UNIVERSITY OF CALIFORNIA, SAN DIEGO

Evolving Stochastic Grammars

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Thomas E. Kammeyer

Committee in charge:

Professor Richard K. Belew, Chairperson
Professor Walter Savitch
Professor Charles Elkan
Professor Eric Mjolsness
Professor Doug Smith
Professor Michael Gribskov

1998

The dissertation of Thomas E. Kammeyer is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1998

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I thank my advisor, Rik Belew, for all of the years of support and advice, and the occasional body-check in the right direction when I went stubbornly astray. I thank him also for being enthusiastic about my work and for being demanding, and most of all for being part of a research community so that I could join that community through him. I would also like to thank Dr. Michael Baker, who not only provided the data used in the work on grammar induction on proteins, but also spent much time with me explaining biological details and interpreting the output of runs.

I would also like to thank Chris, Mark, and Dan for all of those sessions of Marathon and backgammon, and everyone in the Cognitive Computer Science Research Group for all of their input and constructive criticism over the years. My thanks also to the Human Genome Grant and the Powell Foundation for their generous financial support.

Finally, it is impossible to mention support without mentioning the support I've received from my family over the past 30 years, and the encouragement I've received from my my mother and Robert and my brother and sister, John and Beth.

This thesis is dedicated to my mother, for making me what I am, and to Angela, for showing me what I want to be.

VITA

December 27, 1967	Born, Fort Wayne, Indiana
1991	B.A. University of California, San Diego Majors: Computer Science, Philosophy Minor: Psychology
1991–1996	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego (for a total of 11 quarters during these years)
1992	M.S., University of California, San Diego Computer Science and Engineering, September 1992 Project: Evolving Grammars for Sorting Networks
1992–1993	Research Assistant, University of California, San Diego Computer Science and Engineering Professor Richard K. Belew
1998	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

"An Empirical Investigation of an Evolutionary Algorithm's Ability to Maintain a Known Good Solution." Evolutionary Programming VII, In press, 1998.

"Stochastic Context-Free Grammar Induction with a Genetic Algorithm Using Local Search." Foundations of Genetic Algorithms 4, pp. 409–436, 1997.

"Evolving Compare-exchange Networks Using Grammars." Artificial Life, v2n2, pp. 199–237, 1995.

"The Role of Development in Genetic Algorithms." Foundations of Genetic Algorithms 3, pp. 315-332, 1995.

"Learning Sorting Networks By Grammars." Proceedings of the Twelfth National Conference on Artificial Intelligence, Abstract, 1994.

"Evolving aesthetic sorting networks using developmental grammars." Proceedings of the Fifth International Conference on Genetic Algorithms, p. 629 (one-page summary, poster), 1993.

ABSTRACT OF THE DISSERTATION

Evolving Stochastic Grammars

by

Thomas E. Kammeyer
Doctor of Philosophy in Engineering Sciences (Computer Science)
University of California, San Diego, 1998
Professor Richard K. Belew, Chair

Evolutionary algorithms (EAs) are a class of machine learning algorithms which, along with neural networks and simulated annealing, reflect a theme that has become popular in machine learning: the pursuit of algorithms that work because they mimic natural processes. In the case of EAs, a population of individual solutions evolves to solve a machine learning problem. Here, the simulated evolution of a population of stochastic phrase-structure grammars and how derivations from those grammars play a role analogous to biological ontogenesis are explored.

An overview of evolutionary algorithms is presented and a simple EA is discussed in detail, taking note of some extensions important for the purposes of this thesis. The problem of representing definitions of functions on EA genotypes, and representing grammars as a special case of this problem, are discussed. A review of grammar induction provides background and context.

Two EAs to evolve grammars are described, using distinct methods to evaluate grammars' fitnesses. The first EA evolves grammars which derive descriptions of merging networks, and tests these simple merging algorithms for correctness in order to evaluate grammar fitness. The second EA, "EvoGrams," evaluates a grammar by parsing each string in a training set and evaluating the probability of the data given the grammar using these parses. A mechanism to give partial credit to grammars is important in assigning fitness when no complete parses are possible, and a local search method plays an important role in evaluation as well.

The best evolved merging networks from the first EA are tested for the property of being log-sequential sorting networks. A previously unknown sorting network is discovered that combines parts of the recursive definitions from two known networks. EvoGrams is successfully applied to induce grammars for several formal languages, compared with other algorithms, and experiments are presented which address important issues related to performing grammar induction with it. Finally, an attempt to use EvoGrams to perform grammar induction on data from the primary sequences of proteins is presented and analyzed, and further work and improvements suggested by the results are discussed.

Chapter I

Introduction

This thesis focuses on the use of stochastic and nonstochastic phrase-structure grammars as genotypic representations in evolutionary algorithms. Evolutionary algorithms (EAs) are a class of machine learning algorithms which, along with neural (or "connectionist") networks and simulated annealing, reflect a theme that has become popular in machine learning: the pursuit of algorithms that work because they mimic natural processes. In such approaches to designing machine learning algorithms, the practice is to pursue a loose but meaningful metaphorical relationship between what the algorithm does and what nature does. The algorithm is intended as an abstraction of a natural process that preserves enough relevant details to support an argument by analogy: the natural process is observed to accomplish something we've identified as analogous to the goal of a learning algorithm, the algorithm contains analogs to the relevant primitives and operations of the natural process, and applies the latter to the former in a manner analogous to the natural process. If this is all really true, then a standard argument by analogy says that the algorithm will accomplish its goal with the same reliability as the natural process.

EAs have been studied as a cohesive field since the middle to late 1970s in many different forms[46, 10, 56]. In this case, the natural process is evolution and the algorithm's goal is the discovery of highly-fit individuals according to some measure of fitness. The basic operations generally included are mutation (almost always) and crossover (often but not always) on individual genotypes, selection of highly fit individuals, and fitness evaluation. The primitives are sequences of characters representing individuals' genomes, which are organized into a population. Time is discretized into "generations;" each generation, the whole population undergoes fitness evaluation, selection chooses individuals for reproduction with a bias toward individuals having higher fitness, and offspring are generated from individuals or pairs of them using mutation or crossover, or both. Offspring are generated until a new population is filled up, usually using a fixed population size, and the new population is the input to the next generation. This is a simple version of an EA, which admits of many additions and variations.

An important distinction that arose early in evolutionary algorithms research is the distinction between "complex" and "simple" representations. Later, this topic will be discussed in detail; here, we can move the discussion forward using examples alone. A good example of a simple representation is a list of parameters to be passed to the fitness function. If the fitness function is a function of five variables, values for these variables can be specified in the genome of each individual. Two good examples of complex representations come from the genetic programming paradigm, programs encoded as LISP s-expressions, and classifier systems, evolving rule sets that implement production systems[1]. In both complex representations, fitness evaluations require some process of interpretation: running a program on some set of inputs, or invoking a series of rules to reach a goal state. In both cases, this is more complicated than simply plugging parameter values into a formula, and introduces a procedure that must be carried out using the information on the genotype before an individuals fitness can be evaluated.

An important recent development associated with increasingly complex genotypic representations in EAs has been a growing concern with analogies to the "life histories" of individuals in natural populations. This manifests as a concern with issues such as that of the interaction between an EA and learning on the part of indi-

[1]Human-designed, "knowledge-engineered" productions systems built using knowledge elicited from experts are familiar to many people now as expert systems.

viduals prior to fitness evaluation, such as running backpropagation to tune a neural network whose initial weights are genotypically specified ([72, 73]). In the same vein of operations carried out on or using an individual's genotype prior to fitness evaluation is the idea of including a model of ontogeny, morphogenesis, or embryonic growth in an EA, viewed as a mapping from genotypes to phenotypes([54, 40, 14, 43]) Sometimes, these two categories are difficult to distinguish and the choice of how to describe a given algorithm is based on which description is clearer. For example, we might view a classifier system as a growing set of facts added by rules (so the final set of facts when we stop executing rules is the phenotype) and use the metaphor of embryonic growth, or we might view the same system as learning over time using its rules (so the rules and initial facts are the initial phenotype and facts are modified by learning or "individual adaptation"). Depending on our exact interests (the final set of facts or the rules by which we arrive at them), we might adopt either perspective. Often, a given EA will contain elements of both growth and learning, and the border between the two may be difficult to draw[66, 64].

