Finding Minimal Covering Codes Using Greedy Closure Computational Intelligence Algorithms

Justin Maltese
Department of Computer Science
Brock University
St, Catharines, ON, Canada
jmaltese@brocku.ca

***Abstract*—A q-ary error-correcting code C is said to be covering if each possible word in the space is within a given radius r of at least one codeword in C. Finding covering codes with minimal number of codewords is an important problem with several well-documented real-world applications. A considerable amount of research has been performed to establish feasible techniques for finding minimal covering codes. Unfortunately, due to the inherent complexity of the problem nearly all techniques rely on previously found upper bounds, limiting the extent to which new bounds can be discovered. In this paper, we propose a new greedy algorithm capable of finding covering codes. This technique is combined with several computational intelligence methods to form novel greedy closure algorithms capable of finding minimal covering codes without relying on existing bounds. A comparison of each greedy closure algorithm is carried out on a wide variety of parameter sets to determine which algorithms are most effective. Results demonstrate the effectiveness of the proposed algorithms, as each is able to find covering codes whose sizes are typically close to existing bounds. Greedy closure genetic algorithms were found to perform especially well, outperforming all other algorithms in nearly all cases.**

I. Introduction

Often, when sending messages over unreliable communication channels data can become unintentionally corrupted in a variety of ways. In the case of binary data, substitution errors may occur wherein bits are flipped from 0 to 1 or vice-versa. Consequently, error detection and correction are often needed to enable reliable delivery of data. Error-correcting codes provide a means of accomplishing this via adding redundant information, allowing a corrupted message to potentially be recovered either partially or fully. Several comprehensive resources on error-correcting codes can be found via [1], [2].

An $(n, M, d)_q$ code is a collection of M codewords of length n, where each symbol is selected from an alphabet of q symbols. Additionally, each codeword must differ in at least d positions when compared to any other codeword in the code. A code that obtains the maximum possible value of M is deemed *optimal*. Finding optimal codes for different parameter sets is a fundamental problem and has been a topic of interest in the past few decades.

A covering code C of radius r is defined such that for all possible q-ary words of length n, there exists at least one word in C at Hamming distance r or less. Essentially, the space of all possible words is "covered" by the code. Covering codes possess a number of real-world applications, the most famous of which is the football pool problem [3]. Within this work, we are interested in finding covering codes with the smallest possible number of codewords for given values of n, q and r. Finding minimal covering codes is similar to finding optimal codes in that we are attempting to optimize the number of codewords in a code given a set of parameters. Additional similarities are that both problems can be represented graphically and are NP-complete.

Establishing effective techniques for finding minimal covering codes is a non-trivial task. Techniques proposed in previous literature [4], [5], [6] rely on existing bounds, attempting to find a covering code whose size is slightly less than the provided bound. A major disadvantage of this method is that codes are limited to a single size during the search, preventing discovery of codes possessing different sizes. Additionally, this method also yields a number of disadvantages when applied to population-based algorithms:

1) Since candidate solutions must encompass all codewords, bounds become increasingly more infeasible as they grow larger.
2) Large candidate solution sizes are highly susceptible to epistasis [7].

Motivated by the observations above, this paper proposes a new greedy algorithm which can be used to find covering codes. Computational intelligence (CI) algorithms are also incorporated, as they are used to evolve a small list of codewords which act as a seed into the proposed greedy method. These new greedy closure CI algorithms alleviate the issues presented above, since candidate solutions are kept small and codes are no longer limited to a single size during the search. Each of the proposed algorithms are compared on a variety of different parameter sets to determine relative performance. One should note that currently no literature exists on applying CI algorithms to find minimal covering codes, thus this paper aims to help fill the literature gap.

The remainder of this paper is organized as follows: Section II introduces the concept of covering codes and presents relevant background information. In Section III, an overview of various CI algorithms relevant to this work is given. Section IV presents and discusses each of the proposed algorithms. Section V describes the experimental setup used in this study. Section VI presents the results of all experiments performed, including analysis and discussion of observations. Finally, Section VII concludes the paper and suggests avenues for future research.

II. Covering Codes

A. Overview

A *covering code C* is an error correcting code where each q-ary word of length n is at Hamming distance r or less to at least one codeword of C. Codewords of C collectively "cover" all q^n possible words in the space. Formally, let W represent the set of q-ary words of length n. Then, a covering code C of radius r is a subset of W such that for all possible words of W there exists at least one word in C at Hamming distance r or less. That is:

$$C \subseteq W \text{such that } \forall x \in W, \exists y \in C : H_D(x,y) \leq r \qquad (1)$$

Where $H_D(x,y)$ represents the Hamming distance between word x and y. As an example, suppose a covering code is desired with parameters $n = 2, q = 2, r = 1$. The set of all possible words would thus be $W = \{00, 10, 01, 11\}$. A possible covering is $C = \{00, 11\}$. This cover is valid since all words in W are at Hamming distance ≤ 1 to at least one word in C.

Finding a cover of any size is trivial, as W itself is a valid covering code. A considerably more challenging problem is finding codes with the smallest number of codewords for a given set of parameters. The minimum size of a cover using q-ary words of length n and radius of size r is formally expressed as $K_q(n,r)$.

