

## Chapter 2

# Application of Evolutionary Computation to Bioinformatics

Daniel Ashlock

## 1. INTRODUCTION

In solving a scientific problem, one of the most helpful possibilities is that you will see a pattern in your data. It is almost the definition of an interesting scientific problem that it contains some sort of pattern. The patterns that arise in nature are often subtle and escape notice until cleverness or hard work un-cover them. The field of machine learning is a collection of techniques intended to automate the process of pattern discovery. A broad survey of machine learning techniques applied to bioinformatics is given in Pierre Baldi and Soren Brunak (2001). This document introduces a single, relatively versatile machine learning technique called evolutionary computation. A collection of applications of evolutionary computation to bioinformatics is given in Fogel and Corne (2003).

Both machine learning and evolutionary computation have applications far beyond bioinformatics but almost all of the techniques in the domain of machine learning and evolutionary computation have useful applications within bioinformatics. The introduction to evolutionary computation given here is in the form of three examples intended to showcase three substantially different applications of evolutionary computation. The first, while it solves a real problem, is an almost trivial instance of evolutionary computation. It seeks a gapless alignment of 315 sequences in a fashion that permits the discovery of a motif associated with

---

**Daniel Ashlock**     Department of Mathematics, Bioinformatics and Computational Biology Program, Iowa State University, Ames, IA 50010

*Genome Exploitation: Data Mining the Genome*, edited by J. Perry Gustafson, Randy Shoemaker, and John W. Snape.  
Springer Science + Business Media, New York, 2005.

**Table 1**  
**Predictions versus Truth Results for the Most Fit**  
**Finite State Machines Located During the First Set**  
**of Evolutionary Runs.**

<i>Training Data</i>				<i>Crossvalidation Set</i>			
	Prediction				Prediction		
	+	−	?		+	−	?
Good	666	256	78	Good	115	106	29
Bad	287	659	54	Bad	96	125	29

insertion of a mu-transposon. The second example is a more sophisticated but standard application of evolutionary computation, learning patterns in a collection of good and bad primers designed as part of a *Zea mays* genomics project. Once learned, these patterns are then used to reduce the failure rate of primers in subsequent work. The third example is a departure from standard evolutionary algorithms, which fuses evolutionary algorithms with greedy algorithms to create a new type of evolutionary algorithm called a greedy closure genetic algorithm. This algorithm is used to create error correcting DNA bar codes for use in pooled genetic libraries. These bar codes permit the identification of the library, which contributed a given expressed sequence tag. The error correcting property of the bar codes permits the identification of the library even when the area containing the bar code is sequences with some errors.

**2. EVOLUTIONARY COMPUTATION**

Evolutionary computation has been described as a “Swiss army algorithm” in comparison to the Swiss Army knife, which typically has a whole collection of small tools built into it (Fig. 1). This description is both misleading and a good starting point for discussing the strengths of evolutionary computation. A completely general-purpose tool would be useless (probably too heavy to lift). The Swiss army knife is not a multipurpose tool. Careful examination will show the alert scientist that it is a collection of many tools with fairly specific applications. Sure, it is possible to use the screwdriver as a hole punch and the large blade may be useful for both shaving a dowel and trimming a bad spot out of an apple, but the applications of each tool within the knife are fairly limited. Also, the tools you are not using tend to get in your way. The purchase on a Swiss army knife is not as good as the handle on a normal screwdriver. What does this have to do with evolutionary computation? The basic algorithm for evolutionary computation is given in Table 1. This algorithm can solve any problem for which the solutions can be placed in the form of a data structure and for which the quality of the solutions in such data structures can be compared. In spite of this apparent power, the process of getting a problem transformed into a data structure and then creating a useful quality comparison can be insuperably difficult. Even if these hurdles can be overcome, the running time on evolutionary algorithms is typically long and so

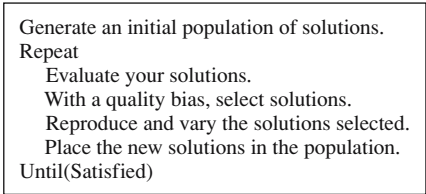


FIGURE 1. Basic algorithm for evolutionary computation.

an evolutionary algorithm may be impractical. Let us start with the very simplest type of evolutionary algorithm, a string evolver.

In a string evolver the population is a population of character strings and the notion of fitness is that of matching a reference string. Examine the following population member (character string), aligned with the reference string “Madam, I’m Adam.”:

**Reference: “Madam, I’m Adam.”**  
**Population Member: “Mad\*g,hI.m Admm!”**  
**Fitness loci: +++ + + +++++ +**

The fitness of the above population member is 10 of a possible 16 because 10 of its 16 characters match the reference string. Once we have a notion of fitness it becomes possible to apply the algorithm given in Table 1.

Generating an initial population of character strings is done by filling in characters at random in each member of the population. We pick strings to reproduce by shuffling the population into groups of four and permitting the best two members of each group of four to reproduce. Their offspring replace the two worst members of the group of four. The relatively small size of the groups (four members) represents a weak bias in favor of fitness; larger groups would yield sharper selection. Reproduction is done by first copying the two strings that are reproducing and then performing *crossover* and *mutation* on the copies. These words have completely different meanings in the context of evolutionary algorithms than they do in biology. Crossover consists of exchanging middle segments of the strings; mutation consists of picking a position in the string and putting in a new random character. An example of crossover is shown in Figure 2, an example of mutation is shown in Figure 3.