Including either of the above two forms of individual change in an EA implies an extension of the basic analogy between natural evolution and evolutionary algorithms. The modification is the inclusion of something that fulfills the role of natural biological "development" in the context of the algorithm. This allows for complex or nondeterministic mappings from genotypes to phenotypes and possible modification of the phenotype before finally determining an individual's fitness.

In this thesis, the primary focus is on an EA in which the genome is a phrase structure grammar, often a stochastic grammar. The emphasis here is on using such an EA as a tool to solve machine learning problems, rather than using it to suggest conclusions about natural biological systems. In one application, the grammars on the genotype are used as "ontogenetic programs" to derive strings that then undergo fitness evaluation. This is very much in the tradition of Lindenmayer systems, or "L-systems" when they are used in conjunction with EAs[62]. In other applications discussed herein, the ontogenetic analogy is less immediate, but it will be argued that these cases admit the same analogy in another form. These other cases resemble classical grammar induction problems, some study of which using genetic algorithms has taken place in the past[82, 83, 74, 58].

In the case of stochastic grammars, there is a parameter tuning algorithm reminiscent in its role for grammars of backpropagation's role in the evolution of neural networks[19, 72]. This algorithm, the "inside-outside algorithm" updates the probabilities in a stochastic context-free grammar to make some training data more probable given the grammar. If we view the grammar as a phenotype decoded from a genetic representation, as we will below, then a series of probability changes based on training data plays a role analogous to learning.

I.A Evolving grammars

In order to apply an EA to the problem of evolving grammars, we need to make several implementation decisions; these are treated in detail in later chapters. Three very important questions are "How are grammars represented in genotypes?", "How do the genetic operators manipulate genotypic representations of grammars?", and "How are grammars' fitnesses evaluated?". These questions are covered in chapters II and IV. The third question, regarding fitness measurement, is also treated in overview here, since it has a strong impact on how we view the analogy between growth from a fertilized egg cell and derivation from a grammar's start symbol.

If the individuals in an EA's population are phrase-structure grammars, and these grammars are to be construed as "developmental programs" in accord with the extension suggested above of the analogy between EA and natural populations, then we must specify what constitutes a phenotype and how grammars as developmental programs are interpreted or executed to produce phenotypes. As in L-systems, we can think of each symbol as type of cell and each step in a derivation as one or more cell-divisions. Thus, the phenotype is a string derived from a grammar's start symbol (analogous to a fertilized egg) using the grammar's rules.

How we measure the fitness of individuals from the EA population effects how we construe the metaphorical relationship between ontogeny in EA and natural populations. If we measure the fitness of the phenotypic string derived from a grammar, the analogy is straight-forward: a genotypic grammar derives a phenotypic string from its start symbol whose fitness determines its chances at reproduction. This sounds like a top-level description of what happens in nature; the fitness function is defined so that its domain is the set of possible phenotypic strings for any grammar which could be written. This approach counts on the availability of some procedure to test or evaluate phenotypes. For example, the strings might represent simple algorithms which could be tested for correct output on sets of inputs. Indeed, this approach will is taken in the application of evolving grammars to discovering sorting algorithms in chapter VI.

The foregoing approach makes the most sense if it isn't possible to write down a training set of phenotypic strings which a highly fit genotypic grammar should be able to derive from its start symbol. If such a training set is available, however, we need to evaluate grammars according to their abilities to account for the training set. In this case, we can employ well know algorithms to ask for each grammar and training string to answer the question "(How) can this string be derived from this grammar?" Such algorithms are called "parsers." Thus, we can distinguish evaluation by deriving a string from a grammar from evaluation by parsing. A "parse" of a string given a grammar summarizes how the string can be derived from the grammar's start symbol using the grammar's rules.

Fitness evaluation by parsing seems to be at odds with our "developmental metaphor," linking a loose model of the ontogenetic process to derivation from a grammar. The second fitness evaluation strategy looks more like something that doesn't happen in nature at all: given a phenotype and a genotype, answer the question "Could this phenotype have developed according to this genotype?", then assign fitness based on the answer to this question for several phenotypes. This seems to be just the opposite of ontogeny, since the phenotypes are supplied and we want to know whether they can be reduced to a grammar's start Consider, however, that evaluating a grammar by trying to parse several strings using it is like deriving a large number of strings from the grammar and then evaluating these. The grammars take on more of the role of species in this case.

The EA described in chapter IV uses evaluation by parsing using a data set. This is a typical grammar induction situation that has been well studied over a long period of time[33, 34, 19]. This work will be interested in inducing em stochastic grammars, a more challenging problem. Stochastic grammars contain an ordinary grammar plus probabilities that allow the grammar to define a probability distribution over strings. In stochastic grammar induction, we are interested not only in finding a grammar which yields at least one derivation of every string from our training data, but also in a grammar that assigns high overall probability to the training data. The goals of stochastic grammar induction subsume those of nonstochastic grammar induction, as the definition of a stochastic grammar extends the definition of a nonstochastic grammar, as will be described in chapter III. Stochastic grammars are of interest as a generalization of probabilistic models of protein sequences in biology, and this application is considered in chapter VII.

I.B A road map

This section gives a guide to the rest of this thesis with emphasis on identifying on which sections a given reader should concentrate. Chapter II gives an overview of EAs, the subject of complex representations, and the evolution of "behaving things." The first section should be read by those unfamiliar with evolutionary computation. The latter parts of the chapter discuss the problems inherent in using EAs to solve machine learning problems to which the solution is a specification of a complex collection of behaviors, and how grammar induction with an EA fits into this regime. Chapter III moves on to a detailed background on grammar induction, considering separately the nonstochastic and stochastic cases. Readers unfamiliar should read at least the definitions of grammars and the grammar induction problems from sections III.B and III.C, as well as the parser description of section B..1. The material immediately following the parser description is a technical discussion of error propagation in the inside-outside algorithm. The final two sections of the chapter discusses what properties make a given learned grammar a good model of its training data, section III.D, and what properties are desirable of the training data, section III.E.

The first three chapters are meant to provide background. The later chapters describe the EAs that were developed as part of the work of this thesis and present and discuss the results obtained using them. In chapter IV, the "EvoGrams" stochastic grammar evolution system is described in detail, including the manner in which grammars are represented on population genotypes, fitness evaluation, and genetic operators. This material is mainly of interest to readers interested in the mechanics and details of the algorithm used here, in more detail than the overviews of similar algorithms provided in earlier chapters. This chapter deals with the details of our approach to using an EA to perform grammar induction.

EvoGrams is used to evolve grammars for several formal languages in chapter V. Many of these formal languages are standard tests in the grammar induction literature, and this chapter will mainly be of interest to readers interested in grammar induction in particular or machine learning generally. We compare EvoGrams's performance to other grammar inducers, and discuss some issues related to stochastic grammar induction in particular. Two sets of results in this chapter are aimed at checking the EA's usefulness in situations involving data analysis when we don't know the nature of the target language. In one case, we analyze the ability of EvoGrams to retain a good solution with which its initial population has been seeded. In the other case, we evolve grammars from data that we know cannot be exactly modeled, and check EvoGrams's ability to capture as much structure as possible.

Chapter VI departs from EvoGrams and discusses a different EA designed to evolve grammars which derive simple sorting algorithms. This different EA does not evaluate by parsing, but instead by deriving. This chapter will be important to readers interested in developmental components in EAs. The genotypic grammars' productions are constructed from one of a few templates. The grammars derive descriptions of sorting networks which are then tested on various inputs to see whether they will correctly sort a list into increasing order. This chapter presents an example in which the analogy between derivation from a grammar and morphogenesis is particularly clear.

