7 arises from the Baldwinian approach and is not additionally strengthened by the explicit use of the Conway algorithm during evolution.

Table 8: GP $(12,6)_2$ Results. Optimal=24. Total 20 runs. (B=breadth first, D=depth first)

ID	Union Arity	Order	Best (# runs)	Avg
1	2	B	24 (19)	22.75
2	2	D	24 (19)	22.75
3	3	B	24 (16)	22
4	3	D	24 (16)	22

Table 9: GP $(13,6)_2$ Results. Optimal=32. Total 20 runs. (CS=Conway Start, B=Baldwin)

ID	Union Arity	Order	CS	B	Best (#runs)	Avg
1	2	B	-	-	27 (11)	26.1
2	2	D	-	-	27 (5)	25.5
3	3	B	-	-	31 (1)	25.6
4	3	D	-	-	27 (1)	24.9
5	2	B	2	-	27 (7)	25.75
6	3	B	2	-	27 (11)	26.1
7	2	B	3	-	31 (1)	26.85
8	3	B	3	-	27 (10)	26.1
9	2	B	4	-	27 (14)	26.45
10	3	B	4	-	27 (6)	25.65
11	2	B	-	√	31 (2)	27.4
12	2	B	3	√	31 (3)	27.6

Table 10: GP $(17,6)_2$ Results. Optimal=256. Total 10 runs. Union Arity=2.

ID	Order	CS	B	Best (# runs)	Avg
1	B	-	-	163 (1)	156.6
2	D	-	-	160 (1)	152.3
3	B	10	-	251 (1)	238.7
4	B	20	-	215 (1)	194.5
5	B	30	-	175 (1)	157.4
6	B	-	√	256 (1)	255.1
7	B	20	√	256 (1)	255.1

5.3 Comparing GA and GP

One goal of this research is to compare the efficacy of the GA and GP for code search. Firstly, it is clear from Tables 5 and 8 that the $(12, 6)_2$ code is easy for both representations, and not worth further scrutiny.

For the $(13, 6)_2$ runs, the results to consider are the superlative configurations for each: the GA configurations #3 and #4 in Table 6, and GP #11 and #12 in Table 9. Ex-

amining the average scores of the solutions for these configurations shows that the GA and GP have equivalent performance. The performance graphs for GA #4 and GP #12 in Figures 3 and 4 show similar convergence characteristics. However, unlike the GA runs, Table 9 shows that the GP runs were more consistent in returning near-optimal codes. This gives suggestive evidence that the GP representation may be worth considering over the GA for this code, and by extension, others in the future. The statistical significance of this requires more runs to establish.

Examining the $(17,6)_2$ code, the GA configurations #3 and #4 in Table 7 and GP #6 and #7 in Table 10 all obtain optimal or near-optimal codes. The performance charts for the GA and GP runs do show some differences. GA #3 and #4 are in Figures 5 and 6, while GP #6 and #7 are in Figures 9 and 10. The best solution curve in all the charts is remarkably similar. However, the population mean curves in the GA charts show a slow convergence in the first 40 generations to a suboptimal level. The GP populations, however, converge to a higher fitness in less than half the time, which suggests a good adaptation of the GP to this search space, perhaps due to reduced epistasis. For comparison, GP #1 in Figure 7 clearly converges to a suboptimal level. GP #3 in Figure 8 shows the most steady progress of all the plots. In fact, this plot suggests that superior results may have been obtained with GP #3 (breadth-first, Conway Start of 10, no Baldwinian) if it were permitted to run to 100 generations.

In summary, the best GA and GP experimental configurations are competitive with one another, and there is no clearly superior approach for all codes examined. On the other hand, the $(13,\ 6)_2$ experiments give suggestive evidence that the GP representation with Baldwinian search was the best performer for this code. The performance plots show that most runs with GA and GP are affected by premature convergence. Better quality results may arise if population diversity strategies were used.