The process of evolving a copy of a string that you already have in hand is not an intrinsically interesting one. It does serve as a simple example of evolutionary computation and serves as a starting point for discussing the design of evolutionary

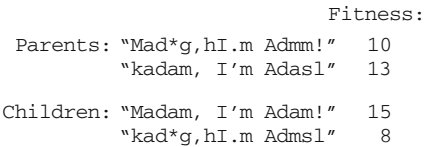


FIGURE 2. Depicted above is crossover. Two selected parent strings are copied and characters in positions 3–14 are swapped. We call 3 and 14 the *crossover points* and the type of crossover depicted is called *two point crossover*.

```
Child: "Madam, I'm Adam!" 15
      |
Mutated Child: "Ma^am, I'm Adam!" 14
```

**FIGURE 3.** Mutation consists of taking one or more character positions and generating new random characters at those positions. Above is depicted mutation at the third character in a string with a net decrease in fitness.

algorithms. A trace of the best string found so far in a run of a string evolver is shown in Figure 4. Notice that the time to the next improvement in fitness is highly irregular but generally increases with time. Initially, crossover can bring together correct sub-strings. As evolution proceeds the population in the algorithm becomes highly inbred and crossover does little. Toward the end of a run, mutation becomes the sole source of progress as we wait for a fortuitous mutation to fill in the one or two missing characters not present in the initial population.

The method of picking parents and placing offspring used in the string evolver example is called *single tournament selection with size four*. The methods of producing variation are called *two point crossover* and *single point mutation*. There is a wealth of possible choices for parent selection, offspring placement, and of variation producing techniques. Many of these are detailed in Ashlock (2004), which is available at <http://www.math.iastate.edu/danwell/EC>

3. FINDING A TRANSPOSON INSERTION MOTIF

In Dietrich, Cui, Packila, Ashlock, Nikolau, and Schnable, (2002), an example of an application of a string evolver to bioinformatics appears. A collection of 315

Best String	Fitness	Appeared in Generation
HadDe Q' /--<jlm'	3	5
HadDe,em3m/<I jm-	4	52
HadDe,em3m/<I jm-	5	54
HadDm,ex3m/#I jmj	6	73
HadDm,eI8m/#I jmj	7	86
HadDm,eI8m[Aj jmt	8	118
HadDm,UI8m[Aj jm.	9	135
MadDm,zI8m4AJ1m.	10	154
Madam,zIXm4AJ1m.	11	163
Madam, InmqAJym.	12	256
Madam, I'mqArHm.	13	327
Madam, I'm AC^m.	14	473
Madam, I'm APam.	15	512
Madam, I'm Adam.	16	647

**FIGURE 4.** A string evolver running. Each time an improvement in the best fitness in the population occurs, the string receiving that fitness is printed together with the generation number in which it appeared.

DNA sequences 129 bases long and centered on a 9 base repeat created during a transposon insertion were acquired. The conjecture is that there is a motif favored by the transposon at the site of insertion but this motif is masked by not knowing which orientation of the sequenced DNA contains the motif. The motif is not tight enough to permit its discovery by inspection or simple statistical analysis. What is desired is to recover the correct orientation by aligning all 315 sequences. There is not sufficient sequence homology at the insertion sites to permit useful alignment via dynamic programming, the standard alignment technology.

The choice of data structure for attempting to solve this problem is fairly obvious. It is a character string of 315 zeros and ones that specifies an alignment of the sequences. A zero means “leave the sequence in its current orientation” while a 1 means “compute the reverse complement of the sequence.” This yields a space of  $2^{314} = 3.34 \times 10^{94}$  possible alignments (the first sequence is never reversed, selecting one of two possible global alignments). The tricky part of designing the evolutionary algorithm in this case is the fitness function. It was decided to minimize the randomness of the alignment of the sequences. In this case the randomness of an alignment (choice of forward or reverse orientations) was estimated by computing the squared deviation of the number of bases of each type at each position from the empirical frequency of those bases in the sequences being aligned. This measure of non-randomness is given in Equation 1.