Returning to EvoGrams, chapter VII presents results from using it to evolve grammars to describe protein sequence data for the globin family. Grammars are found from partial and whole sequences. A variety of methods are used to analyze how learned grammars account for the data by examining which productions are important for accounting for which training strings, and by discussing how the training data are derived from the grammars. An important part of this chapter is the discussion of performing searches in databases of proteins using learned grammars. We also discuss the use of amino acid similarity metrics with grammars to help detect homologs. This chapter is of interest not only to biologists, but also to readers interested in applications of grammar induction.

The final chapter should be of interest to everyone: chapter VIII summarizes and discusses conclusions that can be drawn from the the previous chapters. In this chapter, future work and improvements are also discussed, including potentially more efficient parsers and the possibility of eliminating the parser by estimating the information it provides, both with respect to fitness evaluation and the inside-outside algorithm. An estimated version of the inside-outside algorithm is proposed, and a discussion of potential limits on grammars as a means of biasing the algorithm is discussed.

Chapter II

Evolutionary Algorithms

In this chapter, the evolutionary algorithm is presented on its own, and then its use to evolve grammars is presented. The early sections of this chapter describe evolutionary algorithms and the evolution of “behaving things” generally, and the later sections discuss evolving grammars as an example.

II.A An overview of evolutionary algorithms

Evolutionary algorithms are search and optimization algorithms which work by loosely simulating evolution on a population of data structures, each of which is thought of as the genotype for an organism in the population[46, 38, 56, 10]. This section describes the most basic version of an evolutionary algorithm and some of the embellishments relevant to the work of this thesis. In the final section of this chapter, we will consider the genetic programming paradigm, an important subdivision of evolutionary computing which bears a special relationship to the topics the work pursued in this thesis, and to the subjects discussed below.

Figure II.1 shows an evolutionary algorithm (EA). The algorithm proceeds by initializing a collection of genotypes at generation 0. At each generation, the fitness function, μ, is evaluated on each individual and then a new population is built by applying genetic operators to highly fit individuals to get the population at the

0. Randomly initialize population, P_0 of n genotypes, g_i
1. Set generation number $t = 0$

Repeat until termination condition reached:
    For i from 1 to n do
        2. Decode genotype g_i to a stochastic grammar Γ_i
        3. Tune Γ_i's probabilities using local search
        4. Compute fitness: $f_i = \mu(\Gamma_i)$
    EndFor
    5. Construct P_{t+1}: apply genetic operators
    6. Add one to t

EndRepeat

Figure II.1: A simple evolutionary algorithm.

next generation. The process of building new individuals employs different numbers of operators in different implementations. After the operators are applied, to build the population for the next generation, the generation counter is incremented and the repeat loop starts again unless a termination condition has been reached. In the EA implementations considered in this work, this generational loop repeats up to some set number of generations or until there is little or no variation among the g_i.

This "local search" component of the EA allows each individual to be optimized separately and individually[44]. In some cases, as will be the case for much of the work discussed in this thesis, the local search operator stands in the metaphorical role of individual learning or accumulation of experience[14]. In some variants of the EA, the modified probabilities are copied back into g_i before proceeding; this "Lamarckian" variant of the algorithm is not always practical or desirable, especially if the mapping from g_i to Γ_i is complicated or expensive to invert. Even when the modified probabilities are not copied back into the genotype, the addition of local search to the EA can have an influence on the trajectory of the population as a whole[14, 44].

In chapter VI, an EA is used which evaluates each Γ_i by deriving a string from the grammar and testing the string as a specification of a simple algorithm.

This mapping stands in a metaphorical relationship to biological ontogenesis, or development generally[43, 14]. The grammar is seen in this case as a developmental program gives instructions regarding the growth of the algorithm. This is discussed further later in section II.C.

The overall time complexity of running an EA such as the one shown in figure II.1 varies depending on the costs of fitness evaluation and local search; the cost for the other, common steps is discussed briefly here. Further details regarding the cost of steps 2 and 4 may be found in chapters IV and VI, and appendix B. Let the number of individuals in the population at each generation be n and the maximum length of genotypes be l bytes. Steps 1 and 6 are simple constant time operations and can be ignored for purposes of asymptotic time complexity. Step 0 will require a fixed amount of time per byte per genotype, so initialization will require time $O(ln)$.

For both of the EA's considered in later chapters, step 2 is linear time per genotype, and the overall time it contributes to the loop of which it is a member is thus $O(ln)$. Step 5 is $O(ln)$ as long as the genetic operators and the selection mechanism take time of order l per genotype. These conditions are satisfied for both algorithms considered here and thus the time required to run the EA other than that needed for steps 3 and 4 is $O(ln)$. That is, the EA overhead is asymptotically bounded by the maximum total space needed to store all of the population's genotypes.

II.B Evolutionary algorithms for evolving behaving things

In the previous section the main concern was the mechanics of a generic version of the GA. This section draws a distinction between two broad classes of genotypic encodings based on the population's genotypes. On the one hand, we have the class of genotypic representations used with evolutionary algorithms which are evolving parameter or variable settings which optimize some function's value, and on the other we have the class of representations used to evolve descriptions of functions, algorithms, agents, concepts, or infinite sets. The latter, "behavioral," entities are usually represented intensionally, not extensionally.

The distinction between these two classes of genotypic representations can be drawn in a way analogous to the idea of "essential infinity"[71, 70]. The idea of essential infinity is developed in terms of classes of phrase-structure grammars. If F is a finite language over some alphabet, then we say the F is essentially infinite with respect to some class of grammars C with a given size metric on grammars, if there is a grammar $\Gamma_0 \in C$ such that:

- The set of strings from $\mathcal{L}(\Gamma_0)$ of length $\leq n$ is exactly equal to F, where n is the length of the longest string in F,
- $\mathcal{L}(\Gamma_0)$ is infinite, and
- For any other grammar $\Gamma \in C$ such that $\mathcal{L}(\Gamma) = F$, size$(\Gamma_0) \leq$ size(Γ).

These conditions are drawn from [71], which also gives conditions on the size metric. A finite language that is not essentially infinite with respect to a given grammar class is referred to as "essentially finite."

The main point of use to us here is that a language is essentially infinite with respect to representation by some grammar class when its shortest description by a grammar in that class is as a length-limited subset of an infinite language. When evolving behaving things, they will generally have a range of possible behaviors analogous to the language of a grammar; in the case of learning grammars, discussed below, the correspondence is literal. If the behavior languages for individuals are always essentially *finite* with respect to the class of genotypic representations, then the behaviors are most briefly summarized by listing them.

If individuals' behavioral repertoires can be either infinite or essentially infinite, then the shortest description of some repertoires is intensional rather than extensional. This more intensional representation will, moreover, specify an infinite set of behaviors, of which only a finite number might be relevant. An important point here is that our exact notion of description length plays an important role. The main point of Savitch's work was that whether or not a language is essentially infinite depends on the class of grammars C. The same observation applies here: genotypic representations for evolving highly fit individuals for a given fitness function μ may seem more or less behavioral, as we illustrate below with some examples.

II.B.1 Some illustrative examples

Suppose an EA is searching for the maximum value of a function μ,

$$(\hat{x}_1, \ldots, \hat{x}_n) = \arg \max_{(x_1,\ldots,x_n)} \mu(x_1, \ldots, x_n)$$

. Each member of the population encodes a single tuple $(x_1, \ldots, x_n)$. μ is the fitness function as in figure II.1. The "behavioral repertoires" of these individuals is just the genotypically encoded tuple.

Consider the same EA , but now each genotype encodes a collection of n unary functions $f_i : \Re \mapsto \Re$ and a single parameter value $p \in \Re$. For a given genotype, the tuple encoded is $(f_1(p), \ldots, f_n(p))$. That is, we compute the values of the functions at the parameter in order to find the tuple that is to be the argument to $\mathcal{G}$. Now in order to decode a genotype we must decode the functions f_i, decode the parameter, and apply the functions to the parameter to get a vector for evaluation. This genotype is further toward the behaving thing end of the spectrum of representations, but the functions are only being evaluated at one point, and the behavioral repertoires are still unitary for each individual. To the extent that evaluating any of the f_i is complicated, imprecise, stochastic, or irreversible, this example will seem more like a behaving thing example if the f_i can be tersely described.

