# Chapter 15

# Application to Bioinformatics

This chapter gives examples of applications of evolutionary computation to bioinformatics. We will start with an application requiring only the very simple sort of evolutionary computation from Chapter 2. The fitness function will align binary strings with a type of genetic parasite called a *transposon*. The next application will evolve finite state automata to try to improve the design of polymerase chain reaction (PCR) primers. The third example will use evolutionary computation to locate error correcting codes for DNA, useful in bar codes for genetic libraries. The final application is a tool for visualizing DNA created with finite state automata combined with a fractal technique using an iterated function system.

## 15.1 Alignment of Transposon Sequences

A *transposon* is a form of genetic parasite. A genetic parasite is a sequence (or string) of DNA bases that copies itself at the expense of its host. It appears multiple times, possibly on different chromosomes, in an organism's genome.

In order to discuss transposons, we will need to discuss a bit of molecular biology first. Deoxyribonucleic acid (DNA) is the primary information storage molecule used by living organisms. DNA is very stable, forming the famous double helix in which complementary pairs of DNA sequences bind in a double spiral. This structure gives stability, but means that manipulating DNA requires a good deal of biochemical effort. Because there is a trade-off, in biochemical terms, between stability and usability, DNA is transcribed into a less stable but more usable form: ribonucleic acid (RNA). RNA is then sent to a subcellular unit called a *ribosome* to be converted into protein. Proteins are the workhorse molecules of life, performing much of the active biochemistry. The central dogma of molecular biology is that the information in DNA follows the path given in Figure 15.1.

The complementary binding of DNA bases does not only lend stability to the DNA

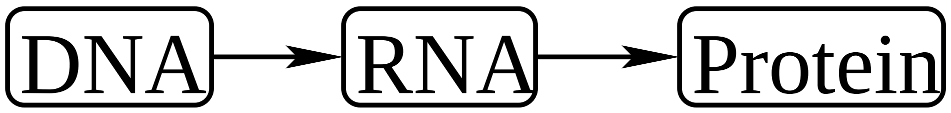


Figure 15.1: The central dogma of molecular biology

molecule, it also enables the process of copying the information. There are 4 DNA bases: **C**, **G**, **A**, and **T**. The bases **C** and **G** bind, as do the bases **A** and **T**. When DNA is copied to make RNA, the RNA that is made is the complement with **C** copied as **G**, **G** copied as **C**, **A** copied as **T**, and **T** copied as **A**.

There are 3 kinds of transposons. Type I transposons are segments of DNA that cause the cell to transcribe RNA from them. This RNA is then transformed back into DNA by an enzyme called *reverse transcriptase* and integrated back into the genome. These transposons are thought to prefer specific positions to reintegrate their copies into the genome. Type II transposons simply copy themselves from DNA directly to DNA. Type III transposons are similar to type II, save that they average much shorter and use a different copy mechanism. Almost any text on molecular biology, e.g. [26], contains a full description of the state of knowledge about transposons and their intriguing relationship with viruses.

In this section, we will be working with the problem of identifying the sequence that type I transposons use to integrate back into the genome. The data set available on the webpage associated with this text (click on data and then Chapter 15) was gathered by finding the point at which a particular transposon integrated into the genome. This is done by comparing a gene with no transposons to the same gene with transposons. Where the genes differ is a transposon site.

The problem is that, while we know where the transposon integrated, we do not know into which strand of DNA it integrated. If there is a particular sequence of DNA bases required for integration, it appears on one strand and its complement appears on the other. This means that, if we want to compare the insertion sites, we must first decide into which strand the transposon integrated.

This is where the evolutionary computation system comes into play. It is used to decide whether to use a DNA sequence as found or to use its reversed complement. This is a binary problem, so the binary string evolvers we learned about in Chapter 2 will be useful.

We use the reverse complement instead of just the complement, because DNA strands have opposite orientations on opposite strands. This means that, if the transposon integrated on the opposite strand, we should not only complement the DNA but reverse it (turn it end-for-end).

**Example 15.1 Reverse complementation.** *The DNA sequence,* **CGATTACTGTG**, *has reverse complementary sequence* **CACAGTAATCG**. *Not only do we apply the swaps* **C**⇔**G** *and* **A**⇔**T**, *but we also rewrite the sequence in reversed order.*

```
01234567890123456789012345678
.---------.---------.--------
CACCGCACCGCACTGCATCGGTCGCCAGC
ACCCGCATCGGTCGCCAGCCGAGCCGAGC
CACCGCATCGGTCGCCAGCCGAGCCGAGC
CACTGCATCGGTCGCCAGCCGAGCCGAGC
GCTCGACACACGGGCAGGCAGGCACACCG
```

Figure 15.2: A gapless alignment of 5 DNA sequences

Suppose that we a sequence containing several hundred examples of transposon insertion sites. We delete the regions of the sequence without transposon sites and line up the regions with insertion sites so that the sites coincide. We then need to specify an orientation for each insertion site sequence, either forward or reverse-complement. This specification will be what we evolve. A binary string gene of length $N$ can specify the orientation of a set of $N$ sequences. For a data set with $N$ sequences, we thus need a binary string evolver that operates on length $N$ strings. It remains to construct a fitness function.

In this case, we presume that there is a conserved motif at the point of transposon insertion. A *motif* is a set of characters, possibly with some wildcards or multi-base possibilities. So,

**C**, **G**, **A** or **T**, anything, **C**, **C**

is an example of a motif that includes 8 sequences: **CGACCC**, **CGTCCC**, **CGAGCC**, **CGTGCC**,**CGAACC**, **CGTACC**, **CGATCC**, and **CGTTCC**. There may also be some other properties, like an above average fraction of **A**s and **T**s. Because of this, there is reason to believe that, when the sequences are placed in their correct alignment, there will be a decrease in the total "randomness" of the base composition of the alignment.

**Definition 15.1** *For a collection $C$ of DNA sequences, define $P_X$, $X \in \{$***C***,* **G***,* **A***,* **T***$\}$ to be the fraction of the bases that are $X$.*

**Definition 15.2** *A* **gapless alignment** *of a set of sequences of DNA bases consists of placing the sequences of DNA on a single coordinate system so that corresponding bases are clearly designated. An example of such an alignment appears in Figure 15.2. (Gapped alignments will be discussed in Section 15.3.)*

The transposon insertion data has to be trimmed to make the DNA sequences the same length. This means that the orientation, either forward, or reversed and complemented, is the only thing that can change about the way a sequence fits into an alignment. We now need a fitness function that will compute the "non-randomness" of a given selection of orientations of sequences within an alignment.

**Definition 15.3** *Assume that we have a gapless alignment of $N$ DNA sequences, all the same length $M$. View the alignment as a matrix of DNA bases with $N$ rows and $M$ columns. Let $X_i$ be the fraction of bases in column $i$ of the matrix of type $X$, for $X \in \{\mathbf{C}, \mathbf{G}, \mathbf{A}, \mathbf{T}\}$. Then, the* **non-randomness** *of an alignment is*

$$\sum_{i=1}^{M}(X_i - P_X)^2.$$

The non-randomness function is to be maximized. Lining up the motif at the point of insertion will yield less randomness. Notice that we are assuming the DNA sequences are essentially random away from the transposon insertion motif. We are now ready to perform an experiment.

**Experiment 15.1** *Write or obtain software for a steady state evolutionary algorithm using single tournament selection with tournament size 7 that operates on binary genes of length $N$. Download transposon insertion sequences from the website associated with this book; $N$ is the number of these sequences. Use two point crossover and probabilistic mutation with probability $\frac{1}{N}$. Use a population of 400 binary strings for 400,000 mating events. Use the non-randomness fitness function. Run the algorithm 100 times and save the resulting alignments. If an alignment specifies the reverse complement of the first sequence, reverse complement every sequence in the alignment before saving it. How often do you get the same alignment? Are alignments that appear often the most fit or are the most fit alignments rare?*

This experiment produces alignments and gives us a baseline notion of an acceptable fitness. With the baseline fitness in hand, let's perform a parameter sensitivity study for various algorithm parameters. We will start with mutation rate.

**Experiment 15.2** *Modify the software from Experiment 15.1 as follows. Take the most common fitness you got in Experiment 15.1 and assume any alignment with this fitness is "'correct." This let's us compute a time-to-solution. Now, repeat the previous experiment, but for mutation rates $12N$, $1N$, $32N$, and $2N$. Report the impact on time-to-solution and the number of runs that fail to find a solution in 500,000 mating events.*

Now, let's look at the effects of varying population size and tournament size.

**Experiment 15.3** *Repeat Experiment 15.2 using the best mutation rate from Experiment 15.2. Use all possible pairs of population sizes 100, 200, 400, 800 and tournament sizes 4, 7, and 15. Report the impact on time-to-solution and the number of runs that fail to find a solution in 500,000 mating events.*

Now that we have a way of aligning the transposon insertion sites, we need a way of finding the motif. A motif is a sequence of DNA bases with wildcard characters. A motif could thus be thought of as a string over a 15-character alphabet consisting of the nonempty subsets of {**C**, **G**, **A**, **T**}. We will encode this alphabet by letting **C**=8, **G**=4, **A**=2, and **T**=1 and by adding the numbers. Thus, the number 9 is a partial wildcard that matches the letters **C** and **T**. With this encoding of a motif, we can use a string evolver to search for a motif. As always, we need a fitness function.

**Definition 15.4** *A* **kth order Markov model** *of a collection of DNA sequences is an assignment to each DNA sequence of length k an empirical probability that the next base will be* **C**, **G**, **A**, *or* **T**. *Such a model is built from a collection of target DNA sequences in the following manner. For each length k sub-sequence S appearing in the target DNA, the number of times the next base is a* **C**, **G**, **A**, *or* **T** *is tabulated. Then, the probabilities are computed by dividing these empirical counts by the number of occurrences of the sub-sequence S. For subsequences that do not appear, the first order probabilities of each DNA base are used.*

**Example 15.2** *Let's take the target DNA sequence:*

```
AAGCTTGCAGTTTAGGGCCCCTGATACGAAAGAAGGGAGGTCCGACAGCCTGGGGCCGAC
TCTAGAGAACGGGACCCCGTTCCATAGGGTGGTCCGGAGCCCATGTAGCCGCTCAGCCAG
GTCCTGTACCGTGGGCCTACATGCTCCACCACCCCGTGACGGGAACTTAGTATCTAGAGT
TATAAGTCCTGCGGGTCCGACAACCTCGGGACCGGAGCTAGAGAACGGACATTAGTCTCC
TGGGGTGGTCCGGAGCCCGTACAGCCGCTCAGCCTAGTCCCGTACCATGGTCCTGCACGC
TCCACCGCCCTGTGACAAGTGTCCTAGTATCTAGAACCGCGACCCAAGGGGGTCCGGACA
AGCAACTTGGCCACCCGGACTAAAACCTGCAGGTCCCTAGCATGTATCAAAGGGCGACTA
ATGTCAGACGGAGAACCCTATGAGGTGTACTACTAACGCTTCCTAGCTAAAAGTTGTGTA
CAGATCCAGATCTCGGCGAGTTTGCCTCCCGAGGATTGTTGACAACCTTTTCAGAAACGC
TGGTATCCAACCTCAACACATCAAGCCTGCATCCGAGGCGGGGGGCCAGGTACTAAGGAG
AAGTCAACAACATCGCACATAGCAGGAACAGGCGTTACACAGATAAGTATTAAATACTGC
TTAGAAGGCATTATTTAATTCTTTACAAAAACAGGGGAAGGCTTGGGGCCGGTTCCAAAG
AACGGATGCCCGTCCCATAGGGTGGTCCGGAGCCTATGTGGCCGGTTAGCCTGGTTCCGT
ACCCAAAATCCTGCACACTCCACCGCTCTGTGGTGGGTGTCCTAGTATTTAAAACTAAAG
```