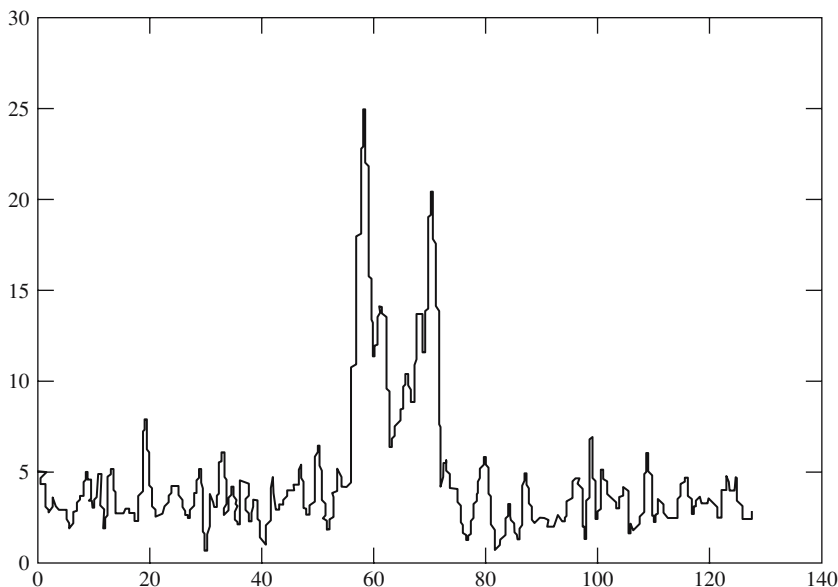
$$f(A) = \sum_{i=1}^{129} \left( \sum_{z \in \{C, G, A, T\}} (N_x - E_x)^2 \right)$$

The number  $N_x$  is the number of bases  $x$  at a given position while  $E_x$  is the expected number of bases of that type, i.e. the number of sequences times the fraction of all bases in the data set of type  $x$ .

Using Equation 1 as a fitness function, a string evolver can be used to perform the alignment. The string is made up of characters each of which specifies the orientation of one of 314 sequences with the first sequence left in its original orientation. Such a string evolver that searched the space of alignments was run 100 times. Out of those 100 runs, 88 returned the same alignment and the same fitness value. This fitness value was the largest found in any of the runs. This makes it likely we are detecting a true optimal alignment of the 315 sequences.

In order to see if there is a motif at or near the point of insertion, a  $\chi^2$ -value was computed for each position of the alignment. The  $\chi^2$ -values are given along the length of the alignment in Figure 5. The unaligned sequences did not show a significantly non-random base composition at any point. The aligned sequences yield a significant deviation from the expected base composition statistics at points immediately flanking the insertion. Readers interested in the biology that underlies this example should read Dietrich, Cui, Packila, Ashlock, Nikolau, and Schnable, (2002).

It is important to consider the question of interaction between the fitness function and the  $\chi^2$ -statistics. In this case the non-randomness of the entire alignment was maximized. Using 60 bases of flanking sequence on either side of the insertion



**FIGURE 5.**  $\chi^2$ -values derived from base composition at each position in an alignment of 315 DNA sequences of length 129 flanking distinct insertion points of a mu-transposon. The alignment used was the best found by the evolutionary algorithm.

point reduces the chance that we are creating a motif by fortuitous arrangement of existing variation in a small region including the point of insertion. The fact that the  $\chi^2$ -values spike in positions immediately flanking the insertion but not in the remainder of the flanking sequence suggests a motif does exist at the point of insertion. It is also important to note that non-randomness was maximized relative to the empirical base statistics of the sequences aligned. Using some larger region to compute expected base frequencies would have made it easier to create a phantom motif by exploiting and aligning existing random sequence features.

#### 4. PCR PRIMER PICKING

A more complex application of evolutionary computation to bioinformatics is that of picking PCR primers. At this point the application is complex enough to raise the issue of representation. The representation used in an instance of evolutionary computation is the way that candidate solutions to a problem are coded as data structures and varied during reproduction. The representation used to align the transposon insertion sites was a character string representation with two point crossover and single point mutation, exactly the same representation as was used in the example string evolver. Before picking a representation for primer picking, we will need a clear specification of the problem.

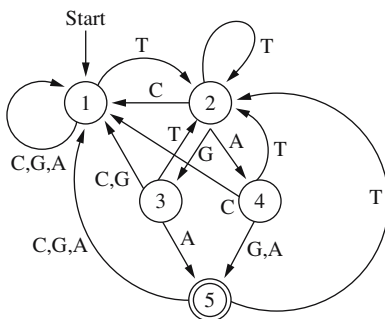
Using a standard primer-picking tool (Primer 3 from the NCSA Biology Workbench), many thousands of primers were designed to amplify sites likely to contain polymorphisms in *Zea mays*. Many of these primers amplified their targets correctly, while others did not. The problem is to distinguish the good primers from the bad primers given that the original primer picking software thought they were all good. We are neglecting technician error in performing the PCR reactions and in scoring the outcomes of the PCR experiments, in effect treating these sources of errors as “noise”. We thus act as if the scoring of primers as good or bad is entirely correct. These primers, scored as good and bad, form the *training data* for our primer picking system.

The experiment designed to find bad primers used evolutionary computation as a machine learning system to attempt to detect any patterns that will help us to tell good primers from bad primers. Note that many of the standard things that make a primer good, such as correct Tm and the presence of a GC-clamp are already in every primer because they were put there by the original primer picking software. This means we will be looking for organism specific patterns that only affect primers in *Zea mays*. In subsequent work on *Zea mays*, multiple primers will be designed for each target and the good/bad classifier created via machine learning will pick from among them those primers most likely to work based on patterns learned from earlier primers.