B. Relationship to Perfect Codes

By the definition of a perfect code P, all words are contained uniquely in the set of Hamming spheres of radius R centered around each codeword in P. Since no words exist outside of codeword spheres, perfect codes are also covering codes by definition. Thus, the following observation arises:

Observation 1a: *Given a code C, if C is a perfect code then C is also a covering code.*

The formulation above trivially extends to the observation that perfect codes are also minimal covering codes. Since by definition perfect codes pack all words into disjoint spheres centred around codewords, removal of any codeword would leave at least one word uncovered. The resultant code would not be valid, as the definition of covering is violated. Therefore, we have:

Observation 1b: *Given a code C, if C is a perfect code then C is also a minimal covering code.*

While perfection implies covering, the opposite does not hold. It is possible for a covering code to contain words which lie in the intersection of multiple Hamming spheres. Therefore, any covering code which possesses this property violates the definition of perfection and thus is not a perfect code.

TABLE I
Upper and lower bounds for the football pool problem

k	Lower Bound	Upper Bound
1	1	1
2	3	3
3	5	5
4	9	9
5	27	27
6	71	73
7	156	186
8	402	486

C. Practical Applications

Covering codes possess a number of important real-world applications. Detailed in [8], applications include data compression, error decoding, football pools, broadcasting in interconnection networks, write-once memory, speech coding and cellular telecommunications. Of these, the football pool problem is the most popular as it is believed to be one of the most important applications in coding theory [3].

The football pool problem deals with correctly predicting the outcomes of k football games, allowing a single incorrect prediction at most. The general idea is that one will win the pool regardless of the game outcomes while simultaneously minimizing cost if they can determine the smallest number of prediction combinations which cover all possible outcomes. Essentially, one is attempting to find a $K_3(k,1)$ code. Note that q is 3 since each game has a ternary outcome of win, loss or draw. Table I displays several football pool problem bounds for $1 \leq k \leq 8$. A comprehensive list of bounds can be viewed via [9].

D. Previous Literature

Determining $K_q(n,r)$ is a challenging problem, especially when n and q are large. Exceptions to this are for parameter sets which possess perfect codes, since perfect codes are also mnimal covering codes by Observation 1b. In the case where a perfect code is not possible $K_q(n,r)$ is often known only to be within a given range, since exhaustive searches are infeasible for all but trivially small values of n and q. Lower bounds of $K_q(n,r)$ are commonly established through mathematical methods, e.g see [10], [11], [12].

The focus of this work is on establishing upper bounds for $K_q(n,r)$ through proving existence by finding a code with those parameters. Previous literature has utilized metaheuristic techniques to perform computational searches. Heuristic functions guide the search towards desirable regions in the search space with the intent of uncovering global optima (minimal covering codes). Wille [5] was the first to utilize this method, applying a simulated annealing algorithm to the 6-match football pool problem as follows:

1) Input M as a parameter to the simulated annealing algorithm, where M is one less than the current best upper bound.
2) Execute the simulated annealing algorithm with the constraint that codes must have M codewords.

Algorithm 1 Tabu Search for Covering Codes

```
1: $S = S_0$
2: overallBestCode = $S$
3: tabuList = []
4: while $termination\ criteria\ not\ satisfied$ do
5:     bestCode = null
6:     for each code C in neighbourhood of S do
7:         if (tabuList.contains(C) $\wedge$ f(C) $>$ f(bestCode)) then
8:             bestCode = C
9:         end if
10:    end for
11:    S = bestCode
12:    if (f(overallBestCode) $>$ f(bestCode)) then
13:        overallBestCode = bestCode
14:    end if
15:    tabuList.push(bestCode)
16:    tabuList.removeFirstIfFull()
17: end while
```

3) If valid covering code found, M is the new best upper bound. Go back to step 1 and repeat with this new upper bound.

The above process was repeated until the algorithm consistently failed to find a new best upper bound. The simulated annealing algorithm utilized a simple neighbourhood function in which codewords are perturbed by swapping it with a word that it covers. The heuristic function f of a covering code C is defined as

$$f(C) = num\_uncovered(C) \tag{2}$$

where $num\_uncovered$ is a function which takes a code C as input, returning the number of words in W that are not covered by C.

In [4], the tabu search method was used to establish upper bounds on the football pool problem. The general flow of this algorithm is shown in Algorithm 1. Note that in Algorithm 1, S represents the current state, S_0 is a randomly generated initial state and $neighbourhood(\cdot)$ is a function which returns all neighbours of a given code.

Two distinct neighbourhood functions were utilized in [4]. The first was identical to the neighbourhood function of [5], described previously in this section. The second variant considers each q-ary vector of size n in lexicographic order. If an uncovered word is encountered, a word in C is perturbed in an attempt to cover it. If the word is successfully covered, the resulting move is a neighbour state. This neighbourhood function proved to be superior, resulting in several new upper bounds.

III. Computational Intelligence Algorithms

This section provides an overview of several basic CI algorithms used within this paper. Algorithms covered include particle swarm optimization, genetic algorithms and differential evolution.

