“Genetic” Programming

Sean Luke
Department of Computer Science
University of Maryland
College Park, MD 20742
http://www.cs.umd.edu/~seanl/

Shugo Hamahashi
Kitano Symbiotic Systems Project
6-31-15 Jingumae, M-31 Suite 6A
Shibuya-ku, Tokyo 150-0001, Japan
http://www.shugo.com

Hiroaki Kitano
Sony Computer Science Laboratory
3-14-13 Higashi-Gotanda, Shinagawa
Tokyo 141 Japan
http://www.csl.sony.co.jp/person/kitano/

Abstract

Much of evolutionary computation was inspired by Mendelian genetics. But modern genetics has since advanced considerably, revealing that genes are not simply parameter settings, but interactive cogs in a complex chemical machine. At the same time, an increasing number of evolutionary computation domains are evolving non-parameterized mechanisms such as neural networks or symbolic computer programs. As such, we think modern biological genetics offers much in helping us understand how to evolve such things. In this paper, we present a gene regulation model for *Drosophila melanogaster*. We then apply gene regulation to evolve deterministic finite-state automata, and show that our approach does well compared to past examples from the literature.

1 EVOLUTIONARY COMPUTATION AND BIOLOGY

Many areas of evolutionary computation, especially genetic algorithms (GA), and genetic programming (GP), are heavily influenced by genetics. This influence can be seen in the adopted terminology of evolutionary computation: chromosomes, genes and genomes, alleles, loci, recombination and mutation, introns, haploidy and diploidy, genotypes and phenotypes. While evolutionary computation has drawn some inspiration from modern genetics, by and large the inspiration for this field has been aging Mendelian ideas. Even foundational concepts like schemata, deception, royal roads, linkage, etc. are usually expressed in terms of a genome as a tuple in a parameterized space of fitnesses influenced by linkage. Even Melanie Mitchell’s highly-regarded GA text [Mitchell 1996] introduces biological genes thus:

> A chromosome can be conceptually divided into *genes*—functional blocks of DNA, each of which encodes a particular protein. Very roughly, one can think of a gene as encoding a *trait*, such as eye color. The different possible “settings” for a trait (e.g., blue, brown, hazel) are called *alleles*.

This is not how it actually works. Mendelian genetics is certainly useful at the macro-level, modelling overall traits of large organisms in stabilized populations with a correspondingly large number of genes. But on the micro level, scientific understanding of DNA has advanced to the point that we know that genes bear relatively little resemblance to simple parameter settings. Genes form a complex network of interrelationships and regulation which, when seeded with initial chemical concentrations distributed throughout cells, results in miniature chemical machines. This is not to say that Mendel is inappropriate for traditional parameterized GA domains — far from it. But a rapidly growing body of evolutionary computation research, comprising many GA and evolutionary programming (EP) domains, and practically all GP domains, is attempting to evolve something quite different: *mechanisms*. Functional programs, neural networks, decision trees, cellular automata, L-systems, finite state automata. For these domains, a micro-level view of genetics seems more appropriate as an inspirational model.

The real merits of GP and GA recombination (among other things) have also recently been called into question (see for example [Angeline 1997a], [Hinterding, Gielewski, and Peachey 1995], [Luke and Spector 1998], [Tate and Smith 1993]) especially in these mechanism-building domains where the interrelationships between “genes” are complex and in extreme cases reduce recombination to little more than randomization. Yet biological systems of similar complexity appear to use recombination quite effectively. Why? One possibility worth considering is that recombination and other genetic-search concepts work especially well with the *biological genetic structures and dynamics for which they have co-adapted for so long.* As we drift further away from anything resembling real genetics (much less real evolution), such mechanisms may have correspondingly less applicability.

Biology has advanced greatly since the founding inspirations behind modern evolutionary computation. It may well be high time we reexamined our biological roots.

In the rest of this paper, we begin by describing our work in modelling an interesting area of developmental genetics, *gene regulation*, which we think has particular utility

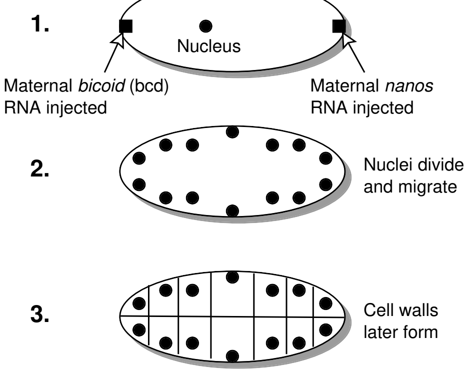

Figure 1: Steps in *Drosophila* Embryonic Development.

