CodeGen: The Generation and Testing of DNA Codewords

David E. Kephart
Department of Mathematics
University of South Florida
Tampa, FL 33620
Email: dkephart@mail.usf.edu

Jeff LeFevre
Department of Engineering
University of South Florida
Tampa, FL 33620
Email: jlefevre@ieee.org

Abstract—With this paper we present algorithms to generate and test DNA code words that avoid unwanted cross hybridizations. Methods from the theory of codes based on formal languages are employed. These algorithms are implemented in user-friendly software, CodeGen, which contains a collection of language-theoretic objects adaptable to various related tasks. Lists of code words may be stored, viewed, altered and retested. Implemented in Visual Basic 6.0, its interface allows for lists of code words to be assembled at varying levels of acceptability from a single main window.

I. INTRODUCTION

A. Motivation

DNA offers us the prospect of massive parallelism in computing. A few basic obstacles to this, however, must be overcome. The simple property of Watson-Crick complementarity, which allows the precise chemical matching of a denatured, single-stranded DNA molecule, or oligonucleotide with an oppositely-oriented complementary strand allows the transfer of information it contains. By assembling particular strands and encoding a problem in these strands, it has been shown that certain NP-complete problems may be solved. This requires identification of specific strands in the resultant soup by renaturization of the DNA in the presence of their complements.

This property would compromise the results of a DNA program if the encoding is selected carelessly. The information conveyed in such strands will be destroyed if they bond (hybridize) with other code words. DNA molecules move about extremely rapidly. Hybridization that can occur during the renaturization process will occur. If this is possible between the encoding sequences then it will happen and the desired computation will not take place.

Further, not all DNA nucleotides are born chemically equal. The higher the percentage of cytosine and guanine in a strand, the lower its hybridization temperature. Without their presence, however, the number of useable distinct code words of any given length must fall. Finally, the piecemeal search for potential unwanted hybridizations between code words demands exponentially increasing amounts of computation time.

This paper presents algorithms which use a theorem from coding theory and language-theoretic properties to address this dilemma for the DNA programmer. These have been implemented in software in a program CodeGen which performs the various tests necessary in polynomial time. In addition, it generates or verifies these properties in a language from a user-friendly interface and enables the storage of languages. It is the extension and overhaul of software presented in [12].

While the program selects DNA settings by default, the interface and object-oriented implementation lend themselves to more general applications.

In section II-A we define necessary terms from coding theory and present the theorem that drives the main algorithm of CodeGen. In section III we present the algorithms which vary the level of “goodness” demanded in tested code words, followed by pseudo-code for the main algorithm. In section III-G, we discuss the object-oriented library created to implement the algorithm, and pseudo-code for a typical algorithm made possible by this approach. In section IV we show the details of the user interface. We conclude with several remarks about present limitations and future development of this project.

B. Past and present work on this problem

This view of the problem is an extension of the framework set up in [14]. The use of computers to assist in the synthetic self-assembly of DNA strands dates from the origin of the laboratory construction of such molecules. In the past, the program of choice has been that presented in [20], and analyses such as those by Winfree [21], Baum [2], Li [15], and more recently, Ari[1]. As mentioned in [12], though, these approaches generally focus on Hamming distance and avoid only inter-molecular hybridizations. They deal with code words of constant length, with the exception of Seeman’s program [20], which may pose some difficulties in being set to accomplish anything else. CodeGen is designed to handle code words of variable length and can randomly generate a set of code words without intra-molecular cross-hybridizations, and with very little initial input.

We begin by defining precisely what we mean by a “good” set of DNA code words.
II. The Problem: What is a “Good” DNA Codeword?

