On the Synthesis of DNA Error Correcting Codes

Daniel Ashlock\(^a\), Sheridan K. Houghten\(^b\), Joseph Alexander Brown\(^c\), John Orth\(^d\)

\(^a\)Department of Mathematics and Statistics
University of Guelph
Guelph, Ontario, Canada
N1G 2W1 dashlock@uoguelph.ca

\(^b\)Department of Computer Science
Brock University
St. Catharines, Ontario, Canada
L2S 3A1 houghten@brocku.ca

\(^c\)Joseph Alexander Brown
School of Computer Science
University of Guelph
Guelph, Ontario, Canada
N1G 2W1 jb03hf@gmail.com

\(^d\)John Orth
Department of Computer Science
Brock University
St. Catharines, Ontario, Canada
L2S 3A1 jo05bi@brocku.ca

Keywords: Evolutionary Computation; Bioinformatics; Representation; Error Correcting Code; Nucleic Acids; Next-Gen Sequencing.

1. Introduction

An error correcting code is a collection of strings over a given alphabet that are well separated from one another. The separation property means that small numbers of errors in transmission of a code word can be both detected (by noting that the word received is not a code word) and corrected (by assuming the code word transmitted is the one most similar to the one that was received). In this study we are creating a code where the transmission channel consists of incorporating the code word, in the form of an oligonucleotide, as a label in a genetic construct and later reading it out when the entire construct is sequenced. In this situation the Hamming metric [15], which only counts substitutions, is inappropriate because sequencers can skip a base or read one that is not there. This means that codes relative to the edit metric or Levenshtein distance [17] are required for the detection and correction of sequencing errors.

The edit distance between two words is the smallest number of single character substitutions, insertions, or deletions that can transform one string into the other. The edit metric is, in fact, a metric in the mathematical sense [15]. Edit metric codes are startlingly unlike codes over the Hamming metric [13]. Very little of the beautiful algebraic theory of Hamming metric codes applies to edit codes. While the edit metric
permits code words with variable length, we will, for reasons of simplicity in both coding and biological applications, work with code words of a fixed length.

An \((n, M, d)\)-code is a code with \(M\) members whose words are of length \(n\) with pairwise minimum distance between code words of \(d\). In fact, in general, only a very small number of additional code words may be obtained by permitting word length to vary [8]. In essence, the longer words are spaced out from one another more and so shorter words obstruct a far larger share of the edit-metric space than longer words.

The first application of DNA-error correcting codes [19] used the code words as embedded tracking labels in an expressed sequence tag (EST) project in \textit{zea mays} (corn). The biochemistry used in the preparation of the genetic constructs used in an EST project limited the codewords to a length of six. Since more than twenty different genetic libraries were pooled, this meant that a \((6,M,3)\)-code was the inevitable choice. Such a code corrects \(3-1=2\) errors, a number that was sufficient to recover 50% of the data in which sequencing errors had made the source of the sequenced EST otherwise unreadable. The advent of next-gen sequencing, with runs that obtain millions of sequence reads, forms a new application domain for application of error-correcting embedded sequence tags. This new application domain motivates this manuscript which pulls together diverse research on the creation of error correcting tags for genetic constructs. The maximum size of a code of a given length and able to correct a given number of errors is a critical planning variable if such tags are to be incorporated in a sequencing project. As before, error correcting tags will enhance the recovery of information about the source of given sequences.

The first studies on edit metric error correcting codes used a fitness function with exponential time complexity [6]. This algorithm solved an immediate biological problem but is computationally unsatisfying. This high time complexity was acceptable only because a relatively short code (of length 6 and able to correct a single error) was required. The algorithm used to find these codes operates by evolving small collections of code words called \textit{code seeds} that are completed to a full code with a greedy algorithm. If we view the greedy algorithm as a type of optimization, then this algorithm is a Baldwinian one. The technical details of this initial algorithm for finding edit metric codes appears in [6]. The fitness of a code seed is the size of the code located by the greedy algorithm starting with the seed. This definition of fitness, resulting code size, is retained in the current study.

In general, when we fix the length of the code words and the number of errors that must be corrected, codes are better if they are bigger (contain more code words). To understand why bigger codes are better, consider the application of tagging biological constructs. A larger code provides more available labels at a given level of error correction and (word length based) difficulty of synthesis.