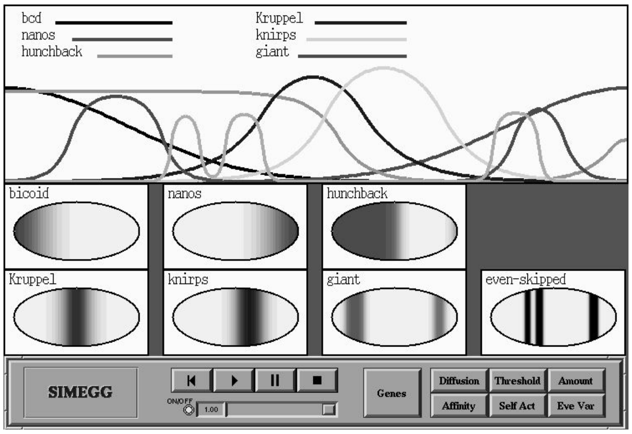

Figure 2: *Drosophila* Gene Regulation Simulation System. This screenshot shows seven high-level regulatory proteins generated by seven respective DNA genes: the main graph shows the stabilized protein concentration (the *y* axis) along the long (horizontal) axis of the embryo. The small embryo pictures depict varying resultant protein concentration as they would appear in typical cell-staining pictures.

to many new domains in evolutionary computation. We then present a preliminary "first stab" at applying some of these ideas to evolutionary computation itself, by evolving deterministic finite-state automata (DFA) which induce the Tomita language set, a popular and nontrivial language induction benchmark [Tomita 1982]. Despite their preliminary nature, our results nonetheless hold their own against past examples from the literature. We hope this will encourage further explorations into modern genetics as inspiration for evolutionary computation.

2 GENE REGULATION

Roughly speaking, the DNA sequence of a gene typically consists of a *promoter*, a series of *exons* separated by *introns*, and a *terminator*. The promoter is a series of DNA codons to which RNA Polymerase binds, initiating the gene transcription process. Exons are those parts of the gene which are converted into RNA, while introns are those parts of the gene that are "edited out". When RNA Polymerase reaches the terminator, it stops the transcription process. But RNA Polymerase doesn't just start transcribing all by itself. It typically needs a push, which DNA provides in the form of one or more *activation* or *enhancer* sites located near the gene. Various proteins readily bind to these sites: some proteins activate RNA Polymerase, giving it the push it needs, while others inhibit the transcription of that gene. Similar activation or inhibition may also occur later in the process (when RNA forms protein, or when protein is further edited). In this way, protein generated from specific genes *regulates* other genes' protein-production ability. This process forms a complex graph of regulatory interrelationships, including even mutual regulation and self-regulation.

Among the best-understood regulatory genes are the high-level regulatory genes responsible for the early developmental process. The embryo of *Drosophila melanogaster* (a fruit fly) is a very popular testbed for experiments on these genes, mostly because it is easy to mutate them in dramatic, highly visual ways, adding extra sets of wings, replacing antenna with legs, etc. Further, *Drosophila's* embryo has special features that make obvious the basic inner workings of gene regulation. Unlike in many organisms, *Drosophila*'s embryo splits its nucleus into many nuclei which migrate to different parts of the embryo. Only after a great many nuclei have formed and migrated do cell walls begin to appear, as shown in Figure 1. This means that for much of the early embryonic development, proteins may freely flow through the embryo unimpeded by cell walls.

During embryonic development, a suite of high-level genes produce proteins which regulate the production of "downstream" genes responsible for producing the proteins which form actual body parts. These regulatory genes also regulate each other's protein production, resulting in a complex web of gene regulation. This gene regulation may be crudely described with a set of if-then rules, for example: "if the protein generated by gene A has a concentration of 0.4 or greater, it inhibits the protein production of gene B with a weight of 0.2". In fact, genes can have a variety of complex inhibitory or activating properties based on narrow ranges of gene protein concentration. Proteins may also have different rates of diffusion within the cell, which adds a spatial dimension to such gene regulation.

Initially, when a mother lays a *Drosophila* egg, she inserts into each end different clumps of RNA, one coding for the *bicoid* protein, and one coding for the *nanos* protein (see Figure 1). *bicoid* and *nanos* inhibit each other's protein production; as a result, at one end of the embryo the *bicoid* protein is prevalent, but at the other end *nanos* is prevalent. This sets up an initial state of protein concentration in the embryo which “bootstraps” regulatory gene production throughout the cell. After a while the system stabilizes with a complex mosaic of regulatory protein concentrations in different parts of the cell. When the cell walls finally form, each cell thus has differing amounts of regulatory protein concentrations; these form new initial conditions for gene regulation within each new cell, causing different cells to form different body parts.