*To build a 2nd ($k = 2$) order Markov model of the DNA sequence, we need to tabulate how many times a* **C**, **G**, **A**, *or* **T** *appear after each of the possible 2-character sequences. This is the work computers were meant to do, and they have, yielding the tabulation:*

| *Sequence* | $N_C$ | $N_G$ | $N_A$ | $N_T$ |
|---|---|---|---|---|
| *CC* | *18* | *25* | *16* | *23* |
| *CG* | *9* | *17* | *9* | *8* |
| *CA* | *11* | *16* | *15* | *12* |
| *CT* | *12* | *14* | *19* | *8* |
| *GC* | *20* | *6* | *10* | *12* |
| *GG* | *13* | *26* | *17* | *20* |
| *GA* | *14* | *14* | *13* | *6* |
| *GT* | *17* | *13* | *14* | *10* |
| *AC* | *17* | *9* | *20* | *11* |
| *AG* | *16* | *19* | *16* | *13* |
| *AA* | *19* | *17* | *17* | *4* |
| *AT* | *10* | *8* | *7* | *7* |
| *TC* | *27* | *3* | *8* | *7* |
| *TG* | *10* | *14* | *5* | *13* |
| *TA* | *13* | *18* | *11* | *10* |
| *TT* | *6* | *7* | *12* | *7* |

*Dividing through by the number of times each 2-character sequence occurs with another base after it yields the second order Markov model for the target above.*

| **Markov model** $k = 2$ | | | | |
|---|---|---|---|---|
| *Sequence* | $P_C$ | $P_G$ | $P_A$ | $P_T$ |
| *CC* | *0.220* | *0.305* | *0.195* | *0.280* |
| *CG* | *0.209* | *0.395* | *0.209* | *0.186* |
| *CA* | *0.204* | *0.296* | *0.278* | *0.222* |
| *CT* | *0.226* | *0.264* | *0.358* | *0.151* |
| *GC* | *0.417* | *0.125* | *0.208* | *0.250* |
| *GG* | *0.171* | *0.342* | *0.224* | *0.263* |
| *GA* | *0.298* | *0.298* | *0.277* | *0.128* |
| *GT* | *0.315* | *0.241* | *0.259* | *0.185* |
| *AC* | *0.298* | *0.158* | *0.351* | *0.193* |
| *AG* | *0.250* | *0.297* | *0.250* | *0.203* |
| *AA* | *0.333* | *0.298* | *0.298* | *0.070* |
| *AT* | *0.312* | *0.250* | *0.219* | *0.219* |
| *TC* | *0.600* | *0.067* | *0.178* | *0.156* |
| *TG* | *0.238* | *0.333* | *0.119* | *0.310* |
| *TA* | *0.250* | *0.346* | *0.212* | *0.192* |
| *TT* | *0.188* | *0.219* | *0.375* | *0.219* |

*For each 2-character sequence, we have the probability that it will be followed by each of the 4 possible DNA bases.*

What use is a $k$th order Markov model? While there are a number of cool applications, we will use these Markov models to baseline the degree to which a motif is "surprising." In order to do this, we will use the Markov model to generate sequences "like" the sequence we are searching. A $k$th order Markov model of a given set of target DNA sequences can be used to find more sequences with the same $k$th order statistics. Let's look at the algorithm for this.

**Algorithm 15.1 Moving Window Markov Generation Algorithm**

**Input:** *A $k$th order Markov model and a number $m$*
**Output:** *A string of length $m$*
**Details:** *Initialize the algorithm as follows. Select at random a sequence of $k$ characters that appeared in the target DNA sequence used to generate the original Markov model. This is our initial window. Using the empirical distribution for that window, select a next base. Add this base to the end of the window and shift the window over, discarding the first character. Repeat this procedure $m$ times, returning the characters generated.*

Algorithm 15.1 can be used to generate any amount of synthetic DNA sequences with the same $k$th order base statistics as the original target DNA used to create the Markov model. This now puts us in a position to define a fitness function for motifs.

**Definition 15.5** *Suppose we have a set of target DNA sequences, e.g. the set of aligned transposon insertion sequences generated in Experiments 15.1-15.3. The* **count** *for a motif is the number of times a sequence matching the motif appears in the target DNA sequences.*

**Definition 15.6** *Suppose we have a set of target DNA sequences, e.g., the set of aligned transposon insertion sequences generated in Experiments 15.1-15.3. The* **synthetic count** *for a motif is the number of times a sequence matching the motif appears in a stretch of synthetic DNA, generated using Algorithm 15.1 with a Markov chain created from the target DNA or an appropriate set of reference DNA.*

**Definition 15.7** *The* **p-fitness** *of a motif is the probability the count of a motif will exceed its synthetic count. The* **$p_N$-fitness** *of a motif is the estimate of the p-fitness obtained using $N$ samples of synthetic DNA. Compute the $p_N$-fitness of a motif as follows. Obtain target and reference DNA (they may or may not be the same). Pick $k$ and generate a $k$th order Markov model from the reference DNA. Compute the count of the motif in the target. Pick $N$ and compute the synthetic count of the motif in $N$ sequences the same length as the target sequence generated with the Markov chain derived from the reference DNA. The fraction of instances in which the synthetic count was at least the count is the p-fitness.*

The $p$-fitness of a motif is to be minimized; the harder it is for the synthetic count to exceed the count of a motif, the more surprising a motif is. Notice that the $p$-fitness is an approximate probability and, so, not only selects good motifs, but gives a form of certificate for their statistical significance. It is important to remember that this $p$-value is relative to the choice of reference DNA. The transposon insertion studies used in this chapter are published in [10] and studied insertion into a particular gene. A good set of reference DNA is thus the sequence of that gene, available on the website for this book. Let's go find some motifs.

**Experiment 15.4** *Write or obtain software for a steady-state evolutionary algorithm using single tournament selection with tournament size 7 that operates on string genes over the motif alphabet described in this section. Download the glu18 gene sequence from the website for this text for use as reference DNA. Build a 5th ($k = 5$) order Markov model from the glu18 code and use it to implement the p-fitness function for motifs in aligned sets of transposon insertion sites from Experiments 15.1-15.3. Use two point crossover and single point mutation in the motif searcher. Use a population of motifs of length 8 for 100,000 mating events with a population size of 400. Perform 100 runs. Sort the best final motifs found in each population by their fitnesses. Report the number of times each motif was found. Are there cases where the sequences specified by one motif were a subset of the sequences specified by another?*

If this experiment worked for you as it did for us, you have discovered a problem with this technique for finding motifs: what a human thinks of as a motif is a bit more restrictive than what the system finds. The system described in Experiment 15.4 managed to find several motifs with high fitness values, *but* appearing in the target sequence only once each. This means that our motif searcher can assemble a motif from rare strings which has a high $p$-value but is not of all that much interest. A possible solution to this problem is to insist numerically that the motifs be more like what people think of as motifs.

**Definition 15.8** *A character in the motif alphabet stands for one or more possible matches. The* **latitude** *of a character in the motif alphabet is the number of characters it stands for minus one. The* **latitude** *of a motif is the sum of the latitudes of its characters.*

**Experiment 15.5** *Repeat Experiment 15.4. Modify the fitness function so that any motif with a latitude in excess of d is awarded a fitness of 1.0 (the worst possible). Perform 100 runs for $d = 5, 6, 7$. Contrast the results with the results of Experiment 15.4.*

It would be possible to perform additional experiments with the motif searcher (you are urged to apply the searcher to other data sets), but instead we will move on to an application of evolutionary algorithms to a problem in computational molecular biology. If you are interested in further information on motif searchers, you should read [28] and look at the Gibbs sampler, a standard motif location tool, [15].

## Problems

**Problem 15.1** *Give a 20-base DNA sequence that is its own reverse complement.*

**Problem 15.2** *The non-randomness fitness function compensates for first-order deviation from uniform randomness in the DNA used by computing the fractions $P_X$ of each type $X$ of DNA base. Write a function that compensates for second-order randomness; the statistics of pairs of adjacent DNA bases.*

**Problem 15.3** *Explain in a sentence or two why the function given in Definition 15.3 measures non-randomness.*

**Problem 15.4** *Give a motif, of the sort used in this section, that matches as few sequences as possible, but also matches each of the following.*

| | | | |
|---|---|---|---|
| AAGCTCGAC | CACGGGCAG | CGGGCAGGC | GGGGCAGGC |
| ACACAGGGG | CACTCCGCC | CTACCAAAG | GTCGCCAGC |
| ACCGGATAT | CACTGCATC | CTCCGTCTA | GTCGCCAGC |
| AGCCGAGCC | CCACCGGAT | CTGTCGATA | GTCGCCAGC |
| CACAGGGGC | CCCCAAATC | CTGTGTCGA | GTGCGGTGC |
| CACCCGCAT | CCCTCATCC | GAGTAGAGC | TCCTAGAAT |
| CACCGCACC | CCGCACCGC | GCTGCGCGC | TCCTGATGG |
| CACCGCATC | CGGCTCGGC | GGAGAGAGC | TTCACTGTA |

**Problem 15.5** *Construct and defend a better fitness function for motifs than the p-fitness.*

**Problem 15.6** *Give an efficient algorithm for checking the count of a motif in a sequence of DNA.*

**Problem 15.7 Essay.** *Explain why the order of the Markov model used in Experiments 15.4 and 15.5 must be shorter than the length of the motifs being evolved to get reasonable results.*

**Problem 15.8 Essay.** *Based on Experiments 15.1-15.3, make a case that the non-randomness fitness function on the data set used is unimodal or polymodal.*

**Problem 15.9 Essay.** *Why is maximizing the non-randomness fitness function the correct choice?*

**Problem 15.10 Essay.** *With transposon data, we have a simple method of locating where the transposon inserted: there is a transposon sequence where before there was none. This gives us an absolute reference point for our alignment and so leaves us to worry only about orientation. Describe a representation for gapless alignment where we suspect a conserved motif but do not know exactly where, laterally in the DNA sequence, that alignment is.*

**Problem 15.11 Essay.** *Taking the minimal description of transposable elements (transposons) given in this section, outline a way to incorporate structures like transposable elements into an evolutionary computation system. If you are going to base your idea on natural transposons, be sure to research the three transposon types and state clearly from which one(s) you are drawing inspiration.*

## 15.2 PCR Primer Design

Polymerase chain reaction (PCR) is a method of amplifying (making lots of copies of) DNA sequences. DNA is normally double-stranded. When you heat the DNA, it comes apart, like a zipper, at a temperature determined by the fraction of **GC** bonds (**GC** pairs bind more tightly than **AT** pairs). Once they are apart, an enzyme called DNA-polymerase can grab individual bases out of solution and use them to build partial double strands. As the DNA cools, it re-anneals as well as being duplicated by the polymerase. A single PCR cycle heats and then cools the DNA with some of it being duplicated by the polymerase.

*Primers* are carefully chosen short segments that re-anneal earlier, on average, than the whole strands of DNA. If we start with a sample of DNA, add the correct pair of primers, a supply of DNA bases, and the right polymerase enzyme, then we will get exponential amplification (roughly doubling in each cycle of the reaction) of the DNA between the two primers. (The primers land on opposite strands of DNA.) A diagram of this process is given in Figure 15.3

```
CCAGTGTTACTAGGCTACTACTGCGACTACG
|||||||||||||||||||||||||||||||
GGTCACAATGATCCGATGATGACGCTGATGC

CCAGTG==>>
||||||
CCAGTGTTACTAGGCTACTACTGCGACTACG

GGTCACAATGATCCGATGATGACGCTGATGC
                         ||||||
                      <==TGATGC
```

Figure 15.3: Double-stranded DNA and single-stranded DNA undergoing replication

The length of each new strand is controlled by time and temperature. Typically, you let the strands grow, on average, *beyond* the annealing point for the complementary primers. Since the primers are on opposite strands, they amplify in opposite directions and only the DNA between them undergoes exponential growth. Primers are length 17-23, typically, and the amplified DNA is a few hundred to several thousand bases. Evolutionary computation can help pick good primers.

Existing primer picking tools make sure that the DNA biophysics of a pair of primers is correct. These tools match the melting temperature of the right and left primer, make sure the primer does not anneal with itself or its partner, and check for other problems that can ruin a PCR reaction. (There are potential problems specific to the organism for which the primers are being designed.) Transposons, discussed in Section 15.1, are a source of duplicate sequences.

A problem current primer picking tools do not address is the problem of duplicate sequences. Given the size of genomes, 20-character DNA sequences (e.g., typical primers) should be unique. If genomes were generated at random, such sequences would be unique. However, many biological processes duplicate sequences within a genome. What effect does this have on a PCR reaction?

