Brock University

Department of Computer Science

Probabilistic Pattern Matching and the Evolution of Stochastic Regular Expressions

Brian J. Ross

Tech. Report # CS-99-01

May 1999

Brock University

Dept. of Computer Science

St. Catharines, Ontario

Canada L2S 3A1

www.cosc.brocku.ca

Probabilistic Pattern Matching and the Evolution of Stochastic Regular Expressions

Brian J. Ross
Brock University, Dept. of Computer Science
St. Catharines, Ontario, Canada L2S 3A1
bross@cosc.brocku.ca
(905) 688-5550 ext. 4284

Abstract

The use of genetic programming for probabilistic pattern matching is investigated. A stochastic regular expression language is used. The language features a statistically sound semantics, as well as a syntax that promotes efficient manipulation by genetic programming operators. An algorithm for efficient string recognition based on approaches in conventional regular language recognition is used. When attempting to recognize a particular test string, the recognition algorithm computes the probabilities of generating that string and all its prefixes with the given stochastic regular expression. To promote efficiency, intermediate computed probabilities that exceed a given cut-off value will pre-empt particular interpretation paths, and hence prune unconstructive interpretation. A few experiments in recognizing stochastic regular languages are discussed. Application of the technology in bioinformatics is in progress.

1 INTRODUCTION

Language inference is a classical problem in machine learning, and continues to be an important and active research topic. The basic problem is, given a set of example behaviours or strings, automatically infer a corresponding language (grammar, automata, expression,...) which generates or recognizes those examples. Genetic algorithms (GA) and genetic programming (GP) have been applied towards languages inference, with varying degrees of success. Although successful inference is possible, the generic inference problem is not entirely well–suited for solution by evolutionary search. There are a number of reasons for this. For example, some genome encodings do not preserve useful language characteristics during crossover. Even small local changes to such genomes can be catastrophic, which does not lend itself well to genetic reproduction and evolutionary search.

An even more acute weakness is that "all or nothing" problems such as the language inference problem are not entirely natural for GP. An acceptable language inference minimally requires that the solution language correctly recognize all positive test cases, and reject all negative ones. This essential criteria may also be supplemented by efficiency concerns, such as a relatively small number of states or grammar rules. The resulting search space is a difficult one to navigate with evolutionary techniques, due to these stringent requirements for language correctness and completeness. On the other hand, it is generally recognized in the GP community that problems which require an "acceptably close" solution are typically the best candidates for successful solution with GP. Pragmatically speaking, giving the fitness function a larger degree of freedom for evaluating a successful solution will substantially increase the chances of the discovery of acceptable solutions.

This research addresses the inference of stochastic regular languages using genetic programming. Stochastic languages are formal languages with probability distribution associated with the language set. The stochastic language inference problem is similar to the classical inference problem, with the additional requirement that the distribution of strings recognized by the stochastic language conform to some desired target distribution. At first, this may seem intuitively more complex than non–stochastic language inference, since it is unclear what impact the determination of probability distributions has on the inference. It turns out, however, that the inclusion of string distributions can simplify the inference problem. Hypothesized languages are now allowed to generate erroneous strings so long as they fall within an acceptably small probability of occurence. In other words, the use of language distributions introduces a more generous degree of freedom for generated solutions. This is ideal in a GP setting, as it simplifies the search space substantially for evolutionary search.

The target language used here is a probabilistic regular expression language, henceforth called Stochastic Regular Expressions (or SRE). Although theoretically weaker than stochastic context–free languages studied elsewhere, it was nevertheless chosen due to both its amenability to concise GP representation, and its ability to naturally solve the substantial number of problems in the "regular language domain". The stochastic regular expression language is closely related to stochastic regular grammars and stochastic finite automata, the latter commonly referred to as Hidden Markov Models in the literature.

Some SRE language implementation issues had to be addressed before GP could be successfully applied to stochastic regular expression problems. Firstly, an efficient implementation of SRE interpretation was necessary. Interpretation of an SRE expression requires that the probability of recognizing a given string is generated. Since intermediate probabilities would be computed during the interpretation of a string, these values can be used to terminate or prune unproductive interpretation paths whose probabilities are smaller than some supplied cut–off probability. Given the extensive testing that is necessary during fitness evaluation, such pruning greatly increases the speed of GP runs. The SRE language is implemented in a grammatical GP system, which permitted the use of syntactic language constraints to further enhance evolution efficiency.

Two example experiments proved that probabilistic language inference is indeed possible with SRE and GP. The more complex of these experiments indicated that the complex search space often resulted in premature convergence. A minor language enhancement to this experiment resulted in failed inferences by the GP system. From this experience, it can be deduced that the fitness evaluation strategy used here is not a general purpose solution to all stochastic language problems, but rather, is suitable to a class of stochastic regular languages whose members are structurally related to one another.

An outline of the paper is as follows. Related work is reviewed in section 2. Section 3 defines the syntax and semantics of the stochastic regular expression language, and discusses the algorithm for processing SRE expressions. Section 4 outlines the genetic programming system used. Two example experiments are discussed in section 5. A discussion and future directions conclude the paper in section 6.

2 RELATED WORK

Formal language induction has a long history as a fundamental problem in machine learning (Fu and Booth 1975*a*, Fu and Booth 1975*b*, Angluin 1992, Sakakibara 1997). The specialized topic of stochastic languages has also been studied for some time (Fu and Huang 1972). A stochastic grammar differs from a conventional grammar in that each grammar rule is marked with a probability associated with its use, and the set of probabilities for a grammar encode a probability distribution for the resulting derived language. (Fu 1982) has an extensive treatment of stochastic grammars, their derivation, and their application in pattern recognition. Stochastic grammars are also more complex than their non–stochastic kin, as the distributions inherent with the language introduce a new dimension of membership criteria. For example, all context free languages are also stochastic context free languages (all probabilities are 1); however, there may be many stochastic context free languages having essentially the same membership set, but vastly different distributions over that set. Language equivalence issues are therefore more discriminating than in a non–stochastic setting. Stochastic context free languages enjoy both expressitivity and tractable properties, for example, the existence of useful inference algorithms (Lari and Young 1990). They have also found practical use in language processing (Charniak 1993).

Stochastic regular languages, albeit descriptively weaker than stochastic context-free languages, have also found their practical niche in applications. Regular languages are definable by finite automata, regular grammars, and regular expressions (Hopcroft and Ullman 1979). Similarly, stochastic regular languages are defined by stochastic versions of these three representations. Examples of work in stochastic grammar inference is in (Maryanski and Booth 1977, van der Mude and Walker 1978, Carrasco and Forcada 1996, Carrasco and Oncina 1998). Stochastic finite automata are defined in terms of Hidden Markov Models (HMM) (Rabiner and Juang 1986). An HMM is a finite automaton with probabilities marking the transition links between nodes. Each node is connected to all other nodes, and so the network itself is maximally connected. When particular transitions are not required, the probabilities associated with those nodes are set to zero. HMMs have found extensive use in language and speech processing (Rabiner 1989, Charniak 1993). Strangely enough, stochastic regular expressions have not been extensively studied; one example paper is (Garg *et al.* 1996).

Language inference has been successfully done using genetic algorithms (GA) and genetic programming (GP). The distinguishing difference between GA and GP approaches is one of denotation: a pure GA uses a binary encoding for the genome, while a GP uses a variable–sized parse tree. Some of the following use encodings with characteristics of both approaches.