In the last example there was an obvious representation (a character string that specified sequence orientations) but the fitness function (minimize overall randomness of the alignment) was not such a clear choice. In building an evolutionary algorithm to pick primers, it turns out that both the fitness function and representation are not obvious. We want a classifier that, given a primer, gives a (often correct) prediction if it is a good or bad primer. Fitness will thus consist of some abstraction of predicting correctly the good/bad status of the primers in the training data. In a set of initial experiments simply scoring the number of correct predictions did not work well.

Given that the problem could be in a motif that is somewhere in the primer, we need a representation that does not assess bases in a manner based on their distance from the end of the primer but rather based on the pattern of surrounding bases. Because of this need to have a non-position-specific assessment we settled on finite state machines as our representation. A finite state machine can process a string of bases, waiting for one of a small set of motifs to appear, and then make a state transition that detects it. The widely used BLAST software incorporates finite state machines to perform essentially this task. As an additional benefit, use of finite state machines permitted a unique sort of incremental fitness reward, described subsequently.

A finite state machine is a collection of states (including a starting state), together with a collection of data-driven transitions among the states. The output of a finite state machine is associated with the transitions or the states themselves. An example of a finite state machine is shown in Figure 6. In this case the output of the FSM is encoded by noting that, when it is in state five, the last three bases it encountered formed a stop codon. In order to give an incremental assessment of



**FIGURE 6.** A finite state machine that recognizes stop sequences in DNA.

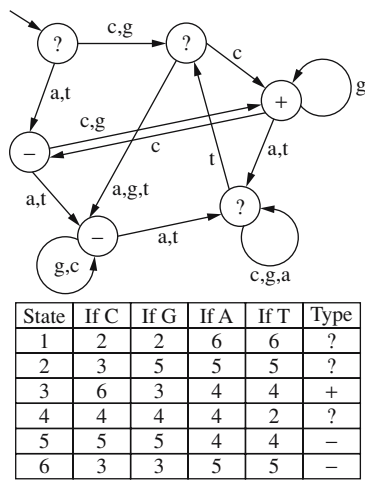
strings of DNA bases we are considering as primers we will make a modification to this standard sort of finite state machine.

The states of the finite state machines used to classify primers have three possible types or labels: ? (don't know), + (good primer), and - (bad primer). These state labels are used to permit the finite state machine to function as a classifier. The fitness of a finite state machine on a training set of primers is computed as follows. Each PCR primer in a set of training data is run through the finite state machine. As the machine passes through each state it is given +1 score if the state label matches the good/bad status of the primer and -1 if it doesn't match. No incremental score is awarded for the don't know states. Fitness is summed over all primers examined. If we imagine the evolutionary algorithm as searching a fitness landscape for good classifiers then the use of this sort of incremental reward scheme acts to smooth the landscape and permit the search to avoid getting stuck. A more complete discussion of these issues appears in Ashlock, Wittrock, and Wen, (2002). A finite state machine of the sort used to classify primers is shown in Figure 7 as both a state transition diagram (picture) and as a table.

Having decided to use finite state machines in our evolutionary computation system, we still need to select methods for generation of an initial population and for generating variations during reproduction. The finite state machines are initialized uniformly at random, filling in both transitions and state labels with uniformly distributed valid values. As with the string evolver, we will have a crossover operator and a kind of mutation. Both of these variation operators need to be re-tooled to contend with the more complex structure of finite state machines. The crossover operator used works by treating the states, including their label and outward transitions, as "characters" and then performing crossover on the "string" of states. Two point crossover of this string of states was used and the designation of the initial state moves with the first state during crossover. The mutation used on the finite state machine modifies the choice of initial state 10% of the time, randomly picks a new destination for one of the transitions 30% of the time, and modifies the label f+; -; ?g on a state 60% of the time.

One hundred evolutionary runs with distinct random starting populations of 600 machines were performed. The best finite state machine from each simulation





**FIGURE 7.** A finite state machine configured for primer classification. Both a pictorial and a tabular representation are given. The starting state, denoted by the rootless arrow, is state 1 and the states in the pictorial representation are numbered clockwise from that point.

was saved for evaluation as a classifier and for use in later sets of runs. Each evolutionary run proceeded for 1000 generations. These evolutionary runs used a set of 2000 primers, half good and half bad, as their training data. In order to ensure that the classifiers were learning patterns rather than just memorizing the training data they were cross-validated on a set of 500 primers, also half good and half bad, that were not part of the training data. In order to use the finite state machines to predict the good/bad status of a primer, primers are run through the finite state machines noting how many of each type of state are encountered. A majority vote is taken on the type of state label encountered. This permits a failure to classify if a majority of the states are of type ?. Table 1 documents the classification abilities of the best finite state machine located in the first 100 runs.