Finally, consider the same EA again with a very similar representation to that just discussed. Now, however, we eliminate the parameter p from the genotype. The parameter is to be selected externally for each individual during or before fitness evaluation. We might, for example, try several values of p and take the best resulting fitness evaluation as the genotype's fitness. Now we must do even more work to evaluate an individual's fitness and the decoding of genotypes is mixed up with fitness evaluation, since we evaluate for several values of p and take the best.

This situation is a fairly good schematic for behaving things representations, especially if we generalize the single p above to multiple parameters, $p_1 \dots p_k$, not all of whose values are necessarily relevant to finding a vector $(\hat{x}_1, \dots, \hat{x}_n) = \arg\max_{(x_1,\dots,x_n)} \mu(x_1, \dots, x_n)$. That is, the coordinate functions f_i now have signatures $f_i : \Re^k \mapsto \Re$ for some $k \geq 0$. Some or all of the parameters may have genotypic initial values which serve as starting points for a search for better values.

The difference between the above examples is the difference between the search spaces being explored by the GA, which is in turn the difference in the form that we want the answer to our optimization problem to take. In the first of the three examples above, running the EA yields a single n-tuple as an answer. In the second case, more is described: the functions f_i in the best individual derive the individual components of the best n-tuple found; the answer is still only a single n-tuple, but its description in terms of functions applied to parameter p tells us something about relationships among elements of the best n-tuple evolved in this representation, especially if we constrain the space of functions from which we draw the f_i to make such interpretation of constraints convenient. Further, genotypes differing only in the value p which are highly fit encode equally good solutions drawn from a common space, the space of p values for which the shared f_i give optimal or near-optimal vectors. In the final case, the functions are the answer to the problem for whatever value of p is provided external to the genome. That is, the last example is one in which we are genuinely looking for mappings which give us the best vector for whatever value of p is provided.

Phrase-structure grammars are clearly an example of behaving things, since evaluation of a grammar requires either that we derive strings from the grammar's start symbol using the grammar's productions or that we try to determine whether it is possible to derive a given string from the grammar. Both of these options present the problem of having a large or even infinite possible set of test cases against which to test the grammar. In the latter case for stochastic grammars, we also need to keep track of how the given string is derived (i.e., what productions are applied in what sequence to arrive at the given string) in order to assign probabilities to strings; thus we need to parse, as opposed to simply recognizing, strings using grammars. In the case of stochastic grammars, the probabilities attached to a grammar's productions may also be modified by a tuning algorithm during fitness evaluation, in which case evaluation and decoding are intermixed as in the third example above. The next two sections discuss issues involved in evolving behaving things, how these issues relate to grammars in particular, and some specific issues related to evolving grammars.

II.B.2 Behaving things and grammars

To evolve behaving things in general and grammars in particular we must address several issues with respect to the application of genetic algorithms to behaving things representations. Many of these are observations that apply to learning representations of behaving things generally and some are specific to the GA. In this section, the general issues are discussed first, followed by the issues specifically related to the GA; both of the latter are concerned with the relationship between the genetic operators used and the particular genotypic representation chosen for writing down descriptions of behavers.

An immediate concern is that behaving things tend to have very large or infinite input or stimulus spaces. In the case of a mapping from one set to another, we can only afford to verify its behavior on a finite subset of its domain. If the domain is infinite, then we can never hope to exercise the mapping over all possible inputs. If the domain is very large, it may be too expensive computationally to perform a complete evaluation. This concern arises in much of machine learning generally, concerning generalization from a training set to the complete domain on which the learning algorithm must perform. In the case of evolutionary algorithms, the concern also translates into a need to get as accurate an evaluation of each population member's fitness as is possible.

On the side of the GA, as mentioned earlier, is that it is population based.

We expect that successful individuals will persist across generations and that the population will contain more than one copy of a highly successful individual. This is because the GA tends to converge on the present best members of its population unless some improvement is being made. Even if the GA doesn't converge so that the entire population is composed of identical individuals, there are still usually many identical or similar solutions present in the population at various generations while the GA is executing. Thus, highly successful individuals will tend to see many test cases over the course of several generations, even if at each generation each individual of the population is only evaluated using a small sample of the possible test cases.

With respect to grammars in particular, fitness evaluation is accomplished either by deriving strings from grammars or by parsing strings using grammars. In the former case, we must have some procedure for judging derived strings as being correct or incorrect. In the latter case, strings are taken from a training set against which the GA tests the grammars in its population at each generation. In the case of stochastic grammars, the result of parsing is that we also know the probability of each parsed string given each grammar. If there are a large number of strings available, we may choose to evaluate each grammar using only some chosen subset of the training set; this subset changes from generation to generation or even individual to individual. The hope is that grammars which persist in the population across many generations will do so because they've been exposed to many different training strings and proved highly fit in the face of all of them.

Usually, testing a behaver involves exposing it to various test cases or stimuli. The presentation of each test case requires the evaluation of the individual's behavior for that test case. This usually requires an interpreter or a simulator which often depends on a pseudo-random number generator. The latter dependency is related to the sampling issues already mentioned. The need for an interpreter points out an important feature of behaving things — they are essentially *programs* with respect to some interpreter. It is important to design the representation so that the class of behaving things that are representable is neither too exclusive nor too inclusive. If only simple behavers can be represented, we can't discover complicated behaviors, if too broad a class of behaviors is representable, learning with the representation may be intractable.

This issue is brought into sharp relief in the case of grammars, whether they are stochastic or nonstochastic, in the light of the Chomsky hierarchy. If we choose to represent only regular grammars, we can't capture center-embedding relationships in the training data and we might miss regularities of interest. In any event, learning context-free languages is out of the question. Context-free grammars allow a greater variety of regularities in the training data to be recognized but at an increased cost of parsing when it is used. If we allow the grammars to be as general as context-sensitive, the parsing problem is already intractable, and if we allow fully general phrase-structure grammars, the parsing problem is uncomputable, since parsing is in that case tantamount to solving the halting problem. Similar considerations, though exclusively regarding tractability and not computability, apply in terms of the size of the search space as we increase the maximum number of nonterminal symbols allowed in a grammar and the maximum number of productions allowed in a grammar, as will be seen later.

With respect to the GA in particular, it is important to arrange the genotypic representation and genetic operators so that they are well-fitted to one another. In the case of evolving behaving things, it is important that the mutation and crossover operators operate on the genotypes in such a way that they can alter and recombine behaviors by altering and recombining parts of the encodings of behaviors. This simple fact is fundamental to evolving anything with behavior or anything like a function or mapping, and is easy to see with respect to grammars; recombining parts of productions allows us to recombine behaviors. That is, recombining a production left-hand side with some other right-hand side changes, at least potentially, the set of strings derivable from the left-hand side. Crossover on genomes might exchange, in addition, whole sets of productions, thus changing the language derived from the grammar encoded on a given individual's genotype. Similarly, point mutations can

	Nature	
Genotypic grammar	Derives	Parses
Derives	Score sequence match	"Parse" phenotypic string
Parses	Parse environmental string	Score derivation match

Table II.1: Fitness evaluation methods depending on whether "nature" or genotypic grammars are used, to generate or parse strings.

change both left- and right-hand sides of productions.

Another important issue with respect to the GA is the extent to which the genetic operators give rise to individuals whose genomes are encodings of well-formed behavers. It is often very difficult to arrange the representation so that crossovers in particular always yield individuals that are at least as fit as their parents. Usually, there is at least some probability that a crossover will yield an individual that is either less fit than both parents or, in addition, completely nonsensical as an encoding of something in the class of behaving things being evolved. Two popular strategies for dealing with such difficulties are repair[43], in which ill-formed or "nonviable" individuals that are allowed a limited number of corrections before they are abandoned as hopeless, and brood selection, in which for each pair of parents many crossovers are done and an computationally inexpensive simplified fitness function, the "brood fitness" is used to choose the best offspring for inclusion in the next generation.