A. Definitions

A finite alphabet, \( \Sigma \), is a finite collection of symbols. A language is a subset of all possible words of finite length over that alphabet. We denoted by \( \Sigma^+ \) all possible words over \( \Sigma \). All possible words over \( \Sigma \) of positive length is denoted by \( \Sigma^* \), and all possible words over \( \Sigma \) of length \( n \) are denoted by \( \Sigma^n \). If there is a function \( \theta : \Sigma \to \Sigma \) with the property that \( \theta(x) = \theta^2(x) = x \) for every symbol \( a \) in \( \Sigma \), then \( \theta \) is an involution on \( \Sigma \). If, for every \( x, y \in \Sigma^* \), \( \theta(xy) = \theta(x)\theta(y) \), then \( \theta \) is a morphism, while if \( \theta(xy) = \theta(y)\theta(x) \), then \( \theta \) is an antimorphism.

Definition 1: Let \( \Delta \) represent the DNA alphabet, i.e., the set \{adenine, cytosine, guanine, thymine\}, or, more succinctly, \{\( A, C, G, T \)\}.

The component nucleotides of a DNA strand bind at one end at the 5' location on a carboxyl ring and at the other end at the 3' location. This orients the strand. Taking the 5' end to be leftmost, the sequence of nucleotides in a DNA strand is a word over \( \Delta \). Watson-Crick complementarity refers to the hydrogen bonds which form between oppositely-oriented \( A \) and \( T \) nucleotides and between oppositely-oriented \( C \) and \( G \) nucleotides. This extends to entire strands, so we have the following.

Definition 2: Let \( \rho : \Sigma^* \to \Sigma^* \) be the extension to \( \Sigma^* \) of the mapping such that \( \rho : A \to T \), \( \rho : T \to A \), \( \rho : C \to G \), and \( \rho : G \to C \). Let \( \sigma \) map a DNA word to its reverse (e.g., \( \rho(AC) = CA \)). Then \( \theta = \sigma \circ \rho \) is the antimorphic involution on \( \Delta \) representing Watson-Crick complementarity.

A collection \( X \) of words over \( \Delta^* \) which avoids unwanted cross-hybridizations between or within its members is the “good” set of codewords we are after. We define its properties in a general setting. Let \( \Sigma \) be an alphabet and let \( X \) be a subset of \( \Sigma^* \).

1) We require that \( X \) be a code, in the sense that if \( x = x_1x_2x_3\ldots x_r = y_1y_2y_3\ldots y_s \), \( x_1, x_2, \ldots, x_r, y_1, y_2, \ldots, y_s \in \Sigma \), then \( r = s \) and \( x_i = y_i \), \( 1 \leq i \leq r \).

2) The involution of an element of \( X \) should not be a proper subword of an element of \( X \). Stated formally, \( \Sigma^+\theta(X)\Sigma^* \cap X = \emptyset \) and \( \Sigma^*\theta(X)\Sigma^+ = \emptyset \). The code \( X \) is then termed \( \theta \)-compliant.

3) The involution of an element of \( X \) should not be the proper subword of the concatenation of two elements of \( X \). \( \Sigma^+\theta(X)\Sigma^* \cap \Sigma^2 = \emptyset \) and \( \Sigma^*\theta(X)\Sigma^+ \cap \Sigma^2 = \emptyset \). We then call \( X \) \( \theta \)-free.

4) No prefix or suffix of length \( k \) of an element of \( X \) should be the involution of a subword of that word which is separated from it by a number of symbols from \( m_1 \) to \( m_2 \). Then \( X \) is said to be \( \theta(m_1, k_1, k_2) \)-subword compliant. For \( k_1 \leq i \leq k_2 \) and any \( w \in \Sigma \), \( w\Sigma^\theta(w)\Sigma^* \cap \Sigma X = \emptyset \), and \( \Sigma^*\theta(w)\Sigma^+ \cap \Sigma X = \emptyset \).

5) If \( \Sigma^\theta(w)\Sigma^+ \cap \Sigma X = \emptyset \) and \( \Sigma^*\theta(w)\Sigma^+ \cap \Sigma X = \emptyset \) for \( w \) in \( \Sigma^k \) where \( \Sigma^w \Sigma \in \Sigma \), then \( X \) is said to be \( \theta \)-\( k \)-free, and \( X \). No subwords of \( X \) of length \( k \) are complements of subwords of \( X \).