In [2] the authors established that the crossover operator in the original (code seed representation) evolutionary algorithm was actually counterproductive and demonstrated enhanced performance with a crossover-free evolution strategy. The lack of scalability of the code-seed representation led to the development of a novel binary variation operator, one that is not really a crossover operator. This new operator incorporates aspects of both crossover and mutation and exploits the fact that the order in which words are fed into the \textit{lexicode algorithm} (Algorithm 1) has a substantial impact on the size of the resulting code. In a preliminary study [1] a number of forms of this variation
operator were explored. The one exhibiting best performance, a modified evolution strategy (ES), is used in this study. It applies the new variation operator to code lengths it had not been tested on previously. In [7] the new variation operator was tested in a geographically structured evolutionary algorithm called a ring optimizer [4, 5]. The standard algorithm exhibited superior performance. In this study the ES is run for longer times on a broad variety of code parameters and improves the known bound for thirteen different sets of code parameters.

The most recently developed algorithm for search for DNA error correcting codes is the Salmon algorithm [18], inspired by the spawning behavior of salmon. This algorithm is similar to ant colony optimization[14], except that the algorithmic analog of pheromones does not evaporate. The algorithm is configured to search for cliques in a graph with vertices that are strings in the DNA alphabet and edges where pairs of words meet the minimum distance restriction for a code. In [18] the algorithm’s performance was optimized on a single set of code parameters. In this study we run the algorithm on several sets of code parameters. The algorithm is memory intensive and so most useful to perform a thorough search for codes with relatively short word length. Since these words are synthesized into biological constructs [19], codes with short word length are the most useful in applications. The algorithm improves the best known code size for five different code parameters.

The remainder of this study is structured as follows. Section 2 gives mathematical background that permits us to leverage the experimental information used to build the table of best code sizes. Section 3 surveys the techniques used to locate large codes over the DNA alphabet. Section 4 presents new best code sizes and a table of best-known code sizes. Section 5 gives application notes and discusses the results and future directions.

2. Mathematical Background

Readers only interested in the application of DNA error correcting codes may safely skip this section. It is included because it is needed to justify many of the entries in Table 3 which gives the best possible sizes of codes for various lengths and minimum distances.

2.1. Preliminaries

Recall that an \((n, M, d)\) edit code is a \(q\)-ary code consisting of \(M\) codewords each of which has length \(n\), and in which all codewords are at edit distance at least \(d\) from each other. This study focuses primarily on DNA codes for which \(q = 4\). Define \(E_q(n, d)\) to be the maximum \(M\) for which there is an \((n, M, d)\) edit code. Such a code is optimal if \(M = E_q(n, d)\).

2.2. Automorphism Group and Equivalence

The automorphism group of a \(q\)-ary edit code is described in [16]. In that paper words are described in terms of their block structure. A block is defined as a maximal run of a single character within a word. It is shown that the automorphism group is dependent on the block structure of words, with the only allowed operations being
to simultaneously permute all symbols of all words, or to simultaneously reverse all words. Two codes $C_1$ and $C_2$ are \textit{equivalent} if one can be transformed into the other by any combination of these operations. That paper also establishes several relationships between different types of codes and between different parameter sets.

2.3. \textit{Bounds on Sizes of Codes}

The results in this section fill in gaps in the table of best known code sizes by using knowledge of the size of a code for one parameter to estimate the size for another parameter.

2.3.1. \textit{Relationships to Other Types of Codes}

\textbf{Definition 1.} The insertion-deletion distance between two strings is the minimum number of insertions and deletions needed to transform one string into another. Insertion-deletion codes are codes that use the insertion-deletion distance.

\textbf{Observation 1.} Since edit distance encompasses insertion-deletion distance, the number of codewords $M$ in an $(n, M, d)_q$ edit code cannot exceed that in an optimal insertion-deletion correcting code of the same parameters.

\textbf{Observation 2.} Since edit distance encompasses Hamming distance, the number of codewords $M$ in an $(n, M, d)_q$ edit code cannot exceed that in an optimal Hamming distance code of the same parameters.

An implication of this observation is that one can use Hamming distance, which is faster to compute than edit distance, as a bounding function in the computation of edit distance. If two words are at Hamming distance $< d$ apart, then they are also at edit distance $< d$ apart. The algorithms implemented in this study speed up computation by finding the Hamming distance first, then computing edit distance only if Hamming distance is acceptable.