II.C Grammars, ontogeny, and fitness evaluation

The option of having genotypic grammars generate or parse strings is mirrored by a similar option with respect to fitness information. Table II.1 summarizes the different variations available; each cell gives the action that must be taken to evaluate fitness for each genotype. A derivation from a genotypic grammar is the metaphorical equivalent of ontogeny, morphogenesis, or at least some part of the broader category of changes which organisms undergo and which are collectively called "development." The grammar itself is a developmental program and the interpreter for the grammar, which derives a string from its start symbol, is the environment in which development occurs and which is necessary for the developmental program to be carried out.

In the table, the source of fitness information, "nature," is either providing labelled example strings (labelled with probabilities in the case of stochastic grammars, with a yes/no indication for nonstochastic grammars), or consists of a grammar or mechanism which we use to parse strings provided by the genotypic grammar. In the latter case, we refer to an imagined pre-existing grammar which essentially forms the solution to the learning problem as the "natural grammar." The language, or set of strings, derivable from the natural grammar is the target language.

In the upper-left cell of the table, we are in the situation of already having some set of strings that are correct solutions and we derive strings from the genotypic grammar and see how well they match the known solutions. That is, we have a sample of the correct language and we generate a sample from the grammar on a genotype and see how similar the strings derived from the genotypic grammar are to the sample strings. One possible candidate for this match score is simply the length of the longest substring. Another interesting possibility is that of using sequence alignment scores of the kind used to align proteins as the similarity metric here.

In the upper-right cell of the table, it is presumed that there is either an actual known grammar that generates the target language, or that there is some computable function that allows us to determine membership in the target language. For example, in chapter VI, the goal is to find grammars that generate sorting networks, members of a class of simple sorting algorithms. We don't actually hope to find a grammar for all sorting networks, but rather a grammar that generates a family of networks that are similar but sort different size arrays. As will be seen, a network is derived from each genotype and then tested against different input arrays to make sure it outputs a sorted array. Thus there is no "natural grammar" in this case, but rather a testing method that tells us whether the network derived from the genotypic grammar sorts correctly.

Recall from above the two main ways of assigning fitness to grammars. The first was to derive a string from a genotypic grammar and evaluate it using some function. This fits the developmental metaphor and the upper-left table cell best: the genotype encodes a grammar, or developmental program, a sequence of symbols is derived from the grammar, or a phenotype grows and is formed according to the program, and the string or organism survives or doesn't survive depending on its relative fitness in its environment[1].

Much of typical grammatical inference falls in the lower-left cell. Here we have some collection of strings from a target language and we seek a correct grammar to capture the syntactic regularities of this training data. We parse each string in the training data using each genotypic grammar and assign fitness based on whether the genotypic grammar correctly accepts or rejects the string, for nonstochastic grammars, or based on the probability assigned to the string, for stochastic grammars. Strings that are not in the target language may also be present in the sample and will be labelled as such. Thus, "nature" comprises in some sense more than just the definition of the target language: it contains some generator of strings plus the target grammar which labels them as being positive examples (members of the target language) or negative examples (not members of the target language).

The second method of evaluating fitness discussed previously is suggested by this case: there are pre-existing strings and the genotypic grammar is used to parse them, and assign probabilities if it is a stochastic grammar. This seems to be in direct conflict with the metaphor just developed. Recall that stochastic grammars assign probabilities to strings of symbols drawn from a finite alphabet. This information is the same as the information gained from deriving a large number of strings from the grammar randomly, choosing productions, when more than one is applicable, according to the grammar's production probabilities. We can arrive at string proba-

[1] In the GA, fitness is the main, often the exclusive, point of contact between individuals in the population and their environment. Indeed, it is sometimes appropriate to say that the fitness function *is* the environment for individuals in a GA population.

bilities by counting how many times each distinct string occurs, and assigning strings probabilities equal to their frequency of occurrence as the output of derivations.

This is the way in which the probability distribution according to a stochastic grammar arises. Let N_D be the number of derivations performed and N_S be the number of distinct strings created by these derivations. If the strings are indexed from 1 to N_S, let c_j be the number of times that the j'th string was derived from the grammar and p_j be the probability of the j'th string according to the grammar. Then formally,

$$p_j = \lim_{N_D \to \infty} \frac{c_j}{N_D}, 1 \leq j \leq N_S \tag{II.1}$$

For complex grammars with many productions and long strings, the value of N_D at which the error between c_j and $\frac{c_j}{N_D}$ becomes small relative to p_j could be very large, but the point is that parsing using to the stochastic grammar is like deriving a large number of strings according to that stochastic grammar and counting occurrences of each distinct string.

If the genotypic grammar is not stochastic, similar observations apply, but the question isn't one of the probability of strings but of their membership in some target language. When we evaluate a grammar by deriving strings and then computing some function f on the derived strings, f is actually telling us about membership of the derived strings in the set of strings that we would like to be able to derive from the genotypic grammar. Parsing answers the question *given* a string of whether it is derivable from the grammar. Once again, the information we get by deriving many strings from the grammar will be equivalent to that obtained by parsing strings. The question is one of whether we wish to evaluate genotypic grammars by sampling the sets of strings derivable from them, or by asking, for a given set of strings, whether they are derivable from the grammar.

How does this similarity of information effect our developmental metaphor? Think of a genotypic grammar as a developmental program for a species and the particular strings derivable from that grammar as the different possible members of that species. Then in evolving a grammar, we are trying to evolve a definition of a species that represents a solution to some problem which we've set up for the GA to solve. Given a particular genotypic grammar Γ, we can ask the question: "To what extent do the members of the species defined by Γ represent solutions to the problem posed the evolutionary algorithm?" The answer can be found by deriving strings from the grammar and by some independent means determining whether they're solutions, or by checking whether known solutions are in the species, which is the language of the grammar. In the case of traditional grammar induction, the two alternatives are methods for checking that the species defined by a grammar generates strings in the target language and not strings outside of the target language. Thus, evolving grammars is metaphorically closer to the idea of evolving developmental programs for species rather than evolving developmental programs for individuals.

The lower-right cell represents a method in which we must have some set of strings whose membership in the target language is unknown. If we evaluate each with the parsing method available for the target language, this case will collapse into the lower-left cell, since we now have a set of correctly classified strings to parse. Alternately, we could look at the actual derivations found for each string according to the "natural grammar" and compare these with derivations found by parsing the same strings using a genotypic grammar. This is a richer source of information than is available in the lower-left cell, but such information about the correct answer to the problem posed the GA will usually not be available. Finally, we could preclassify or evaluate probability for the strings according to the genotypic grammar and then check these answers using the natural grammar.

II.D Genetic programming

An important and widely employed subset of evolutionary algorithms arose from John Koza's "Genetic Programming Paradigm" (GP)[56, 53]. In GP, an evolutionary algorithm is used to evolve programs in the LISP programming language. These programs are written as "s-expressions." If f is a scalar-valued function of two arguments then we write $f(1,2)$ as:

$$(f\ 1\ 2)$$

If g is another function of one argument which returns a scalar, then we can write $f(g(1),2)$ as a LISP expression as follows:

$$(f\ (g\ 1)\ 2)$$

In its most basic form, GP is concerned with evolving expressions like those above. The fitness function exercises each genotypic program over some test domain and scores the program according to how correct it is. The genotypic program considered as a function often has either a large domain, a large range, or both. In any event, its repertoire of behaviors is the set of possible input-output pairs, which will often be large and sometimes itself infinite in theory. GP uses a crossover operator that exchanges subexpressions of the genotypic expressions in its population. There are often several mutation operators which may add a new random subexpression to a program or change an entire subexpression. There is also a more conventional mutation operator which makes small changes.

Applying GP to evolve a program to perform some computation involves choosing a fitness function and some functions and constants for the genotypic representation to use. Choosing functions and constants is how one chooses a specific problem representation in genetic programming. In a now-common variant, the genotypic programs are also allowed to contain "automatically defined functions"(ADFs). These are functions that evolve on the genotype as a result of abstracting subexpressions into the bodies of LISP function definitions[57]. ADFs are both defined and used in genotypic programs in GP.