6) If \( \Sigma^+ \) is replaced by \( \Sigma^* \) in 2, 3, 5, or 5, then \( X \) is said to be strictly \( \theta \)-compliant, \( \theta \)-free, \( \theta(m, k_1, k_2) \)-subword compliant, or \( \theta \)-\( k \).

If a set of code words does not have property 1, there is no way to decode sequences of code words when they are identified. Some code word sequences will “factorization” in more than one way.

If a set of DNA code words does not have property 2, then some code word will hybridize with the interior portion of another sequence in the code and neither one will be identifiable in further computation. This is shown in Figure 1(a).

![Fig. 1. Intermolecular Cross-Hybridizations](#)

If a set of DNA code words does not have property 3, then some code word will “glue” two other code words together and destroy all three for purposes of computation. This is shown in Figure 1(b).

![Fig. 2. Intra-Molecular Cross-Hybridizations](#)

If a DNA code does not have property 4, with values \( k, m_1 \), and \( m_2 \) dependent on the specific conditions of the computation, then some sequences can form “hairpins” as shown in Figures 2(a) and (b). These strands are then rendered useless.

![Fig. 3. Cross-Hybridizations Avoided by \( \theta \)-\( k \) Codes](#)

Finally, with property 5, a DNA code avoids the above and various other cross-hybridizations, as shown in Figure 3. In particular, as proven in [13], if \( X^2 \) is \( \theta \)-\( k \), and there are no words in \( X \) with fewer than \( k \) symbols, then no cross-hybridizations in \( X \) or \( X^* \) are possible.

We now require some definitions from coding theory.
Definition 3 (Flower): The flower automaton of a language \( X \) over a finite alphabet \( \Sigma \) is the finite state automaton \( \mathcal{F}(Q, I, T, E) \) where \( I = T = \{0\} \subset Q \) and \( E \subset Q \times \Sigma \times Q \) which recognizes \( L \) in the following sense. If \( w \) is a word in \( L \) of length \( s \), there is a path \( \pi = e_1 e_2 \cdots e_s \), \( e_1, \ldots, e_s \subset E \) in \( \mathcal{F} \) such that \( e_i = (q_i, w_i, q_{i+1}) \), \( w_i \) is the \( i \)th symbol of \( w \), and \( q_1 = q_{s+1} = 0 \), but \( q_i \neq 0 \) if \( 1 < i \leq s + 1 \).

We say that \( w_i \) is the label of \( e_i \), and \( w \) is the label or \( \pi \). As a labelled graph, \( \mathcal{F} \) looks like a flower with a petal for each word starting and ending at the same central state.

Definition 4 (Square): The direct product of a flower automaton with itself gives a new automaton \( \mathcal{P}(Q', I', T', E') \) with \( Q' \subset Q \times Q \), and \( \lambda((q_1, q_2), a, (q_3, q_4)) \in E' \) if and only if \((q_1, a, q_3) \in E \) and \((q_2, a, q_4) \in E \). Then \( \mathcal{P} \) is the square automaton of \( X \).

The square automaton is ambiguous if there exists a path \((p_0, q_0) \rightarrow (p_1, q_1) \rightarrow \cdots \rightarrow (p_n, q_n) \rightarrow (0, 0)\), such that \( p_i \neq q_i \) for some \( 1 \leq i < n \). CodeGen uses the following theorem from coding theory as the basis for its main algorithm.

Theorem 1: A language over a finite alphabet is a code if and only if its square automaton is unambiguous.

Ambiguity in the square automaton of a language \( X \) means that \( X \) is not a code, for two distinct paths with the same label may then be traced in the flower automaton of \( X \) from 0 to 0, so there is a word in \( X^* \) equal to two distinct concatenations of words in \( L \), contrary to the definition in item 1.

B. Combinatorial Facts

CodeGen implements an algorithm based on Theorem 1. The program attempts to take advantage of combinatorial coding facts to speed up the code word verification and generation processes. For instance, if the request is for a language of words all of the same length or if the language to be tested consists of same-length code words, then it is code and no check for this property is made.