We have developed a gene regulation simulator for the *Drosophila* embryo, shown in Figure 2. As it turns out, our model is very close to actual biological results. But more interestingly, we can also model so-called “knock-out” mutations, eliminating an entire regulatory gene in the DNA, or adding new unusual ones. This changes the resultant final mosaic of protein concentrations, which accurately predicts changes in body part location and form.

3 A NEW GENE MODEL

Our work in modelling *Drosophila* embryonic development, DNA coding, and gene regulation has suggested to us a number of interesting approaches to evolutionary computation which take advantage of modern genetics. As a first experiment in this area, we have created a deterministic finite-state automata (DFA) induction mechanism inspired by ideas in gene regulation. While this is admittedly preliminary work, we have been pleasantly surprised by the results in comparison to the literature. To compare our DFA-induction mechanism with other examples from the literature, we have performed two separate experiments on the Tomita language set, shown in Table 1.

There has been past work in evolving gene-regulation-like mechanisms, but to our knowledge most of it has modelled gene regulation for theoretical biology or artificial life (see for example [Behera and Nanjundiah 1997], [Chiva and Tarroux 1994]), as opposed to applying it to non-biological evolutionary computation problems.

3.1 THE GENE MODEL

In our experiments, we used a traditional generational GA approach, creating an initial population, assessing the fitness of all its members, then repeatedly performing tournament selection (size 7), followed by recombination and mutation, to generate a new population to assess. However, in our model a genome is simply a multiset of one or more genes, each tagged with a floating-point locus value between 0.0 and 1.0 which defines its position on a (presumably large) linear chromosome.

The idea of a genome of arbitrary size is obviously not new. There is a considerable body of work in genomes of varying size or form. Much of this has grown out of the desire to evolve arbitrary-sized neural networks, rule sets, or symbolic computer programs. Evolutionary programming attempted such things from its very inception (in [Fogel *et al* 1966]); a noted recent example is GNARL, which used EP to evolve neural networks [Angeline 1994]. Nearly all of GP (beginning with [Cramer 1985], [Koza 1992]) uses arbitrary-length genomes, as its original impetus was to evolve symbolic functions from LISP s-expressions. From GP have grown a variety of genomes for computer program evolution, including arbitrarily-long linear genomes for stack machines [Spector and Stoffel 1996] or recurrent graphs of symbolic functions [Teller 1996]. Other interest in arbitrary-length genomes has sprung from attempts to improve on GA’s fixed-length vector genotypes while still evolving for a fixed number of parameter settings. Best known in this area is the *messy GA* [Goldberg *et al* 1993], later extended in [Kargupta 1996]. Lastly, some theoretical work has dealt with genes as unordered sets of arbitrary objects [Radcliffe and George 1993].

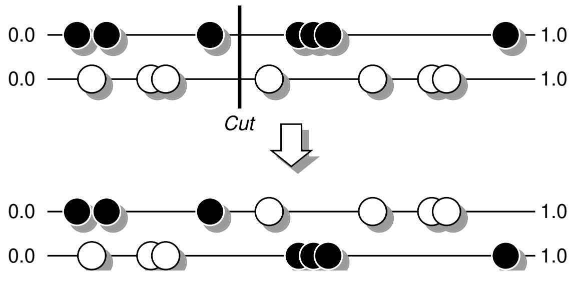

Figure 3: One-Point Crossover. On linear genomes with an arbitrary number of genes with floating-point loci.

Perhaps more closely related to our work is a body of literature interested in applying a truly “DNA-like” genome to GA problems. Genomes in such approaches have typically been long strands of DNA-like codons, often even using a four-letter A,T,G,C alphabet. In these strands are found genes delimited by start and stop codon sequences. For example, [Fullmer and Miikkulainen 1991] used such a genome to evolve neural networks. [Jakobi 1995] introduced a similar encoding to attack problems in evolutionary robotics. [Wu and Lindsay 1995] examined the dynamics of non-coding segments (parts of the strands that did not define genes) in similar genomes. [Burke *et al* 1998] continued this work, adding more ideas from biology, including multiple reading frames (allowing genes to overlap) and homologous crossover (recombination at points where codons are most similar). Unfortunately, most work in this area is exploratory; few such papers have compared this approach empirically to other EC work in nontrivial domains.