2.3.2. \textit{Relationships between Parameter Sets}

\textbf{Observation 3.} Since we can always create an $(n + 1, M, d)_q$ code by adding a single letter to the end of each word in an $(n, M, d)_q$ code, we have $E_q(n + 1, d) \geq E_q(n, d)$. This process is called lengthening. An implication of this observation is that the columns of Table 3 are nondecreasing.

\textbf{Theorem 1.} $E_q(n, d) \leq q \cdot E_q(n - 1, d)$.

\textit{Proof:} For each symbol $0 \leq i \leq q - 1$, take an optimal $(n - 1, M, d)_q$ code and lengthen it by inserting the symbol $i$ at the start of each codeword. Call the resulting codes $C_i$, $0 \leq i \leq q - 1$.

By taking all codewords from $C_0, \ldots, C_{q-1}$, we have an upper bound on the number of codewords in an optimal $(n, M, d)_q$ code $C$. If this were not the case, then at least one of the subcodes of $C$ obtained by selecting all codewords of $C$ starting with the same character would be larger than an optimal $(n - 1, M, d)_q$ code, which is a contradiction. Therefore, $E_q(n, d) \leq q \cdot E_q(n - 1, d)$. $\square$
2.3.3. Known values of $E_q(n, d)$

Observation 4. If $d > n$ then $E_q(n, d) = 1$.

Observation 5. If $n = d$ then $E_q(n, d) = q$.

Observation 6. The set of all $q$-ary vectors of length $n$ forms a code with minimum distance $d = 1$. Therefore $E_q(n, d) = q^n$ if $d = 1$.

Theorem 2. If $d = 2$ then $E_q(n, d) = q^{n-1}$.

Proof: From Theorem 1, $E_q(n, 2) \leq q \cdot E_q(n-1, 2)$. Also, by taking all words of length $n - 1$ and appending a parity check that is the sum of all previous columns mod $q$, we obtain a $(n, M, 2)_q$ code with $q^{n-1}$ codewords. Thus we have $E_q(n, 2) = q \cdot E_q(n-1, 2)$. $\square$

Figure 1: Mean and best fitness in an evolutionary run searching for length eight distance five codes. The algorithm uses a population of 500 with a size three seed representation and both crossover and mutation.

3. Survey of Techniques

Many of the techniques for finding edit metric error correcting codes incorporate some version of Conway’s lexicode algorithm, Algorithm 1. Informally, this algorithm functions as follows. Place the DNA words of length $n$ in lexicographical order. Initialize an empty set $C$ of words. Scanning the ordered collection of words, select a
word and place it in $C$ if it is at edit distance $d$ or more from every word already placed in $C$. The original application of the algorithm is entirely theoretical [12]. In a space with regular structure, such as the Hamming space, theorem and proof techniques are superior for the location of codes. It is the highly irregular structure of edit metric spaces [13] that make the use of Conway’s algorithm and its variations necessary.

**Algorithm 1. Conway’s Lexicode Algorithm**

**Input:** An alphabet $A$, a minimum distance $d$ and an ordered subset of $S \subset A^n$.

**Output:** $CLA(S)$, a subset of $S$ that has pairwise minimum distance $d$.

**Details:**

Initialize an empty set $R$ of words.

Traverse the members $s \in S$ in order

If $s$ is at least distance $d$ from every member of $R$,
add $s$ to $R$

Return $R$ as $CLA(S)$.

<table>
<thead>
<tr>
<th>L/d</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>256</td>
<td>64</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1024</td>
<td>256</td>
<td>36</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>4096</td>
<td>1024</td>
<td>96</td>
<td>20</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>16384</td>
<td>4096</td>
<td>311</td>
<td>57</td>
<td>14</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>$4^7$</td>
<td>$4^7$</td>
<td>1025</td>
<td>164</td>
<td>34</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>$4^7$</td>
<td>$4^9$</td>
<td>3451</td>
<td>481</td>
<td>90</td>
<td>25</td>
<td>10</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>$4^{10}$</td>
<td>$4^9$</td>
<td>11743</td>
<td>1463</td>
<td>242</td>
<td>57</td>
<td>17</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>$4^{14}$</td>
<td>$4^{10}$</td>
<td>40604</td>
<td>4574</td>
<td>668</td>
<td>133</td>
<td>38</td>
<td>13</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>$4^{12}$</td>
<td>$4^{14}$</td>
<td>*</td>
<td>*</td>
<td>1894</td>
<td>338</td>
<td>79</td>
<td>28</td>
<td>12</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>$4^{13}$</td>
<td>$4^{12}$</td>
<td>*</td>
<td>*</td>
<td>5517</td>
<td>879</td>
<td>188</td>
<td>54</td>
<td>20</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

* - run time too long.