If \( \Sigma \) has \( n \) symbols, and if the involution on \( \theta \) is such that \( \theta(a) \neq a \) for any \( a \in X \) (which implies that \( n \) is even) then there are some limits on how many words are in \( X \) if \( X \) is \( \theta \)-\( k \). There are a maximum of \( n^k/2 \) distinct subwords of length \( k \) which can be used in \( X \). For code words of lengths \( k_1 \leq k \) up to length \( k_2 \geq k \) there are at the most

\[
\sum_{n_1}^{m-1} c^i + \sum_{m}^{n_2} c^i \left(\frac{n}{2}\right)
\]

words available to choose from.

CodeGen informs the user and does not attempt to answer the request for the generation of more than this many \( \theta \)-\( k \) words.

III. Algorithm for Finding and Detecting “Good” DNA CODEWORDS

The basic algorithms of CodeGen answer requests for randomly-generated code words with any number of the \( \theta \) properties and tests any list of code words for specified \( \theta \) properties. The structures it uses are easily accessible, so that the most costly computation involved is a string operation. The testing process is recursive, a methodical check for ambiguous paths in the square, as suggested by Theorem 1.

A parameter indicating the desired \( \theta \) properties is passed from routine to routine until finished. We explain first the data structures and how code and \( \theta \) property detection is accomplished using the TRACE routine and the CHECK subroutine. Then we present pseudo-code for the TRACE routine.

A. Necessary Data Structures

The structure representing a flower automata such as \( \mathcal{F} \) we will call FLOWER. It is an array of the lists of state-transitions in \( \mathcal{F} \). Thus, each list has entries of the form \((i, j)\), where \( i \) and \( j \) are the numbers of the source and target state, respectively, of an edge in \( \mathcal{F} \). It expands and contracts as code words are added to or deleted from \( X \) – as opposed to reconstructing it from scratch with each change in the collection of code words.

The structure representing a square automaton such as \( \mathcal{P} \) is an array with only two lists in it: first, edges in \( \mathcal{P} \) originating at \((0, 0)\), and, second, edges whose source is not \((0, 0)\). Expansion of This structure, which we will call SQUARE, also expands with each code word added. A copy of the path added to FLOWER and a pointer to the FLOWER is passed to the routine, and edges sharing a label are sought out and added to the appropriate path of SQUARE.

Finally, a structure SOURCES is used to maintain a list of states in SQUARE that are sources of some edge in \( \mathcal{P} \). This list will be depleted as TRACE fails to identify a path of the desired type leading from each state in it.

At the heart of TRACE is a simple routine CHECK to determine whether a given state on a path being traced satisfies the property, fails to satisfy that property, or whether the trace should simply continue. As examples we describe here how CHECK is used by TRACE to determine if \( X \) is a code, if \( X \) strictly \( \theta \)-free or if \( X \) is strictly \( \theta \)-compliant.

B. Identifying a Code

To verify that \( X \) is a code, CodeGen traces through SQUARE until an ambiguous path is identified – whereupon it returns TRUE, or until it exhausts all possible paths or reaches an excessive recursion depth. In either of the latter cases it returns FALSE.

To do this, the state \((0, 0)\) is passed to TRACE as the only member of a “list” of initial states and list of final states. Each new state found on a path from \((0, 0)\) in SQUARE is passed to CHECK. If the current state is \((i, j)\) and \( i \neq j \), it sets a flag which it passes to deeper recursions of TRACE (if there are any). If \( i = j = 0 \) and the flag has already been set, then CHECK returns the constant DONE. This will cause TRACE to return TRUE. Otherwise, if \( i = j = 0 \) and the flag has not been set, or if \((i, j)\) is not found in SOURCES, then CHECK returns the constant NO_PATH. Then TRACE returns FALSE, or else its recursion unwinds one level. In any other case, CHECK returns the constant CONTINUE, TRACE compiles
a list of the target states of edges where the source each is (i, j), and recurses through it.