Algorithm 2 Standard GBest PSO

```
1: Create and initialize a swarm, S, with candidate solutions in $n_x$ dimensions
2: while $termination\ criterion\ not\ satisfied$ do
3:     for each particle $i$ in $S$ do
4:         if $f(S.\vec{x}_i) < f(S.\vec{y}_i)$ then
5:             $S.\vec{y}_i = S.\vec{x}_i$
6:         end if
7:         if $f(S.\vec{y}_i) < f(S.\vec{\hat{y}})$ then
8:             $S.\vec{\hat{y}} = S.\vec{y}_i$
9:         end if
10:    end for
11:    for each particle $i$ in $S$ do
12:        Update velocity of particle $i$ using Equation (3)
13:        Update position of particle $i$ using Equation (4)
14:    end for
15: end while
```

A. Particle Swarm Optimization

Particle swarm optimization (PSO) is a metaheuristic optimization algorithm developed by Kennedy and Eberhart [13], inspired from the flocking behavior of birds. A PSO algorithm maintains a swarm of particles, where each particle represents a candidate solution to the optimization problem under consideration. Particles adapt their positions by moving towards the position of the best candidate solution found by the particle's neighbourhood and the best candidate solution found by the particle itself.

The PSO algorithm is summarized in Algorithm 2. One should note that the algorithm makes use of a star neighbourhood topology [13], which uses the social component to attract particles to the overall best position of the swarm. Alternatively, one could use a ring topology, which results in neighbourhoods of particle attraction [14].

Algorithm 2 also uses synchronous updating [15] of particle positions and best positions, where all particles are first updated, followed by an update of all best positions. Particle positions are initialized randomly within the bounds of the defined search space. Initial particle velocities should be set to zero, as shown in [16]. Algorithm 1 makes use of two update equations on line 12 and 13, defined as:

$$\begin{aligned} S.\vec{v}_i(t+1) &= \omega S.\vec{v}_i(t) + c_1\vec{r}_1(S.\vec{y}_i(t) - S.\vec{x}_i(t)) \\ &\quad + c_2\vec{r}_2(S.\vec{\hat{y}}(t) - S.\vec{x}_i(t)) \end{aligned} \tag{3}$$

$$S.\vec{x}_i(t+1) = S.\vec{x}_i(t) + S.\vec{v}_i(t+1) \tag{4}$$

where $S.\vec{v}_i$ is the velocity of particle i in swarm S, $S.\vec{x}$ is the current position of particle i in swarm S, $S.\vec{y}_i$ is the personal best position of particle i in swarm S, $S.\vec{\hat{y}}$ is the global best position of swarm S, $\vec{r}_1$ and $\vec{r}_2$ are vectors of random numbers sampled from a uniform distribution in the range [0,1], ω is the inertia weight, c_1 is the cognitive weight and c_2 is the social weight.

B. Genetic Algorithms

The genetic algorithm (GA) [17], [18] is an optimization technique which mimics the real-world process of natural selection. Survival of the fittest is a central idea in GAs, with fitter individuals guiding the search. Similar to nature, the fittest individuals are more likely to survive and mate in GAs, essentially carrying on their features into the next generation. A population of candidate solutions is maintained, referred to as *strings* or *chromosomes*. Chromosomes possess a fixed number of dimensions, each referred to as an individual *gene*.

Arguably the most challenging aspect of GAs is choosing how to represent candidate solutions as chromosomes, a process referred to as *encoding*. Determining a suitable encoding is problem specific, with certain problems possessing relatively more obvious encodings than others. The simplest forms of encoding are seen for problems which can be represented as binary strings.

1) Fitness Evaluation: To determine the quality of an individual chromosome, it is necessary to utilize a heuristic function which produces a single scalar value. The resultant value is referred to as the *fitness* of the individual and represents their desirability with respect to the problem at hand. Thus, fitness functions are entirely problem dependent. In the case of a minimization problem, lower fitness values are deemed better; the opposite is true for maximization problems.

2) Mating Selection: At each generation in a GA, individuals are mated together to produce offspring which persist into the next generation. Selection of parents must be done in such a way that promotes fit individuals while also maintaining diversity of the entire population. If the current best individuals are selected to produce every single child in the next generation, the GA becomes susceptible to premature convergence into local minima through lack of diversity. Thus, selection of parents must be performed in such a way that is biased towards more fit individuals while still allowing less fit members a chance to be chosen. The most common selection strategies fitting these criteria are:

- **Tournament Selection** - k individuals are randomly selected out of the current population to participate in the tournament. From these individuals, the winner of the tournament is deemed to be the individual with the best fitness function value. The value of n determines the amount of selection pressure, with larger tournaments essentially leading to more selection pressure.
- **Roulette Wheel Selection** - Each individual in the current population is assigned space on a roulette wheel proportionate to their fitness. More fit individuals occupy a larger portion of the wheel, giving them a higher chance of being selected. This type of selection is also referred to as *fitness proportionate selection.*

3) Crossover: To perform the mating operation, two individuals are essentially combined by exchanging genes to form children. This recombination process is formally referred to as *crossover*. Crossover essentially serves as the main recombination operator within GAs and is largely responsible for producing the population at each generation. Selection of crossover is dependent on the problem at hand. Continuous problems often require different crossovers than discrete problems.

A crossover commonly used is the one-point crossover. This operator is performed by randomly choosing a point in a chromosome string and exchanging all genes after that point using two selected parents. As an example, let $p_1 = 10011100$ and $p_2 = 11101011$ be two parents selected for crossover. Assume that the crossover point is randomly chosen to be after the fourth gene. Then, the two children produced would be

$$c_1 = 1001|1011$$
$$c_2 = 1110|1100$$

4) Mutation: To overcome local minima, a *mutation* operator is used to perform a random minor alteration of a chromosome. Unlike the crossover operator, mutation requires only one selected individual. Mutation is used to increase exploitation, providing additional diversity and preventing premature convergence. An example of mutation would be the bit flip method, which simply selects a random position within a binary chromosome and flips the bit at the selected position.