Table 1: Size of the codes resulting from running Algorithm 1

3.1. Code-seed Based Algorithms

The earliest work on DNA error correcting codes, appearing in [6, 19] and revisited in [2], was all performed with an evolutionary algorithm that used a modified version
of Algorithm 1 as a fitness function. Conway’s algorithm completes an initial empty set of words to form a code. We call these codes the Conway edit codes and Table 1 gives examples of sizes for these codes. The modification used is to start with a seed set of words rather than an empty set of words. The evolutionary algorithm then evolves this seed with fitness equal to the size of the code returned by the seeded Conway’s algorithm. This code seed consists of a set of three words of the proper length. If two code words in the seed violate the distance bound for the code, that seed is awarded a fitness of zero.

Since Conway’s algorithm is a greedy algorithm, we called the algorithm a greedy closure evolutionary algorithm. The representation is an indirect one. The data structures making up the population do not include complete descriptions of codes. Rather, they provide a set of controls for a greedy algorithm. One of the nice properties of this representational strategy is that a very large final structure (the code) can be implicitly specified by a very small data structure (the code seed). For length six distance three codes, for example, a 3-word code seed requires 36 bits of information to specify, while a specification of which words are in the best known code requires 2,226 bits. This advantage grows with word length. For a length 11 distance 3, code the numbers are 264 bits versus 893,288.

![Figure 2: Mean and best fitness in an evolutionary run searching for length seven distance five codes. The algorithm uses a population of 500 with a size three seed representation using only mutation.](image_url)

The original algorithm used both mutation and crossover. Mutation consists of randomly replacing one of the seed words with a new word generated uniformly at random. Crossover shuffles the words in the union of two code seeds and deals them
out to parents at random, subject to the restriction that, if one word appears twice, then one copy of it goes to each parent. The reason for revisiting the algorithm in [2] was to investigate the hypothesis that the crossover operator was worse than useless. The study confirmed the hypothesis: without crossover the performance of the algorithm is substantially superior.

The reason that the crossover operator was so ineffective is best explained with the concept of infertility. In an evolving population we say that two members of the population are fertil when there is a high probability that the possible results of crossing them over have fitness high enough to survive. If this probability is low the two members of the population are infertile. Figure 1 illustrates the phenomenon. The algorithm is elitist and so the best fitness increases in a slow stair-step fashion. The mean fitness, however, plummets between the first and second generation. This is because crossover yields a large number of seeds containing two words that violate the code’s minimum distance and so have zero fitness. For comparison, Figure 2 shows a similar run performed without crossover. The stair-stepped best fitness is still present but the huge initial drop in fitness is not present.

Algorithm 2. Modified Evolution Strategy

```
Create a random initial population of code seeds
Loop(Generation Many Time)
  Sort the population
  Loop over less-than-maximum-fitness seeds
    Select at random from seeds with maximum fitness
    Copy the max-fit seed over a less than max-fit seed
    Replace one word in the copy
    Evaluate fitness of the copy
  End Loop
End Loop
Report best result
```

The crossover-free algorithm developed in [2] is a form of modified evolution strategy (ES) [9]. It is given as Algorithm 2. In a standard ES, a new generation of structures is created by generating variations of a set of best structures in the current population of fixed size, often the single current best structure. The typical applications of ES are to real optimization where ties in fitness values are rare making “best” a usually unambiguous notion. The fitness values here, code sizes, are integers drawn from a small range. We therefore modify the standard ES by generating variations of all maximal fitness members of the current generation. A comparison of this ES with the standard seed-based algorithm with crossover appears in [2].

3.1.1. Seed Size

In the initial studies a seed size of three was used in all the algorithms. This was based on a few unpublished preliminary studies on seed size. We provide here data
and a discussion of the issue of seed size. The starting point of the discussion is the following theorem, proved in [2].

**Theorem 3.** Suppose \( k_1 < k_2 \). Then all codes that can be located using seeds of size \( k_1 \) can be located with codes of size \( k_2 \).