If both members of a primer pair are inside a duplicated sequence in an organism, then they will amplify both copies. If the duplicates are identical, this isn't a problem for the PCR reaction (it may be one for the biologist). Often, though, duplicated sequences have diverged, and, so, many different sequences are amplified. These amplifications happen at slightly different exponential rates and diverge from each another exponentially. In practice, primer pairs in a duplicated sequence are unusable.

Another problem is created when one member of a primer pair is part of a duplicated sequence. If the number of copies of this sequence is small, then the exponential amplification of the paired sequence permits things to work properly. Some transposons have a ridiculous number of copies and, in this case, one half of the primer pair anneals in so many places that the amplification of the region flanked by the primer pair goes nowhere. So what do we do?

The technique that follows was developed in the context of a large sequencing project in corn. Tens of thousands of primers were designed and tested. The results of these tests can be used to create a fitness function for evolved predictors that guess which primers will or will not work. (This data is available on the website associated with this book.)

The basic idea is this. Use primer picking software to generate multiple primers for a given sequence target. Performance predictors, trained on past results, examine these primers and rate them. The more highly rated primers will have a better chance of working, if the predictors are correctly generalizing about the sequence features that make primers work or fail. We also save a set of primers with known performance on which to test our predictors; this is called *cross-validation.*

We need to chose a representation for our primer performance predictors. In this case, a finite state automaton is a natural choice, as it can process strings of characters and embed its opinion of them in its state space. Figure 15.4 shows a finite state automaton specialized for use on DNA. The automaton shown is a Moore automaton with states labeled with the automaton's output.

In order to train finite state automata to predict PCR performance, we need a fitness

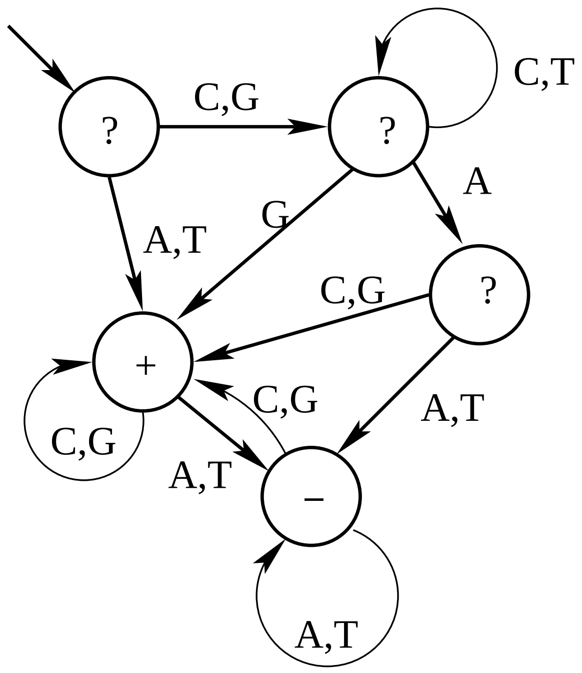


Figure 15.4: A finite state automaton that can be driven by DNA and used as a performance predictor (The input alphabet is **C, G, A ,T** and the output alphabet is +, **-**, **?**, interpreted as "good", "bad", and "don't know.")

function. The data for training is in the format shown in Figure 15.5 and is available on the text website. We will divide the data into randomly selected training and cross-validation sets, with $\frac{1}{5}$ of the sequences (selected uniformly at random) in the cross-validation set. Primers in the training set are marked with a 0, 1, or 2; those marked with 0 are considered "good"; the others are considered "bad." The fitness function will select for automata that end in a + state on a "good" primer and a **-** state on a "bad" primer. (Note: we are treating primers that don't amplify their targets and those that amplify multiple targets as "bad.")

**Definition 15.9** *The* **raw prediction fitness** *of a finite state automaton is computed as follows. Initialize the fitness to zero. Run each primer in the training set through the finite state automaton. If it ends in a + state for a good primer or a - state for a bad primer, add 1 to the fitness. If the automaton ends in a* **?** *state, add nothing to the fitness. Otherwise, subtract 1 from the fitness.*

**Experiment 15.6** *Write or obtain software for evolving Moore automata with transitions driven by the DNA alphabet and state labels* +, **-**, **?**. *Divide randomly the available primer data from the text website into training ($\frac{4}{5}$) and cross-validation ($\frac{1}{5}$) data. Treat the states*

```
...
2 CTCCACTATAGCTGCCGTCG
2 TACAGGGACATCTGGATGGG
0 CTGCAGTACATCTACCACCACC
0 TGCAGAGCTTCGAGCACC
0 CGATCAGCATGTTCATTTGC
1 CAAGGAGGGAGTGATTCAGC
1 AAGAACAGCACTCAATCGGG
1 CAAGGAGGGAGTGATTCAGC
...
```

Figure 15.5: Format for primer training data (Numerical codes are: 0 = primer works; 1 = primer amplifies multiple targets; 2 = primer does not amplify.)

*of the Moore automaton as indivisible objects with the string of states forming the basis of the crossover operator. Perform two point crossover. Use three point mutation. Each single point mutation should change a transition destination, state label, or the initial state. Set the probabilities so that all arrows and labels in a given FSA have the same chance of being affected. Evolve a population of 400 finite state primer predictors for 100,000 mating events using a steady state algorithm with size 7 tournament selection. Let the predictors have 32 states. Run 30 populations.*

*Report both the fitness tracks and a cross-validated final fitness for each run. This latter is computed by assessing the fitness of the entire final population on the cross-validation data and taking the best automaton. Also, report how often the most fit automaton according to cross validation is also the most fit according to the fitness on the training data. Report the density of each of the three state types (*+, -, ?*) in each population. What do these densities suggest? Be sure to save the best predictor from each run for later use.*

The results of this experiment suggest a couple of modifications. Leaving the automata the option of saying “I don't know” gives the system flexibility, but is it happening too often? Also, since there are a finite number of examples of good and bad primers in the training set, there is a possibility of falling into a useless local optimum: the predictor could predict all primers were of whatever type is most common. To avoid these pitfalls, let's improve the experiment.

**Experiment 15.7** *Take the available primer data and divide it into good and bad primers. Randomly select, from whichever sort are more common, a number of examples equal to the number that are less common and then discard the excess of the more common type. This set of training data is now* balanced, *removing the option “guess whatever is most common.” Modify the evolutionary algorithm from Experiment 15.6 to add a lexical fitness: the number* *of* **?** *results, to be minimized. Rerun Experiment 15.6 both with and without the lexical fitness. Discuss the effect of balancing the training data and the impact of using the lexical fitness.*

At this point, we will redesign the fitness function. Insisting that the predictor get the fitness right at its final state is somewhat brittle. Perhaps there are automata that are on the right track, but get the final answer wrong.

**Definition 15.10** *The* **incremental reward fitness function** *is computed in almost the same manner as the raw prediction fitness function. The difference is that the fitness is scored at each state transition. This yields more finely grained fitness information. As a good primer runs through the predictor, 1 is added for each* **+** *state and 1 is subtracted for each* **-** *state with* **?** *still yielding a reward of 0. The opposite is done for bad primers.*

**Experiment 15.8** *Using the non-lexical version of the software, repeat Experiment 15.7 with the incremental reward fitness function in place of the raw fitness function. Document the impact. Examine your best predictor: are there lots of* **?***s near the initial state?*

Now we can try applying some other techniques from earlier chapters. Since there are many patterns in the training data (distinct possible sources of fitness), it follows that different runs will find different patterns. How do we combine patterns from distinct evolutionary runs?

**Definition 15.11** *The practice of* **hybridization** *consists of initializing an evolutionary algorithm with superior genes from multiple populations that have already been evolved.*

**Experiment 15.9** *Repeat Experiment 15.8 incorporating the 30 best-of-run automata saved during Experiment 15.8 into the initial population (in addition to random predictors). Does this impact the results?*

There is a problem with hybridization as performed in Experiment 15.9: it does not control for the effect of unmodified added evolution on the original populations. The 30 hybridized automata evolved through 200,000 mating events, while the others only evolved through 100,000. The next experiment will take far longer to run, but should yield a more meaningful test of the utility of hybridization.

**Experiment 15.10** *Repeat Experiment 15.8 with the following modifications. Set the experiment to run for 100,000 mating events, but save the best-of-run automata (according to the cross validation data) at mating event 50,000. Now, initialize a new set of 30 runs with these best-of-run automata included in the initial population and run them for 50,000 mating events. We have two sets of runs, both run for 100,000 mating events, but with the second set benefitting from hybridization. In addition to reporting the other performance measures, discuss the impact of hybridization.*

The number of states used is a measure of the amount of information a finite state automaton can store. The experiments performed thus far yield a baseline for performance. Let's check the sensitivity of the system to the number of states.

**Experiment 15.11** *Repeat Experiment 15.9 but with 48 and 64 states. What impact does this change have on the baseline and hybridized runs?*

For our last experiment, let's check the sensitivity to mutation rate.

**Experiment 15.12** *Repeat Experiment 15.10 using only the number of states that performed best. Use 1, 5, and 7 point mutation and compare with the 3 point mutation used in Experiment 15.10. What impact does this change have on the baseline and hybridized runs?*

This section is a modest introduction to using machine learning to improve primer design. The technique of hybridization is a potentially valuable one. The incremental reward fitness function is an example of a redesign of a fitness function that makes the hill climbing functionality of an evolutionary algorithm more effective. There are a number of other possible technologies for this sort of machine learning - Markov modeling of good and bad primers, for example. There are also other EC-techniques we could use, such as graph-based algorithms. We now leave primers for a much stranger application, DNA bar codes, with a new type of fitness function.

## Problems

**Problem 15.12** *Is a predictor that has a* **?** *on all its states in a local optimum or a big flat space with uphill paths at its edge? Defend your conclusions.*

**Problem 15.13** *Would a real function optimizer benefit from hybridization? Explain.*

**Problem 15.14** *Prove that there is a finite state automaton that can achieve maximal raw prediction fitness on the training data. Assume no primer appears in the training set twice.*

**Problem 15.15** *Explain why it is impossible to receive a reward on every state transition when computing incremental reward fitness, no matter what finite state automaton you use.*

**Problem 15.16** *The system developed in this section runs primers through the finite state automaton one at a time. Come up with a fitness function that scores finite state automata on pairs of primers that are used together.*

**Problem 15.17** *Is 32 states a reasonable number for the task in this section? Your answer should involve mathematics, probably counting arguments.*

**Problem 15.18 Essay.** *One of the advantages of GP-automata is that deciders compress the bandwidth of the environment. Specify and defend a decider language which uses GP-automata with 3 or 5 base windows (instead of a single base at a time as the finite state automata do), permitting GP-automata to be used in place of finite state automata.*

**Problem 15.19 Essay.** *Primers work or fail in pairs. That means that a primer that might work perfectly well with a different partner may receive a bad score with the partner with which it was tested. Given this, can we still hope to get useful results from the primer prediction system given in this chapter? Is it important that we are picking the best from among multiple primers when we use the system to select new primers?*

**Problem 15.20 Essay.** *Address the following statement. The finite state automaton whose existence was proved in Problem 15.14 would not perform well on the cross validation set.*

**Problem 15.21 Essay.** *Would hybridization help more with the grid-robot tasks in Chapters 10 and 12 or with playing Iterated Prisoner's Dilemma?*

**Problem 15.22 Essay.** *In Chapter 10, several representations are used for Tartarus controllers: strings, parse trees, and GP-automata. Rank them by the relative benefit you think they would get from hybridization.*

**Problem 15.23 Essay.** *One of the more controversial ideas in evolutionary computation is whether there are building blocks that can be brought together by crossover. The reason for the controversy is mostly failure to think on the part of various vociferous proponents and opponents of the idea. The truth (tm) is that some problems have neat easy-to-assemble building blocks, and others don't. Your topic: can the degree to which hybridization improves performance be used as an objective probe for the presence of building blocks?*

## 15.3 DNA Bar Codes

Our goal in this section is to find an algorithm for creating error correcting codes for DNA libraries. These codes can be used to identify the source that contributed that DNA as part of a sequencing project. We will take some long detours and, along the way, invent a new type of evolutionary algorithm.