While our approach has similarities to past “DNA-like” work, we have chosen to represent genomes expressly without long strands of codons. There are good reasons for our approach. First, large DNA-like strands are computationally very expensive for recombination, mutation, and gene-determination. Second, they impose arbitrary encoding requirements (codon alphabets, and gene length predetermined by start and stop codons) which are often problematic for many problem domains.

<table>
<tr><th>Language</th><th>Description</th></tr>
<tr><td>1</td><td>1*</td></tr>
<tr><td>2</td><td>(10)*</td></tr>
<tr><td>3</td><td>(0|11)*(1*|(100(00|1)*))</td></tr>
<tr><td></td><td>Any string without an odd number of consecutive 0's after an odd number of consecutive 1's</td></tr>
<tr><td>4</td><td>1*((0|00)11*)*(0|00|1*)</td></tr>
<tr><td></td><td>Any string without more than 3 consecutive 0's</td></tr>
<tr><td>5</td><td>(((1|0)(1|0))*(1|0))|((11|00)*((01|10)(00|11)*(01|10)(00|11)*)*(11|00)*)</td></tr>
<tr><td></td><td>Any string of even length which, making pairs, has an even number of (01)'s or (10)'s</td></tr>
<tr><td>6</td><td>((0(01)*(1|00))|(1(10)*(0|11)))*</td></tr>
<tr><td></td><td>Any string such that the difference between the number of 1's and 0's is a multiple of 3</td></tr>
<tr><td>7</td><td>0*1*0*1*</td></tr>
</table>

Table 1: The Seven Tomita Languages.

Third, in most eukaryotes, especially higher-order ones, DNA genes can be rather sparse. For example, the actual amount of coding DNA in humans is about 3 percent (the rest is roughly 27% intragenic noncoding DNA, such as introns, and about 70% extragenic DNA — see [Strachan 1992]). For this reason, we felt a more appropriate model was one which treated a genome as broad expanse on which individual genes were occasionally found, but where the actual makeup of non-coding DNA was immaterial.

We performed recombination with a simple (and simplistic) form of one-point crossover, shown in Figure 3: we selected two arbitrary genomes, picked a random value from 0.0 to 1.0, then swapped between the genomes all the genes with loci greater than the random value. Our initial experiment did not perform inversion, bulk transposition, etc. However, genes were permitted to *migrate* by adding a small amount of Gaussian noise (in our experiment, 0.1 std. dev.) to their loci. Obviously there are many more sophisticated ways of dealing with gene movement (including transposons or actual gene duplication), which we hope to try in the future.

Gene migration brings up an interesting area of study, which we hope to examine more closely in the future: genes clustering together as protection from recombination. The genetic literature is filled with theoretical analysis of gene migration, including crossover "hot-spots", supergene clusters, and operons and statistical explanations for the unusual ordering of homeobox genes [Brown 1992]. These ideas are closely related to recent work in *self-adaptive* evolutionary computation, where the evolutionary system evolves its recombinative mechanism during the run. See [Angeline 1995] for an excellent survey of this area.

During genome mutation, with a certain probability (0.1) each gene in the genome was mutated as described in the next section. Next, genes were deleted, then new genes were added. Addition and deletion used a simple binomial distribution — that is, random values from 0.0 to 1.0 were chosen repeatedly until a number was chosen greater than some probability (0.03 for adding genes, 0.05 for deleting them). The number of random values chosen indicated the number of genes to be deleted (which were picked at random) or the number of new genes to be added. In the future we hope to attempt a more sophisticated approach, perhaps to test the utility of gene duplication, etc.

4 EVOLVING FINITE-STATE AUTOMATA

A deterministic finite-state automata (DFA) is a directed graph of one or more *states* connected with *transition edges*, which determines whether or not a particular string of symbols (in our case, 0's and 1's) is a member of some regular language. One state in the DFA is labelled the *starting state*, and zero or more states are labelled *accepting states*. A state may be a starting state, an accepting state, both, or neither. Transition edges are labelled with symbols in the language (1 or 0), and every state must have exactly one outgoing edge for each symbol in the language. The DFA begins in the starting state, and as it reads each symbol in turn along the string, it transitions along the appropriate edges to a new states. When the string ends, if the DFA is in an accepting state, then the string is in the language.

The mapping of genomes to DFAs is natural: genes are states in the DFA, and a gene-regulation-like mechanism determines state transitions. In our experiments, each gene had boolean value indicating whether or not it was an accepting state: upon gene-mutation, this value flipped with a 0.2 probability. Each gene also had a floating-point value from 0.0 to 1.0 which determined its *starting-state candidacy*: the gene with the lowest such value was the starting state for the DFA. Upon gene mutation, this candidacy value was randomized with a probability of 0.2.