This theorem, while true, ignores a critical issue related to the fitness landscape of the search. The apparent implication of the theorem is “longer seeds are no worse and may, in fact, be better.” Since Table 3 contains dozens of examples of beating the size of the Conway code, which uses seed size zero, it follows that larger seeds are better in many cases. Experimental testing in [2] strongly suggests that good seed size varies in a manner idiosyncratically dependent on the code parameters. Table 2 demonstrates that the probability that a random seed is fertile drops off with seed size. This probability is likely to be proportional to the probability that changing one of the words in a code seed will transform it into an invalid seed. Putting all this together, it is clear that some seed size bigger than zero is best for evolutionary search. What evidence is available suggests that the optimal seed size grows slowly with code size.

<table>
<thead>
<tr>
<th>Seed Length</th>
<th>Length 7 Distance 3</th>
<th>Length 6 Distance 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.000 ± 0.000</td>
<td>1.000 ± 0.000</td>
</tr>
<tr>
<td>2</td>
<td>0.982 ± 0.008</td>
<td>0.956 ± 0.012</td>
</tr>
<tr>
<td>3</td>
<td>0.951 ± 0.013</td>
<td>0.883 ± 0.019</td>
</tr>
<tr>
<td>4</td>
<td>0.903 ± 0.017</td>
<td>0.745 ± 0.023</td>
</tr>
<tr>
<td>5</td>
<td>0.861 ± 0.020</td>
<td>0.640 ± 0.024</td>
</tr>
<tr>
<td>6</td>
<td>0.775 ± 0.023</td>
<td>0.502 ± 0.022</td>
</tr>
</tbody>
</table>

Table 2: This table gives the probability that a random code seed of size \( k \) will not have two words at less than distance D (will have nonzero fitness).

If larger seeds are able to potentially find better codes but also have a higher probability of being invalid there is the question of which effect is stronger. To test the relative strength of the two phenomena, a set of 10,000 random valid sample seeds were generated for length seven distance three codes; seeds were generated at random until 10,000 valid ones were located. This was done for seeds of size two, three, and four. Figure 3 shows the distribution of resulting code sizes for these seeds. At the level of resolution of the figure, the three distributions are indistinguishable. This suggests that the benefit of using larger code sizes is very small compared to the infertility effects.
Algorithm 3. Conway Variation Operator

**Input:** One or more \((n, M, d)\)-codes \(C_1, \ldots, C_k\), a random material rate \(R \geq 0\) (an integer), and a minimum distance \(d\).

**Output:** An \((n, M, d)\) code.

**Details:**

Let \(Q\) be the union of \(C_1, \ldots, C_k\).

Generate \(R\) random words and add them to \(Q\).

Shuffle the set \(Q\) into a random order.

Apply Algorithm 1 to \(Q\), returning \(CLA(Q)\).

Figure 3: Shown are the distribution of codes sizes derived from 10,000 samples for length seven distance three codes for seeds of length 2, 3, and 4.
3.2. Conway Variation Operator Based Algorithms

The great weakness of the code seed algorithm, even without crossover, is that the fitness function has to traverse the entire set of $4^n$ words in the string space. This means the algorithm time is exponential in the length of the code words. For codes of length $\leq 9$ the algorithm can function. For longer codes, a different algorithm needed to be designed. It also uses a variation on Conway’s lexicode algorithm retooled into a variation operator: Algorithm 3. The Conway variation operator operates by taking the union of two codes, adding several randomly generated words to that union, shuffling the resulting set of words randomly, and then extracting a new code by applying Conway’s lexicode algorithm to the words in their shuffled order. The operator thus stays within type (valid $(n, M, d)$-codes). The addition of random words introduces words not used before into the process and so plays a role similar to that of the mutation operator in a more standard representation.

Most variation operators can be classified as crossover-like (binary) or mutation-like (unary). The Conway variation operator incorporates elements of both. The number of random words added during application of the Conway variation operator is called the random material rate. The random material rate is similar to the rate of a mutation operator. In addition a single code, two codes, or more than two codes could be shuffled together before Conway’s algorithm is applied, meaning that there are natural unary, binary, trinary, and other forms of the Conway variation operator.