With respect to regular languages, an early work in evolving finite automata is in (Zhou and Grefenstette 1986). They used a GA with a binary encoding of the automata as a set of state transitions, capped at a size of 8 states. A weakness of this encoding is that the represented automata are susceptible to destructive effects during crossover and mutation. Their unspecified fitness function scores language performance (ability to accept positive strings and reject negative examples) and automata size.

(Dunay *et al.* 1994)'s approach is similar to (Zhou and Grefenstette 1986), except that finite automata are denoted in GP–style nested S–expression notation.

(Dupont 1994) uses an automata–theoretic partition representation for regular languages. This has the advantage of preserving language properties of chromosomes during GA reproduction, unlike the more fragile FA represention in (Zhou and Grefenstette 1986). His fitness function scores both language performance and automata size. He successfully evolved a large set of regular languages, including the benchmark Tomita languages (Tomita 1982).

(Brave 1997) uses an abstract "cellular encoding" representation for deterministic FA's, which builds the network structure of a FA during interpretation. The intention of this denotation is to preserve structural properties of a language during evolutionary reproduction. His automata are embellished with boolean operators which permit automata composition. The fitness function tallied the number of correctly classified sentences. All but one of the Tomita languages were successfully inferred using this technique.

(Longshaw 1997) uses a straight–forward state–transition representation for automata. However, his GA uses a population seeded with correct but overly general automata. Specialized reproduction operators manipulate automata by duplicating or refining states. The overall intention is to refine the general automata into a more specific one for the language in question. His fitness function scores example classification performance and automata size.

(Svingen 1998) uses a GP on regular expressions. Regular expressions are directly encoded as program trees, and fitness is based on correct example classification. He successfully evolved the Tomita languages.

Context–free languages have also been studied. (Wyard 1991) uses a GA to evolve context–free grammars. Chromosomes takes the form of lists of production rules, which guarantees correctness at all times. The fitness function scores example classification performance. Two simple CFG's were successfully evolved.

(Lankhorst 1994) uses a vector encoding to represent grammar productions. His fitness function is more involved than most others, as it scores example classificaton performance, the length of substrings of examples correctly classified, the degree of determinism of grammars, and the ability of the grammar to generate correct strings not included in the example set. These additional evaluation considerations give the GA more information with which to drive evolution. He applied the GA to a number of CFG and regular languages.

(Lucas 1994) uses a binary–encoded normal form for CFG productions, which preserves language properties during reproduction, and promotes convergence. His fitness strategy scores example classification and grammar size.

(Sen and Janakiraman 1992) applies a GA towards inferring deterministic push-down automata, which is an alternative to the grammar representation for CFG's. Fitness scores example recognition performance, and whether the PDA attempts to erroneously 'pop' an empty stack. (Lankhorst 1995) extends this idea towards nondeterministic push-down automata. His fitness additionally considers prefix sizes and the stack size after a string has been consumed.

(Dunay and Petry 1995) use a Turing machine representation in their GA experiments. Although this powerful notation can denote the entire set of languages in the Chomsky, it does not necessary mean that search will be easy to accomplish, given the inherent enormity of the search space in question. To solve some relatively simple examples of regular, context–free and context–sensitive languages, they used a compositional approach, in which the GA had access to TM building blocks evolved in earlier runs.

Finally, the evolution of stochastic languages has been studied. (Schwehm and Ost 1995) uses a GA for evolving stochastic regular languages. Two different encodings are studied – production rules with probabilities, and quotient automata. The fitness function uses grammar complexity (number of productions), a modified χ^2 test for distribution conformance, and a measure of the grammar's ability to accept prefixes of the target grammar. A few experiments were performed, and their GA performance compares well with standard regular–language inference algorithms.

(Kammeyer and Belew 1997) uses a GA to evolve stochastic context–free grammars. They use a liberal representation for grammars in which correct grammars are parsed from the genome when evaluated; this permits intron or junk material to be included in chromosomes. The fitness function evaluates the size of test example prefixes consumed by a grammar, and uses cross-entropy to evaluate distribution conformance. They also use a local search technique for finding production probabilities during evolution. A couple of CFG's were successfully evolved.

3 STOCHASTIC REGULAR EXPRESSIONS

3.1 Language Definition

The target language for the GP system is stochastic regular expressions, or SRE. The language is very similar to one in (Garg *et al.* 1996), which is used for modeling the qualitative behaviour of stochastic discrete event systems. Amongst other properties, they prove that probabilistic regular language operations such as choice, concatenation, and Kleene-closure forms a closed language, and hence an algebra. Although a few basic properties will be illustrated here, the reader is referred to (Garg *et al.* 1996) for further details. It is assumed the reader is familiar with basic concepts from formal language theory (Hopcroft and Ullman 1979).

Two language variations, SRE and Guarded SRE (or gSRE), are used. We first define SRE. Let α range over alphabet $\sum^*$, E range over SRE expressions, n range over positive integers ($0 \leq n \leq 1000$), and f range over decimal values with a precision of 2 decimal places ($0 \leq f < 1.00$). The syntax of SRE is recursively defined as:

$$E \ ::= \ \alpha \ | \ \sum_i E_i(n_i) \ | \ E_1 : E_2 \ | \ E^{*f} \ | \ E^{+f}$$

Without loss of generality, the empty string ϵ is not included in the alphabet.

The operators have the following meaning:

1. *Atomic action* α : The action α is generated.

2. *Choice* $\sum_i E_i(n_i)$: This denotes a probabilistic choice of terms. Each choice expression E_i can be chosen with a probability:

$$\frac{n_i}{\sum_j n_j}$$

   For example, given the expression $E_1(3) + E_2(5)$, the term E_1 can be chosen with a probability of 3/8 and E_2 with a probability of 5/8.

3. *Concatenation* "$E_1 : E_2$" : Term E_1 is interpreted, followed by that of E_2.

4. *Kleene Closure* E^{*f} : Term E can be repeatedly executed 0 or more times, and each iteration occurs with a probability of f. The probability of E terminating execution is $1 - f$.

5. *+Closure* E^{+f} : Term E executes once, after which it repeatedly executes 0 or more times using the same probability scheme as Kleene closure. +Closure is an abbreviation for the following:

$$E^{+f} \ \equiv \ E : E^{*f}$$

The Guarded SRE language is identical to SRE, except that a guarded choice operator is used instead of the general choice in 2 above:

6. *Guarded Choice* $\sum_i E_i'(n_i)$, where $E' = (\alpha_i : E_i)$ or $E' = \alpha_i$, and $\forall \alpha_i, \alpha_j : \alpha_i \neq \alpha_j$: Here, each term in the choice expression is either prefixed with a unique atomic action that is found nowhere else in the expression, or consists of a unique action by itself. This makes guarded choice deterministic, unlike SRE's nondeterministic choice.

Note that, even with guarded choice, gSRE is still a nondeterministic language, since the closure operators are nondeterministic. The rest of the discussion in this section pertains equally to both SRE and gSRE.

A derivation of a conventional regular expression E is the set of sentences, or strings over the alphabet, derivable from it. This defines the language $L(E)$ of E. This notion of language derivation is similarly applicable to SRE, except that each string has a probability value associated with it, and hence the language itself is associated with a probability distribution of its members.