Greedy algorithms are familiar to people who study programming or discrete math. (We defined them in Chapter 7 on page 179.) A few, like the algorithms for finding a minimal-weight spanning tree, can be proven to yield optimal results. Other problems, like graph coloring or the Traveling Salesman problem, admit a plethora of greedy algorithms, all of which yield suboptimal results. While it would seem that the control of greedy algorithms is a natural target for evolutionary computation, relatively few methods have been devised.

There are several possible approaches. The approach explored in this section seeks to deflect the behavior of a greedy algorithm by giving it a small hint. The hint is the target of our evolutionary computation, and we call the technique used to evolve good hints a greedy closure evolutionary algorithm.

**Definition 15.12** *A* **greedy closure** *evolutionary algorithm is an evolutionary algorithm which uses a representation consisting of partial structures called* seeds. *The seeds are completed (closed) with a greedy algorithm. The quality of the complete structure, as finished by the greedy algorithm, is the fitness of the seed.*

The structures created from the seeds while evaluating the fitness function will be said to have *grown* from those seeds. In order to understand the bioinformatic application in this section, DNA bar codes, we will need both a small amount of additional molecular biology and some basic theory of error correcting codes. We begin with the error correcting codes.

## Error correcting codes

An *error correcting code* is a collection of strings to be sent over a possibly noisy communications channel. While any collection of strings is technically a code, the science of error correcting codes seeks to create codes that permit us to correct some of the errors that occur during transmission. Thus, a complete error correction system contains not only the code but a decoding algorithm. Let's look at an example.

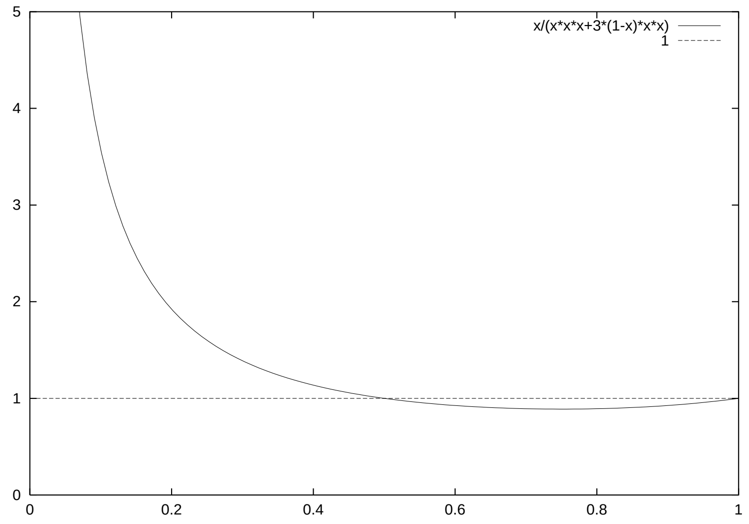
**Example 15.3** *Imagine a pair of neighbors one of which is selling a car and the other of which is contemplating purchasing the car. The neighbors live across a ravine from one another and must cross an arroyo to reach one another. The person selling the car has told the buyer that they must decide if the price is acceptable by 5:00 p.m.: otherwise, the car goes to another buyer who is offering a higher price but is not a friend and neighbor. At 3:50 p.m., a huge storm blows up and wipes out the bridge and the phone lines (and the cell tower for you high tech types). The potential buyer must get a yes or a no to the seller. The neighbors walk out into the backyards of their houses and try to talk over the sound of the flood waters. The seller realizes that it is almost impossible to hear and yells something three times. What can happen?*

*Well "yes" and "no" don't sound that similar, but, with raging flood waters, there is a chance of mishearing what was said. Let's assume there is a probability $\alpha$ of mishearing the result. Then, we get a simple binomial distribution (see Appendix A) which tabulates this way:*

| *Answers misheard* | *Probability* | $\alpha = 0.1$ | $\alpha = 0.2$ |
|---|---|---|---|
| *0* | $(1-\alpha)^3$ | *0.729* | *0.512* |
| *1* | $3\alpha(1-\alpha)^2$ | *0.243* | *0.384* |
| *2* | $3\alpha^2(1-\alpha)$ | *0.027* | *0.096* |
| *3* | $\alpha^3$ | *0.001* | *0.008* |

*A code requires a decoding algorithm. In this case, we will take a majority vote on the answers heard. What does this do to the probability of error? Well, for* $\alpha = 0.1$*, the chance of error drops from 0.1 (with only one yell) to 0.028 for majority vote over three yells. This is about a 3.5-fold decrease in the chance of error. When* $\alpha = 0.2$*, the improvement is from 0.2 to 0.104, about a 1.9-fold improvement. Let's plot this fold improvement:*

$$FoldImprovement(\alpha) = \frac{\alpha}{\alpha^3 + 3\alpha^2(1-\alpha)} = \frac{1}{3\alpha^2 - 2\alpha^3}.$$



*When* $\alpha$ *is small, fold improvement in the chance of understanding correctly with three yells is huge, with a vertical asymptote at zero. The technique ceases to help at* $\alpha = 0.5$ *(as one would expect). The behavior for* $\alpha > 0.5$ *is weird, but no one would use a communications channel with more that a 50% chance of miscommunication.*

The code used in the example is called the *odd length repetition code of length 3*. When working with error correcting codes, the usual thing is to send bits; flipping a bit constitutes an error. If we repeat each bit an odd number of times, then the received bits can be decoded with a simple majority vote. This means that any communications channel that has the chance of flipping a bit $\alpha < 0.5$ can be used with any degree of accuracy at all. The more times you repeat the bit, the more likely you are to decode the bit correctly. What is the price? Repeating the bit uses up a lot of bandwidth.

A repetition code of length $2n + 1$ can decode $n$ errors, but it is not very efficient. A code is a collection of strings or *code words*. The code words of the length 3 repetition code are $\{000, 111\}$. Any code has a set of code words, and they are the words that are sent down the communications channel. The *received words* are the ones we try to correct. If we receive a code word, we assume that there were no errors. If we receive a word that is not a code word, then we try to find the code word closest to the received word. In this case, the notion of closest used is the *Hamming metric* which defines the distance between two words to be the number of positions in which they disagree.

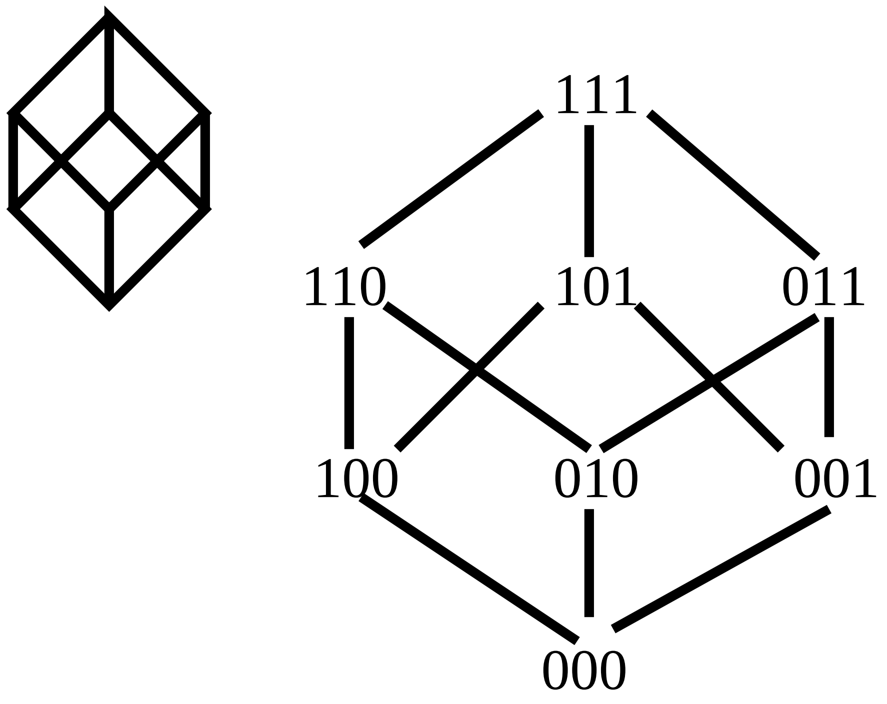


Figure 15.6: A 3-cube formed by joining the words of length 3 over the binary alphabet with edges when at a Hamming distance of 1

If we take the rate at which we can send bits on the channel times $\alpha$, we get the *fundamental rate* of the channel. Claude Shannon proved that you can use a channel at any rate below its fundamental rate with any positive probability of error - i.e., you can get the error probability down to any level you like above zero. Shannon's theorem does not tell you how to construct the code – it only proves the code exists. Most of the current research on error correcting codes amounts to finding constructions for codes that Shannon proved must exist decades ago.

At this point we change viewpoint a little to get a geometric understanding of error correcting codes. The code words in the yelling-over-the-flood example, 000 and 111, are at opposite corners of the 3-hypercube shown in Figure 15.6. If we take the binary words of length $n$ and join those which have Hamming distance 1, then we get an $n$-hypercube. This is the underlying space for standard error correcting codes. Code words are, geometrically, vertices in the hypercube.

A *ball* is a collection of vertices at distance $r$ or less from a distinguished vertex called the center. The number $r$ is the radius of the sphere. *Hamming balls* are sets of vertices of a hypercube at Hamming distance $r$ or less from a distinguished vertex called the center. If each word of a code is in a Hamming ball of radius $r$ that is disjoint from the ball of radius $r$ around any other code word, then any set of $r$ errors during transmission leave the received word closer to the transmitted word than to any other code word. This means a code that is a set of centers of disjoint Hamming balls of radius $r$ can decode up to $r$ errors.

We call a Hamming ball of radius $r$ an $r$-ball. A collection of centers of disjoint $r$-balls is called a *sphere packing* of radius $r$. The problem of finding good error correcting codes is identical to that of packing spheres into a hypercube. A good introduction to error correcting codes is [29]. A book that puts codes into an interesting context and continues on into interesting fundamental mathematics is [36].

This view of codes words as sphere centers will be fundamental to understanding the algorithm that produces DNA bar codes. Another useful fact is left for you to prove in the Problems. We call the smallest distance between any two code words the *minimum distance* of the code. If the minimum distance between any two words in a code is $2r + 1$, then the code is a packing of radius $r$ spheres. We now know enough coding theory to continue on to the molecular biology portion of this section.

## Edit Distance

DNA sequencers make errors. If those errors were always substitutions of one DNA base for another, we could correct them with a version of the binary error correcting codes, upgraded to use the 4-letter DNA alphabet. Unfortunately, sequencing errors include finding bases that are not there (insertions) and losing bases that are there (deletions). These errors are called, collectively, *indels*. Our first task is to find a distance measure that can be used to count errors in the same way that the Hamming distance was used to count bit flips.

**Definition 15.13** *The* **edit distance** *between two strings is the minimum number of single character insertions, deletions, and substitutions needed to transform one string into the other.*

From this point on we will denote the Hamming distance between two strings $x$ and $y$, $d_H(x, y)$, and the edit distance, $d_E(x, y)$. It is easy to compute Hamming distance, both algorithmically and by eyeball. In order to compute the edit distance, a more complex algorithm is required.

**Algorithm 15.2 Edit Distance**

**Input:** *Two L-character strings a,b*
**Output:** *The edit distance $d_E(a, b)$*
**Details:**

```
int dEdit(char a[L],char b[L]){//edit distance

int i,j,q,r,s,M[L+1][L+1];

  for(i=0;i<=L;i++){//initialize matrix
    M[i][0]=-i;
    M[0][i]=-i;
  }
  for(i=1;i<=L;i++)for(j=1;j<=L;j++){//fill in the dynamic programming matrix
    q=M[i-1][j-1];
    if(a[i-1]!=b[j-1])q--;
    r=M[i-1][j]-1;
    s=M[i][j-1]-1;
    if(s>q)q=s;
    if(r>q)q=r;
    M[i][j]=q;
  }

  return(-M[L][L]);  //the lower right corner is -(edit distance)

}
```

The edit distance algorithm is a modification of a dynamic programming algorithm used to perform sequence alignment. If you are interested in the connections between sequence alignment and the computation of edit distance, read [18]. The edit and Hamming distances have a one-sided relationship. In the Problems, you will prove that Hamming distance is an upper bound on edit distance. We now do an example to show that the separation between Hamming and edit distance can be almost the length of the strings.