The outcomes given here show some patterns are being located but the finite state machines are not yet classifying well enough to have a substantial impact on the number of bad primers used. A score of 240 correct, 192 wrong, and 59 undecided is less than one would hope for. Examining the distribution of best fitnesses it appeared that a few of the evolutionary runs had discovered interesting patterns and many had not. In an attempt to consolidate these patterns in a single finite state machine we performed a second set of evolutionary runs that hybridized the finite state machines found in the first set of runs. For another instance of hybridization in the context of evolutionary computation see (Ashlock and Joenks, 1998). The hybridization runs are identical to the first set except that 100 members of the initial random population are replaced with the best-of-run finite state machines saved during the first 100 runs. The other members of these initial populations are still generated uniformly at random. A second set of hybridizations was performed,

**Table 2**  
**Predictions versus Truth Results for the Most Fit**  
**Finite State Machine Located During the Second Set**  
**of Evolutionary Runs, the First Set of Hybridization.**

<i>Training Data</i>				<i>Crossvalidation Set</i>			
	Prediction				Prediction		
	+	−	?		+	−	?
Good	626	337	37	Good	104	137	9
Bad	181	802	17	Bad	79	161	10

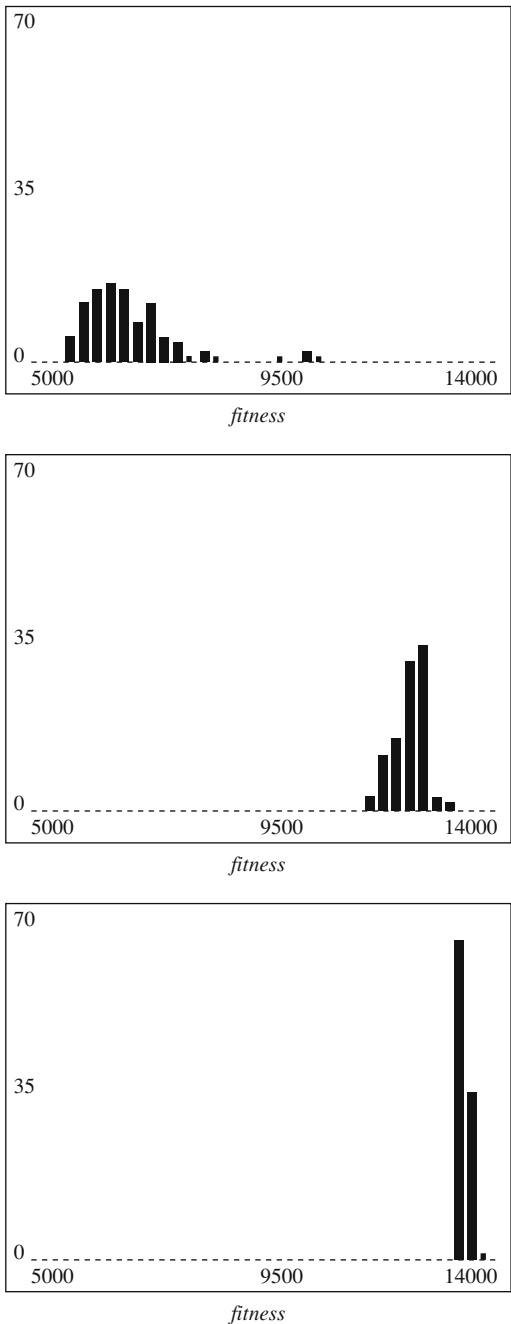
using the best-of-run finite state machines from the first set of hybridization runs. The outcomes for both sets of hybridization runs are given in Tables 2 and 3.

Examining the cross-validation results for the two sets of hybridizations we see improved performance for the first set of hybridizations and degraded performance appearing in the second. This means that, in the second set of hybridization runs, the finite state machines had gone beyond finding general patterns in the training data and had begun to find highly specific patterns that were relevant only to the training data. The prediction scores on the training data show this, as do histograms of end-of-run best fitnesses given in Figure 8. The fitness on the training data climbs impressively from the initial set of runs to the first hybridization. It also climbs substantially from the first hybridization to the second. This second improvement is at the expense of useful performance, as gauged by cross validation.

The “memorization” of the training data in the second set of hybridization runs is an instance of a problem that can appear in many machine learning systems: over-training. If you spend too much time training on a set of data you will become almost perfectly able to recognize patterns that give you a good score on that training data. Many of these patterns do not exist outside of the training data. This phenomenon of over-training means that the use of cross validation on data not used in training is a critical part of evaluating the success of any machine learning experiment. The best classifier found in the first set of hybridization runs is good

**Table 3**  
**Predictions versus Truth Results for the Most Fit**  
**Finite State Machine Located During the Third Set**  
**of Evolutionary Runs, the Second Set of**  
**Hybridization.**

<i>Training Data</i>				<i>Crossvalidation Set</i>			
	Prediction				Prediction		
	+	−	?		+	−	?
Good	636	364	0	Good	127	107	16
Bad	171	829	0	Bad	95	132	23



**FIGURE 8.** Histograms of the distribution of fitnesses. Top to bottom, for the first set of simulations, for the first set of hybridizations, and for the second set of hybridizations.

enough to have an impact on primer location. It makes 265 correct predictions, 216 bad predictions, and cannot decide about 19 of the primers. Since different runs may have found classifiers that recognize different patterns, it is also possible that a voting scheme among several classifiers will yield even better performance. That possibility has not yet been evaluated.