Alternatively, an intuitive way to consider SRE expressions is that every expression defines a specific probability function over strings in Σ^*:

$$E : \Sigma^* \rightarrow p \qquad (0 \leq p \leq 1)$$

Using a denotational semantics style of representation (Stoy 1977), the probability function for SRE expression E is denoted by $[\![E]\!]$, and its application to a particular string s is denoted $[\![E]\!]s$, which denotes the probability associated with string s in the language $L(E)$.

A probability function model of SRE is now given. Let $s = \alpha_1 ... \alpha_n \in \Sigma^*$.

- Atomic actions:

$$\begin{array}{lll} [\![\alpha]\!]\beta = 1 & if & \alpha = \beta \\ [\![\alpha]\!]\beta = 0 & if & \alpha \neq \beta \end{array} \tag{1}$$

- Choice (including guarded choice):

$$[\![\sum_i E_i(n_i)]\!]s = \sum_k \left(\frac{n_k}{\sum_j n_j} \cdot [\![E_k]\!]s \right) \tag{2}$$

  Since every term might recognize s, the overall probability for a choice expression is the sum of all the term probabilities with respect to s.

- Concatenation:

$$\begin{array}{ll} [\![E_1 : E_2]\!]s & = \sum_{i=1}^{n}([\![E_1]\!]\alpha_1 ... \alpha_i \cdot [\![E_2]\!]\alpha_{i+1} ... \alpha_n) \\ & \quad + \ [\![E_1]\!]s \cdot [\![E_2]\!]\epsilon \\ & \quad + \ [\![E_1]\!]\epsilon \cdot [\![E_2]\!]s \end{array} \tag{3}$$

  In the first summation, s is decomposed into two substrings, each of which may be consumed by a concatenated expression. Even though one term may recognize its substring argument, if the other term does not recognize its respective substring, then that term returns a probability of 0, and the overall probability for that instance of decomposition is 0. The rest of the formula represents the cases when one entire expression consumes s, while the other consumes ϵ. If these other cases do not succeed, then they return 0.

- Kleene closure:

$$\begin{array}{l} [\![E^{*f}]\!]\epsilon = 1 - f \\ [\![E^{*f}]\!]s = \sum_{i=1}^{n}(f \cdot [\![E]\!]\alpha_1 ... \alpha_{i-1} \cdot [\![E^{*f}]\!]\alpha_i ... \alpha_n) \\ \qquad\qquad + \ f \cdot [\![E]\!]s \cdot [\![E^{*f}]\!]\epsilon \qquad\qquad : s \neq \epsilon \end{array} \tag{4}$$

The first formula accounts for empty strings, as the only way an iterated expression should recognize an empty string is by not iterating. The other formula recursively defines the general case. Here, one iteration of E will consume some portion of s, and the rest of s is consumed by further iterations. The final term in this formula represents when the first iteration consumes the entire string. It is assumed that an iteration of a loop always consumes some non–empty string. Otherwise, the semantic model would have to account for Kleene closure iterating indefinitely on an argument, which is not useful behaviour.

- +Closure:

$$\begin{array}{rl} [\![E^{+f}]\!]s = & \sum_{i=1}^{n}([\![E]\!]\alpha_1...\alpha_i \cdot [\![E^{*f}]\!]\alpha_{i+1}...\alpha_n) \\ & + \; [\![E]\!]s \cdot [\![E^{*f}]\!]\epsilon \end{array} \tag{5}$$

  This is similar to the non–empty argument formula for Kleene closure, except that the expression E will consume part of the string before iterations commence. This can be seen by the lack of f value in the formula.

The nondeterministic nature of regular expressions is modeled in the above by multiple argument decomposition in the concatenation and closure operators. Nondeterminism can also arise in the (nonguarded) choice operator.

Membership in SRE is reflected by SRE expressions returning non–zero probabilities for particular strings:

$$\begin{array}{l} s \in L(E) \;\; iff \;\; [\![E]\!]s > 0 \\ s \notin L(E) \;\; iff \;\; [\![E]\!]s = 0 \end{array}$$

Definition 3.1 *All probability functions pf must adhere to the following two characteristics*
(Subrahmaniam 1979):

(i) for all x_i in the sample space of the experiment:

$$\sum_i pf(x_i) = 1 \tag{6}$$

(ii) For every event x_i:

$$0 \leq pf(x_i) \leq 1 \tag{7}$$

□

Consequently, if SRE expressions are to define well–formed probability functions, all expressions must similarly respect these requirements.

Theorem 3.1 *The SRE operators are well–formed probability functions.*
Proof: *The proof uses structural induction on SRE expressions. We show conditions (i) and (ii) of Defn 3.1 hold for all operators. Let $s \in \Sigma^*$.*

(a) Atomic actions: trivially.

(b) Choice: i. From equations 2 and 6:

$$\sum_i \sum_k \left(\frac{n_k}{\sum_j n_j} \cdot [\![E_k]\!] s_i \right)$$

By the induction hypothesis,

$$\sum_i [\![E_k]\!] s_i = 1.$$

Thus we have,

$$\sum_k \frac{n_k}{\sum_j n_j} = 1.$$

ii. From equation 2, the greatest value for the sum

$$\sum_k \left(\frac{n_k}{\sum_j n_j} \cdot [\![E_k]\!] s \right)$$

occurs when $[\![E_k]\!]s = 1$ *for all* k. *In this case, the sum reduces to*

$$\frac{\sum_k n_k}{\sum_j n_j} = 1$$

Equivalently, when all $[\![E_k]\!]s = 0$, *the summation is zero. And when any* $0 < [\![E_k]\!]s < 1$, *the resulting summation is a fraction between 0 and 1. Hence it is a probability.*

(c) Concatenation: i. From equation 6:

$$\sum_i [\![E_1 : E_2]\!] s_i$$

Using equation 3, because s_i *ranges over all* Σ^*, *this becomes:*

$$\sum_{i,j} [\![E_1]\!] s_i \cdot [\![E_2]\!] s_j$$

This translates to:

$$(\sum_i [\![E_1]\!] s_i) \cdot (\sum_j [\![E_2]\!] s_j)$$

By the induction hypothesis, this simplifies to:

$$1 \cdot 1 = 1.$$

ii. Given a concatenation,

$$[\![E_1 : E_2]\!] s$$

By the induction hypothesis, each of E_1 *and* E_2 *return probabilities* $0 \leq p_i \leq 1 (i = 1, 2)$. *Hence their product* $p_1 \cdot p_2$ *must likewise be a probability.*

(d) Kleene closure:

i. Starting with equation 6:

$$\sum_i [\![E^{*f}]\!] s_i$$

Using equations 4, it translates as follows:

$$\begin{aligned} &= [\![E^{*f}]\!]\epsilon + \textstyle\sum_i [\![E^{*f}]\!] s_i \quad (s_i \neq \epsilon) \\ &= (1-f) + \textstyle\sum_i f \cdot [\![E]\!] s'_i \cdot [\![E^{*f}]\!] s_i \quad (s_i \neq \epsilon) \end{aligned}$$

By the inductive hypothesis:

$$\textstyle\sum_i [\![E^{*f}]\!] s_i = (1-f) + f \cdot 1 \cdot \sum_i [\![E^{*f}]\!] s_i$$