C. Differential Evolution

DE, introduced in [19], is a simple yet efficient evolutionary computation technique which optimizes a problem by iteratively improving a set of candidate solutions. Within DE, candidate solutions are referred to as *agents* and the set of maintained agents is known as a *population*. Agents within the population, initially placed at random positions in the search space, are updated by combining a number of agents from the population. Like a typical evolutionary algorithm, DE performs optimization using selection, crossover and mutation operators. Concerning stopping criterion, DE continues optimization until a maximum number of generations are reached or an ideal solution is encountered.

1) Trial Position Creation: DE differs from other evolutionary algorithms through utilizing the concept of a *trial position*. A trial position represents a newly created position that an agent has the potential to move to. At each generation DE iterates through the current population, creating a trial position for each agent through the use of a recombination operator. If the created trial position has better fitness than the current position of the agent, the trial position is accepted and the agent moves to the trial position. Otherwise, the trial position is simply discarded and the agent retains its current position. Thus DE is considered an elitist strategy, since the average fitness value of the population will never decrease between generations. While a variety of methods exist to create a trial position in DE, the most commonly used version is DE/rand/1/bin [20]. Creation of a trial position t for an agent x in DE/rand/1/bin is shown in Algorithm 3. Note that within Algorithm 3 P represents the current population, D is the dimensionality of the problem to be optimized, $rand()$ is a function which returns a random floating point value in the range $[0, 1)$, CR is the *crossover probability* and F is the *differential weight*. Both CR and F are user-defined parameters, discussed further below.

Algorithm 3 DE Trial Position Creation

```
1: $a \in P, a \neq x$
2: $b \in P, b \neq x \wedge b \neq a$
3: $c \in P, c \neq x \wedge c \neq b \wedge c \neq a$
4: $i_{rand} \in \{1, 2, ..., D\}$
5: for $(i = 1;\ i \leq D;\ i = i + 1)$ do
6:     if $rand() < CR \vee i = i_{rand}$ then
7:         $t_i = a_i + F \cdot (b_i - c_i)$
8:     else
9:         $t_i = x_i$
10:    end if
11: end for
```

2) Parameter Selection: Three main parameters exist in DE, defined *a priori* to optimization. The population size NP refers to the number of agents that will be in the population at each generation. The crossover probability CR is a value between [0,1] which controls the crossover operation, essentially representing the chance that a dimension of the trial position dimension will be chosen from a linear combination of three randomly chosen agents. However, at least one randomly chosen dimension of the trial position is guaranteed to be a crossover result, seen from the condition $i = i_{rand}$ on Line 6 in Algorithm 3. In practice, CR controls the level of exploration seen in the search.

Another crucial parameter to DE is the differential weight F. F acts as a scaling factor for mutation, constrained to $[0, 2]$. Mutation within DE is self-adaptive to the problem surface similar to Covariance Matrix Adapatation Evolutionary Strategies [21]. In early generations, mutation magnitude is large due to agents typically having radically different positions in the search space. As evolution continues, the population converges and agents become more similar, reducing the overall mutation magnitude. Choosing an appropriate value of F is critical as it controls the speed and robustness of the search. The problem of premature convergence into local minima is overcome by choosing a suitably large value of F. However, choosing values that are too large may lead to erratic behaviour and a severely decreased convergence rate.

Since the choice of control parameters are critical to performance of the DE algorithm, a variety of literature exists on the topic. Ali and Törn [22] propose an adaptive strategy for the differential weight F, which changes during evolution according to the equation

$$F = \begin{cases} max(F_{min}, 1 - |\frac{f_{max}}{f_{min}}|), & \text{if } |\frac{f_{max}}{f_{min}}| < 1. \\ max(F_{min}, 1 - |\frac{f_{min}}{f_{max}}|), & \text{otherwise} \end{cases} \quad (5)$$

where f_{min} is the minimum fitness value in the population, f_{max} is the maximum fitness value in the population and F_{min} is an input parameter such that $F \in [F_{min}, 1]$. This strategy is used to create a more diversified search early on and an intensified search as the population is converging in later generations. Previous literature [22] has shown this method to be quite effective despite its simplicity. However, one should note that when $f_{min} < 0$ and $f_{max} > 0$, F experiences large fluctuations and the method loses its effectiveness. Several other methods have been proposed for adaptive parameter control in DE, such as [23], [24], [25].

Algorithm 4 GreedyCover

```
1: $W = lexicographic\_sort(q, n)$
2: $L = \emptyset$
3: for $(i = 0;\ i \leq W.length;\ i++)$ do
4:     $isCovered = false$
5:     for $(j = 0;\ j \leq L.length \wedge\ !isCovered;\ j++)$ do
6:         if $H_D(W[i], L[j]) \leq r$ then
7:             $isCovered = true$
8:         end if
9:     end for
10:    if $!isCovered$ then
11:        $L.add(W[i])$
12:    end if
13: end for
```

IV. Proposed Algorithms

This section proposes and discusses several new algorithms for finding covering codes. A greedy covering algorithm is presented along with five new greedy closure CI algorithms.