6. CONCLUSION

This research builds on the pioneering work of Haas and Houghten [4], who established the viability of GAs for generating binary error correcting codes. The contribution of our research is to re-examine the generation of error correcting codes, using a variety of evolutionary computation strategies. This paper shows that, although a variety of techniques are feasible, different techniques may be especially applicable to particular codes. For example, the direct modulo gene representation improved the maximum fitness achieved, as well as the frequency with which these maximized results were obtained for both the $(13,6)_2$ and $(17,6)_2$ codes without the use of the Conway algorithm. The benefit of the Conway Finish algorithm demonstrated in [4] was reaffirmed by the superior performance of the GA configurations that used it.

Another contribution of this paper is the demonstration of the effectiveness of using GP to generate error correcting codes. Elsewhere, GP has been applied to the maximum clique generation problem [11, 6]. Soule *et al.* [11] established that a simple language could express optimal approximations to the problem of maximum clique – a finding that heavily influenced the design of our GP language. With our language, we demonstrated that GP is as equivalently powerful to GAs in the problems examined. The program tree representation, and its integration with the Conway Lexicode Algorithm and the use of the Baldwin search, allowed for performance that surpassed that of the GA approach in some cases. One advantage of the GP representation seems to be the reduction of negative effects of epistasis, which may be more acute in GA representations. Furthermore, the issues imposed by a fixed GA chromosome length are absent in a dynamic GP program tree representation. Further research exploring the ability for a GP system to discover optimal codes for code settings that do not have a single known optimum code size is required to establish these benefits.

Future work involving the integration of more heuristic information could prove to aid in the quality of results obtained. For instance, a measure of "near cliqueness" and a representation able to describe non-clique or partially incompatible codes could prevent premature convergence. Multi-objective scoring may be useful in this regard. Finally, the use of our best techniques should be considered for use in discovering new optimal codes in the future.

7. REFERENCES

[1] B. Carter and K. Park. How good are genetic algorithms at finding large cliques: an experimental study. *Boston University*, 1993.

[2] J. Conway and N. Sloane. Lexicographic codes: Error correcting codes from game theory. *IEEE Transactions on Information Theory*, IT-32(3):337–348, 1986.

[3] G. M. U. ECLab. ECJ: Java-based evolutionary computation research system, 2000–2011.

[4] W. Haas and S. Houghten. A comparison of evolutionary algorithms for finding optimal error-correcting codes. In *Proceedings of the third IASTED International Conference on Computational Intelligence*, pages 64–70, 2007.

[5] R. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950.

[6] T. Haynes and D. Schoenefeld. Clique detection via genetic programming. In *Preceedings of the First Annual Conference on Genetic Programming*, 1996.

[7] A. Jagota and L. Sanchis. Adaptive, restart, randomized greedy heuristics for maximum clique. *Journal of Heuristics*, 7:565–585, 2001.

[8] E. Marchiori. A simple heuristic based genetic algorithm for the maximum clique problem. In *Proceedings of the 1998 ACM symposium on Applied Computing*, pages 366–373, 1998.

[9] M. Pelillo. *Encyclopedia of Optimization*, chapter Heuristics for Maximum Clique and Independent Set. Kluwer Academic Publishers, 2001.

[10] R. Poli, W. Langdon, and N. McPhee. *A field guide to genetic programming*. Lulu, 2008.

[11] T. Soule, J. Foster, and J. Dickinson. Using genetic programming to aproximate maximum clique. In *Preceedings of the First Annual Conference on Genetic Programming*, pages 400–405, 1996.

[12] D. Whitley, V. S. Gordon, and K. Mathias. Lamarckian evolution, the baldwin effect and function optimization. In *PPSN III*, pages 6–15. Springer-Verlag, 1994.