The Conway variation operator was introduced and tested in both its unary and binary forms in [1]. The test found that performance of the Conway variation operator was roughly comparable to, but worse than, the code-seed algorithm for code parameters where both could be used. The study also found the binary form of the operator to be superior to the unary form. The primary reason for using the Conway variation operator is that it can be used to rapidly find suboptimal codes for word lengths much longer than those on which seed-based algorithms can operate. This means that the Conway variation operator based algorithms are no use in improving Table 3 but may have substantial utility in designing codes for application. The discussion of this subject continues in Section 5.

3.3. The Salmon Algorithm

We now describe a special purpose algorithm that was used to improve the bounds on the size of $(n, M, 3)$-codes for $n = 6, 7, 8$ and $(n, M, 4)$-codes for $n = 7, 8$. This algorithm was first presented and optimized for a single set of code parameters in [18]. The current study applies the algorithm to several additional sets of code parameters, obtaining new bounds on the maximum size of codes. A salmon algorithm is a population-based path finding algorithm that is inspired by the behavior of salmon swimming upstream to spawn. The algorithm was originally conceived as a solution for the traveling salesman problem. Described here is the clique finding version of the algorithm that can be used to find codes. The salmon algorithm combines concepts found in two other metaheuristics: evolutionary algorithms and artificial ant algorithms. From EA’s we borrow the idea of fitness based selection and inheritance. From artificial ant algorithms we borrow the idea of preference for using components that have been part of previous good solutions.
Algorithm 4. The Salmon Algorithm

\textbf{Input:} A desired length \( n \), distance \( d \), retention fraction \( \phi \), reproductive fraction \( \sigma \), population size \( P \).

\textbf{Output:} An \((n, M, d)\) − code.

\textbf{Details:}

Precompute the distance \( d \) compatibility graph \( G \) for words of length \( n \).

Initialize a flow vector indexed by \( V(G) \) to a constant value

Generate \( P \) salmon, each containing a clique

\textbf{Repeat Indefinitely}

Sort salmon by their clique size.

The top \( \sigma \) fraction of salmon reproduce

The best salmon is copied into the new population

New salmon probabilistically retain a \( \phi \) fraction of their parent’s clique.

Each salmon adds clique compatible vertices to its clique, choosing in proportion to flow.

Flows are updated: each salmon adds its clique size to the flow of the vertices in its clique.

\textbf{End Repeat}

Report best clique as the output code.

The algorithm begins by creating a population of salmon and a set of flow values on a combinatorial graph. The vertices of this graph are words of length \( n \) over the DNA alphabet. Two words are adjacent in the graph only if they satisfy the minimum edit distance \( d \) for the desired code. This means that \((n, M, d)\)-codes are cliques in this graph. The flow values are stored in an array indexed by all the vertices, where the value at each vertex equals the amount of flow through that vertex. Flow does not have a direction. The algorithm initializes all flows to some positive value.

Each salmon consists of two lists. The first is a tabu list, which is a list of the vertices that have been added to the current clique under construction. The second is a memory list, which is a copy of the completed tabu list from the salmon’s parent. There is also a candidate list which is associated with each salmon, but not part of the partial clique constructed so far. The candidate list contains all vertices that have not been added to the clique which are connected to all vertices currently in the clique. (They are candidates for addition to the clique.) To begin the algorithm a randomly selected clique is placed in each salmon’s memory list.

New salmon generation happens in the following fashion. Each salmon builds a new clique. First, each vertex from the salmon’s memory list (its parent’s clique) is added to the tabu list with some probability \( \phi \). The value \( \phi = 0.8 \) is used in this study. Next we add vertices by selecting from the candidate list using fitness proportional selection on the flow values. The salmon continues adding vertices until the candidate list is empty. Vertices may leave the candidate list both by being transferred into the salmon’s clique or when a vertex that is not adjacent to a candidate is added to the salmon’s clique.
Table 3: Shown are the best known values of \( E_4(n, d) \), code sizes for DNA edit metric codes with distance \( d \geq 3 \). Values improved in this paper with a novel algorithm are shown in bold. The italicized values are those improved because a parameter above them improved enough to raise it or because a new Conway code was computed.