A. Greedy Covering Algorithm

We present a new greedy algorithm designed to find covering codes, referred to as *GreedyCover*. A list of codewords L is maintained by the algorithm, initially empty. The algorithm executes as follows: Step through all possible q^n words of the space in lexicographic order. For each word w, if it is not covered by at least one codeword in L add w to L. w is now trivially covered, since $H_D(w, w) = 0$. Continue this process until all possible q^n words have been considered. When the algorithm terminates, L is a valid covering code since all words in the space are covered. The entire GreedyCover algorithm is shown in Algorithm 4.

One should note that the effectiveness of this algorithm grows with r. When a word w is added to L, all words within radius r of w are covered, removing the need to add any of them to L in the future. The number of words covered by adding w into L is equivalent to the size of the Hamming sphere of radius r centered around w, which is

$$\sum_{i=0}^{r} \binom{n}{i} (q-1)^i \quad (6)$$

It is obvious from the above equation that larger values of r will result in more words being covered, increasing the influence of a single addition into L.

B. Greedy Closure CI Algorithms

A *greedy closure* CI algorithm evolves a short list of codewords, utilizing a greedy algorithm to transform (close) the partial codeword list into a full code. An example of a greedy closure evolutionary algorithm can be found in [26]. We propose five new greedy closure CI algorithms which each utilize the GreedyCover algorithm. Each algorithm employs a CI technique to evolve a "seed", which is fed into the GreedyCover algorithm and serves as the initial codeword list L. Each codeword in the initial list can have a large impact on the outcome of the GreedyCover algorithm, influencing the resultant code.

One should note that the impact of each individual codeword grows with r as described in Section IV.A. Theoretically, evolving seeds rather than entire codes should significantly reduce epistasis and allow large bounds to be tackled. The proposed greedy closure algorithms are:

1) PSO-GC: The PSO algorithm is used to evolve a collection of candidate solutions, where each candidate solution serves as a seed into the GreedyCover algorithm.

2) DPSO-GC: As a consequence of being designed for continuous optimization problems, the PSO algorithm possesses several disadvantages when tasked with a discrete optimization problem such as finding a minimal covering code. The primary issue is that particles simply move their position towards their personal best and the swarm global best, but do not necessarily inherit any of the values. In terms of codes, this is analogous to simply moving each codeword towards desirable codewords but not actually changing into the codewords themselves. Compounded with the Hamming cliff problem, desirable codewords are almost never inherited from a particles personal best or the swarm global best during the update phase.

To aid in alleviating the above problem, a discrete PSO (DPSO) variant is proposed as a modification of the PSO algorithm. DPSO is identical to traditional PSO, however it's position update phase differs slightly. Given a particle p, each decision variable of p has a chance to mutate into the decision variable of the swarm global best at the same index. If the variable does not mutate, then it is given a chance to mutate into the decision variable of the personal best position of p at the same index. These changes require two additional parameters, which are the probability of global best mutation and the probability of personal best mutation.

The DPSO-GC algorithm utilizes DPSO to evolve a collection of candidate solutions, where each candidate solution serves as a seed into the GreedyCover algorithm.

3) DE-GC: The DE algorithm is used to evolve a collection of candidate solutions, where each candidate solution serves as a seed into the GreedyCover algorithm.

4) DDE-GC: The DE algorithm, inherently designed for continuous optimization, also possesses the same problems as PSO when tasked with discrete optimization (discussed above). A discrete DE (DDE) variant is proposed to improve the performance of DE on discrete problems. A simple modification is employed to the trial position creation phase, detailed in Algorithm 5. Crossover and mutation is now separated in DDE, requiring the introduction of a mutation probability parameter (denoted MP).

Algorithm 5 DDE Trial Position Creation

```
1:  $a \in P, a \neq x$
2:  $b \in P, b \neq x \wedge b \neq a$
3:  $c \in P, c \neq x \wedge c \neq b \wedge c \neq a$
4:  $i_{rand} \in \{1, 2, ..., D\}$
5:  for ($i = 1$; $i \leq D$; $i = i + 1$) do
6:      if $rand() < CR \vee i = i_{rand}$ then
7:          $t_i = a_i$
8:          if $rand() < MP$ then
9:              $t_i = t_i + F \cdot (b_i - c_i)$
10:         end if
11:     else
12:         $t_i = x_i$
13:     end if
14: end for
```

The DDE-GC algorithm employs DDE to evolve a collection of candidate solutions, where each candidate solution serves as a seed into the GreedyCover algorithm.

5) GA-GC: The GA is used to evolve a collection of candidate solutions, where each candidate solution serves as a seed into the GreedyCover algorithm.

V. Experimental Setup

Within this section, the experiment setup is described. This includes algorithm parameters along with an overview of the experimental framework developed during the course of this work

A. Algorithm Parameters

All algorithms used an identical number of iterations and candidate solutions to ensure an unbiased comparison as recommended in [27]. Specifically, a total of 200 iterations with 400 candidate solutions were used for each algorithm. All additional parameters, described below, were determined through empirical trials by the author.

1) PSO Algorithms: Both PSO-GC and DPSO-GC utilized an initial particle velocity of zero. To ensure convergent weight values [28], ω was set to 0.729844, c_1 was set to 1.496180 and c_2 was set to 1.496180. To handle boundary constraints, bests were only updated if their positions remained within the legal bounds of the search space. This method ensured that particles were always attracted back into the search space should they happen to exit. The DPSO-GC algorithm utilized a 10% chance for decision variables to mutate to the swarm global best and a 5% chance for decision variables to mutate to the personal best.