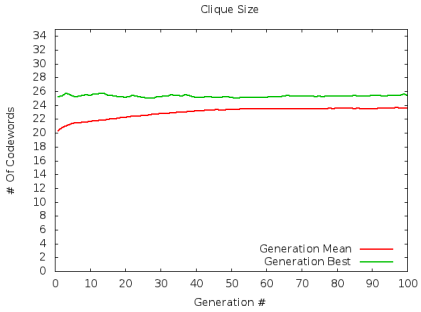

Figure 3: GA $(13,6)_2$ Configuration 4

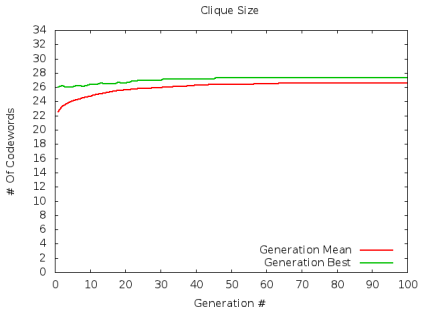

Figure 4: GP $(13,6)_2$ Configuration 12

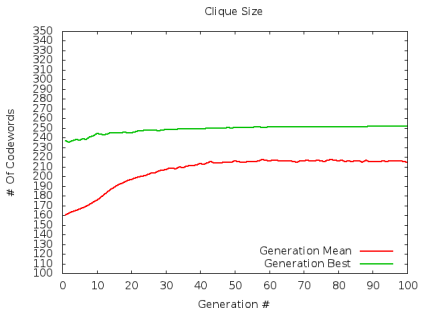

Figure 5: GA $(17,6)_2$ Configuration 3

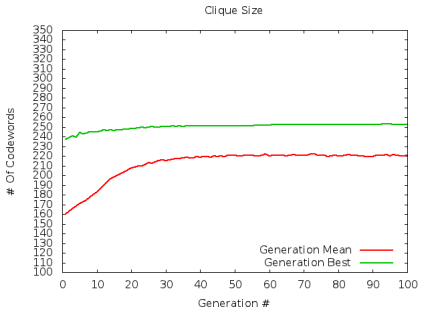

Figure 6: GA $(17,6)_2$ Configuration 4

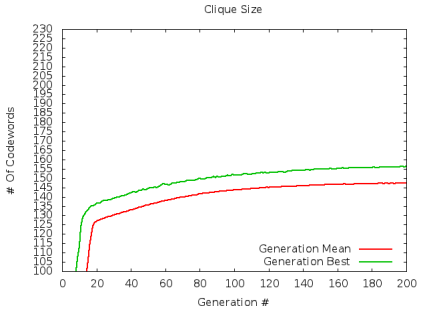

Figure 7: GP $(17,6)_2$ Configuration 1

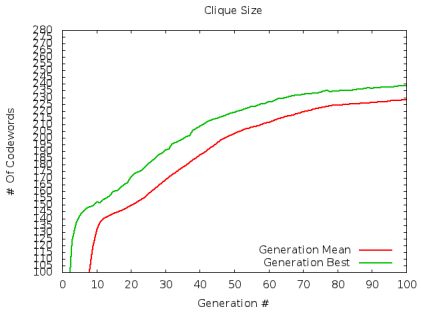

Figure 8: GP $(17,6)_2$ Configuration 3

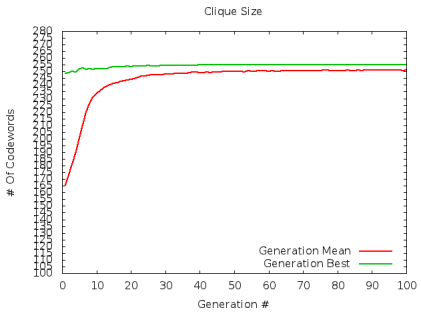

Figure 9: GP $(17,6)_2$ Configuration 6

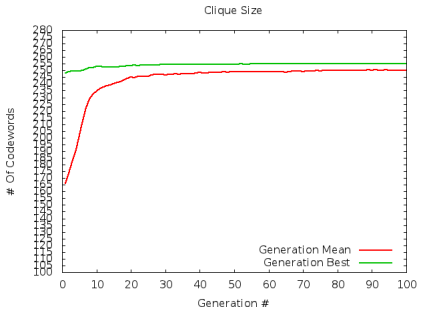

Figure 10: GP $(17,6)_2$ Configuration 7