**Example 15.4** *Notice:*

$$d_H(CACACACACA, ACACACACAC) = 10$$

*while*

$$d_E(CACACACACA, ACACACACAC) = 2.$$

*To see that the edit distance is two, delete the last character and insert it as the first.*

## Conway's Lexicode Algorithm

We will use *Conway's lexicode algorithm* as the greedy algorithm in our greedy closure evolutionary algorithm. It is a greedy algorithm that permits us to build error correcting codes. A good discussion of its use for standard (binary, Hamming) codes appears in [9].

**Algorithm 15.3** Conway's Lexicode Algorithm

**Input:** *A minimum distance $d$, and alphabet $A$, and a word length $n$*
**Output:** *A code $C$ with minimum distance $d$ over $A^n$*
**Details:**

*Place the list of all words of length $n$ over $A$ in lexicographical (alphabetical) 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 distance $d$ or more from each word placed in $C$ so far.*

Conway's lexicode algorithm is a greedy algorithm that creates a code that is constructively of minimum distance $d$. As long as the space of words can be alphabetized, the algorithm produces a code, no matter what notion of distance is used. This turns out to be critical for finding error correcting codes for the edit metric. The standard constructions for error correcting codes relative to the Hamming metric don't seem to have versions over the edit metric. Briefly, the edit metric is far messier than the Hamming metric. Let's do an example.

**Example 15.5** *Suppose we run Conway's algorithm on the edit metric space for 5-letter DNA words. Then the resulting set of words at pairwise edit distance at least 3 is:*

| | | |
|---|---|---|
| *AAAAA* | *AACCC* | *AAGGG* |
| *AATTT* | *ACACG* | *ACCAT* |
| *ACGTA* | *ACTGC* | *AGAGT* |
| *AGGAC* | *ATATC* | *ATTAG* |
| *CAACT* | *CAGTC* | *CATGA* |
| *CCCCA* | *CCGAG* | *CGCGC* |
| *CGTTG* | *CTAGG* | *CTCTT* |
| *CTTCC* | *GAAGC* | *GATCG* |
| *GCATT* | *GCTAA* | *GGCAG* |
| *GGGCT* | *GTGGA* | *TAATG* |
| *TAGCA* | *TCCTC* | *TCGGT* |
| *TGACC* | *TGTAT* | *TTCAA* |

## DNA Bar codes, at last

We now have all the parts needed to create a greedy closure evolutionary algorithm to locate error correcting codes for the edit metric over the DNA alphabet. We still lack, however, the motive for doing so. As we noted in Section 15.1, some organisms have a great deal of repeated sequences. The human genome project fragmented human DNA in several different ways, sequenced the fragments, and then fitted overlapping fragments together like a puzzle. In an organism like corn, with far more repeated sequences than humans, the step of fitting the puzzle together isn't possible. The repetitive nature of the sequences makes too many of the puzzle pieces look the same.

A related problem is that of locating the genes in an organism. A gene is a stretch of DNA that makes a protein. Most DNA is not part of a gene, rather it is "junk" DNA. Junk DNA may in fact be junk, or it may be a transposon sequence, or it may play a regulatory role. In any case, most applications of genomics need to know where the genes are. Genes can be located by sequencing their mRNA transcripts. While genes may be hard for humans to spot, an organism "knows" where its genes are; it can transcribe them. An *expressed sequence tag* (EST) is exactly an mRNA transcript. A complex biochemical process can be used to intercept transcribed genes, transform the mRNA into complementary DNA (cDNA). This cDNA is then placed in constructs in e-coli (a kind of bacteria). A collection of e-coli carrying cDNA is called a *genetic library.* Which ESTs are present in a given bacteria is random, and, so, an EST sequencing project is a random sampling of the transcribed genes. The bacteria can be grown, increasing the amount of the cDNA. Primer annealing sites in the constructs placed in the e-coli permit selective amplification of the cDNA, providing enough DNA for sequencing. So what is the problem?

Most genes are not transcribed all the time. Heat shock genes in plants require the plants to be subjected to heat stress before they are transcribed. Genes that confer resistance to a parasite are typically transcribed only when the parasite is present. Genes used in development of a young organism cease being transcribed in the adult. There are thousands of genes in a given organism that are only transcribed in some weird circumstance.

When preparing a genetic library, samples are taken from as many organismal states as possible. An EST sequencing project in corn, for example, will use libraries prepared from different tissues, developmental stages, and different stress states (such as drought or disease). For economic reasons, these libraries are pooled before sequencing. A *DNA bar code* is a short sequence of DNA incorporated into the genetic construct placed in the e-coli. This bar code is used much the way bar codes are used in grocery stores: to identify the product. Each tissue, developmental stage, and stress type is assigned its own bar code. When a pooled library is sequenced, the bar codes allow the researchers to figure out which states stimulate which genes. If the bar codes happen to be drawn from an edit metric error correcting code, then sequencing errors that hit the bar code may not prevent identification of the bar code. With this motivation, let's move on to the algorithm for finding sets of bar codes.

| Code Sizes | Minimum Distance | | | | | | |
|---|---|---|---|---|---|---|---|
| Length | 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 |

* denotes big.
- denotes empty.

Table 15.1: Size of DNA edit-metric lexicodes found with the unmodified lexicode algorithm

The primary attribute of a code, after its length and minimum distance, is its size. A large code is one that packs more spheres into the same string space. All codes found by the lexicode algorithm (Algorithm 15.3) have the property that they cannot accept any more words. However, they need not be as large as possible. Our evolutionary algorithm searches for larger codes within a fixed word length and minimum distance.

**Experiment 15.13** *Implement Conway's lexicode algorithm for the edit metric over the DNA alphabet. Run the algorithm for the following parameter sets: length 5, distance 3;*

*length 6, distance 3; length 8 distance 5. Verify both the sizes (from Table 15.1) and the membership in the (5,3) case (from Example 15.5). Record the running time of the algorithm in all three cases. Now, modify the algorithm to first check the Hamming distance. Since Hamming distance exceeds edit distance, if a word is too close to a word already in the code in the Hamming sense, then it is too close in the edit sense. This can be done in two ways: (i) either scan for Hamming rejection against all words in the code first, then scan for edit rejection, or (ii) check Hamming and then edit rejection of a potential new word against each word in the code. Try both possible modifications and report the impact on runtime.*

Our evolutionary algorithm will search for a length $n$ minimum distance $d$ code. The structure we will evolve (our seeds) is a set of 3 words at mutual distance $d$. Instead of starting Conway's algorithm with an empty code $C$, we will use a seed as the starting point. Let us now define our fitness function.

**Definition 15.14** *The* **greedy closure fitness with Conway's algorithm** *or* **greedy fitness***, for short, is computed as follows. Initialize the code in Conway's algorithm with a set $S$ of words already at mutual distance $d$. Run the algorithm. The fitness of $S$ is the size of the resulting code.*

A fact we have not yet established is that the size of codes produced by Conway's algorithm can vary when different seeds are used. A simple sampling experiment can settle this question.

**Experiment 15.14** *Using the fastest version of the lexicode algorithm found in Experiment 15.13, implement the greedy fitness function. Evaluate this function on 20,000 sets of three words of length 6 with a minimum distance of 3 generated at random over the DNA alphabet. To get such sets of words: generate a first word; generate a second repeatedly until it is edit distance 3 or more from the first word; generate a third word repeatedly until its edit distance from the first and second word is at least 3. Plot a histogram showing the number of codes of each size found. Compare your results with Figure 15.7.*

In the sampling experiment, we saw that the result of the lexicode algorithm without a seed, 96, is slightly better than the mode code size of 95 for length 6 distance 3 codes. The best, 103, contained just over 7% more words. Since longer bar codes are expensive in terms of biochemical success in creating libraries, squeezing in a few more bar codes at a given length is worth the trouble. From a mathematical perspective, getting some idea as to how large the codes can be is itself interesting. In any case, we see that seeds do change the behavior of Conway's algorithm, and, so, seeds can "control" the algorithm. But how?

A code with minimum distance $d$ is made of words that are at least distance $d$ apart in the string space the code is drawn from. When we select a word to be in the code, we exclude all words within distance less than $d$ of the selected word. This means that a seed,

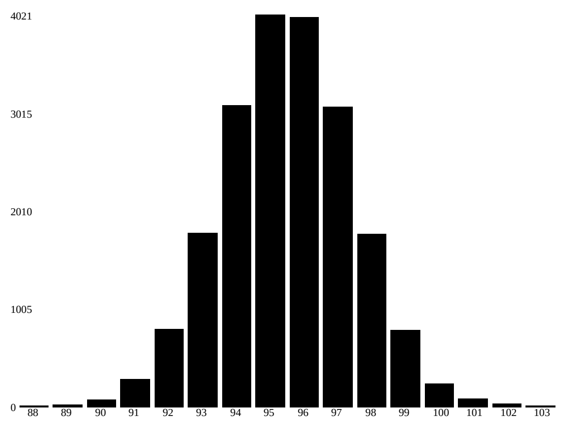


Figure 15.7: A histogram showing the distribution of sizes of length 6, distance 3 edit-metric codes on the DNA alphabet located in 20,000 random samples of 3-word seeds (The largest code located by sampling has 103 code words.)

which selects a few words at the beginning, excludes a large number of other words from consideration. This means that the control that seeds have over the behavior of the lexicode algorithm is pretty substantial, but also quite unpredictable. A word chosen to be in the seed excludes a word the algorithm would normally have chosen. This in turn causes other words to be chosen, and a domino effect cascades through the code. Not only does the seed change the size of the code, but it also changes the membership of the code far more than one might suppose given the size of the seed.

Since understanding the impact of seed choice on code size is difficult, choosing seeds is a sensible task for an evolutionary algorithm. An evolutionary algorithm does not require understanding to function. We have a representation, the 3-word seed, and we have a fitness function, the greedy fitness of seeds. We still need variation operators.

**Definition 15.15** *For two seeds,* **uniform exclusive crossover** *is performed as follows. If two seeds have words in common, then we leave one copy in each seed. The words not in common are pooled and then randomly assigned during crossover. Uniform exclusive crossover is similar to uniform crossover for string genes but (i) it does not have positions the way a string does and (ii) is does not permit duplication of words by crossover.*

**Definition 15.16** *We define* **seed point mutation** *to consist of changing one character in one uniformly selected word within a seed to a new character selected uniformly at random.*

**Definition 15.17** *We define* **seed word mutation** *to consist of changing one word in a seed to a new word selected uniformly at random.*

A seed is a collection of words, so far three words, that obey the minimum distance rule for the code the size of which we are trying to maximize. All three of the variation operators defined above have the potential to create seeds that violate this minimum distance rule. We extend the fitness function by awarding a fitness of zero to any seed that violates the minimum distance criterion. We are ready to construct the first evolutionary algorithm.

**Experiment 15.15** *Write or obtain code for the following steady state evolutionary algorithm. Use size 7 tournament selection. Operate on a population of 200 seeds containing three words each. The algorithm should use the greedy fitness function to evolve codes of length $n = 6$ and minimum distance $d = 3$. Use uniform exclusive crossover 50% of the time and no crossover in the remainder of the mating events. Optionally, use seed point mutation or seed word mutation. Perform 100 runs using both mutation operators on each new seed and also 100 runs using one or the other mutation operator with equal probability.*

*Save the maximum and population average fitness of those population members that do not have fitness zero. Also, save the number of zero fitness seeds. Give histograms of the best final fitness for each of the 3 sets of runs using different mixes of mutation operators. Based on fitness information, does the appearance of a new best fitness have a subsequent impact on average fitness or the number of zero fitness individuals? Which type of mutation turned in the best performance?*

The above is our first implementation of a greedy closure evolutionary algorithm. In the Problems we explore other possible targets for this sort of algorithm. As a tool for locating bar codes, it avoids the problem of finding an encoding that stores an entire code. Selecting roughly 100 code words from $4^6$ length 6 DNA words is a daunting problem. Especially since the minimum distance constraint creates a vast degree of interdependence among the words. The greedy closure algorithm we used fails badly to make a global search of the space of codes; instead, it searches some subset of those codes with great efficiency. It is also a completely new type of evolutionary algorithm, and, so, the "knobs" or operational parameters will need to be explored.

**Experiment 15.16** *Repeat Experiment 15.15, using the mutation operator(s) that turned in the best performance, but modify the crossover probability and perform runs with 0%, 25%, 75%, and 100% chances of doing crossover. What is the impact?*

Another critical parameter is seed size.