2) DE Algorithms: DE-GC and DDE-GC set CR = 0.7, F = 1.0. An MP value of 0.30 was used for the DDE-GC algorithm. To ensure boundary constraints, positions of violating agents were mapped back into the search space using the modulus function.

TABLE II
ALGORITHM PERFORMANCE FOR BINARY COVERING PROBLEMS

		Problem				
Algorithm	**Metric**	$K_2(13,4)$	$K_2(15,5)$	$K_2(16,6)$	$K_2(18,7)$	$K_2(20,8)$
GA-GC	Best	24(5/10)	25(1/10)	17(3/10)	18(1/10)	20(8/10)
	Average	24.5	26.5	17.7	18.9	20.2
PSO-GC	Best	28(9/10)	29(1/10)	19(6/10)	21(5/10)	22(9/10)
	Average	28.1	30.5	19.4	21.5	22.1
DPSO-GC	Best	27(1/10)	29(1/10)	18(2/10)	20(1/10)	22(10/10)
	Average	28	30.3	18.9	21.2	22
DE-GC	Best	28(9/10)	30(1/10)	19(10/10)	21(5/10)	22(10/10)
	Average	28.1	30.9	19	21.5	22
DDE-GC	Best	27(6/10)	29(1/10)	18(2/10)	20(5/10)	21(3/10)
	Average	27.4	29.9	18.8	20.5	21.7
	Best Known:	**16**	**16**	**12**	**12**	**12**

TABLE III
ALGORITHM PERFORMANCE FOR TERNARY COVERING PROBLEMS

		Problem				
Algorithm	**Metric**	$K_3(8,3)$	$K_3(9,4)$	$K_3(9,5)$	$K_3(10,4)$	$K_3(10,5)$
GA-GC	Best	38(2/10)	26(1/10)	9(4/10)	60(1/10)	20(8/10)
	Average	39.5	27.1	9.6	61.9	20.2
PSO-GC	Best	44(1/10)	30(1/10)	10(6/10)	70(4/10)	22(9/10)
	Average	46.7	30.9	10.4	70.6	22.1
DPSO-GC	Best	44(2/10)	30(7/10)	9(2/10)	68(2/10)	21(3/10)
	Average	45.8	30.3	9.8	69.4	21.7
DE-GC	Best	45(3/10)	30(2/10)	10(8/10)	69(6/10)	22(9/10)
	Average	45.9	30.8	10.2	69.4	22.1
DDE-GC	Best	44(2/10)	29(1/10)	7(1/10)	68(1/10)	21(1/10)
	Average	45.4	29.9	9.5	69.1	21.9
	Best Known:	**27**	**18**	**6**	**36**	**12**

TABLE IV
ALGORITHM PERFORMANCE FOR QUATERNARY COVERING PROBLEMS

		Problem				
Algorithm	**Metric**	$K_4(5,2)$	$K_4(6,2)$	$K_4(6,3)$	$K_4(7,3)$	$K_4(7,4)$
GA-GC	Best	21(1/10)	64(6/10)	18(4/10)	56(1/10)	14(3/10)
	Average	22.7	65.3	18.6	57.2	14.7
PSO-GC	Best	28(9/10)	86(1/10)	20(3/10)	64(2/10)	16(9/10)
	Average	28.1	90	20.7	65	16.1
DPSO-GC	Best	25(2/10)	76(1/10)	19(1/10)	64(3/10)	15(4/10)
	Average	26.6	86.5	20.1	64.8	15.6
DE-GC	Best	27(1/10)	89(1/10)	20(2/10)	64(2/10)	16(10/10)
	Average	27.9	91.9	20.8	64.9	16
DDE-GC	Best	26(3/10)	86(1/10)	20(6/10)	63(2/10)	15(4/10)
	Average	26.9	88.3	20.4	63.8	15.6
	Best Known:	**16**	**52**	**14**	**32**	**10**

TABLE V
ALGORITHM PERFORMANCE FOR QUINTARY COVERING PROBLEMS

		Problem				
Algorithm	**Metric**	$K_5(4,2)$	$K_5(5,3)$	$K_5(5,2)$	$K_5(6,2)$	$K_5(6,3)$
GA-GC	Best	11(1/10)	10(1/10)	51(6/10)	213(2/10)	39(2/10)
	Average	12.9	10.9	51.8	216.3	40.1
PSO-GC	Best	14(3/10)	11(2/10)	61(2/10)	254(1/10)	44(1/10)
	Average	14.7	11.8	62.3	261.2	45.2
DPSO-GC	Best	13(1/10)	11(2/10)	61(5/10)	248(1/10)	44(1/10)
	Average	13.9	11.8	61.6	257.7	45.2
DE-GC	Best	14(1/10)	11(1/10)	61(4/10)	258(1/10)	45(9/10)
	Average	14.9	11.9	61.8	261.3	45.1
DDE-GC	Best	13(1/10)	11(5/10)	58(1/10)	252(3/10)	44(5/10)
	Average	13.9	11.5	60.8	253.5	44.5
	Best Known:	**11**	**9**	**35**	**125**	**25**

3) Genetic Algorithm: For the GA-GC algorithm, the crossover rate and mutation rate were set to 0.85 and 0.10, respectively. Selection was done tournament style with 3 participants used in all cases. Elitism was employed, ensuring that the overall best individual of each generation persisted into the next generation.