## 5. DNA BAR CODE LOCATION

Many useful evolutionary algorithms are hybrids. The word “hybrid” is used in a completely different fashion from its use in the preceding section. In this case a hybrid algorithm is a combination of evolutionary computation with some other type of algorithm. In this case we combine evolutionary computation with a *greedy algorithm*. We will use this greedy algorithm, under the control of evolved structures, to locate an error correcting bar code over the DNA alphabet. The errors corrected will not be biological but rather will permit us to recognize embedded DNA bar codes in constructs even when a sequencing error has modified them.

Greedy algorithms are familiar to most computational professionals. They are algorithms that use a greedy rule (e.g. make the best possible next move) to try to accomplish some goal. A few greedy algorithms, like those for finding a minimal-weight spanning tree in a weighted network, can be proven to yield optimal results. Other problems, like graph 14 coloring or the traveling salesman problem, admit a plethora of greedy algorithms all of which cannot be shown to yield optimal results. While it would seem that the control and improvement of greedy algorithms is a natural target for evolutionary computation, relatively few attempts have been made. It turns out that there are several possible approaches. The approach explored in the current paper consists of making small modifications in the order of presentation of potential parts of a growing structure as a means of deflecting the greedy algorithm’s behavior. The role of evolutionary computation is to locate modifications of the order of presentation that result in better structures. The resulting structure is not a standard one for evolutionary computation and the crossover used is probably macromutational (Peter, 1997) in character. The greedy algorithm we will use is Conway’s lexicode algorithm.

### Algorithm 1 Conway’s Lexicode Algorithm

*Input:* A minimum distance  $d$  and a word length  $n$ .

*Output:*  $An(n,d)$  – code.

*Algorithm:*

*Place the DNA words of length  $n$  in lexicographical order. Initialize an empty set  $C$  of words. Scanning the ordered collection of DNA words, select a word and place it in the code if it is distance  $d$  or more from each word placed in code so far.*

An  $(n; d)$ -error correcting code is a collection of  $n$ -character strings (in this case strings of DNA) that have the property that any two of them are at least  $d$

errors apart. The notion of error must be chosen to fit the problem. In this case, since sequencing errors can change, remove, or insert an apparent DNA base, the relevant notion of error is the edit metric. In the edit metric the distance between two strings of DNA is the minimum number of single base additions, deletions, or substitutions required to transform one string into the other. The edit distance of CGATT and GGAT, for example, is two: change the initial C to a G and delete the terminal T. Since no single edit will transform one of these strings into the other, their edit distance is exactly 2.

The  $(n; d)$ -error correcting codes over the DNA alphabet will be used as error tolerant bar codes for genetic constructs. In order to correct errors, we note that any collection of less than  $1/2d$  errors leaves us nearer to the actual bar code used than any other. Error correction consists of checking what bar code, among those used in any genetic construct, is closest in edit distance to the one apparently sequenced at the bar code's location in the genetic construct.

Examining Algorithm 1, the reader will see that the algorithm extends a partial code as it goes along. Since the algorithm considers the potential code words in a fixed order and starts with an empty set of code words the algorithm is deterministic. Run many times it always produces the same result. In order to get a different code out of the algorithm we would need to present the words to the algorithm in a different order. A *greedy closure evolutionary algorithm* exploits a restricted method of permuting the order in which the words are considered. Conway's lexicode algorithm is modified by specifying a short list of words, called a seed, that start as members of the code. The seeds used here have three members. Once a seed is chosen, Conway's algorithm is used to "close" the partial code represented by the seed. The words not in the seed are presented in the same order as in the standard algorithm. Since each word chosen to be in the code prevents any other word within  $d$  edits from being in the code even a small seed can have a huge impact on the membership and size of the code.

The structure to be evolved by the evolutionary algorithm is the seed. The fitness of a seed is the size of the resulting error correcting code. Notice that bigger codes are better because you get more bar codes with the same error correcting potential.

We have chosen seeds as the data structure to evolve. In order to complete a representation for the bar code location problem we must also specify the crossover and mutation operators for the data structure. Crossover of two seeds consists of copying two parents and then randomly shuffling the words between the copies, save that any words that appear twice send one copy to each child. Mutation consists of replacing one word in a seed with a new word generated at random. All of the variation operators listed here have the potential to create seeds, which violate the minimum distance for the code. Such seeds are awarded a fitness of zero and so removed from the population by the selection process. We call these seeds invalid. For valid seeds, the size of the code resulting from application of the lexicode algorithm to the seed is the fitness of that seed.

Before we run an evolutionary algorithm to locate bigger error correcting codes (equivalently: sets of DNA bar codes) it would be a good idea to compute

**Table 4**  
**Size of DNA Edit-Metric Lexicodes Found with the Unmodified Version of Conway’s Lexicode Algorithm.**

Code Size		Minimum Distance $d$						
Length $n$		3	4	5	6	7	8	9
3		4	—	—	—	—	—	—
4		12	4	—	—	—	—	—
5		36	8	4	—	—	—	—
6		96	20	4	4	—	—	—
7		311	57	14	4	4	—	—
8		1025	164	34	12	4	4	—
9		3451	481	90	25	10	4	4
10		*	1463	242	57	17	9	4
11		*	*	668	133	38	13	4

\* Big  
- empty

the rough size we could expect codes to be. Table 4 gives the sizes of the codes located by the unmodified version of Conway’s algorithm for several values of the length  $n$  of the bar code and the minimum distance  $d$  between any two strings in the code.