**Experiment 15.17** *Repeat Experiment 15.16, using the crossover rate that turned in the best performance. Change the algorithm so that it uses seeds of size 1, 2, and 4. What is the impact of varying seed size?*

Let us also check the impact of population size and sharpness of selection.

**Experiment 15.18** *Repeat Experiment 15.17, using the seed size that turned in the best performance. Survey all possible combinations of population sizes, 100, 200, 400, and tournament sizes, 4, 7, and 15. What is the impact?*

The structure of these experiments is not a sound one. Experiments 15.15-15.18 assume that once we have found an optimum for one parameter relative to the algorithms current settings, it remains optimal. If we knew there was no interaction between, say, the mutation operator(s) and the tournament size, then we would not have a problem. A complete factorial study, however, would take an inordinate amount of time. You may want to do a final project that is either a sparse factorial study or fills in parts of an ongoing one.

This section barely scratches the surface of both edit metric error correcting codes (note that decoding is left as an exercise) and of the application of greedy closure evolutionary algorithms. Other applications are suggested in the problems. A natural thought is to attempt to apply the setup in this chapter to standard (binary, Hamming) error correcting codes. The author has done so and failed to improve on the known best codes for a given length and minimum distance. Given that the mathematical theory is far more beautiful for standard codes, it is not surprising that a messy technique like evolutionary algorithms cannot outperform it. Nevertheless, please contact the author if you manage a breakthrough.

## Problems

**Problem 15.24** *Reread Example 15.3. Compute a general formula for the fold-change in the probability of misunderstanding a single bit message when the probability of misunderstanding each individual bit is $\alpha$ and a length $2n+1$ repetition code is used. The improvement is for the code over the single bit as in the example. Plot the function for $2n+1=5$ in a manner like that in the example for $2n+1=3$.*

**Problem 15.25** *Prove that if a collection $C$ of code words has the property that, for any $u, v \in C$, the Hamming distance from $u$ to $v$ is at least $2r+1$, then the Hamming ball of radius $r$ around any code word in $C$ contains no other code word in $C$.*

$$M_3 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

**Problem 15.26** *Suppose that we have a matrix $M_k$ whose columns are every binary word of length $k$,* except *the all-zero word, in counting order. The matrix $M_3$ is shown above. Let $HC_k$ be the set of words that are the null space of the matrix, i.e., binary vectors $\vec{x}$ of length $2^k-1$ such that $M_k * \vec{x} = \vec{0}$. For example, since*

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

*we see that $\vec{x} = (1,0,0,1,1,0,0)$ is in $HC_3$. Prove that $HC_k$ is a code with minimum Hamming distance 3 between any two words.*

**Problem 15.27** *Enumerate (list the members of) $HC_3$, defined in Problem 15.26.*

**Problem 15.28** *Let $d_H(x,y)$ be the Hamming distance between two strings $x$ and $y$, and $d_e(x,y)$ be the edit distance. Prove that*

$$d_E(x,y) < d_H(x,y).$$

**Problem 15.29** *Compute the edit distance and show a minimal sequence of edits for all pairs of the following words:* {**ACGTA**, **GCTAA**, **AAGGG**}.

**Problem 15.30** *Review Section 7.4. Outline a greedy closure algorithm for finding Costas arrays.*

**Problem 15.31** *Outline a greedy closure algorithm for the Traveling Salesman problem. Is the Traveling Salesman problem a natural target or a poor one?*

**Problem 15.32** *Prove that a code found with Conway's algorithm, using a seed or not, is maximal in the sense that no larger code with the same length and minimum distance contains it.*

**Problem 15.33** *Using the edit distance lexicode algorithm, give a decoding algorithm for edit metric lexicodes. Assume you are using DNA bar codes of length $n$ and minimum distance $d = 2r+1$. Given a received (sequenced) word, you should return either a member of the code $C$ or an error message (if the received word is not closer to one code word than another).*

**Problem 15.34 Essay.** *A direct encoding of an error correcting code would require a gene that picks out the members of the code from the space of words. Is such a direct encoding practical for the type of code located in Experiment 15.15?*

**Problem 15.35 Essay.** *Is Conway's algorithm specific to the Hamming or edit metric or can it be used with any notion of distance? With what kinds of notions of distance can it be used?*

## 15.4 Visualizing DNA

In this section, we will make a substantial departure from applied bioinformatics and enter the realm of speculative bioinformatics. In the course of this, we will create a data driven, evolvable fractal visualization tool. The starting point is a fairly well known type of fractal algorithm called a chaos game.

### Chaos game fractals

A *chaos game* is characterized as the process of generating a fractal by accumulating the positions of a moving point. This moving point is repeatedly displaced toward one of a fixed set of points, e.g., the vertices of an equilateral triangle. Figure 15.8 shows the Sierpinski triangle. It is generated by a chaos game in which a moving point is displaced, in each iteration, halfway from its present position toward a randomly selected vertex of a triangle. Figure 15.8 is plotted over 100,000 iterations of this process.

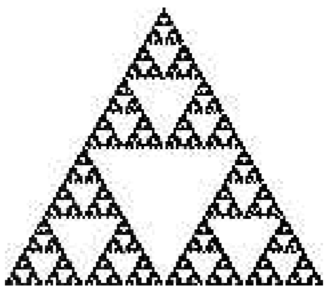


Figure 15.8: The Sierpinski Triangle generated by a 3-cornered chaos game

**Algorithm 15.4 Simple chaos game**

**Input:** *A collection of fixed points in the real plane*
**Output:** *A set of points in the real plane*
**Details:**

*A point, called the* moving point*, is initialized to the position of one of the fixed points. An* updating *of the moving point's position is performed by choosing one of the fixed points uniformly at random and then averaging the current position of the moving point with that of the fixed point. The moving point is moved halfway to the chosen fixed point.*

*A series of updatings are made to* burn in *the moving point. This process permits the moving point to enter, up to the resolution of plotting, the fractal attractor that is characteristic of the chaos game. Typically, a few hundred updatings are more than enough to burn in*

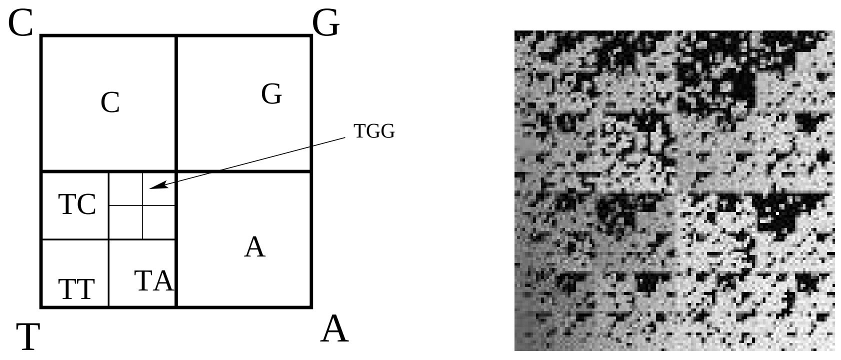


Figure 15.9: The diagram in the left half of this figure shows how sequence data subdivides the square in a 4-cornered chaos game. Such a chaos game, driven by HIV sequence data, is displayed in the right half of the figure.

*the moving point. Subsequent to the burn-in, the chaos game continues to update the moving point's position. In the post burn-in period, the moving point is plotted.*

The character of the fractal resulting from a chaos game is controlled by the number of fixed points being used and the order in which those points are chosen to specify the direction of motion for the moving point. This latter point is key. The Sierpinski triangle is generated by using the vertices of a regular triangle, with the next point chosen uniformly at random. If, instead, we have data with some degree of non-uniformity, then the points in the resulting fractal are a subset of the fractal obtained by driving with uniform random data.

If the 4 points at the vertices of a square are used as the fixed points, the chaos game produces a dense subset of the square. The averaging toward the corners of the square produces the diadic rationals in each coordinate. If the fractal is visualized, the square simply fills in. This represents an opportunity for the visualization of DNA or RNA data in a manner discussed in [20] and [12]. If we assign each corner of the square to one of the 4 DNA bases, then deviations from uniformity of the nucleic acid sequence will appear as gaps in the square filled in by the fractal process.

Figure 15.9, for example, demonstrates the results if we drive a chaos game on the square with sequence data from an HIV virus. As each base, **C**, **G**, **A**, **T**, is handed to the fractal process, the moving point is moved halfway from its current position to the corner of the square associated with the base. The averaging or halfway moves subdivide the square by sequence data as shown in the left part of Figure 15.9. The resulting gaps indicate subsequences the do *not* appear in the HIV genome. In this case, many of the gaps can be attributed to the HIV virus's lack of methylization sites.

Interpretation of chaos game fractals such as those shown in Figure 15.9 requires a good deal of biological knowledge. The lack of methylization sites is only obvious in Figure 15.9 if you know the sequences for methylization sites and can picture where they are on the chaos game's square. This problem becomes more acute when an attempt is made to use these techniques to derive visual representations of protein sequences. Proteins are built out of 20 building blocks, called amino acids, rather than the 4 bases of DNA or RNA. In [33], both placing the 20 amino acids in a circle and extending the fractal into a third dimension are attempted. As one would expect, the interpretation difficulties grow.

A number of biological issues can be used to inform the choices made when designing a biological representation for a fractal. The map from nucleic acid to protein reads DNA in triples, producing 64 *codons*. These codons are in turn taken by a many-one map (the *genetic code*) onto the 20 amino acids as well as a *stop codon*. This stop codon indicates the end of transcription of a given sequence of DNA. The many-one map that forms the genetic code is the same in almost all organisms. The choice of which of several possible codons to use to specify a given amino acid, however, has a substantially organism-specific character. These biological considerations will factor into the design of evolvable fractals. Our next step is to generalize the chaos game.

## Iterated function systems

Chaos games are a particular type of iterated function system [7]. In an *iterated function system* (IFS), a number of maps from the Cartesian plane to itself are chosen. These maps are then called in a random order, according to some distribution, to move a point in a manner similar to the chaos game. The orbit of this point in the metric space is called the *attractor* of the iterated function system. In [7], a number of theorems about iterated function systems are established. In order to get a well behaved fractal, the maps in the iterated function system must have the following property.

**Definition 15.18** *Let $d(p, q)$ be the distance in the Cartesian plane between points $p$ and $q$. A function, $f : \mathbb{R}^2 \to \mathbb{R}^2$, from the plane to itself is called a* **contraction map***, if, for any pair of points $p$, $q$,*

$$d(p, q) > d(f(p), f(q)).$$

An iterated function system made entirely of contraction maps has a bounded fractal attractor. A rich class of maps that are guaranteed to be contraction maps are similitudes.

**Definition 15.19** *A* **similitude** *is a map that performs a rigid rotation of the plane, displaces the plane by a fixed amount, and then contracts the plane toward the origin by a fixed scaling factor. The derivation of a new point $(x_{new}, y_{new})$ from old point $(x, y)$ with a similitude that uses rotation $t$, displacement $(\Delta x, \Delta y)$, and scaling factor $0 < s < 1$ is given by:*

$$x_{new} = s \cdot (x \cdot Cos(t) - y \cdot Sin(t) + \Delta x) \tag{15.1}$$

$$y_{new} = s \cdot (x \cdot Sin(t) + y \cdot Cos(t) + \Delta y) \tag{15.2}$$

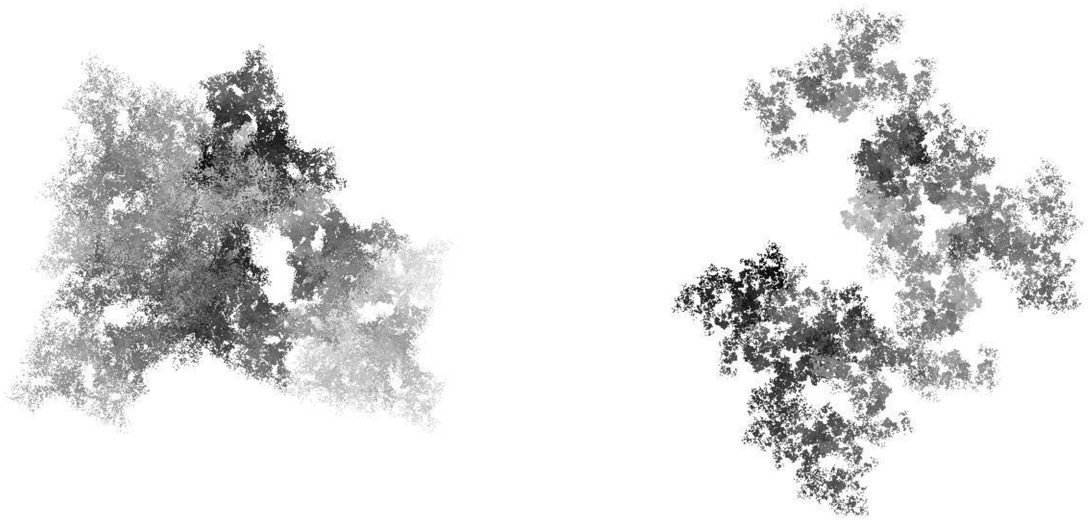


Figure 15.10: The fractal attractors for the iterated function systems given in Example 15.6

To see that a similitude must always reduce the distance between two points, note that rotation and displacement are isometries (they do not change distances between points). This means any change is due to the scaling factor which necessarily causes a reduction in the distance between pairs of points. Let's look at a couple of iterated function system fractals.