This works because, according to Theorem 1, a path from (0, 0) to (0, 0) exists in $P$ such that $i \neq j$ for some state (i, j) on the path if and only if X is not a code. TRACE returns TRUE if and only this is the case. It returns false not only when a path leads nowhere, but a path is the dual tracing of a petal in $F$.

C. $\theta$ Freedom

To check whether a code X is strictly $\theta$-free, CodeGen forms the flower automaton $F\theta$ of the involution of X, i.e., the flower automaton recognizing {\( \theta(x) : x \in X \). It then forms a square automaton $P\theta$ by taking the direct product of $F\theta$ with $F$.

It then determines whether there is a path in $P\theta$ between a state (0, i) and a state (0, j) which passes through at most one state of the form (k, 0), where $k \neq 0$. To do this, CodeGen assembles a list of all states (0, i) in $Q'(P\theta)$. This list is passed to the TRACE routine as initial and terminal states (see III-E). TRACE then follows paths from these initial states, calling the CHECK routine for each new state on each path.

If the current state is in the list of terminal states, CHECK returns the constant DONE, and TRACE returns TRUE. If the current state is of the form (j, 0), (j \neq 0) it tests whether a flag has been set. If the flag has not been set, it sets the flag, and returns the constant CONTINUE. If the flag has already been set, CHECK returns the constant NO\_PATH, causing TRACE to return FALSE or to unwind its recursion one level. If the current state is not the source of any edge in $P\theta$, CHECK will also return NO\_PATH. In every other case, CHECK returns the constant CONTINUE. Then TRACE assembles a list of the target states of edges of which the current state is a source, and to recurse through that.

This works because, if a code is not $\theta$-free, then, by the definition in II-A, item 3, there is an element $x$ of $O(X)$ and elements $y$ and $z$ of $\Sigma^*$ such that $y\theta(x)z \in X^2$. Let yxz = y1y2 \cdots yty(x)1y2 \cdots y(x)z1z2 \cdots zn, where $s, t \geq 0$ and $x = x1 \cdots x_s$, while $y1, y2, \ldots, yk \in \Sigma$ for $1 \leq i < s, 1 \leq j \leq r$ and $1 \leq k \leq t$. Then $\theta(x)$ is the label of a path $\pi(\theta)$, where $\pi1 = f1 \cdot \cdots f_s$ and $\pi2 = g1 \cdots g_{r+s+t-u}$ are petals in $F$, (one of which may be $\lambda$), and where the label of $f_{r+1} \cdot \cdots f_u$ followed by the label of $g1 \cdots g_{r+s+t-u-u+r+1} = g_{s+t+1}$ is $\theta(x)$. Let $q_{v1}$ be the source of $f_{r+1}$, and let $q_{v2}$ be the target of $f_s$, where $1 < i < s$, and let $q_{v3}$ be the target of $g_s$, where $1 < i < s - u$. Let $q_{v4}$, similarly, be the source of $e_1$ in $\pi$, $1 \leq i \leq s$, where the target of $f_s$ is 0 = $q_{v4}$ = $q_{v5}$. Then (q_{v6}, q_{v2}) = (q_{v6}, q_{v5}) is in $Q\theta \times Q$, and $q_{v6}$ is the source, and $q_{v5}$, the target of an edge in $P\theta$ for each $1 \leq i \leq s$, by its definition, so that these edges form a path $\pi'' = e_1e_2 \cdots e_s$ in $P\theta$. Since $\pi1$ and $\pi2$ are petals in $F$, there is at most one the one state, $q_{v6}$, in the interior of $\pi''$ such that $q_{v5}$ or $q_{v6}$ \neq 0. Conversely, the tracing of such a path shows the existence of a sequence of edges in $F$ beginning at symbol $r+1$ (r \geq 0) of some petal $\pi1$ and either ending within that petal or after reaching its end continuing $s$ edges into another petal $\pi2$ of $F$, the label of which is equal to the $\theta(x)$ for some $x \in X$. But then there exist $u, z \in \Sigma^*$ such that $y\theta(x)z = w_1w_2$, where $w_1$ is the label of $\pi1$. Thus, $X$ is not $\theta$-free.