The evolutionary algorithm was run 100 times for 100 generations in for each of ten different sets of parameters  $n$  and  $d$ . Results from these runs are given in Table 5. The greedy fitness evolutionary algorithm outperformed the plain lexicode algorithm for all parameter sets tested. An example of a (6; 3)-error correcting code in DNA for the edit metric is given in Table 6.

**Table 5**  
**Comparison of DNA Edit Metric Code Sizes for the Plain Lexicode Algorithm and the Greedy Fitness Evolutionary Algorithm. The Figures in Parenthesis are the Number of Times the Best Result was Located in 100 Runs.**

Length	Minimum Distance	Plain Lexicode	Evolutionary Algorithm
4	3	12	16 (18)
5	3	36	41 (2)
5	4	8	11 (1)
6	3	96	106 (2)
6	4	20	25 (11)
6	5	4	9 (9)
7	3	311	329 (2)
7	4	57	63 (1)
7	5	14	18 (12)
7	6	4	7 (92)

**Table 6**  
**An Instance of a Maximum Size (6; 3)-Error Correcting Code Among**  
**those Locate by the Evolutionary Algorithm. This Code has 106**  
**Members, All at Mutual Edit Distance at Least 3. The Unmodified**  
**Version of Conway’s Lexicode Algorithm Locates a 96 Member Code.**

GTGCTC	ATTGGC	ACGGOG	CGOCTG
GACTAA	AGGAGC	GAAGOG	ATACTG
OOCAGC	TAGTGC	TTGACG	GTTGTG
GOCOOC	ACATGC	GCTAOG	COGATG
CGGOCC	ATCCAC	GTCOGG	TGAATG
AAAEOC	TAOGAC	CAGCGG	AOCTTG
TTTCOC	CTGGAC	AGAGGG	TATTTG
AGOGCC	GGCAAC	TCTGGG	TGACCA
TCAGOC	AATAAC	AACAGG	CTGGCA
CAGACC	CGATAG	CGTAGG	CGCACA
CTAACC	TCTTAC	GGGTGG	ACAACA
CACTOC	TGGGTC	TTATGG	TATACA
ATGTOC	CCTGTC	GOGCAG	TOCTCA
GGTTCC	TTCATC	AGTCAG	CTTTCA
TGOCGC	GAATTC	CATGAG	ATGOGA
GATOGC	TAOCOG	TOCAAG	ACOGGA
GOGGGC	CCAOGG	GTAAAG	CGGGGA
CAAGGC	CTOGCG	AAATAG	TCGAGA
ATTGGC	ACGGCG	CGOCTG	GGAAGA
AGGAGC	GAAGOG	ATACTG	GCTTGA
TAGTGC	TTGACG	GTTGTG	OOCCAA
ACATGC	GCTAOC	COGATG	TAGCAA
ATOCAC	GTCCGG	TGAATG	GTTCAA
TAOGAC	CAGOGG	AOCTTG	GCAGAA
CTGGAC	AGAGGG	TATTTG	TGTGAA
GGCAAC	TCTGGG	TGACCA	CAAAAA
AGGAGC	GAAGOG		

The application for the error correcting codes in the edit metric is to provide embeddable bar codes for cDNA libraries. Because these bar codes are to be embedded in constructs there are a number of constraints on the sequence that may be used that are driven by biology beyond the need for error correction. In making the constructs various restriction enzymes are used which cut a DNA strand at a particular pattern. We must avoid creating additional instances of this pattern either within our bar codes or as a side effect of embedding our bar codes into the construct. Sub-strings of the form TT or AAA will interfere with use of the construct because of a long sting of T’s near the point were the bar code is embedded.

It turns out that modifying the evolutionary algorithm to deal with such constraints is not difficult. In the seed generator, mutation operators, and in the greedy algorithm, a short piece of code is called that checks the acceptability of each string relative to the biological constraints.

## 6. SUMMARY

Three examples of applications of evolutionary computation to bioinformatics are presented here. The first differs from the simple introductory example, the string evolver, only by having a different fitness function. With a fitness function that minimized the randomness of an alignment of a set of 315 sequences the string evolver was able to bring a motif correlated with transposon insertion over the threshold of detectability. While quite simple as an example of evolutionary computation, this application solved a real problem. Because of the essential simplicity of evolutionary computation, the entire software development effort needed to solve that problem took an afternoon.

The creation of finite state machines to learn how to second-guess a standard primer-picking package was a more difficult effort. A large amount of data, in the form of scored primers, was required before the effort began. The choice of a representation was not as obvious as in the first example. Finite state machines are well able to pick out a pattern no matter where it appears along the length of the primer and so were selected as the representation for our classifiers. The standard fitness function—scoring the number of correct predictions—did not work well in preliminary studies. The use of an incremental reward, computed as a primer traverses the finite state machine, worked well enough to have an impact on future costs. Classifiers located with the incremental fitness function will decrease the number of bad primers used. Substantial room remains for improvement. The evolutionary algorithm used to evolve finite state machines to classify primers (equivalently: to learn the patterns in the scored primer training set) was a fairly standard evolutionary algorithm. The only feature not completely standard was the fitness function. It is also worth reminding the reader that this evolutionary algorithm over-trained the finite state machines when permitted to hybridize a second time. It is not possible to over-emphasize the need for cross validation when learning from data.