4.1 CHEMICAL TEMPLATES

When modelling something like gene regulation, it is important to come up with a way to determine how well a protein's expression pattern fits with a gene's activation regions. The way this is done in biology is through arbitrary and infinite chemical interrelationships. Obviously this would be nontrivial to model tractably in a computer.

In our model, genes are instead given special objects we call *chemical templates*. One chemical template in a gene represents the expression pattern of the protein generated by that gene, and other chemical templates represent various activation sites for the gene. A chemical template is an array T of n tuples $(T_{s,0}, T_{p,0}), \ldots, (T_{s,n-1}, T_{p,n-1})$. For all $0 \leq i < n$, $T_{p,i}$ is a floating-point value from 0.0 to 1.0, and $T_{s,i}$ is 0 or 1, indicating whether or not $T_{p,i}$ is "turned on". T must be of at least some minimum size m (in our experiments, $m = 1$), and $T_{s,i}$ must be 1 for all $0 \leq i < m$. The function $Match()$ returns the degree to which a chemical template A "matches" or "fits" with some other chemical template (B), with 1.0 being the worst match and 0.0 being the best match, using a normalized nearest-neighbor function which ignored "turned off" areas:

$$Match(A,B) = \sqrt{\frac{\sum_{i=0}^{\min(|A|,|B|)-1} (A_{p,i} - B_{p,i})^2 A_{s,i} B_{s,i}}{\sum_{i=0}^{\min(|A|,|B|)-1} A_{s,i} B_{s,i}}}$$

The reason for using arbitrary-length arrays with "turned off" areas within them is that it allows the template matching between gene A and gene B to be independent of the template matching between gene A and gene C. This permits self-adaption such that two genes may "migrate" towards or away from each other independent of their relationships with other genes in the genome. In the future, we hope to further study this and other plausible approaches.

In our model, genes had three chemical templates which determined state transitions: an *expression* template, a *reading-0* template and a *reading-1* template. For each state in the DFA, the receiving state of the outgoing reading-0 edge or reading-1 edge was simply that gene whose expression template most closely matched the first gene's reading-0 template or reading-1 template, respectively.

5 THE FIRST EXPERIMENT

We performed two experiments in inducing the Tomita language set. In our first experiment, we compared gene-regulation with three examples from the literature which used the Tomita language set as their domain.

The standard experimental methodology for most Tomita language induction experiments in the literature is to attempt to induce a mechanism which properly classifies all positive and negative examples in a limited training set of binary strings up to some length. Afterwards, this mechanism is tested for generality on the full population of binary strings of that length. We used the same accuracy measurement which was used by the other experiments both as a raw fitness metric and as the final generalization accuracy, namely:

$$\frac{\textit{correct negative examples} + \textit{correct positive examples}}{\textit{all negative examples} + \textit{all positive examples}}$$

Tomita Lang.	Positive Examples	Negative Examples
1	ϵ, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111	0, 10, 01, 00, 011, 110, 11111110, 10111111
2	ϵ, 10, 1010, 101010, 10101010, 10101010101010	1, 0, 11, 00, 01, 101, 100, 1001010, 10110, 110101010
3	ϵ, 1, 0, 01, 11, 00, 100, 110, 111, 000, 100100, 110000011100001, 111101100010011100	10, 101, 010, 1010, 1110, 10001, 111010, 1001000, 11111000, 0111001101, 1011, 11011100110
4	ϵ, 1, 0, 10, 01, 00, 100100, 001111110100, 0100100100, 11100, 010	000, 11000, 0001, 000000000, 00000, 11111000011, 1101010000010111, 1010010001, 0000
5	ϵ, 11, 00, 1001, 0101, 1010, 1000111101, 1001100001111010, 111111, 0000	1, 0, 111, 010, 000000000, 1000, 01, 10, 011, 1110010100, 010111111110, 0001
6	ϵ, 10, 01, 1100, 101010, 111, 000000, 10111, 0111101111, 100100100	1, 0, 11, 00, 101, 011, 00000000, 010111, 10111101111, 11001, 1001001001, 1111
7	ϵ, 1, 0, 10, 01, 11111, 000, 00110011, 0101, 0000100001111, 00, 00100, 011111011111	1010, 00110011000, 0101010101, 1011010, 10101, 010100, 101001, 100100110101

Table 2: Positive and Negative Training Examples. Used in Experiment 1 and in [Angeline 1997b]. A very slightly different training set was used in [Angeline 1994] and [Waltros and Kuhn 1992].