D. $\theta$-Compliance

To check whether a code X is strictly $\theta$-compliant, CodeGen forms $F\theta$ and $P\theta$ and assembles a list of $P\theta$ states with one zero coordinate as described above in III-C.

Then TRACE attempts to find a path in $P\theta$ which demonstrates that X is not $\theta$-compliant, whereupon it will return TRUE. Otherwise will return FALSE. Here the CHECK routine sets no flags. If the current state is in the list of terminal states, CHECK returns the constant DONE. Otherwise, if the current state is of the form (j, 0), CHECK returns the signal NO\_PATH. In any other case, CHECK returns CONTINUE. TRACE responds to these signals as described in III-B and III-C.

The justification of this algorithm is similar to that for strict $\theta$ freedom, and the non-strict versions of both require but minor variations.

E. Main Algorithm

We present here pseudo-code for TRACE. It is made up of two functions, MULTITRACE and SINGLETRACE.

MULTITRACE receives a list of initial states, a list of terminal states, and instructions as to which property to check for. It walks through the list of initial states, passing a copy of each, along with the list of terminal states and test level, to SINGLETRACE. If any such call returns TRUE, it halts and returns TRUE. If the list of initial states is exhausted, it halts and returns FALSE.

The main function is SINGLETRACE, (see Algorithm 1) which performs the recursive trace into SQUARE, calling CHECK to determine whether the test levels have been satisfied. If so, TRACE returns TRUE (“Yes, a violating trace has been found.”). If it exhausts all paths leading from that state without meeting the required test levels, it returns FALSE. Its parameters are S (a state in SQUARE), T (the list of terminal states), L (the parameter indicating the test level, i.e., the property to be tested for), F (a constant holding the flags for CHECK), and D (the current recursion depth).

Note the use of M, a maximum value to restrain the depth of recursion for the routine. Currently this value is set to $f^4$, where $f$ is the number of states in $F$.

The local boolean variable SingleTrace holds the current state of the trace. The local variable Check holds the return-value of the CHECK routine.

F. Computation Time

This algorithm uses calls to standard string routines, and the number of these calls is proportional to the total depth of all recursions it makes. Since each such call either returns a TRUE and halts, or eliminates an element of SOURCES, i.e., one state in $P$ or $P\theta$, the computation time – at the most – is proportional to $O(s^2)$, where $s$ is the number of states in $P$ or $P\theta$. It is therefore $O(s^2)$. 

Algorithm 1 SINGLETRACE(S, T, L, F, D)

1: SingleTrace ← FALSE
2: D ← D + 1
3: if D < M then
4:     Check ← CHECK(S, L, F)
5:     if Check = DONE then
6:         Return TRUE
7:     else if Check = NO_PATH then
8:         Return FALSE
9:     else
10:        i ← 1
11:        while SingleTrace = FALSE and i < |Q| do
12:            SingleTrace ← SINGLETRACE(Q[i], T, L, F, D)
13:            i ← i + 1
14:        end while
15:        if SingleTrace = FALSE then
16:            Remove S from SOURCES
17:        end if
18:    end if
19: else
20:    Return FALSE
21: end if

Suppose that \( \Sigma \) contains \( n \) symbols \( a_1, \ldots, a_n \). Every edge in the flower has one of these \( n \) labels. The total number of edges in the flower, \( l \), is the total length of all words in \( X \) by construction. If \( x_1, x_2, \ldots, x_n \) are the number of occurrences of \( a_1 \) through \( a_n \) in all elements \( X \), then there must be

\[
\sum_{i=1}^{n} x_i^2 = s^2 \]

edges in \( P \). Thus, \( s \), the number of states in \( P \) is \( O(l^2) \), where \( l = \sum_{i=1}^{n} x_i \). But by the observation above, the computation time of whether \( X \) is a code is \( O(l^4) \).