B. Experimental Framework

To perform the necessary experiments in this work, the Computational Intelligence for Covering Codes (CICC) framework was developed. Since this work is the first to utilize CI algorithms to find covering codes, the framework is the first of its kind and will be useful for any future offshoots of this work.

Within the CICC framework codewords are represented as integers, stored as binary sequences in memory. In the case where $q = 2$, bit operations can be used to directly calculate codeword compatibility quickly. When $q > 3$, the integer representation of each word is converted into base q when computing distance. In this case, bit operations cannot be used and thus compatibility calculation is much slower. Consequently word compatibility is not computed at runtime, rather it is precomputed and accessed in $O(1)$ time via a matrix of bits. Such a matrix is deemed a "compatibility matrix", defined as follows: If the bit at index i, j = 1, $H_D(i,j) \leq r$, therefore i covers j and vice-versa. If the index is 0, then neither i or j cover each other and thus are incompatible.

Compatibility bit matrices for each problem are generated in advance and stored in .matrix files, loaded into memory by the program when needed. Using this strategy, large values of n and q can be tackled for as many runs as needed, since the matrix only needs to be loaded into memory once for each set of runs of a given problem.

VI. EXPERIMENTAL RESULTS AND DISCUSSION

The performance of the proposed greedy closure CI algorithms were compared over a plethora of parameter sets, ensuring a diverse and comprehensive testing environment. Tables II-V present a performance overview of each algorithm on a set of binary, ternary, quaternary and quintary covering problems, respectively. The best ever found code size along with the average code size over all runs is displayed for each algorithm. Additionally, numbers in brackets identify the amount of times a best result was obtained in comparison to the total number of runs, set to 10. Each problem displays a current best known bound in bold at the bottom of each table along with a best performing algorithm, highlighted in gray.

Tables II-V present several interesting observations. GA-GC obtained the lowest best and average values for nearly all problem sets, outperforming all other algorithms in most instances. It is clear that GA-GC is undisputably superior to all other algorithms for finding minimal covering codes. GA-GC seems to be adept at finding covering codes with small amounts of words, likely due to the effectiveness of the selection, crossover and mutation combination seen in GAs. The superiority of GC-GA is unsurpising, as it is the only algorithm inherently designed to tackle discrete problems.

Another notable observation is the effectiveness of the proposed discrete variants. Both DPSO-GC and DDE-GC exhibit an enhanced ability to find minimal codes in comparison to their non-discrete variants. In almost all problem instances, both discrete variants yielded lower best and average values in comparison to both PSO-GC and DE-GC. This observation leads one to conclude that the modifications seen in DPSO-GC and DDE-GC are effective for improving the performance of PSO and DE on discrete problems such as finding minimal covering codes. DDE-GC performed notably well for the $K_3(9,5)$ problem, achieving the lowest best and average code size out of all algorithms.

Analyzing the effectiveness of the GreedyCover closure technique, it is clear that the approach is adept at finding small covering codes when combined with CI algorithms. However, considerable empirical evidence within this work suggests that performance ranges widely based on the given problem parameters. Notable performances include the $K_2(15,5)$, $K_2(20,8)$, $K_3(9,5)$, $K_4(7,4)$ and $K_5(4,2)$ problems. Unfortunately, performance was quite poor for certain problems, e.g. $K_5(6,2)$, $K_4(7,3)$, $K_3(10,4)$. One should note that each greedy closure CI algorithm becomes more effective as r increases (see Section IV.A), an observation which is apparent when examining outcomes of the experiments performed.

Overall, performance of the proposed algorithms was quite satisfactory. In most cases found code sizes were close but not identical to the size of the current bound, which is expected given the complexity of the problem. Since finding minimal covering codes is NP-complete, it is very difficult to find global optima, especially considering that the search space size grows extremely quickly with q and n. Note that difficult problem sets were deliberately chosen to present a challenge to the algorithms; if trivial problem sets had been used, algorithms would have likely matched many more bounds.

VII. Conclusion

This work investigated the effectiveness of combining CI algorithms with a novel greedy approach to find minimal covering codes. Five new greedy closure algorithms were proposed, formally referred to as PSO-GC, DPSO-GC, DE-GC, DDE-GC and GA-GC. Performance of these algorithms were compared over a variety of challenging binary, ternary, quaternary and quinary covering problems.

Results demonstrate the effectiveness of the proposed algorithms. While in most instances current bounds expectedly were not matched identically, each greedy closure algorithm exhibited the ability to find codes whose size was near the current best. The GA-GC algorithm performed especially well, outperforming all other algorithms on nearly every problem instance. This performance disparity is attributed the traditional GA algorithm, as it is inherently designed to solve discrete problems.

An interesting observation was the performance of both discrete algorithms, DPSO-GC and DDE-GC. Both algorithms exhibited an enhanced ability to find minimal covering codes in comparison to their non-discrete variants. From this, it was concluded that the modifications proposed in DPSO-GC and DDE-GC improve discrete problem performance by addressing weaknesses present in algorithms designed for continuous problems.

There are many opportunities for future work in this area. A straightforward avenue would be to improve upon the proposed GreedyCover algorithm, increasing its effectiveness when combined with CI algorithms. Entirely new greedy algorithms could be proposed as well. Another interesting offshoot would be to develop creative new greedy closure CI variants and compare their performance against the existing algorithms in this work.