Doing some algebraic manipulation:

$$\begin{aligned} & f \textstyle\sum_i [\![E^{*f}]\!] s_i - \sum_i [\![E^{*f}]\!] s_i = f - 1 \\ & \textstyle\sum_i [\![E^{*f}]\!] s_i (f-1) = f - 1 \\ & \textstyle\sum_i [\![E^{*f}]\!] s_i = 1 \end{aligned}$$

Note that the division by $f-1$ is permitted because $f < 1$ by definition.

ii. By induction on the length of a string s, it can be shown that

$$0 \leq [\![E^{*f}]\!] s \leq 1$$

The base case is when $s = \epsilon$, in which case the probability is $p = 1 - f$ from the first equation in 4, and $0 < p \leq 1$. For an arbitrary $s \neq \epsilon$, the probability from the second equation in 4 is:

$$\sum_{i=1}^{n} (f \cdot [\![E]\!] \alpha_1 ... \alpha_{i-1} \cdot [\![E^{*f}]\!] \alpha_i ... \alpha_n) \ + \ f \cdot [\![E]\!] s \cdot [\![E^{*f}]\!] \epsilon$$

By incorporating the second term into the first term's summation, this is rewritten:

$$\sum_{i=1}^{n} (f \cdot [\![E]\!] \alpha_1 ... \alpha_i \cdot [\![E^{*f}]\!] \alpha_{i+1} ... \alpha_n)$$

where $\alpha_m \alpha_n = \epsilon$ when $m > n$. By the inductive hypothesis over s, $\sum_i [\![E^{*f}]\!] \alpha_{i+1} ... \alpha_n$ is a probability. Furthermore, by the structural induction of expressions, $\sum_i [\![E]\!] \alpha_1 ... \alpha_i$ is a probability. Hence their product with f is a probability.

(e) +Closure: Similar to (c) and (d) above.

$\square$

3.2 Implementation of an SRE Processor

Given a regular expression, determining whether particular strings are members of its language is a tractable problem (Sipser 1996, Hopcroft and Ullman 1979). There are different ways in which this may be performed. One technique is to convert the regular expression into an equivalent nondeterministic finite automaton, which can be done in polynomial time. Once this is done, a graph–searching algorithm reads a string character by character, marking states of the FA that are still elligible as paths towards an acceptance state. An advantage of the FA approach is that the nondeterministic FA can be polynomially–time translated into a deterministic FA, which will then have more efficient recognition characteristics during language recognition.

Alternatively, regular expressions can be symbolically interpreted directly. The behaviour of each regular expression operator has a corresponding operational semantics, which can be used to define a regular expression interpreter. This may be done from the perspective of either language generation or language acceptance. One technical requirement of the expression interpretation approach is that the interpreter must be able to handle the nondeterministic nature of expression derivations, since regular expressions are naturally nondeterministic in nature. The expression interpretation is similar to the FA approach, in that there is a mapping between the states of a translated FA and the derivation paths used by the interpreter when processing an expression.

Stochastic regular language recognition uses the same basic recognition schemes as conventional regular languages, with the additional requirement that probabilities be computed for strings. For example, if a FA is derived for a stochastic language, then the links are marked with probabilities. The overall probability of accepting a given string is then computed by computing the product of all the transition probabilities used from the start state to the final accepting state. This probabilistic FA is known as a Hidden Markov Model or HMM (Rabiner and Juang 1986). Therefore, given a stochastic regular language as defined by SRE, the formulae of section 3.1 are incorporated into a translated FA or a SRE interpreter.

The SRE recognition system uses the expression interpretation approach described above. The operational semantics use two relations. One relation, $\longrightarrow$ over $E \times (\Sigma, p) \times E$, where p is a probability, represents single action transitions of expressions. This relation is denoted,

$$E \xrightarrow{(\alpha,p)} E \qquad (\alpha \in \Sigma)$$

The other relation, $\Longrightarrow$ over $E \times (\Sigma^*, p) \times E$, is the transitive closure of $\longrightarrow^*$, and models the generation of strings:

$$E \xrightarrow{(\alpha_1,p_1)} \cdots \xrightarrow{(\alpha_k,p_k)} E \;\equiv\; E \xRightarrow{(s,p_k)} E \qquad (s = \alpha_1 ... \alpha_k)$$

Figure 1 contains transitional rules for the relations, which define the structural operational semantics of the SRE operators (Hennessy 1990). These inference rules define an abstract interpreter for SRE expressions, and are the basis of an SRE recognizer. In fact, with languages such as Prolog, these rules can be compiled into Prolog statements, and then directly interpreted using Prolog's inference engine (Clocksin and Mellish 1994). Furthermore, multiple solutions are obtained for nondeterministic SRE expressions using Prolog's builtin backtracking mechanism.

The actual implementation of the SRE processor uses the above fundamental ideas. The operational semantics implemented are a superset of the rules in Figure 1. The

Action $\dfrac{}{\alpha \xrightarrow{(\alpha,1)} \epsilon}\ (\alpha \neq \epsilon)$

Choice $\dfrac{E_j \xrightarrow{(\alpha,p)} E'}{\sum E_i(n_i) \xrightarrow{(\alpha,q)} E'}\ (q = p \cdot \frac{n_j}{\sum_i n_i})$

Concat$_1$ $\dfrac{E \xrightarrow{(\alpha,p)} E'}{E : F \xrightarrow{(\alpha,p)} E' : F}$

Concat$_2$ $\dfrac{F \xrightarrow{(\alpha,p)} F'}{\epsilon : F \xrightarrow{(\alpha,p)} F'}$

Kleene$_1$ $\dfrac{}{E^{*f} \xrightarrow{(\epsilon,1-f)} \epsilon}$

Kleene$_2$ $\dfrac{E \xrightarrow{(\alpha,p)} E'}{E^{*f} \xrightarrow{(\alpha,p\cdot f)} E' : E^{*f}}\ (\alpha \neq \epsilon)$

+Closure $\dfrac{E \xrightarrow{(\alpha,p)} E'}{E^{+f} \xrightarrow{(\alpha,p)} E' : E^{*f}}\ (\alpha \neq \epsilon)$

Strings$_1$ $\dfrac{E \xrightarrow{(\alpha,p)} \epsilon}{E \xRightarrow{(\alpha,p)} \epsilon}$

Strings$_2$ $\dfrac{E \xrightarrow{(\alpha,p)} E' \qquad E' \xRightarrow{(s,q)} E''}{E \xRightarrow{(\alpha s,pq)} E''}$

Figure 1: Transitional semantics of SRE

implementation uses a logical grammar definition of SRE, which is part of the DCTG-GP system (Ross 1999) (see Section 4). Prolog's backtracking is advantageously used to investigate different paths of an expression's derivation. In addition, string recognition is performed by pattern matching on an argument string and the generated string as shown in the transitional semantics: when a match occurs, the current derivation path is correct, while mismatches cause the current derivation to backtrack and test another possible nondeterministic path. For example, one instance of backtracking may try different terms in a Choice expression, while another may unwrap an iterative expression a varying number of times. Such backtracking is assured of terminating because of the finite size of input strings to be checked, as well as the assertion within the SRE semantics that empty strings ϵ can never be generated within the generative component of iterative operators (they can only be generated when the iteration terminates).