In terms of our earlier discussion in this chapter, GP is a paradigm example of the evolution of behaving things. Quite often, the behavioral repertoires for genotypic programs will be infinite. Even when repertoires are finite, it will often be the case that they are essentially infinite, since we usually do not desire a program to perform a computation which we suspect could be efficiently performed with a look-up table.

We can relate the work of this thesis to GP readily in three distinct ways:

1. by appealing to a similarity between derivations from grammars and LISP s-expressions,

2. by invoking the equivalence of grammars and formal automata, and

3. by pointing to GP work designed to solve grammar induction problems[2].

In figure II.2 we show a simple GP program with imagined functions $\{f(\cdot,\cdot), g(\cdot), h(\cdot,\cdot)\}$, and imagined constants $\{0, 1, 2, 3\}$. To the right of this GP program, we show a derivation tree and a grammar. The grammar in question is capable of generating the tree shown. Note that the nesting of parentheses in the s-expression captures the same relationships as the parent-child relationships in the derivation tree. In fact, LISP stores s-expressions as tree-like data structures, though using a slightly different representation in which each tree node is represented as a list. Note that in figure II.2, the grammar derives exactly one string, the GP program. This could be generalized, or only one derivation from a grammar taken as the program to be tested. In either case, the derivation process has a "developmental" flavor: the GP program "grows" out of the grammar's start symbol.

Note that the grammar in figure II.2, the grammar evolves only the tree corresponding to the GP tree shown at the left. Thus, the grammar and the s-expression are descriptions of the same GP program. Why might we prefer the representation of this program at the far right of the figure to the representation at the left? First, the grammatical representation captures the hierarchic nature of the s-expression in a linear text string. Each rule corresponds to a function call and since the ordering of the rules on the genotype need not be fixed, functionally related groups of rules can be clustered on the genotype. This could be built into the representation or we could

[2]Readers unfamiliar with phrase-structure grammars should skip ahead and read some of the introductory material in chapter III to better understand what follows.

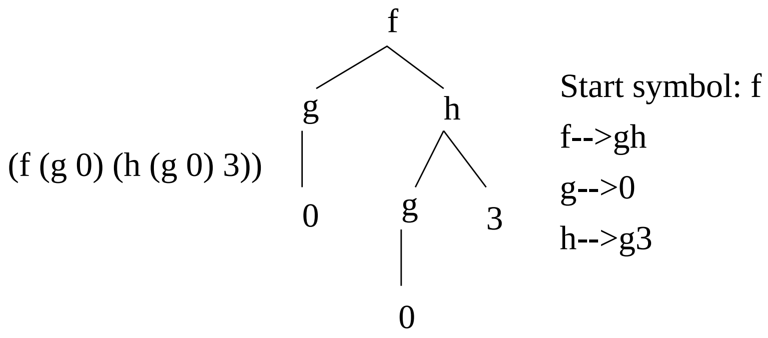


Figure II.2: A simple GP program with equivalent derivation tree from a grammar that generates it.

allow it to happen or not in evolution according to whether our EA did it naturally. Further, the grammatical representation presents us with the opportunity to allow the evolution of general program specifications, as grammars, to build specific programs for different sizes of input. For example, we will discuss the evolution of grammars which derive efficient sorting algorithms for different sizes of lists in chapter VI.

Another connection between genetic programming and evolving grammars is based on the correspondence between classes of grammars and classes of computing machines. For example, the class of *regular* grammars is capable of recognizing or generating the same set of languages as the finite automata. Context-free grammars correspond in similar fashion to push-down automata. Most generally, unconstrained grammars have computational power equivalent to a Turing machine. Thus, for any computation for which we wish to evolve a program, we could imagine evolving a grammar instead. This correspondence is of most apparent in case in which GP has been used to discover programs that implement machines equivalent to grammars[15].

Lastly, GP itself has been directly applied to learning grammatical systems which themselves are intended to model plant development, L-systems. In these cases, the genotype describes a collection of rules which are used to build a string. In this case, the correspondence is between a grammar and an s-expression instead of between a derivation from a grammar and an s-expression[48].

Chapter III

Grammar induction

This chapter gives necessary background and introduces previous work in the field of grammar induction. Grammar induction is a heavily-studied subject in the field of machine learning and pattern recognition, as well as in linguistics, psychology, and also philosophy. A complete survey of the field is well beyond the scope of this chapter, and what follows is mainly restricted to the subject of nonstochastic[1] and stochastic grammar induction as they relate to the subsequent chapters.

Consider the task of writing a grammatical model of a natural language. For English, as an example, we might start a grammar using productions like those in figure III.1. "S" is the start symbol and stands for "sentence," "NP" stands for "noun phrase," "VP" for "verb phrase," "Det" for "determiner," "N" for "noun," "Adj" for adjective, "V" for "verb," "PP" for "prepositional phrase," and "P" for "preposition." The start symbol stands for a complete sentence and the rewrite rules tell us how sentences and their constituents can be formed.

An example of a string which can be derived from this grammar is: "the computer sat on the antistatic mat." Though this example is very simple-minded, it points out how grammars for natural languages specify what strings count as "grammatical," or as members of the language, and also describes the structure of the

[1] We use the term "nonstochastic" instead of "deterministic" because the latter means something else when spoken of grammars.

Start symbol: S

$$
\begin{array}{rcl}
S & \rightarrow & NP\ VP\ . \\
NP & \rightarrow & Det\ N \\
NP & \rightarrow & Det\ Adj\ N \\
VP & \rightarrow & V \\
VP & \rightarrow & V\ PP \\
PP & \rightarrow & P\ NP \\
Det & \rightarrow & \text{the} \\
N & \rightarrow & \text{computer} \\
N & \rightarrow & \text{mat} \\
Adj & \rightarrow & \text{antistatic} \\
V & \rightarrow & \text{sat} \\
P & \rightarrow & \text{on}
\end{array}
$$

Figure III.1: Some possible beginning productions in a grammar for English.

grammatical strings. The parse tree for a string in the language of this grammar identifies its constituents, such as its subject and main verb, and other syntactic entities.

Though the primary concentration in the following chapters is on the stochastic version of the problem, nonstochastic and stochastic grammar induction have much in common. First, we give some antecedents regarding nonstochastic grammar induction in section III.B. Section III.C adds information regarding the computational stochastic case, and the final three sections discuss costs involved in evaluating grammars, some figures of merit which have been used in grammar induction, and the quality of the training data used to induce grammars.

III.A Historical Antecedents

Grammar induction has long been studied in a variety of fields spanning scientific research and philosophical inquiry[25]. Some of its most important roots lie in the desire to understand how human beings acquire language, and how such acquisition is possible. One particularly contentious debate concerns the extent to which linguistic ability is "innate" as opposed to being learned by human children purely from the data available to them. The question is, how much prior information do children possess at birth about a correct grammar for the language they learn to speak? A strict "nativist" would say that linguistic ability is encoded in our genetics, and a childs emerging ability to communicate with language is a natural part of ontogeny. To the contrary, an adherent of the "empiricist" view would say that children are born with only a very general ability to learn and that they apply such a mechanism to learning language.

Another particularly interesting hypothesis originally formulated by the famous linguist Noam Chomsky is that the human brain at birth already contains or encodes so-called "linguistic universals" that form the antecedents of linguistic behavior for all human languages[21]. This view seeks to explain how children are able to acquire any language even though the sum of the data available to them as they grow up seems insufficient. Linguistic universals would be a bias built into the human brain for acquiring the kinds of linguistic structure our species tends to use; they thus represent a compromise between the extremes of strict nativism and strict empiricism.

Chomksy also distinguished between language use and linguistic ability. The former is referred to as performance and is distinguished from the latter, competence. How can we be construed to have anything like a grammar, or any idealized formal system for language, encoded in our brains given that we make mistakes and produce fragmented, ungrammatical constructs in speaking and writing? The idealized aspect of linguistic knowledge wherein no mistakes are ever made is called linguistic competence. Performance arises as a process of making use of the machinery of speech and writing and it is here that slips of the tongue or pen are accounted.