The creation of larger sets of error correcting DNA bar codes used a new type of evolutionary algorithm called a greedy closure evolutionary algorithm. The basic notion is to first choose a greedy algorithm that extends partial structures. In this case Conway's lexicode algorithm is the greedy algorithm. The representation for this type of evolutionary algorithm is a small initial part of the structure, in this case a seed of three initial DNA bar codes. The fitness is the quality (in the case of error correcting codes: size) of the final structure constructed by the greedy algorithm. While we used this technique to find larger sets of DNA bar codes it has many other possible applications.

These three examples, while quite different from one another, do not do justice to the breadth of evolutionary computation. Evolutionary computation has been used since at least the 1960s (Fogel, Owens, and Walsh, 1965) with techniques similar to those used in the primer-picking example. Foundational works in the area include (Goldberg, 1989; Holland, 1992) which introduce a type of evolutionary computation called *genetic algorithms*. Evolution of variable sized structures, including whole computer programs, comes under the name of *genetic programming*



(Kinnear, 1994; Kinnear, and Angeline, 1994; Koza., 1992 and 1994). While these techniques have in common the basic structure given in Figure 1, they each incorporate unique features and potential pitfalls.

Evolutionary computation started as a machine learning and optimization techniques, having been discovered many times in many places. It was not practical until the late 1980s when the size of widely available computers grew to where it could support the long run times required. The algorithms used by evolutionary computation are fast to write, slow to run, and easy to specialize for particular tasks. In general, pure evolutionary computation does not perform well on problems that have been studied for a long time. This is because pure evolutionary computation is too simple to take advantage of expert knowledge about problems. Hybrid evolutionary algorithms, where evolutionary computation is blended with other techniques, can incorporate expert knowledge and does often compare well with or beat other techniques. The DNA bar codes are an example of a high performance hybrid technique.

Evolutionary computation is an option for problem solving best used in the initial, exploratory stages of a project. Algorithms that are less general-purpose can almost always out-perform evolutionary computation. Such specialized algorithms, while they supplant evolutionary computation techniques, may require knowledge gained with initial studies of the problem that used evolutionary computation. In summary, evolutionary computation is so easy to use that it is a good choice for brain-storming and prototyping. It is also quite a lot of fun.

## 7. REFERENCES

- Ashlock, D.A., and Joenks, M., 1998, ISAc lists, a different representation for program induction. In *Genetic Programming 98, Proceedings of the Third Annual Genetic Programming Conference*, pages 3–10. Morgan Kaufmann.
- Ashlock, D.A., Wittrock, A., and Wen, T.-J., 2002, Training finite state classifiers to improve pcr primer design. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 13–18. IEEE Press.
- Ashlock, D.A., 2004, *Optimization and Modeling with Evolutionary Computation*. Springer-Verlag, Inc. New York, NY.
- Baldi, P., and Brunak, S., 2001, *Bioinformatics, the Machine Learning Approach*, second edition. MIT Press, Cambridge, MA.
- Dietrich, C., Cui, F., Packila, M., Ashlock, D., Nikolau, B., and Schnable, P.S., 2002, Maize mutansposons are targeted to the 5' utr if the gl8a gene and sequences flanking mu target site duplications throughout the genome exhibit non-random nucleotide composition, *Genetics*, **160**:697–716.
- Fogel, G.B., and Corne, D.W., 2003, *Evolutionary Computation in Bioinformatics*. Morgan Kaufmann Publishers, Boston, MA.
- Fogel, L.J., Owens, A.J., and Walsh, M.J., 1965, Artificial intelligence through simulated evolution. In *Biophysics and Cybernaetic Systems: Proceedings of the 2<sup>nd</sup> Cybernetic Sciences Symposium*, pages 131–155.
- Goldberg, D.E., 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA.
- Holland, J.H., 1992, *Adaption in Natural and Artificial Systems*. The MIT Press, Cambridge, MA.

- Kinnear, K., 1994, *Advances in Genetic Programming*. The MIT Press, Cambridge, MA.
- Kinnear, K., and Angeline, P., 1994, *Advances in Genetic Programming, Volume 2*. The MIT Press, Cambridge, MA.
- Koza., J.R., 1992, *Genetic Programming*. The MIT Press, Cambridge, MA.
- Koza., J.R., 1994, *Genetic Programming II*. The MIT Press, Cambridge, MA.
- Peter, J., 1997, Angeline. Subtree crossover: Building block engine or macromutation? in: *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J.R. Koza, K. Deb, M. Darigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo, eds, pages 9–17. Morgan Kaufmann, 1997.

Genome Exploitation

Data Mining the Genome

Gustafson, J.P.; Shoemaker, R.; Snape, J.W. (Eds.)

2005, XVI, 252 p., Hardcover

ISBN: 978-0-387-24123-4