One advantage of a stochastic language is that the computed probabilities of strings can be used as an efficiency mechanism during expression recognition. The implementation of the SRE recognizer is such that the probability of intermediate strings are always known throughout the interpreter. When the current probability becomes smaller than a user–supplied threshold, the current derivation path can be forced to terminate. This prunes derivations of an expression which yield probabilities too small to be of consequence. Of course, setting this threshold too large results in inaccurate probability values for recognized strings, and may even erroneously reject legal strings. However, for many experiments, especially with large strings to be recognized, this speeds up processing significantly.

4 GENETIC PROGRAMMING SYSTEM

4.1 Grammatical SRE and gSRE

The GP system used for the SRE experiments is the DCTG-GP system (Ross 1999). DCTG-GP performs grammar–based genetic programming, in which the target language of the evolved program population is defined in terms of a context–free grammar (Lucas 1994, Whigham 1995, Wong and Leung 1995, Geyer-Shulz 1997). A major advantage of grammatical GP systems is that the search space is syntactically constrained so that evolution is given a helpful push towards program structures that are more sensible for the problem at hand. The grammar used by DCTG-GP is a definite clause translation grammar, or DCTG (Abramson and Dahl 1989). A DCTG is a logical version of a context-free attribute grammar. Each DCTG production has a syntactic component, which defines a context–free syntax rule. In addition, each production can have included with it one or more semantic components. A semantic component defines some characteristic of syntactic component to which it is attached. For example, one important SRE characteristic that is defined in the DCTG grammar is the string recognition algorithm of Section 3.2. Since the operational semantics of the SRE operators are very modular in nature, their recognition behaviours can be encoded with the grammar rules that define the syntax of the operators themselves. The overall result of this is a compact definition of the SRE language, in which the syntax and semantics are conveniently unified together.

One syntactic constraint applied to both SRE and gSRE in the experiments in section 5 is the following. Although not specified in the grammar of SRE (or gSRE), the grammatical definition of SRE disallows iterative operators to be directly nested within one another. In other words, expressions such as

$$((E)^{*f})^{*f} \qquad or \qquad ((((E)^{*f})^{+f})^{*f})^{+f}$$

are not allowed. The reason for this restriction is a pragmatic one. When GP was performed without this restriction, many programs had multiply nested iterative expressions. Such expressions are relatively expensive to interpret, due to the variety of nondeterministic paths possible for interpreting them. In addition, nested iteration typically results in strings with very low probabilities, since there is a probabilistic factor f associated with executing every nested iterative expression. Moreover, the expense of nested iteration is not justified by results, since any of these expressions can be replaced with a semantically equivalent expression that uses only one iterative operator. This restriction does *not* imply that an expression like

$$((a : E^{*f})^{*f}$$

is illegal, since the concatenation operator means that the iteration operators are not directly nested.

gSRE is also encoded as a syntactic constraint of SRE. Rather than permitting any SRE expression as a term within a choice operator, only uniquely guarded terms are permitted.

4.2 Other GP System Details

DCTG-GP uses standard GP strategies, such as tournament or roulette–wheel selection, and steady–state or generational evolution. Relevant experimental parameters will be illustrated in section 5. The system is implemented in Sicstus Prolog 3 on both Windows 98 and Silicon Graphics platforms.

5 EXPERIMENTS

5.1 General Strategy

The inference of a stochastic language can be considered to involve two different objectives. Given a training set of positive (and possibly negative) examples, one task is to infer a language which correctly classifies the training examples. This is equivalent to non–stochastic language inference. An additional task required for stochastic language inference, however, is to ascertain the stochastic distribution of the training examples. One might naively presume that a statistical analysis of the training set could be performed, and the results applied to the inferred language. Unfortunately, the situation is typically more complicated than this, because the representation of the stochastic language as used in the hypothesis will not likely permit a straight–forward application of the final string distributions to its internal encoding. For example, if an HMM representation is used in hypotheses, finding appropriate probability values for intermediate links in the network that will correspond to the example set distribution is a challenging task. The significance of the problem of determining distributions for HMM's and context–free languages has spawned specialized training algorithms (Lari and Young 1990, Charniak 1993).

The inference strategy undertaken with the GP experiments is to let evolution determine stochastic distributions in concert with example classification. Since SRE incorporates probability values directly in expressions, treating numeric probability fields is straight–forward in GP. It was found that this approach was sufficient for many experiments undertaken. In fact, it was discovered that evolution using local search for fine–tuning probability parameters lent no advantage over simple evolution of the parameters.

The training sets used in the GP experiments consist of sets of positive examples for the target language to be inferred. Each member of the set is a string, along with its frequency with respect to the total number of strings in the set (typically 1000). Since the format of the target languages is already known via a stochastic regular expression or grammar, generating these sets is straight–forward. Unlike conventional language inference, the implicit probability distributions in training example sets permits stochastic languages to forgo the need for negative examples. This is because the inference of a distribution that matches that of the training set will automatically account for 'negative examples', which have 0 probability in the distribution.

Stochastic language inference incorporates an implicit degree of error in any inferred solution. This has ramifications on the GP fitness evaluation described below. It also can be used to boost efficiency of computations performed during inference. As detailed in section 3.2, string recognition can be pre-empted when intermediate probabilities become smaller than some threshold limit set for the experiment. Similarly, the test set can be pruned of strings whose frequency is below some limit set by the user. This limit parameter should be set with the recognition threshold in mind. For example, if the threshold is set to 0.001, then the test set limit could be likewise set to 1 for a test set of size 1000. Of course, there may be many nondeterministic derivations of an expression when recognizing a string, and all the probababilities of these derivations will be summed to an overall probability for that string. The less discriminating the recognition threshold and test set limit, the more precise (albeit slower) the results.

Since GP experiments use a steady–state algorithm, there are not any discrete generations. For convenience, however, a new generation is said to have occurred every K reproductions, where K is the population size. Between generations, the test set is regenerated. This prevents overfitting to one set of test data, and reflects the nature of the stochastic languages, as each test set reflects a sampling from the actual distribution. One disadvantage, however, is that a discrete test set is an approximation of the real distribution of the language, and hence this introduces an unavoidable measure of noise. This noise is compensated by the fact that multiple test sets are used during successive generations, and their cumulative effect should reflect a more accurate model of the target distribution. However, the population is not reevaluated for each newly generated test set, and so the fitnesses of much of the population may be legacy values from earlier generations. This is acceptable, because the test sets used for those generations are presumed to be as statistically valid as those from any other generation.

The fitness evalution strategy used in the experiments is a modified χ^2 test (Press *et al.* 1992). The known distribution is taken to be the set T of test examples, and the experimental set will be the results of the SRE recognition algorithm on each member $t_i \in T$. Each test set example string is given to the SRE processor, and an overall probility $Pr(t_i)$ for that string is computed. Non–membership is reflected in a probability of 0. The fitness formula is:

$$Fitness = \sum_{t_i \in T} \begin{cases} \frac{(d_i - (Pr(t_i) * N))^2}{d_i} & : Pr(t_i) \geq 0 \\ \frac{|t_i| - |maxpref_i|}{|t_i|} \cdot d_i & : Pr(t_i) = 0 \end{cases}$$

where d_i is the frequency of example t_i in test set T, $N = |T|$, and $maxpref_i$ is the maximum prefix of t_i recognized. The first term is the χ^2 formula, and it is used when the example string t_i is completely recognized. The second formula is used when only a prefix of t_i is recognized. Its value is inversely proportional to the size of the prefix recognized. Should none of t_i be recognized, then this value becomes $2 \cdot d_i$ (a normal χ^2 formula would use just d_i). This prefix scoring gives credit to expressions that recognize portions of the examples, which helps drive evolution towards expressions that recognize complete examples.