We think this is an unfortunate metric, since the number of negative examples is often far fewer than the number of positive examples (or vice versa). For future work we suggest using one of two metrics we think are better:

$$\frac{\frac{\textit{correct negative examples}}{\textit{all negative examples}} + \frac{\textit{correct positive examples}}{\textit{all positive examples}}}{2}$$

$$\min\left(\frac{\textit{correct negative examples}}{\textit{all negative examples}}, \frac{\textit{correct positive examples}}{\textit{all positive examples}}\right)$$

Additionally, experiments typically report the number of evaluations (presentations of the training set to the mechanism) before it is able to properly classify the training set. An assessment of a mechanism should consider both the number of evaluations and the generalization accuracy.

The three examples in the literature were:

- [Waltros and Kuhn 1992] developed a finite state machine using a trainable recurrent neural network. They report the average of five runs per language with strings of length ten or less.
- [Angeline 1994] introduced GNARL, a recurrent neural network developed using evolutionary programming. The paper reports the results of a single run per language, using a population of 50 networks and at most 1000 generations. [Angeline 1994] used strings of length ten or less.
- [Angeline 1997b] revisited the Tomita language set with an inventive GP-like mechanism called a MIPs Net, evaluated over strings of length fifteen or less. A MIPs Net is a GP program consisting of a main tree and zero or more subtrees forming *automatically defined functions* (ADFs) [Koza 1994]. Each tree in Angeline's MIPs Nets could call any other trees; thus the program had a recurrent call-graph. Angeline restricted the trees to five maximum, and reported the results of two experiments. [Angeline 1997b] performed a single run per language consisting of five subruns of 250 generations each, with a population size of 100. In the first subrun, MIPs Nets in the population used a single tree. If no solution was found, for the next subrun (250 generations later) the population was reinitialized with two trees each, then three, etc., until all five subruns were completed or a solution had been found for the training examples.
- We also include a second experiment in [Angeline 1997b] which hard-set the size of the MIPs nets to between 0 and 4 subtrees beyond the main tree. 55 runs were performed on Tomita language 7 for each of the five settings of MIPs net size, again using strings of length fifteen or less.

In comparison, we performed 50 runs per language, and averaged the results. Each run had a population of 500, and ran for 100 generations or until a solution was found which correctly classified all its training examples. The training set we and [Angeline 1997b] used is shown in Table 2. Other experiments differed slightly from this training set (by one or two examples per language). [Waltros and Kuhn 1992] and [Angeline 1994] used strings of size 10 or less as their training and generalization strings, while [Angeline 1997b] and our experiment used strings of size 15 or less. Shorter strings are slightly easier to generalize to.

Table 3 compares the results of our experiment with the others in the literature. We think that our approach, unoptimized as it is, works well. In many cases we achieved generalization that was as good as or better than others for the same language, using many fewer evaluations. In those cases where our generalization was not as good, we had fewer evaluations (or vice versa). In no case were we lower both in generalization accuracy and number of

Tomita Lang.	Gen. 100%	Avg. Evals	Gen. Accuracy Avg.	Var.	Best
First Experiment 50 runs per language, strings size 15 or less.					
1	31	30	88.39	0.0391	100.00
2	7	1010	84.00	0.0232	100.00
3	1	12450	66.28	0.0174	100.00
4	3	7870	65.25	0.0324	100.00
5	0	13670	68.65	0.0147	82.94
6	47	2580	95.94	0.0269	100.00
7	1	11320	67.69	0.0221	100.00
[Angeline 1997b] Number of ADFs increases until solution found. 1 run per language, strings size 15 or less.					
1		2200			100.00
2		6600			80.04
3		80400			73.70
4		32200			64.08
5		85200			68.97
6		35100			100.00
7		71300			53.08
[Angeline 1997b] Just language 7, with fixed (0–4) numbers of ADFs. 55 runs for each ADF size, strings size 15 or less.					
7/0		46285	50.98	0.0328	
7/1		43134	54.60	0.0339	
7/2		40215	52.06	0.0298	
7/3		35529	52.74	0.0202	
7/4		35187	54.91	0.0232	
[Angeline 1994] 1 run per language, strings size 10 or less.					
1		3975			100.00
2		5400			96.34
3		25050			58.87
4		15775			92.57
5		2050			49.39
6		21475			55.59
7		12200			71.37
[Waltros and Kuhn 1992] 5 runs per language, strings size 10 or less.					
1		3034	88.98		100.00
2		4523	91.18		100.00
3		12329	64.87		78.31
4		4393	52.50		60.92
5		1587	44.94		66.83
6		2138	23.19		46.21
7		2969	36.97		55.74

Table 3: Results of the First Experiment. Shown with four similar experiments from the literature. *Gen 100%* is the number of runs with a fully general solution (up to the max string length). *Avg. Evals* is the average number of evaluations per run before a solution to the training set was found. *Gen. Accuracy* is the percentage of all strings (up to the max string length) correctly classified.