The computation-times of \( \theta \)-compliance and \( \theta \)-freedom is proportional to \( O(s'^2) \) where \( s' \) is the number of states in \( \mathcal{P}_\theta \). Since this is

\[
\sum_{i=1}^{n} x_i y_i,
\]

where \( y_i \) is the total number of occurrences of \( \theta^{-1}(a_i) \) in \( X \), they are clearly of the same order.

Note that, for this reason, setting \( f^4 \) as the maximum recursion depth \( (f = l - |X| + 1) \) is a reasonable allowance.

In designing CodeGen, we observed that checking the remaining \( \theta \) properties using the same algorithm demands a disproportionate amounts of memory. The program makes these checks using standard string methods on appropriately constructed objects. The computation time turns out to be linear. We discuss the implementation next.

G. Object-Oriented Implementation

CodeGen was designed in Visual Basic 6.0, which offers a limited form of OOP. Its objects therefore have no inheritance. There are eight language-theoretic objects used: the classes Point, Pointlist, Alphabet, Word, WordList, Involution, Automaton, and Language.

1) The Point class is a wrapper for tuples, with .X and .Y as its primary members.
2) The PointList class wraps an array of Points, useful for storing paths in an Automaton and offers expected functions as .Add, .Remove, etc.
3) The Alphabet class treats a set of symbols like a bag, and is also able to display its members properly. It ignores attempt to .Add symbols it already contains.
4) The Word class wraps a string and makes it behave like a word.
5) The WordList class wraps an array of Words, and its .Add function returns FALSE when an attempt is made to add a word it already contains.
6) The Involution class is contains the assignment pattern of an involution. It is therefore also the germ of future, less specialized mapping or function objects.
7) The Automaton class is where the TRACE algorithm takes place. This class knows if it represents a flower or square automaton and behaves like a finite graph. Its most significant data member is Paths, an array of PointLists which hold the edges traced in the automaton, as in the graph representing it. Its .MakeFlower and .MakeSquare functions behave as described in III-A.
8) The Language class has, of course, a WordList and an Involution. It holds three Automatons as well, representing its flower automaton (\( F \)), square automaton (\( P \)), and the product of the involution of its flower with its flower (\( \mathcal{P}_\theta \)). It offers the methods .AddRandomWord, .AddWord, and .Test, which call the TRACE routine of the appropriate automata. The results are then available in public class members.

These objects enable CodeGen to bound the number of string operations called for testing \( \theta(k, m_1, m_2) \)-subword compliance and for testing whether a code is \( \theta \)-k, by constructing smaller language objects and attempting to at the involution of certain words to it.

IV. THE CodeGEN INTERFACE

The interface allows the user to obtain new sets of code words or expand an existing set of code words. All capabilities are presented on a main screen, consisting of three panels labelled Alphabet, Involution, and Word (Figure 4).

These may be accessed in any order, but an alphabet must be selected or created before an involution can be set, and an involution must be selected before testing or generating code words. After the language has been tested and verified, it can be saved.

A. Alphabet

The DNA alphabet \( \{A, T, C, G\} \) is present and selected by default. Other alphabets can be created, saved or deleted and may contain up to 256 numeric or character symbols. The number of alphabets is currently limited to twelve.
B. Involution

The DNA (Theta) involution is present and selected by default, and cannot be altered. Up to three different involutions may be defined over each alphabet. Morphism may be selected for regular concatenation of words, or Antimorphism for Watson-Crick complementarity, as discussed in II-A.

C. Word

Most of the language operations take place on three tabs in Word (although see IV-D): Test Level, Source, and Parameters.

Test Level sets the desired level of ambiguity and properties of codewords. Code is basic (item 1 in II-A). Checking Subword Compliant enables detection of the ‘hairpin’ DNA hybridization (item 4 in II-A), avoiding complementary subwords of prefixes or suffixes of individual words. The disallowed subword (hybridization) length is chosen in k. The distances between the subword and its complement (the length of the “hairpin” loop) cannot be shorter than the value set in m1 or longer than the value set in m2.