5.2 Experiment 1: Stochastic Iteration

The first experiment uses a simple stochastic regular language which can be naturally encoded in SRE. The main intention of this experiment is to test the evolvability of stochastic Kleene closure as modeled in SRE. The target language is a stochastic rendition of a regular language suggested by Tomita[1] from his popular benchmarks for machine learning (Tomita 1982). The target language written in SRE is:

$$L_1 \quad = \quad a^{*.5} : b^{*.5} : a^{*.5} : b^{*.5}$$

This is a non–trivial language, especially in the stochastic domain, as the overall distribution of each a and b term in all the strings should conform to the given probability of 0.5. These terms may also generate empty strings, should iterations terminate immediately.

The parameters for the experiment are in Figure 1. Most are self–explanatory, and the fitness function strategy was discussed earlier in Section 5.1. The initial population is oversampled, and the running population is pruned from it using tournament selection. Replacement is done using a reverse tournament selection (a sample of K members are randomly selected, and the member with the lowest fitness is selected to be replaced).

[1]His language is $a^*b^*a^*b^*$.

Table 1: Parameters (L_1)

Parameter	Value
Target language	gSRE
Terminals	a, b
Fitness function	modified χ^2
Generation type	steady-state
Initial population size	750
Running population size	500
Unique population members	yes
Maximum generations	50
Probability of crossover	0.90
Probability of mutation	0.10
Probability internal crossover	0.90
Probability terminal mutation	0.75
Probability numeric mutation	0.50
Numeric mutation range	± 0.1
Max reproduction attempts	3
Initial population shape	ramped half&half
Min/max depth initial popn.	6, 12
Max depth offspring	24
Tournament size	5
Test set size	1000
Max test string size	approx. 20
Min test example frequency	3
SRE probability limit	0.0001

Mutation is performed on either terminal or nonterminals. If a nonterminal is to be mutated, there is a 0.5 probability that it should be a numeric field. When a numeric field is selected for mutation, its current value is perturbed $\pm 10\%$ of the entire range for that numeric type (a range of ± 100 for integers, and ± 0.1 for probabilities).

A test set is generated before every generation. Initially, 1000 strings are generated for L_1, and their frequencies are tallied. The maximum string size is approximately 20 (some may exceed this length). Should there be less than 3 instances of a given string, it is pruned from the test set. This means that there are typically between 55 to 60 unique strings in the test set, each of which has its particular frequency for that particular sample of the language. The number of unique strings in the test set is important for χ^2 analyses, as it is equivalent to bin size in the χ^2 formula.

Table 2: Summary L_1

Total runs		15
# unique examples		65
Avg. test set χ^2 (50 cases)		142.22
Fitness	min	89.4 (χ^2=88.8)
	max	251.55 (χ^2=203.75)
	avg	127.91 (χ^2=124.27)

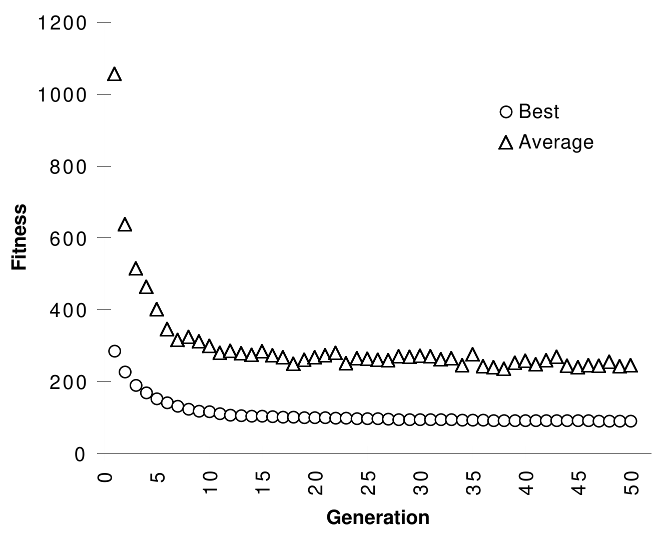

Figure 2: Fitness curves (avg 15 runs)

Summary statistics for the best solutions from 15 runs are given in Figure 2. These values are obtained using a common test set, since each run will have used a different test set during its prior evolution. An average χ^2 test of the test sets themselves is included, in order to better evaluate the expression results. 50 pairs of random test sets were generated. One of the pair was fixed as the independent variable, while the other was the dependent variable. The sets were filtered for frequencies below the minimal test example frequency (2 in Figure 1), and the χ^2 was computed. The resulting 50 χ^2 values were averaged.

A performance chart of the best and average population fitness averaged for 15 runs is in Figure 2. It can be seen that convergence to a local optimum has largely occured by generation 10.

The best solution found ($\chi^2 = 88.8$), is:

$$a^{*.5} : b^{*.44} : a^{.49} : b^{.49}$$

This is a nearly perfect solution, and the iterative probabilities within the range of what might be expected given the stochastic error inherent with the random test sets. The second best solution (χ^2=89.63) is:

$$a^{*.52} : b^{*.49} : a^{.37} : (a(94) + b(759))^{*}.54$$

The last term is interesting, in that the erroneous choice of a is not too acute a problem, given the low probability of choosing it (0.11).

One of the poorer solutions (χ^2=132.85) is:

$$(a(468) + b(235))^{*.25} : a^{*.55} : (a(182) + b(963))^{*.73}$$

The inaccuracy occurs with the first term, which erroneously permits b to occur too frequently, even though the low probability of 0.25 for the enclosing iteration helps reduce its likelihood.

The worst solution obtained (χ^2=203.75) is:

$$\begin{aligned}&(a^{*.58} : (a(362) + b(805))^{+.67})^{+.04}\\ &\quad : (b^{+.2} : b^{+.59} : ((a : (a(364) + b(320))^{*.08}(320)\\ &\qquad +b(947) : (a(191) + b(141))^{+.67})^{+.2}\\ &\qquad : ((a(352) + b(360))^{+.56} : b^{+.59})^{*.04})^{*.04})^{*.04}\end{aligned}$$

Note the repetition of particular numeric fields, such as 320 and 0.04, which is a sign of population convergence. Simplifying this expression by removing iterative probabilities less than 0.10 and expanding +Closure terms, it becomes:

$$a^{*.58} : (a(362) + b(805))^{+.67}$$

which is obviously a suboptimal solution. This example shows the nature of introns within SRE expressions: virtually any expression can be intron code, so long as the associated choice or iterative probability is low enough.

$$\begin{array}{lll} S \rightarrow a\ S & & (0.2) \\ S \rightarrow b\ A & & (0.8) \\ A \rightarrow a\ B & & (0.7) \\ A \rightarrow b\ S & & (0.3) \\ B \rightarrow a\ A & & (0.4) \\ B \rightarrow b\ B & & (0.1) \\ B \rightarrow \epsilon & & (0.5) \end{array}$$

Figure 3: Target language

5.3 Experiment 2: Stochastic Regular Grammar

The second experiment evolves a more complex stochastic regular language. The target language L_2 is taken from (Carrasco and Forcada 1996), and is defined by the stochastic regular grammar in Figure 3. Each production has a probability on the right, which denotes the probability that rule is selected with respect to the other productions for that nonterminal.