Tomita Lang.	Gen. Accuracy Avg.	Var.	Best	Train. 100%	Gen. 100%
Second Experiment 20 runs per language, strings size 15 or less.					
1	94.98	0.0500	100.00	20	14
2	94.77	0.0498	100.00	20	3
3	90.52	0.0504	100.00	15	10
4	94.69	0.0499	100.00	20	18
5	76.69	0.0564	100.00	1	1
6	83.09	0.0654	100.00	13	12
7	90.76	0.0510	100.00	14	6

Table 4: Results of the Second Experiment. *Gen. Accuracy* is the percentage of all strings (length 15 or less) correctly classified. *Train 100%* is the number of runs in which a solution was found which correctly classified all 200 training cases. *Gen 100%* is the number of runs in which the solution was fully general to the language (for strings length 15 or less).

evaluations.[1] For the one experiment which provided variances (the second experiment in [Angeline 1997b]), our results were statistically significantly better (using a two-sample, two-tailed t-test at 95%), with fewer evaluations.

6 THE SECOND EXPERIMENT

Our second experiment compared our gene-regulation-inspired approach with [Brave 1996], again using language induction over the Tomita Language set.

Brave evolved deterministic finite-state automata for language induction using *cellular encoding* [Gruau 1992], a novel mechanism for automated graph-generation. Cellular encoding evolves GP-like trees containing graph-grammar rules which define how a graph "grows" out of an initial "embryonic" node through a series of node splits and labelling procedures. Brave used a restricted form of cellular encoding to define the graph structure of DFAs induced for a given language. In his paper, Brave evolved DFAs inducing the Tomita language set, using 500 positive and 500 negative cases. The paper does not specify the the selection procedures for these cases, nor their maximal length. The population size was 10,000, with a maximum of 100 generations, and the fitness metric was simply the number of incorrectly classified sentences (0 being the ideal).

[1]Missing statistics can make it hard to compare these figures. One crude metric to use is to consider how often in our fifty first-experiment runs a 100% general solution was found. This suggests the expected number of runs necessary to achieve a near-perfect score. An example: for the Tomita language 1, we achieved on average an 88.39% generalization, while [Angeline 1997b] achieved a 100% generalization. But we achieved 100% generalization in 31 runs out of 50, suggesting that only two runs (60 evaluations) on average would be needed to find a 100% generalized result, versus 2200 evaluations in [Angeline 1997b].

Brave reports that for all languages except for language 6, the cellular encoding system found 100% correct solutions within the 100 generations, and that all six solutions were fully generalizable. The experiment was repeated ten times for language 6, but no solution was found, presumably because of a lack of generality in the particular cellular encoding mechanism chosen.

To compare our approach with Brave's findings, we randomly generated 100 positive and 100 negative cases (selected from the corpus of strings length 15 or less). This is one fifth the number of cases used in [Brave 1996]. Using this as a training set, twenty runs were performed using a population size of 500, for 100 generations each, so the maximal number of evaluations for all twenty runs combined equalled the maximal number of evaluations for one run in [Brave 1996]. The results are summarized in Table 4. For each language, anywhere from one to eighteen of our twenty runs resulted in a fully generalizable solution.

7 CONCLUSION

Our experiments attempted to apply gene regulation and other modern genetics ideas to evolutionary computation in evolving an inductive mechanism for the Tomita language set. In the future we hope to also include gene duplication, circular or multiple linear chromosomes, recombinative hot-spots, and a host of other interesting issues that were ignored in our first experiments. Still, we think that these preliminary experiments performed well compared to other language induction examples from the literature.

We think that this area is wide open for experimentation, and that our results demonstrate that such experiments need not be mere proofs of concept. Discoveries in genetics have come rapidly in the last decade or so, changing our understanding of genes and genomes and their dynamics. We think modern genetics holds much promise in furthering the field evolutionary computation, and hope that this work encourages more exploration in this interesting area.

References

Angeline, P. 1995. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence: A Dynamic Systems Perspective,* M. Palaniswami, Y. Attikiouzel, R. Marks, D. Fogel and T. Fukuda eds. 152–163. Piscataway, NJ: IEEE Press.

Angeline, P. 1997a. Subtree crossover: building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97).* J. Koza *et al*, eds. 240–248. San Francisco: Morgan Kaufmann.