<table>
<thead>
<tr>
<th>n \ d</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>44</td>
<td>11</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>114</td>
<td>28</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>353</td>
<td>65</td>
<td>18</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1119</td>
<td>176</td>
<td>38</td>
<td>14</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>3451</td>
<td>495</td>
<td>97</td>
<td>27</td>
<td>11</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>11743</td>
<td>1463</td>
<td>249</td>
<td>60</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>40604</td>
<td>4574</td>
<td>676</td>
<td>139</td>
<td>41</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>40604</td>
<td>4574</td>
<td>1894</td>
<td>338</td>
<td>79</td>
<td>30</td>
<td>13</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>40604</td>
<td>7744</td>
<td>5517</td>
<td>879</td>
<td>188</td>
<td>54</td>
<td>13</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>40604</td>
<td>15206</td>
<td>5517</td>
<td>879</td>
<td>188</td>
<td>54</td>
<td>14</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>40604</td>
<td>17780</td>
<td>6419</td>
<td>880</td>
<td>196</td>
<td>61</td>
<td>22</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>40604</td>
<td>18308</td>
<td>9220</td>
<td>1784</td>
<td>333</td>
<td>102</td>
<td>37</td>
<td>11</td>
<td>4</td>
</tr>
</tbody>
</table>

The flow network is updated after the salmon have finished building their cliques. Each salmon adds an amount of flow to each vertex in its clique which equals the size of its clique. Thus, vertices that are part of large cliques will accumulate more flow than those that are part of small cliques. This is similar to the way an ant deposits pheromone. An important difference is that pheromones evaporate while flow numbers do not.

Once the salmon’s cliques and flow numbers have been updated, the salmon are then sorted based on clique size. The most fit fraction \( \sigma \) of salmon spawn and produce offspring. The value \( \sigma = 0.5 \) was used in this study. Reproduction of salmon is crossover-free. The tabu list of the parent is copied to the memory list of the child. The algorithm is elitist with the most fit salmon copied into the new generation unmodified.

The critical area of the salmon algorithm in terms of run time complexity is the roulette selection of candidate vertices. The number of vertices that must be selected this way is \((1 - \phi) \times C_l\), where \( C_l \) is the size of the codeword clique. If the number of candidate vertices is \( C_\alpha \), then we will traverse \( \frac{C_\alpha}{C_l} \) of them on average before selecting each one. Thus, the order of the algorithm is \( O(C_\alpha \times C_l) \). There is no way of telling, a priori, how much \( C_\alpha \) and \( C_l \) will increase for each gain of one in codeword length.

Experimentally we can see that the size of the cliques increases by a factor of about 3.1 for each added letter. If the candidates list increases by the same amount, we would
expect the run time to increase by a factor of about 10 for each additional letter. This is very close to the actual results. Average run time for 1000 generations on the (6,3) case was 435 seconds. For (7,3) the time was 5000 seconds, and for (8,3) it was 52,000 seconds. The increase in run time per generation is exacerbated by the fact that the larger cases also require a greater number of generations to converge (2500 for length 6 vs. 5000+ for length 8). This exponential run time increase is the reason the salmon algorithm was not used on any case where word length exceeded 8.

The salmon algorithm is the source of the current best (6,3), (7,3), (8,3), (7,4) and (8,4) codes. Because it requires precomputation of a graph with \(4^n\) nodes, the algorithm does not scale well. A careful parameter study for one set of code parameters, length six, distance three, was performed in [18].

4. New Results and Table of Code Sizes

Table 3 summarizes the current state of knowledge for the largest known code for distances \(3 \leq d \leq 11\) and lengths \(1 \leq n \leq 16\). We now give the origin of improvements over the most recent table of best values published in [7].

- The salmon algorithm is responsible for improvements from \(M = 108\) to 114, from \(M = 329\) to 353, and from \(M = 1025\) to 1119 for codes of distance three and length 6,7, and 8 respectively. It also improved distance four codes of length 7 and 8 from \(M = 63\) to 65 and \(M = 164\) to 176 respectively. These improvements are in the most difficult parts of the table.
- Running the ES (crossover free) algorithm in production runs yielded improvements in the form of the following codes: \((9, 495, 4), (9, 97, 5), (9, 27, 6), (9, 5, 8), (10, 249, 5), (10, 60, 6), (10, 20, 7), (11, 676, 5), (1, 139, 6), (11, 41, 7), (11, 16, 8), (12, 30, 8), and (12, 13, 9).\
- Extension of the table of known Conway codes together with Observation 3, improving on mathematically derived bounds, is responsible for improvements at \(d = 3, n = 11, 12, 13, 14, 15, 16, d = 4, n = 11, 12, d = 5, n = 12, 13, 14, d = 6, n = 12, 13, 14, d = 7, n = 12, 13, 14, d = 8, n = 13,\) and \(d = 9, n = 13.\)

While the salmon algorithm provided improvements in the most studied and most useful set of cases, short codes capable of correcting a single error, the largest number of improvements came, in a sense, from the action of Moore’s law. In the several years since we last attempted to compute the size of various Conway codes, computers have improved and so 21 bounds improved by simply reattempting single runs of Conway’s lexicode algorithm for large code spaces. The ES-algorithm continues to yield improvements in the middle of the table.