**Example 15.6** *An iterated function system is a collection of contraction maps together with a distribution with which those maps will be applied to the moving point. In Figure 15.10 are a pair of fractal attractors for iterated function systems built with 8 similitudes. These similitudes are called uniformly at random.*

| *First IFS* | | | | *Second IFS* | | | |
|---|---|---|---|---|---|---|---|
| *Map* | *Rotation* | *Displacement* | *Scaling* | *Map* | *Rotation* | *Displacement* | *Scaling* |
| *M1* | *4.747* | *( 0.430, 0.814)* | *0.454* | *M1* | *2.898* | *(-0.960, 0.253)* | *0.135* |
| *M2* | *1.755* | *(-0.828, 0.134)* | *0.526* | *M2* | *3.621* | *( 0.155, 0.425)* | *0.532* |
| *M3* | *3.623* | *( 0.156, 0.656)* | *0.313* | *M3* | *5.072* | *( 0.348,-0.129)* | *0.288* |
| *M4* | *0.207* | *(-0.362, 0.716)* | *0.428* | *M4* | *3.428* | *(-0.411,-0.613)* | *0.181* |
| *M5* | *2.417* | *(-0.783, 0.132)* | *0.263* | *M5* | *4.962* | *(-0.569, 0.203)* | *0.126* |
| *M6* | *1.742* | *(-0.620, 0.710)* | *0.668* | *M6* | *4.858* | *(-0.388,-0.651)* | *0.489* |
| *M7* | *0.757* | *( 0.444, 0.984)* | *0.023* | *M7* | *5.953* | *(-0.362, 0.758)* | *0.517* |
| *M8* | *4.110* | *(-0.633,-0.484)* | *0.394* | *M8* | *1.700* | *(-0.696, 0.876)* | *0.429* |

*The similitudes in this example were generated at random. The rotation factors are in* *the range $0 \leq \theta \leq 2\pi$ radians. The displacements are selected uniformly at random to move the origin to a point with $-1 < x, y < 1$. The scaling factor is chosen uniformly at random in the range $0 < s < 1$.*

## 15.5 Evolvable Fractals

Our goal is to use a data driven fractal, generalizing the 4-cornered chaos game, to provide a visual representation of sequence data. It would be nice if this fractal representation could work smoothly with DNA, protein, and codon data. These sequences, while derived from one another, have varying amounts of information and are important in different parts of cells operation. The raw DNA data contains the most information and the least interpretation. The segregation of the DNA data into codon triples has more interpretation (and requires us to work on DNA that is transcribed as opposed to other DNA). The choice of DNA triplet used to code for a given amino acid can be exploited, for example, to vary the thermal stability of the DNA (more **G** and **C** bases yield a higher melting temperature), and, so, the codon data contains information that disappears when the codons are translated into amino acids. The amino acid sequence contains information focused on the enzymatic mission of the protein. This sequence specifies the protein's fold and function without the codon usage information muddying the waters.

Given all this, we design an iterated function system fractal which evolves the contraction maps used in the system as well as the choice of which contraction map is triggered by what biological feature. For our first series of experiments, we will operate on DNA codon data, rich in information but with some interpretation. Our test problem is reading frame detection, a standard and much studied property of DNA. Reading frame refers to the three possible choices of groupings of a sequence of DNA into triplets for translation into amino acids. Figure 15.11 shows the translation into the three possible reading frames of a snippet of DNA. Only the first reading frame contains the **ATG** codon for the amino acid, Methionine (which also serves as the "start" codon for translation), and the amino acid ,**TAG** (one of the three possible "stop" codons).

The correct reading frame for a piece of DNA, if it codes for a protein, is typically the frame that is free of stop codons. Empirical verification shows that frame-shifted transcribed DNA is quite likely to contain stop codons, which is also likely on probabilistic grounds for random models of DNA. We remind you that random models of DNA must be used with caution; biological DNA is produced by a process containing a selection filter, and, therefore, contains substantial non-random structure. Figure 15.9 serves as an example of such non-random structure.

```
ATG GGC GGT GAC AAC TAG
Met Gly Gly Asp Asn Stp

A TGG GCG GTG ACA ACT AG
. Trp Ala Val Thr Ala ..

AU GGG CGG TGA CAA CTA G
.. Gly Arg Gly Gln Val .
```

Figure 15.11: A piece of DNA translated in all 3 possible reading frames (Amino acids are given by their 3-letter codes which may be found in [31].)

## A fractal representation

The data structure we use to hold the evolvable fractal has two parts: a list of similitudes and an index of DNA triples into that list of similitudes. This permits smooth use of the fractal on DNA, DNA triplets, or amino acids by simply modifying the way the DNA or amino acids are interpreted by the indexing function. A diagram of the data structure is given in Figure 15.12. Each similitude is defined by 4 real parameters in the manner described in Equation 15.1. The index list is simply a sequence of 64 integers that specify, for each of the 64 possible DNA codon triplets, which similitude to apply when that triplet is encountered.

| Interpretation | Contains |
|---|---|
| First similitude | $t_1 \quad (\Delta x_1, \Delta y_1) \quad s_1$ |
| Second similitude | $t_2 \quad (\Delta x_2, \Delta y_2) \quad s_2$ |
| | $\cdots$ |
| Last similitude | $t_n \quad (\Delta x_n, \Delta y_n) \quad s_n$ |
| Index | $i_1, i_2, \ldots, i_{64}$ |

Figure 15.12: The data structure that serves as the gene for an evolvable DNA driven fractal (In this work, we use $n = 8$ similitudes, and so $0 \le i_j \le 7$.)

In order to derive a fractal from DNA, the DNA is segregated into triplets with a specific reading frame. These triplets are then used, via the index portion of the gene, to choose a similitude to apply to the moving point. The IFS is driven by incoming DNA triplets.

This representation permits evolution to both choose the shape of the maximal fractal (the one we would see if we drove the process with data chosen uniformly at random) and which DNA codon triplets are associated with the use of each similitude. Any contraction map has a unique fixed point. The fixed points of the 8 similitudes we use play the same role that the 4 corners of the square did in the chaos game shown in Figure 15.9.

We need variation operators. The crossover operator performs a one point crossover on the list of 8 similitudes, treating the similitudes as indivisible objects, and also performs two point crossover on the list of indices. We will used two mutation operators. The first, termed a *similitude mutation*, modifies a similitude selected uniformly at random. It picks one of the 4 parameters that define the similitude, uniformly at random, and adds a number selected uniformly in the interval [-0.1,0.1] to that parameter. The scaling parameter is kept in the range [0,1] by reflecting the value at the boundaries so that numbers $s > 1$ are replaced by $2 - s$ and values $s < 0$ are replaced by $-s$. The other parameters are permitted to move outside of their initial range. The second mutation operator, called an *index mutation*, acts on the index list by picking the index of a uniformly chosen DNA triple and replacing it with a new index selected uniformly at random.

Aside from a fitness function, we now have all the machinery required to evolve fractals. For our first experiment, we will attempt to tell if DNA is in the correct reading frame or not. The website associated with this text has a file of in-frame and out-of-frame DNA available. We will drive the IFS alternately with these two sorts of data and attempt to get the IFS to plot points in different parts of the plane when the IFS is being driven by distinct types of data.

**Definition 15.20** *The* **separation fitness** *of a moving point process $P$, e.g., an IFS, being driven by two or more types of data is defined as follows. Compute the mean position $(x_i, y_i)$ when the IFS is being driven by data type $i$. The fitness is*

$$SF(P) = \sum_{i \neq j} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

*where $x_i \in \{0, 1\}$.*

**Experiment 15.19** *Write or obtain code for evolving iterated function systems with the representation given in Figure 15.12. Use the crossover operator. The evolutionary algorithm should be generational, operating on a population of 200 IFS structures with size 8 single tournament selection. In each tournament, perform a similitude mutation on one of the new structures and an index mutation on the other.*

*To perform fitness evaluation, initialize the moving point to (0,0) and then drive the IFS with 500 triplets of in-frame data and 500 bases of out-of-frame data, before collecting any fitness information; this is a burn-in as was used in the chaos game. After burn-in, compute the mean position of the moving point for each type of data while alternating between the two types of data using 100-400 triples of each data type. Select the length, 100-400, uniformly at random. The mean position data for each of the two data types may be used to compute the separation fitness.*

*Perform 30 runs of length 500 generations. Report the fitness tracks and estimate the average number of generations needed to reach the approximate final fitness. If you have skill with graphics, also plot the fractals for the most fit IFSs using different colors for points plotted while the IFS is being driven by different data types. Report the most fit IFS genes.*

Experiment 15.19 should contain some examples that show there is a very cheap way for the system to generate additional fitness. If we were to take an IFS of the type used in Experiment 15.19 and simply enlarge the whole thing, the separation fitness would scale with the picture. This suggests that we may well want to compensate for scaling.

**Definition 15.21** *The* **diameter** *of a moving point process is the maximum distance between any two plotted points generated by the moving point process. For an IFS, the diameter should only be computed after the IFS has been burned in.*

**Definition 15.22** *The* **normalized separation fitness** *of a moving point process $P$, e.g., an IFS, being driven by two or more types of data is the separation fitness divided by the diameter of the moving point process.*

**Experiment 15.20** *Repeat Experiment 15.19 using the normalized separation fitness instead of the separation fitness. Also, reduce the number of generations to 120% of the average solution time you estimated in Experiment 15.19. Comment on the qualitative differences of the resulting fractals.*

There is a second potential problem with our current experimental setup. This problem is not a gratuitous source of fitness as was the scaling issue. This issue is an aesthetic one. A very small scaling factor moves the moving point quite rapidly. If our goal is to separate two sorts of data, then a good IFS would have well separated regions and would move points into those regions as fast as possible via the use of tiny scaling factors.

**Experiment 15.21** *Repeat Experiment 15.20, but modify both initialization and similitude mutation so that scaling factors are never smaller than $a$. Perform runs for $a = 0.5$ and $a = 0.8$. What impact does this modification have on the fitness tracks and on the pictures generated by the most fit IFS?*

## Chaos Automata

The IFS representation we've developed has a problem that it shares with the chaos game: it is forgetful. The influence of a given DNA base on the position of the moving point is decreased by each successive scaling factor. To address this problem we introduce a new representation called *chaos automata.* Chaos automata differ from standard iterated function systems in that they retain internal state information. This gives them the ability to visually associate events that are not nearby in the sequence data.

The internal memory also grants fractals generated with chaos automata a partial exemption from self-similarity in the fractals they specify. In the IFS fractals generated thus far, various parts of the fractal look like other parts. When driven by multiple types of input data, a chaos automaton can "remember" what type of data it is processing, and, so, plot distinct types of shapes for distinct data. Two more-or-less similar sequences separated by a unique marker could, for example, produce very different chaos-automata based fractals by having the finite state transitions recognize the marker and then use different contraction maps on the remaining data.