Table 3: Summary L_2

Total runs		50
# unique examples		35
Avg. test set χ^2 (50 cases)		99.75
Fitness	min	66.39 (χ^2=65.06)
	max	281.93 (χ^2=274.1)
	avg	199.5 (χ^2=193.26)

The experimental parameters for these runs are identical to those in Figure 1. The summary for 50 runs are in Figure 3. A performance plot for the best fitness and average population fitness averaged for the 50 runs is in Figure 4.

The best solution ($\chi^2 = 65.06$) is:

$$\begin{gathered} ((a : (a(674) + b(895))^{*.03})(895) + b : b(960))^{*0.37} \\ : (b : (a : (a)^{+.02}(674) + b : b(284))^{*.29} \\ : (a : a^{+.02}(895) + b : b(284))^{*.29} \\ : b : (a(490) + b(524))^{*.03})^{*.09} \\ : b : a^{+.02} : (a : a^{+.02}(895) + b(284))^{*.37} \end{gathered}$$

Simplifying by expanding +Closures and removing terms with probabilities less than 0.03,

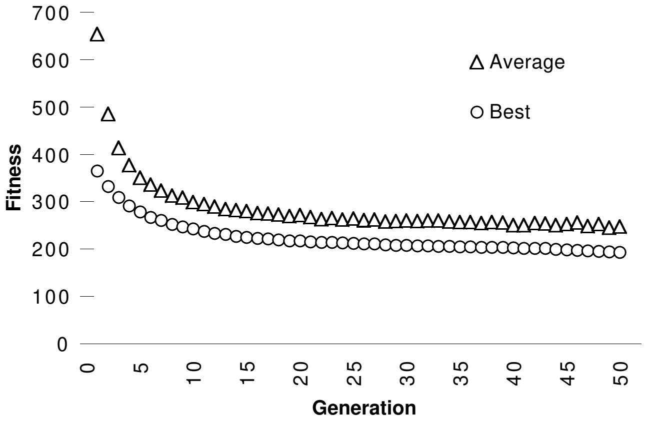

Figure 4: Fitness curves (avg 50 runs)

this becomes:

$$
\begin{aligned}
&(a : (a(674) + b(895))(895) + b : b(960))^{*0.37} \\
&\quad : (b : (a : a(674) + b : b(284))^{*.29} \\
&\qquad : (a : a(895) + b : b(284))^{*.29} : b)^{*.09} \\
&\quad : b : a : (a : a(895) + b(284))^{*.37}
\end{aligned}
$$

It is difficult to see how this expression maps to the target grammar of Figure 3, and an intuitive mapping may not even exist. However, its χ^2 is impressive compared to the test set average.

5.4 Limitations

Many language inference algorithms are easily thwarted by target languages having characteristics antagonistic to the peculiarities of the algorithm in question. Often, these languages are only subtlely different from ones that the algorithms have no problems inferring.

The GP paradigm suffers a similar limitation. A variation of the language in section 5.3 was tried,

$$L_2' = L_2(9) + bbaaabab(1)$$

which is just language L_2 with an additional string *bbaaabab* with probability 10%. 50 runs were performed using the same parameters as figure 1. None of the runs found an acceptably close solution: the best solution had a fitness of 259 and $\chi^2 = 234$ (bin size = 40).

One reason that GP had problems evolving L_2' can be attributed to the linguistic characteristics of SRE. Even though the above definition of L_2' is a concise statement of the language, the evolutionary process tries to unify the term *bbaaabab* and L_2 together in a regular expression. This is difficult to do, because this string is an anomaly with respect to the other strings in L_2. Considering the stochastic regular grammar used to generate L_2, it is clear that strings are derived progressively and incrementally from one another, and so strings of L_2 equal in length to *bbaaabab* are natural extensions of smaller strings of the language. The anomalous string, however, is not derivable from L_2, and hence a natural model of the union of these languages in SRE cannot be inferred. This is especially true given that *bbaaabab* has a 10% probability, which makes it a populous member. If it had a smaller probability, it might be ignored as noise.

The above must be considered in light of the linguistic nature of all formal languages: some representations more naturally model particular languages than others. Even though regular expressions, finite automata and regular grammars have the same expressive power, some languages are more naturally and concisely denoted by regular expressions than by finite automata, and vice versa. It could be the case that another representation language, for example HMM's, may more naturally denote L_2' than SRE.

6 CONCLUSION

This paper presented a new means for evolving stochastic regular languages. Using a probabilistic version of regular expressions as a language for evolution, genetic programming is capable of evolving accurate expressions for stochastic regular languages. However, some stochastic regular languages are more amenable to successful evolution than others. It can be speculated that languages in which members have structural similarities with one another are the most suitable for this paradigm. For more complex languages, more sophisticated evolutionary techniques may be required.

It was found during experimentation that SRE had no evolutionary advantage over gSRE with respect to the quality of solutions discovered. On the other hand, SRE expressions were less efficient to process, and runs took much longer than the gSRE ones.

The use of SRE in a genetic programming context presents advantages over other evolutionary experiments with stochastic languages. One advantage is that SRE is akin to a programming language, with operators that have syntactic and semantic definitions akin to conventional languages. Since GP is typically applied towards such languages as Lisp, the encoding and processing of SRE within a GP environment is straight–forward. More importantly, however, is that SRE has linguistic advantages over finite automata and regular grammars: some stochastic languages are more naturally encoded in SRE than these other representations. The L_1 experiment is a clear example of this point. The linguistic clarity of L_2 is less apparent, although the solution is not overly complex compared to the target grammar.

Like (Svingen 1998), this work uses a regular expression language directly for GP. His work required fairly large populations and parallel populations in order to evolve the Tomita languages. The fitness strategy used here is similar to that used in (Lankhorst 1994, Schwehm and Ost 1995), in that both language recognition performance and prefix consumption are taken into consideration.

There are many directions for future work. The GP strategies used here were fairly conventional, and more sophisticated approaches may be more applicable to stochastic languages. In the experiments, the wide degree of qualitative variations between runs indicates that evolution quickly gets stuck at suboptimal solutions. Parallel subpopulations may help in this regard. Although it was found that local search using hill–climbing over numeric fields was not advantageous to evolution, it is worth investigating the utility of more sophisticated local search techniques akin to those used in stochastic context–free languages (eg. the inside–outside algorithm).

Currently, the applicability of SRE in bioinformatics problems is being investigated. A fundamental problem in DNA and protein sequencing is to determine a common pattern shared amongst a family of sequences (Brazma *et al.* 1995), which can be used for both search and analytical purposes. A number of techniques, such as HMM's and regular pattern languages, are used for this purpose. SRE is a natural vehicle for this problem area, since its regular expression basis conforms to the pattern languages commonly used (eg. that used in the PROSITE database (Hofmann *et al.* 1999)), while its stochastic features conveniently model the probabilistic characteristics of DNA sequences themselves.

Acknowledgement: Thanks to Tom Jenkyns for helpful discussions about probability. This research is supported by NSERC Operating Grant 138467-1998.

References

Abramson, H. and V. Dahl (1989). *Logic grammars*. Springer-Verlag.

Angluin, D. (1992). Computational Learning Theory: Survey and Selected Bibliography. In: *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*. ACM Press. pp. 351–369.