An early attempt to model language acquisition within computer science is found in Ethan Mark Gold's article "Language Identification in the Limit"[37]. Gold is concerned with a model in which the learner is presented with an endless succession of labelled examples. After each example arrives, the learner makes a new guess about a correct definition of the target language. In the case of grammar induction this would mean making a new guess at a correct grammar, and hear we mean the stronger sense of "correct," $\mathcal{L}(\Gamma_{\text{est}}) = \mathcal{L}(\Gamma_{\text{target}})$. The learner is said to identify the target language in the limit if after some finite amount of time all subsequent guesses are the same and all are correct. This paradigm is meant to reflect human language learning as a convergence to the correct language definition, and has as an important component that no observer or teacher ever informs the learner that he or she has reached the correct grammar, though the learner may receive feedback about individual sentences during learning in the former of judgements about grammaticality made by an already competent language user. Languages which can be learned under this model are referred-to as "identifiable in the limit."

Gold was mainly concerned with the identifiability in the limit of languages drawn from various sets or classes of languages given various regimes of "information presentation." An important distinction for Gold is the difference between a "text" and an "informant" as an information presentation method. A text for a target language $\mathcal{L}(\Gamma_{\text{target}})$ is a sequence of strings $\{x_t \mid x_t \in \mathcal{L}(\Gamma_{\text{target}})\}$, where $t \geq 1$. A text is required to meet the condition that each string in the target occurs at least once in its sequence, which thus may need to be infinite if the language to be learned is infinite. An informant tells whether a given string is in the target language, and thus allows inclusion of negative examples. The informant serves as an oracle concerning membership in the target language. Gold made a further distinction between learning a generator of a language and learning a tester for a language. The former is capable of generating all strings in the language in some predetermined order, and the latter is capable of answering the question of membership in the language for a given string. Since we're interested in learning grammars, which are amenable to treatment both as generators and testers, this distinction doesn't figure prominently in the discussion here.

Gold's results are summarized in theorem 1. These results underscore the importance of the availability of negative examples. Without some availability of negative examples, only finite languages can be distinguished, and only from other finite languages. Later, we will see a possible alternative source of information to negative examples in the form of a probability distribution over the strings in a language. Identifiability in the limit represents an attempt to formalize the language learning task successfully performed by humans in computational terms. It is somewhat different from the grammar induction task to be stated below.

Theorem 1 (Summary of results from [37])

1. *No class of languages containing at least one infinite language is identifiable in the limit from text on the representation of recognizers.*
2. *When learning from an informant (whether the learner chooses the examples to submit to the informant, or the choice of examples is imposed), the primitive recursive languages are learnable The primitive recursive languages are a superclass of the context-sensitive languages.*
3. *A "primitive recursive text," one in which x_t is a primitive recursive function of t and all strings of the target occur at least once, suffices to learn* generators *for recursively enumerable languages, the top of the Chomsky hierarchy and the limit of known computability.*

III.B Nonstochastic grammar induction

To start, let us formally state the grammatical inference problem which was sketched in chapter I. The following definitions will be extended in section III.C and used throughout the rest of this thesis. Section III.A will then give a brief overview of important past work in grammar induction. The grammar induction problem as set forth here is based heavily on the formulation popular in machine learning and is discussed extensively in Fu and Booth's pair of review articles from 1975 [33, 34].

III.B.1 Definitions: The Grammar Induction Problem

An alphabet is a finite set Σ satisfying the constraint that $\sigma_1 \in \Sigma$ and $\sigma_2 \in \Sigma$ implies that $\sigma_1\sigma_2 \notin \Sigma$, where $\sigma_1\sigma_2$ is the concatenation of σ_1 and σ_2. Σ^* is the set of all sequences of zero or more symbols drawn from Σ. Such sequences are called "strings over sigma" and the * operation thus defined is called the Kleene-star operator. Σ^* is sometimes called the "Kleene-star closure" or "Kleene closure" of Σ. For a set S, S^+ denotes the set $S^* - \epsilon$, where ϵ is the empty string[2]. A language over Σ is a subset, finite or infinite, of Σ^*. In other words, a language $\mathcal{L}$ is a set of strings over some alphabet Σ, or $\mathcal{L} \subseteq \Sigma^*$.

With the above basic terminology in place we can define the main object of study here:

Definition 1 *A phrase-structure grammar (a "grammar" hereafter) is a 4-tuple* $\Gamma = (\Sigma, N, S, R)$ *where:*

> Σ *is an alphabet containing what are called the* terminal *symbols,*
>
> N *is an alphabet containing what are called the* nonterminal *symbols,* $(N \cap \Sigma) = \emptyset$*,*
>
> S *is the grammar's start symbol, and*
>
> R *is a set of "productions" or "rewrite rules," each of which is an element of* $(N \cup \Sigma)^+ \times (N \cup \Sigma)^*$*, with the constraint that the first element of the pair always contains at least one symbol from* N*.* $r_i \in R$ *is written* $r_i = (\mathrm{lhs}_i, \mathrm{rhs}_i)$*.*

For each $r_i \in R$, lhs_i is called r_i's left-hand side, and rhs_i is called r_i's right-hand side. Productions are most often written in the form $\mathrm{lhs}_i \rightarrow \mathrm{rhs}_i$ instead of $(\mathrm{lhs}_i, \mathrm{rhs}_i)$. A grammar Γ defines a language, $\mathcal{L}(\Gamma)$ over its terminal alphabet. The elements of $\mathcal{L}(\Gamma)$ are exactly those strings of terminals that can be derived from the productions R of Γ. Each step of a derivaton replaces a substring matching some rule's left-hand side

[2] The empty string is the unique string of length zero.

$\Gamma = (\Sigma, N, S, R)$ and $\forall r \in R$, r has the form:

Regular grammars	**Context-free grammars**
$$\begin{array}{lcl} N & \rightarrow & \Sigma N \\ N & \rightarrow & \Sigma \\ N & \rightarrow & \epsilon \end{array}$$	$$N \rightarrow (N \cup \Sigma)^*$$
(a)	**(b)**

Figure III.2: The form of productions allowed in regular grammars, (a), and context-free grammars, (b).

the rule's right-hand side. Derivations must contain a finite number of steps. More briefly, $\mathcal{L}(\Gamma)$ is the set of $\sigma \in \Sigma^*$ which are derivable from Γ's start symbol.

A regular grammar is one in which all rules have one of two forms: either a single nonterminal on the left and a terminal followed by a nonterminal on the right, a single nonterminal on the left and a single terminal on the right, or a single nonterminal on the left and the empty string on the right. A context-free grammar is a grammar such that all $r_i \in R$ satisfy the constraint that the left-hand side is a single nonterminal. Often these are shown schematically, as in figure III.2, in which N stands for any nonterminal, Σ stands for any terminal, and the empty string is notated as ϵ. The languages derivable from regular and context-free grammars are called regular and context-free languages, respectively. In this thesis we will always be attempting to induce regular or context-free grammars, though we will attempt to use them in at least one instance to model more general languages to the extent that they can be used to do so. Thus, the left-hand sides of productions in the grammars *induced* will always be single nonterminals.

Figure III.3 shows a pair of grammars, one which derives bit strings with even parity (an even number of 1's) and one which derives all strings of properly nested and matched parentheses. Grammar (a) is regular and grammar (b) is context-free. Typically, the sets N and Σ need not be listed explicitly. It is understood that the set of symbols making up the productions are the terminals and nonterminals, and that symbols appearing on left-hand sides comprise the set of nonterminals. The distinction is clear since all left-hand sides for these two varieties of grammars are made up of a single nonterminal. The terminal and nonterminal sets will be listed explicitly only when necessary for clarity.