If the Θ Compliant box is checked, this enables verification of the Θ-compliant property. If the Θ Free box is checked, this enables verification of the Θ-free property. (See items 2 and 3 in II-A.) Verification of the Θ-k property (item 5 in II-A) – the strongest of the Θ properties – is enabled if the Θ-k box is checked. This rules out the presence of complementary subwords (cross-hybridizing sequences) of the length set in k, if this is possible (see II-B).

Source is where the language is actually created. It has four radio buttons.

1) Language file This displays all of the previously saved language files in the current directory, and allows the user to select one. These language files contain the relative alphabet and involution, plus the words and previously defined properties of the language. Once a language file is selected, all of the words are displayed and the user may add or delete words, or test the language.

2) Import file The directory information is displayed to allow the user to locate and select a text file of words to be imported into the current language. Words in the text file should be one per line and are imported with only two restrictions: they must contain symbols from the current alphabet, and no words may be duplicated. After importing words, the new language is displayed in the Language View pop-up window and can be tested for the desired properties.

3) Keyboard Words may be added one at a time directly from the keyboard. Again, the only limitation is that the words must consist of symbols from the current
alphabet. Words cannot be duplicated. The list of words currently in the language is displayed and can be modified or tested at any point.

4) Random The program will generate random code words according to the chosen Test Levels and in amounts, lengths and content specified in Parameters. Words are reported one at a time as they are generated until the request is met or until the process exceeds program limitations (see III-E). The code words generated and all valid language properties are then re-displayed in the Language View window.

**Fig. 6. Random Language Generation Parameters Selection Tab**

**Parameters** sets values for the generation of code words. **MINimum** and **MAXimum** word lengths display and can be adjusted. **Max Reps** sets the maximum number of consecutive symbol repetitions, while **GC Content** sets the maximum percentage of Cs or Gs in each word for DNA based languages. **Total Words** displays and sets the number of words in the language. All parameters change to reflect the properties of words currently in the language. When set to a desired values, words in these amounts, lengths, and content are generated using **Word→Source→Random→Generate.**

**Fig. 7. Language Display Screen**

D. Language View window

On the menu bar, **View→Show Language** displays all words and properties such as test levels selected and parameters of the current language will be displayed in a separate window. The language may then be tested at these levels using the **Test** button. This tests the language at the currently selected levels and automatically updates them on the screen to the values the language is found to support. The **Status** label changes to read ‘Verified at current levels’ when this occurs. At that point, **Save** will store the language, i.e., all of its words, parameters, and verified levels for future use.

Finally, the user may change desired test levels and retest the current language, delete any or all of the words, create a new language as spelled out in IV-C, or set up new alphabets (see IV-A) and involutions (see IV-B).

V. LIMITS AND WHAT IS YET TO BE DONE

CodeGen v.1 is an object-oriented program that has been developed in Microsoft Visual Basic 6.0. It currently runs only on Microsoft Windows platforms. The recursion in the main algorithm demands at least 128 MB of memory. It is an ongoing project, and version 2 should be a translation to Java for multi-platform use, and to take advantage of the more complete object-oriented features of that language.

At present, CodeGen cannot generate or operate with words or alphabets consisting of more than 256 symbols, although the language size is limited only by available memory. Not all random generation requests are possible, and CodeGen will at present generate a maximum of 4096 code words before giving notice that the search is incomplete. Because the word-generation technique uses a typical pseudo-random number generator and may produce words that are already in the language, it may fail to produce the requested words even if the request is combinatorially possible to satisfy.

The languages and properties are only theoretical and they are based upon those properties described in [13]. Sequences generated by CodeGen are formed with these various constraints in mind.

This project is part of the wider theoretical effort to facilitate generation of code words and DNA sequences for various application purposes including biotechnology and computation encoding. The true test of these properties is in the laboratory, with code words made up of actual oligonucleotides possessing the various θ properties.

VI. ACKNOWLEDGMENTS

The production of these algorithms, their programmatic implementation, and the character of its user interface are in large part creditable to the continuous firm guidance and valuable advice of Natasha Jonoska. The authors take responsibility for any and all shortcomings.

This work has been supported by grant EIA-008615 from the National Science Foundation, USA.
REFERENCES