References

[1] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland Publishing, 1980. [Online]. Available: http://dx.doi.org/10.1002/9781118032749.fmatter

[2] V. Pless, *Introduction to the Theory of Error-Correcting Codes*. John Wiley & Sons, Inc., 1998. [Online]. Available: http://dx.doi.org/10.1002/9781118032749.fmatter

[3] H. Hamalainen, I. Honkala, S. Litsyn, and P. Ostergard, "Football pools–a game for mathematicians," *The American Mathematical Monthly*, vol. 102, no. 7, pp. 579–588, 1995. [Online]. Available: http://www.jstor.org/stable/2974552

[4] P. Ostergard, "Constructing covering codes by tabu search," *J. COMBIN. DES*, vol. 5, pp. 71–80, 1997.

[5] L. Wille, "The football pool problem for 6 matches: A new upper bound obtained by simulated annealing," *Journal of Combinatorial Theory, Series A*, vol. 45, no. 2, pp. 171 – 177, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0097316587900124

[6] P. van Laarhoven, E. Aarts, J. van Lint, and L. Wille, "New upper bounds for the football pool problem for 6, 7, and 8 matches," *Journal of Combinatorial Theory, Series A*, vol. 52, no. 2, pp. 304 – 312, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0097316589900368

[7] D. Ashlock, L. Guo, and F. Qiu, "Greedy closure evolutionary algorithms," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, vol. 2, 2002, pp. 1296–1301.

[8] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein, *Covering Codes*, ser. North-Holland Mathematical Library. Elsevier Science, 1997. [Online]. Available: https://books.google.ca/books?id=7KBYOt44sugC

[9] G. Keri, "Tables for bounds on covering codes," http://www.sztaki.hu/~keri/codes/, accessed: 2015-11-03.

[10] L. Habsieger and A. Plagne, "New lower bounds for covering codes," *Discrete Mathematics*, vol. 222, no. 13, pp. 125 – 149, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0012365X0000011X

[11] D. Li and W. Chen, "New lower bounds for binary covering codes," *Information Theory, IEEE Transactions on*, vol. 40, no. 4, pp. 1122–1129, Jul 1994.

[12] P. Ostergard and A. Wassermann, "A new lower bound for the football pool problem for six matches," *Journal of Combinatorial Theory, Series A*, vol. 99, no. 1, pp. 175 – 179, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0097316502932607

[13] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, vol. 4, Nov 1995, pp. 1942–1948.

[14] R. Eberhart, P. Simpson, and R. Dobbins, *Computational Intelligence PC Tools*. San Diego, CA, USA: Academic Press Professional, Inc., 1996.

[15] J. Rada-Vilela, M. Zhang, and W. Seah, "A performance study on synchronous and asynchronous updates in particle swarm optimization," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '11. New

York, NY, USA: ACM, 2011, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/2001576.2001581

[16] A. Engelbrecht, "Particle swarm optimization: Velocity initialization," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, June 2012, pp. 1–8.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[18] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[19] R. Storn and K. Price, "Differential evolution &ndash; a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, Dec. 1997. [Online]. Available: http://dx.doi.org/10.1023/A:1008202821328

[20] K. V. Price, "New ideas in optimization," D. Corne, M. Dorigo, F. Glover, D. Dasgupta, P. Moscato, R. Poli, and K. V. Price, Eds. Maidenhead, UK, England: McGraw-Hill Ltd., UK, 1999, ch. An Introduction to Differential Evolution, pp. 79–108. [Online]. Available: http://dl.acm.org/citation.cfm?id=329055.329069

[21] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation," in *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, May 1996, pp. 312–317.

[22] M. M. Ali and A. Törn, "Population set-based global optimization algorithms: Some modifications and numerical studies," *Comput. Oper. Res.*, vol. 31, no. 10, pp. 1703–1725, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1016/S0305-0548(03)00116-3

[23] J. Brest, V. Zumer, and M. Maucec, "Self-adaptive differential evolution algorithm in constrained real-parameter optimization," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 215–222.

[24] M. Omran, A. Salman, and A. Engelbrecht, "Self-adaptive differential evolution," in *Computational Intelligence and Security*, ser. Lecture Notes in Computer Science, Y. Hao, J. Liu, Y. Wang, Y.-m. Cheung, H. Yin, L. Jiao, J. Ma, and Y.-C. Jiao, Eds. Springer Berlin Heidelberg, 2005, vol. 3801, pp. 192–199. [Online]. Available: http://dx.doi.org/10.1007/11596448_28

[25] A. Qin and P. Suganthan, "Self-adaptive differential evolution algorithm for numerical optimization," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 2, Sept 2005, pp. 1785–1791 Vol. 2.

[26] D. E. McCarney, S. Houghten, and B. J. Ross, "Evolutionary approaches to the generation of optimal error correcting codes," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: ACM, 2012, pp. 1135–1142. [Online]. Available: http://doi.acm.org/10.1145/2330163.2330320

[27] A. Engelbrecht, "Fitness function evaluations: A fair stopping condition?" in *Swarm Intelligence (SIS), 2014 IEEE Symposium on*, Dec 2014, pp. 1–8.

[28] F. Van Den Bergh, "An analysis of particle swarm optimizers," Ph.D. dissertation, Pretoria, South Africa, 2002, aAI0804353.