5. Application Notes and Next Steps

In this study we report best known code sizes for codes whose lengths and distances encompass those values useful for providing embeddable error-correcting tags
for genetic constructs. We do not provide actual codes for a number of reasons. First, print is not the correct medium for transferring this kind of data. Beyond that, however, applications typically cannot use generic codes - those that have a correct length and minimum distance but which may not satisfy other biochemical requirements.

In [19], a code of length 6 and minimum distance 3 was used successfully in an EST project. The DNA tags were embedded at a specific point in a genetic construct shown in Figure 4. The biochemistry used to make the construct placed the following restrictions on code words:

1. No code word may end in T.
2. No code word may contain TT or AAA.
3. After the code word is incorporated into the construct, the genetic sequences GAATTC and GCGGCCGC must not appear.

The last rule corresponds to the need to avoid active sites for restriction enzymes and so can, to some extent, be chosen when a project is planned.

Incorporating restrictions like those given above into the search algorithms is easy. The salmon algorithm becomes faster because the excluded words would be removed during pre-computation of the underlying graph, reducing the effective problem size [18]. For the algorithms that incorporate some version of Conway’s algorithm, the words are excluded either from the initial alphabetized list or at the point where random material is incorporated.

Other restrictions may exist. In some cases the code words will need to have a designated GC-content. This becomes more of an issue for longer code words which have a higher internal melting temperature. A DNA-tag design tool is currently under construction and potential users are invited to send in their design requirements so that they can be considered.

In cases where the number of tags required is substantially smaller than the largest known code, the Conway crossover operator based algorithm is the natural choice. The algorithm is modified by incorporating a check for satisfaction of additional biochemical constraints in the all DNA words used by the algorithm. This small modification will permit the algorithm to generate codes that satisfy the constraints. In comparison to the other algorithms, only this one does not need to traverse or enumerate the full space of words and, so, can locate usefully large suboptimal codes rapidly.

5.1. Decoding

After embedding DNA-tags into a collection of genetic constructs and subsequently sequencing those constructs, the tags must be located and, if there are errors, decoded. The first foray into application of error-correcting DNA tags used a (6,76,3)-code that obeys all the restrictions listed in the previous section. A distance-three code can correct one error. Tags were located within the construct by aligning the sequencing output with the non-tag structure of the construct. If the sequence in the tag’s position is one
of the tags used, no decoding is required. If it is not, then decoding was performed by searching for a tag in the code at edit distance one from the sequence observed in the tag position. If such a tag exists, then that tag is the decoding result. If no such tag exists, then the sequenced tag is not decodable. We call this an exhaustive decoding algorithm.

If the code has a short length, then exhaustive decoding is practical. The enormously higher throughput available with next-gen sequencing will require longer codes able to correct more errors. This means that more sophisticated decoding is required. A start on finding better decoding techniques appears in [11, 10].

The principal technology used in decoding edit-metric codes are side effect machines (SEMs) [3]. A side effect machine generalizes a finite state machine by adding a counter to each state and removing the idea of accepting and denying states. A tag to be decoded is run through a side effect machine. The pattern of counts that results is a vector of numbers. The side effect machine is thus a function from strings to numerical vectors. Once this transformation is made, simple clustering or machine learning algorithms can be used as fitness functions to locate an SEM that maps error patterns connected with a code word into easily recognized patterns in Euclidean space. Both direct decoding and iterative partitioning (locking side effect machines) of tags were explored and neither was found to be superior in all cases.

The first two studies[11, 10] do not solve the problem of decoding, but they make substantial progress. Efficient decoding of DNA-tags remains an area of active research. Improved decoding techniques together with design of applied tools to smoothly design DNA error-correcting tags that meet user-specified biochemical constraints are the obvious next steps in this research program.

References