Comparison with the iteration function system fractals already presented motivates the need for this innovation in the representation of data driven fractals. The problem addressed by incorporating state information into our evolvable fractals is that data items are forgotten as their influence vanishes into the contractions of space associated with each contraction function. An example of a chaos automata, evolved to be driven with DNA data, is shown in Figure 15.13.

```
Starting State:6

  Transitions:                   Similitudes:
 If  C  G  A  T    Rotation    Diplacement Contraction
------------------------------------------------------
 0)  3  2  3  3  : R:0.678 D:( 1.318, 0.606) S:0.905
 1)  5  3  5  3  : R:1.999 D:( 0.972, 0.613) S:0.565
 2)  7  7  2  3  : R:0.521 D:( 1.164, 0.887) S:0.620
 3)  3  0  0  3  : R:5.996 D:( 0.869, 0.917) S:0.805
 4)  0  0  0  5  : R:1.233 D:( 0.780,-0.431) S:0.610
 5)  5  5  5  7  : R:1.007 D:(-0.213, 0.706) S:0.623
 6)  3  7  3  4  : R:3.509 D:( 0.787, 0.767) S:0.573
 7)  1  5  5  2  : R:0.317 D:( 0.591, 0.991) S:0.570
```

Figure 15.13: A chaos automaton evolved to visually separate two classes of DNA (The automaton starts in state 6 and makes state transitions depending on inputs from the alphabet {**C**, **G**, **A**, **T**}. As the automaton enters a given state, it applies the similitude defined by a rotation (R), displacement (D), and shrinkage (S).)

Chaos automata are modified finite state automata. Each state of the chaos automaton has an associated similitude, applied when the automaton enters that state. Memory is supplied by the finite state automaton and the similitudes serve as the contraction maps. A chaos automaton is an IFS with memory. Note we have made the, somewhat arbitrary, choice of associating our contraction maps with states rather than transitions. We thus are using

"Moore" chaos automata rather than "Mealy" chaos automata. Algorithm refuseCHAUT specifies how to use a chaos automaton as a moving point process.

**Algorithm 15.5 Using a chaos automaton**

**Input:** *A chaos automaton*
**Output:** *A sequence of points in the plane*
**Details:**

```
Set state to initial state.
Set moving point (x,y) to (0,0).
Repeat
    Apply the similitude on the current state to (x,y).
    Process point (x,y).
    Update the state according to input with the transition rule.
Until (out of input).
```

In order to use an evolutionary algorithm to evolve chaos automata, we need variation operators. We will reuse the previously defined similitude mutation. We will use a two point crossover operator. This crossover operator treats the vector of nodes as a string of indivisible objects. The integer that identifies the initial state is attached to the first state in the string of states and moves with it during crossover. There are three kinds of things that could be changed with a mutation operator. Primitive mutation operators are defined for each of these things and then used in turn to define a master mutation operator that calls the primitive mutations with a fixed probability schedule. The first primitive mutation acts on the initial state, picking a new initial state uniformly at random. The second primitive mutation acts on transitions to a next state. It selects one such transition uniformly at random and then selects a new next state uniformly at random. The third primitive mutation applies a similitude mutation to a similitude selected uniformly at random. The master mutation mutates the initial state 10% of the time, a transition 50% of the time, and mutates a similitude 40% of the time. For our first experiment, we will test our ability to evolve chaos automata to solve the reading frame problem.

**Experiment 15.22** *Modify the software from Experiment 15.21, including the lower bound on the scaling factor for similitudes, to use chaos automata. What impact did this have on fitness?*

Let's now test chaos automata on a new problem. In a biological gene, there are regions called *exons* that contain the triples that code for amino acids. There are also regions between the exons, called *introns* that are spliced out of the mRNA before it is translated into protein by ribosomes. We will use chaos automata to attempt to visually distinguish intron and exon data.

**Experiment 15.23** *Repeat Experiment 15.22 but replace the in-frame and out-of-frame DNA with intron and exon sequences downloaded from the website for this text. Report the fitness tracks. Do the chaos automata manage to separate the two classes of data visually? Report the diameter of the best fractal found in each run as well as the fitness data.*

When developing chaos automata, the author and his collaborators found that tinkering with the fitness function yielded a substantial benefit. We will now explore some new fitness functions. We begin by developing some terminology. To efficiently describe new fitness functions, we employ the following device: the moving point, used to generate fractals from chaos automata driven by data, is referred to as if its coordinates were a pair of random variables. Thus $(X, Y)$ is an ordered pair of random variables that gives the position of the moving point of the chaos game. When working to separate several types of data, $\{d_1, d_2, \ldots, d_n\}$, the points described by $(X, Y)$ are partitioned into $\{(X_{d_1}, Y_{d_1}), (X_{d_2}, Y_{d_2}), \ldots, (X_{d_n}, Y_{d_n})\}$, which are the positions of the moving points of a chaos automata driven by data of types $d_1, d_2, \ldots, d_n$, respectively. For any random variable $R$, we use $\mu(R)$ and $\sigma^2(R)$ for the sample mean and variance of $R$. Using this new notation, we can rebuild the separation fitness function of a moving point process $P$, with $d_1$ and $d_2$ being the in-frame and out-of-frame data.

$$SF(P) = \sqrt{(\mu(X_{d_1}) - \mu(X_{d_2}))^2 + (\mu(Y_{d_1}) - \mu(Y_{d_2}))^2} \tag{15.3}$$

The problem of having fractals made of sparse sets of points is only partially addressed by placing the lower bound on the scaling factor within the similitudes. Our next function will encourage dispersion of the points in the fractal while continuing to reward separation by multiplying the separation by the standard deviation of the position of the moving point.

**Definition 15.23** *The* **dispersed separation fitness** *for a moving point process $P$ is given by:*

$$F_3 = \sigma(X_{d_1})\sigma(Y_{d_1})\sigma(X_{d_2})\sigma(Y_{d_2})SP(P).$$

**Experiment 15.24** *Repeat Experiment 15.23 with dispersed separation fitness in place of separation fitness. In addition to the information recorded previously, track the diameter of the resulting fractals over the course of evolution. Compare this with the diameters recorded in Experiment 15.24. Also, check to see if the fractals visually separate the data.*

If your version of Experiment 15.24 worked the way ours did, then you got some huge fractals. The dispersed separation fitness function over-rewards dispersion. This too can be fixed.

**Definition 15.24** *The* **bounded dispersed separation fitness** *for a moving point process $P$ is given by:*

$$F_4 = Tan^{-1}(\sigma(X_{d_1})\sigma(Y_{d_1})\sigma(X_{d_2})\sigma(Y_{d_2}))SF(P).$$

**Experiment 15.25** *Repeat Experiment 15.24 using bounded dispersed separation fitness in place of dispersed separation fitness. Did the new fitness function help the dispersion problem? As before, report if the fractals visually separate the data.*

We have not made a study of the sensitivity of the evolution of chaos automata to variation of the algorithm parameters. This is not the result of laziness (though the length of this chapter might justify some laziness), but rather because of a lack of a standard. The meaning of the fitness values for chaos automata is quite unclear. While the fitness functions used here did manage to visually separate data during testing, higher fitness values did not (in our opinion) yield better pictures. The very fact that the metric of picture quality is "our opinion" demonstrates that we do not have a good objective fitness measure of the quality of visualizations of DNA. If you are interested in chaos automata, read [2] and [3]. You are invited to think up possible applications for chaos automata. Some are suggested in the Problems.

## Problems

**Problem 15.36** *The dyadic rationals are those of the form*

$$q = \sum_{i=-n}^{\infty} x_i 2^{-i}.$$

*Run a chaos game on the square with corners* $(0,0), (0,1), (1,1)$, *and* $(1,0)$. *Prove that the $x$ and $y$ coordinates of the moving point are always a diadic rational.*

**Problem 15.37** *Is the process, "move halfway from your current position to the point $(x, y)$," a similitude? Prove your answer by showing it is not, or by identifying the rotation, displacement, and contraction.*

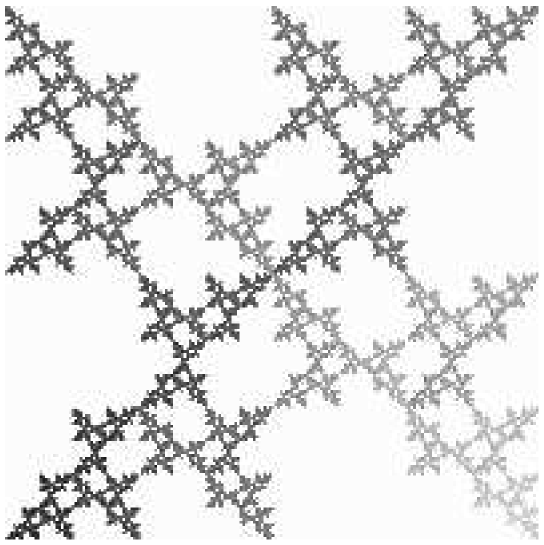
**Problem 15.38** *When the chaos game on a square is driven by uniform random data it fills in the square. Suppose that instead of moving halfway toward the corners of the square, we move 40% of the way. Will the square still fill in? If not, what does the resulting fractal look like?*

**Problem 15.39** *Consider the following modification of the chaos game on a square. Number the corners 0, 1, 2, 3 in the clockwise direction. Instead of letting the moving point average toward any corner picked uniformly at random, permit it only to move toward a point other than the next one* $(mod\ 4)$ *in the ordering. What does the resulting fractal look like?*

**Problem 15.40** *Prove that chaos games are iterated function systems.*

**Problem 15.41** *For the 8 similitudes associated with the first IFS in Example 15.6, compute the fixed point of each similitude to 4 significant figures. Plot these fixed points and compare with the corresponding fractal.*

**Problem 15.42** *For the 8 similitudes associated with the second IFS in Example 15.6, compute the fixed point of each similitude to 4 significant figures. Plot these fixed points and compare with the corresponding fractal.*



**Problem 15.43** *What variation of the chaos game on the square produced the above fractal?*

**Problem 15.44** *Prove that a contraction map has a unique fixed point.*

**Problem 15.45** *True or false? The composition of two contraction maps is a contraction map. Prove your answer.*

**Problem 15.46** *Suppose that the HIV-driven chaos game in Figure 15.9 is* $512 \times 512$ *pixels. How many DNA bases must pass though the IFS after a given base* $b$ *to completely erase the influence of* $b$ *on which pixel is plotted?*

**Problem 15.47** *When evolutionary algorithms are used for real function optimization the number of independent real variables is called the dimension of the problem. What is the dimension of the representation used in Experiment 15.19?*

**Problem 15.48** *When evolutionary algorithms are used for real function optimization the number of independent real variables is called the dimension of the problem. What is the dimension of the representation used in Experiment 15.22?*

**Problem 15.49** *What problems would be caused by computing the diameter of an IFS without burning it in first?*

**Problem 15.50** *Assume we are working with $k$ different types of data and have $k$ disjoint circles in the plane. Create a fitness function that rewards a moving point process for being inside circle $i$ when plotting data type $i$.*

**Problem 15.51** *Suppose that, instead of contracting toward the origin by a scaling factor $s$ in a similitude, we had distinct scaling factors $s_x$ and $s_y$ which were applied to the $x$ and $y$ coordinates of a point. Would the resulting modified similitude still be a contraction map? Prove your answer.*

**Problem 15.52 Essay.** *Create a parse tree language, for genetic programming, that must give a contraction map from from the real line to itself.*

**Problem 15.53 Essay.** *Would two chaos automata that achieved similar fitness values on the same data using the bounded dispersed separation fitness produce similar pictures?*

**Problem 15.54 Essay.** *Suppose we had a data set consisting of spam and normal e-mail. Outline a way to create a fractal from the character data in the e-mail. Assume you are working from the body of the e-mail, not the headers, and that the number of recipients of an e-mail has somehow been concealed.*

**Problem 15.55 Essay.** *When trying to understand the behavior of evolutionary algorithms, we have used the metaphor of a fitness landscape. Describe, as best you can, the fitness landscape in Experiment 15.19.*

**Problem 15.56 Essay.** *When trying to understand the behavior of evolutionary algorithms, we have used the metaphor of a fitness landscape. Describe, as best you can, the fitness landscape in Experiment 15.22.*

**Problem 15.57 Essay.** *Suppose that we have a black and white picture. Construct a fitness function that will encourage the type of fractal used in Experiment 15.19 to match the picture.*

**Problem 15.58 Essay.** *Define* chaos GP-automata *and describe a problem for which they might be useful.*