Angeline, P. 1997b. An alternative to indexed memory for evolving programs with explicit state representations. In *Proceedings of the Second Annual Conference on Genetic Programming (GP97)*, J.R. Koza *et al*, eds. 423–430. Morgan Kaufmann.

Angeline, P., G.M. Saunders, and J.B. Pollack. 1994. An evolutionary algorithm that constructs recurrent neural networks. In *IEEE Transactions on Neural Networks*, 5(1).

Behera, N. and V. Nanjundiah. 1997. Trans gene regulation in adaptive evolution: a genetic algorithm model. *Journal of Theoretical Biology* 188. 153–162.

Brave, S. 1996. Evolving deterministic finite automata using cellular encoding. In *Proceedings of the First Annual Conference on Genetic Programming (GP96)*, J.R. Koza *et al*, eds. 39–44. MIT Press.

Brown, T. 1992. *Genetics: A Molecular Approach*. London: Chapman and Hall.

Burke, D., K. DeJong, J. Grefenstette, C. Ramsey, and A. Wu. 1998. Putting more genetics into genetic algorithms. In *Evolutionary Computation*. 6:4.

Chiva, E. and P. Tarroux. 1994. Studying genotype-phenotype interactions: a model of the evolution of the cell regulation network. In *Proceedings of the International Conference on Evolutionary Computation (PPSN-III), Lecture Notes in Computer Science #866*, Y. Davidor, H-P. Schwefel, and R. Männer, eds. 22–35. Springer-Verlag.

Cramer, N.L. 1985. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, J. Grefenstette, ed. 183–187. Hillsdale, NJ: Lawrence Erlbaum Associates.

Fogel, L.J., A.J. Owens, and M.J. Walsh. 1966. *Artificial Intelligence through Simulated Evolution*, New York: John Wiley.

Fullmer, B. and R. Miikkulainen. 1991. Using marker-based genetic encoding of neural networks to evolve finite-state behavior. In *Proceedings of the First European Conference on Artificial Life (ECAL91)*.

Goldberg, D.E., K. Deb, H. Kargupta, and G. Harik. 1993. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth Annual International Conference of Genetic Algorithms*. 56–64.

Gruau, F. 1992. Cellular encoding of boolean neural networks with a cell rewriting developmental process. In *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN92)*. J.D. Schaffer and D. Whitley, Eds. 55–74. IEEE Computer Society Press.

Hinterding, R., H. Gielewski, and T.C. Peachey. 1995. The nature of mutation in genetic algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms,* L.J. Eshelman, ed. 65–72. San Francisco: Morgan Kaufmann.

Jakobi, N. 1995. Harnessing morphogenesis. In *International Conference on Information Processing in Cells and Tissues*.

Kargupta, H. 1996. The gene expression messy genetic algorithm. In *Proceedings of the IEEE International Conference on Evolutionary Computation*.

Koza, J.R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Koza, J.R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Luke, S. and L. Spector. 1998. A revised comparison of crossover and mutation in genetic programming. *In Genetic Programming 1998: Proceedings of the Third Annual Conference (GP98)*. J. Koza *et al*, eds. 208–213. San Francisco: Morgan Kaufmann.

Mitchell, M. 1996. *An Introduction to Genetic Algorithms*, 161–162. MIT Press.

Radcliffe, N. and F. George. 1993. A study in set recombination. *Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA93).* 22–30. San Fransisco: Morgan Kaufmann.

Strachan, T. 1992. *The Human Genome*. Oxford: BIOS.

Spector, L. and K. Stoffel. 1996. High-performance, parallel, stack-based genetic programming. In *Proceedings of the First Annual Conference on Genetic Programming (GP96)*, J.R. Koza *et al*, eds. 394–399. MIT Press.

Tate, D.M., and A.E. Smith. 1993. Expected allele coverage and the role of mutation in genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms,* S. Forrest, ed. 31–37. San Francisco: Morgan Kaufmann.

Teller, A. 1996. Neural programming and an internal reinforcement policy. In *Late-breaking Papers at the Genetic Programming 1996 Conference (GP96)*, J.R. Koza, ed. 186–192. Stanford Bookstore.

Tomita, M. 1982. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society.* 105–108.

Waltrous, R. and G. Kuhn. 1992. Induction of finite-state automata using second-order recurrent networks. In *Advances in Neural Information Processing 4,* J. Moody, S. Hanson, and R. Lippmann, eds. 309–316. San Fransisco: Morgan Kaufmann.

Wu, A. and R. Lindsay. 1995. Empirical studies of the genetic algorithm with non-coding segments. *Evolutionary Computation*. 3:2.