Brave, S. (1997). Evolving deterministic finite automata using cellular encoding. In: *Proc. Genetic Programming 1997* (John R. Koza *et al*, Ed.). Morgan Kaufmann. Stanford University, CA, USA. pp. 39–44.

Brazma, A., I. Jonassen, I. Eidhammer and D. Gilbert (1995). Approaches to the automatic discovery of patterns in biosequences. Technical Report 113. Dept. of Informatics, University of Bergen, Norway.

Carrasco, R.C. and J. Oncina (1998). Learning Deterministic Regular Grammars from Stochastic Samples in Polynomial Time. Technical Report DLSI-96-01. Universidad de Alicante.

Carrasco, R.C. and M.L. Forcada (1996). Inferring Stochastic Regular Grammars with Recurrent Neural Networks. In: *Proceedings 3rd International Colloqium on Grammatical Inference (ICGI96)* (I. Miclet and C. de la Huguera, Eds.). Springer Verlag. pp. 274–286. LNAI 1147.

Charniak, E. (1993). *Statistical Language Learning*. MIT Press.

Clocksin, W.F. and C.S. Mellish (1994). *Programming in Prolog (4th ed)*. Springer-Verlag.

Dunay, B.D. and F.E. Petry (1995). Solving Complex Problems with Genetic Algorithms. In: *Proc. 6th Intl. Conf. on Genetic Algorithms* (L. Eshelman, Ed.). Morgan Kaufmann.

Dunay, B.D., F.E. Petry and B.P. Buckles (1994). Regular Language Induction with Genetic Programming. In: *Proc. 1st IEEE Conference on Evolutionary Computation*. pp. 396–400.

Dupont, P. (1994). Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG method. In: *2nd Intl. Coll. on Grammatical Inference and Applications*. Springer-Verlag. pp. 236–245.

Fu, K.S. (1982). *Syntactic Pattern Recognition and Applications*. Prentice-Hall.

Fu, K.S. and T. Huang (1972). Stochastic Grammars and Languages. *International Journal of Computer and Information Sciences* **1**(2), 135–170.

Fu, K.S. and T.L. Booth (1975*a*). Grammatical Inference: Introduction and Survey – Part I. *IEEE Transactions on Systems, Man, and Cybernetics* **5**(1), 95–111.

Fu, K.S. and T.L. Booth (1975*b*). Grammatical Inference: Introduction and Survey – Part II. *IEEE Transactions on Systems, Man, and Cybernetics* **5**(4), 409–423.

Garg, V.K., R. Kumar and S.I Marcus (1996). Probabilistic Language Framework for Stochastic Discrete Event Systems. Technical Report 96-18. Institute for Systems Research. http://www.isr.umc.edu/.

Geyer-Shulz, A. (1997). The Next 700 Programming Languages for Genetic Programming. In: *Proc. Genetic Programming 1997* (John R. Koza *et al*, Ed.). Morgan Kaufmann. Stanford University, CA, USA. pp. 128–136.

Hennessy, M. (1990). *The Semantics of Programming Languages – An Elementary Introduction Using Structural Operational Semantics*. John Wiley and Sons.

Hofmann, K., P. Bucher, L. Falquet and A. Bairoch (1999). The PROSITE database, its status in 1999. *Nucleic Acids Research* **27**(1), 215–219.

Hopcroft, J.E. and J.D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley.

Kammeyer, T.E. and R.K. Belew (1997). Stochastic Context-free Grammar Induction with a Genetic Algorithm Using Local Search. In: *Foundations of Genetic Algorithms IV* (R.K. Belew and M. Vode, Eds.). Morgan-Kaufmann.

Lankhorst, M.M. (1994). Grammatical Inference with a Genetic Algorithm. *Proceedings of the 1994 EUROSIM Conference on Massively parallel Processing Applications and Development* pp. 423–430.

Lankhorst, M.M. (1995). A Genetic Algorithm for the Induction of Pushdown Automata. In: *1995 IEEE Intl. Conference on Evolutionary Computation*. IEEE Press.

Lari, K. and S.J. Young (1990). The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language* **4**, 35–56.

Longshaw, T. (1997). Evolutionary learning of large grammars. In: *Proc. Genetic Programming 1997* (J.R. Koza *et al*, Ed.). Morgan Kaufmann. Stanford University, CA, USA. pp. 406–409.

Lucas, S. (1994). Structuring chromosomes for context-free grammar evolution. In: *Proceedings 1st International Conference on Evolutionary Computation*. pp. 130–135.

Maryanski, F.J. and T.L. Booth (1977). Inference of Finite-State Probabilistic Grammars. *IEEE Transactions on Computers* **C26**, 521–536.

Press, W.H., S.A. Teukolsky, W.T. Vetterling and B.P. Flannery (1992). *Numerical Recipes in C*. 2 ed.. Cambridge University Press.

Rabiner, L.R. (1989). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE* **77**(2), 257–286.

Rabiner, L.R. and B.H. Juang (1986). An Introduction to Hidden Markov Models. *IEEE ASSP Magazine* pp. 4–16.

Ross, B.J. (1999). Logic-based Genetic Programming with Definite Clause Translation Grammars. In: *Proc. GECCO-99* (J.R. Koza *et al*, Ed.).

Sakakibara, Y. (1997). Recent Advances of Grammatical Inference. *Theoretical Computer Science* **185**, 15–45.

Schwehm, M. and A. Ost (1995). Inference of Stochastic Regular Grammars by Massively Parallel Genetic Algorithms. In: *Proc. 6th Intl. Conf. on Genetic Algorithms*. Morgan-Kaufmann.

Sen, S. and J. Janakiraman (1992). Learning to construct pushdown automata for accepting deterministic context-free languages. *Applications of Artificial Intelligence X: Knowledge-Based Systems* **1707**, 207–213.

Sipser, M. (1996). *Introduction to the Theory of Computation*. PWS Pub. Co.

Stoy, J. (1977). *Denotational Semantics*. MIT Press.

Subrahmaniam, K. (1979). *A Primer in Probability*. Marcel Dekker, Inc.

Svingen, B. (1998). Learning Regular Languages Using Genetic Programming. In: *Proc. Genetic Programming 1998* (J.R. Koza *et al*, Ed.). Morgan Kaufmann. pp. 374–376.

Tomita, M. (1982). Dynamic construction of finite automata from examples using hill-climbing. In: *4th Annual Conf. of the Cognitive Science Soc.*. pp. 105–108.

van der Mude, A. and A. Walker (1978). On the Inference of Stochastic Regular Grammars. *Information and Control* **38**, 310–329.

Whigham, P.A. (1995). Grammatically-based Genetic Programming. In: *Proceedings Workshop on Genetic Programming: From Theory to Real-World Applications* (J.P. Rosca, Ed.). pp. 31–41.

Wong, M.L. and K.S. Leung (1995). Learning Programs in Different Paradigms using Genetic Programming. In: *Proceedings 4th Congress of the Italian Association for AI*. pp. 353–364.

Wyard, P. (1991). Context Free Grammar Induction Using Genetic Algorithms. In: *Proceedings 4th International Conference on Genetic Algorithms*. pp. 514–518.

Zhou, H. and J.J. Grefenstette (1986). Induction of Finite Automata by Genetic Algorithms. In: *Proc. 1986 IEEE Intl. Conference on Systems, Man, and Cybernetics*. IEEE Press. Atlanta, GA. pp. 170–174.