Below each grammar in figure III.3 is a derivation of a string in the language of each grammar. The strings are $001010 \in \mathcal{L}(\Gamma_{\text{parity}})$ and $()(()) \in \mathcal{L}(\Gamma_{\text{parens}})$. These derivations also show the typical style of writing derivations from grammars: each step is shown by writing an arrow with a rule label over it and the string after the arrow is the result of applying the rule labelling the arrow to the string before the arrow. Labels are usually only included when of interest or for clarification and some steps may be collapsed using the special notation $\omega_1 \Rightarrow \omega_2$ where $\omega_1, \omega_2 \in (N \cup \Sigma)^*$ are strings of terminals and nonterminals, called "sentential forms," such that zero or more applications of rules transforms ω_1 into ω_2. Thus, we can define $\mathcal{L}(\Gamma)$ another way as $\{\sigma \in \Sigma^* | S \Rightarrow \sigma,$ where S is the start symbol of $\Gamma\}$. Below each derivation is a parse tree mirroring the structure of the derivation; each internal node and its immediate children correspond to a single rule application. The root of a parse tree is the start symbol and the leaves are terminals. Since the order of the steps of derivations can be changed in some cases without changing the string derived, the relationship between parse trees and derivations is many to one in general, and parse trees are often used to indicate the structure of a string according to a given grammar.

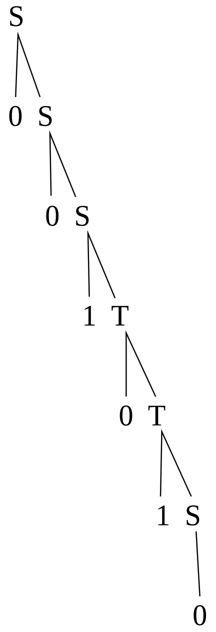
The task for a grammar induction program is to start with some finite set of example strings, the training data E, over some alphabet, Σ, and induce a grammar, Γ that correctly classifies all of the strings in E. For each $\sigma \in E$, Γ assigns labels to strings that match the labels given in E. E is usually thought of as a set of ordered pairs of the form (σ, label) where $\sigma \in \Sigma^*$ and label $\in \{+, -\}$. The label classifies the string it accompanies as being a positive or negative instance, depending on whether it is in the target language or not. E naturally induces two disjoint sets, $E^+ = \{\sigma \mid (\sigma, +) \in E\}$ and $E^- = \{\sigma \mid (\sigma, -) \in E\}$. Thus, the grammar induction

Γ_{parity}: Even parity

Start symbol: S

r_1	S	$\rightarrow$	0S
r_2	S	$\rightarrow$	1T
r_3	S	$\rightarrow$	0
r_4	T	$\rightarrow$	0T
r_5	T	$\rightarrow$	1S
r_6	T	$\rightarrow$	1

$$\mathrm{S} \xrightarrow{r_1} \mathrm{0S} \xrightarrow{r_1} \mathrm{00S} \xrightarrow{r_2} \mathrm{001T} \xrightarrow{r_4} \mathrm{0010T} \xrightarrow{r_5} \mathrm{00101S} \xrightarrow{r_3} 001010$$



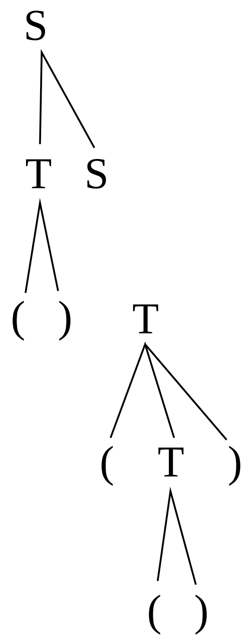
(a)

Γ_{parens}: Parentheses

Start symbol: S

r_1	S	$\rightarrow$	TS
r_2	S	$\rightarrow$	T
r_3	T	$\rightarrow$	(T)
r_4	T	$\rightarrow$	()

$$\mathrm{S} \xrightarrow{r_1} \mathrm{TS} \xrightarrow{r_4} \mathrm{()S} \xrightarrow{r_2} \mathrm{()T} \xrightarrow{r_3} \mathrm{()(T)} \xrightarrow{r_4} \mathrm{()(())}$$



(b)

Figure III.3: Two example grammars. (a) is regular and (b) is context-free. Below each grammar is an example of how a string in the language of that grammar is derived, and below each derivation is a corresponding parse tree.

task is to find a grammar, Γ_{est}, such that:

$$\begin{aligned} \mathcal{L}(\Gamma_{\text{est}}) \cap E^+ &= E^+ \\ \mathcal{L}(\Gamma_{\text{est}}) \cap E^- &= \emptyset \end{aligned} \tag{III.1}$$

That is, we seek a grammar Γ_{est} generalizing E^+ to $\mathcal{L}(\Gamma_{\text{est}}) \supseteq E^+$ in a way that takes into account common substrings or patterns shared among the individual strings in E^+, with the constraint that no strings from E^- be included.

In doing this, we are implicitly imagining that there is some "target grammar," Γ_{target}, defining the "target language," $\mathcal{L}_{\text{target}} = \mathcal{L}(\Gamma_{\text{target}})$, and that the best solution to the problem is a grammar Γ_{est} satisfying $\mathcal{L}(\Gamma_{\text{est}}) = \mathcal{L}(\Gamma_{\text{target}})$. Often it is not reasonable to expect that we will be able to learn such an accurate depiction of a finite training sample drawn from $\mathcal{L}_{\text{target}}$, as will be seen below, and we must settle for a grammar satisfying the earlier conditions of equation III.1, though we may choose among many such grammars according to some bias we have concerning what sort of model of E we would most like.

Even if finding the best or "exact" answer is not possible, we may be able to induce a grammar to distinguish the target language from some particular set of strings smaller than all of $\Sigma^* - \mathcal{L}_{\text{target}}$. In this case, we are trying to distinguish some pair of languages $\mathcal{L}^+ \supseteq E^+$ and $\mathcal{L}^- \supseteq E^-$. The "true" grammar induction task is the case in which $\mathcal{L}^+ = \mathcal{L}_{\text{target}}$ and $\mathcal{L}^- = (\Sigma^* - \mathcal{L}_{\text{target}})$. Practically speaking, we may be inducing a grammar meant to distinguish two kinds of strings, and we may know in advance that the set of negative instances seen will never exhaust the complement of the positive instances. Thus, finding a grammar to satisfy equation III.1 means finding a grammar to distinguish a target language from *all* other strings, or as finding a "separating grammar" to reliably distinguish between two sets of strings. Both are legitimate goals for applied grammatical inference.

III.B.2 Prior Results in Grammar Induction

Our grammar induction task definition arises from a concern for learning formal languages efficiently, as opposed to Gold's concern with providing a model of human natural language learning. As a result, identifiability in the limit is open-ended, requiring only that eventually the learner settle on a single hypothesis without any time limit; the previous definition in Fu and Booth's pair of seminal articles is concerned with learning as good a grammar as possible from finite sample in a finite amount of time, with very little concern for how humans go about learning language.

The first of the two articles by Fu and Booth discusses regular and context-free grammar induction and some applications; the second discusses tree grammars, not discussed here, and stochastic grammar induction, discussed below. One of the more important definitions from Fu and Booth concerns the nature of the positive training data, E^+ and is worth stating exactly as they wrote it:

Definition 2 (Definition 10 from [33]) *A positive sample S^+ of a language $L(G)$ is* structurally complete *if each rewrite rule of G is used in the generation of at least one string in S^+.*

A structurally complete sample "exercises" all of the rules in the grammar of whose language it is a sample. Most grammar induction methods must make the assumption, often implicitly, that the positive training examples comprise a structurally complete set. Even a structurally complete set is not wholly adequate, however, as will be discussed in section III.E.

A "definite grammar," Γ is a grammar for a finite language, $|\mathcal{L}(\Gamma)| < \infty$. A "recursive" grammar, in the terminology of Fu and Booth's article[3], is any grammar for an infinite language. Equivalently, a recursive grammar $\Gamma = (\Sigma, N, S, R)$, has at least one nonterminal derivable from the start symbol which derives a string including itself, and can also derive at least one string of terminals, or the empty string. Formally, $\exists X \in N$ such that $S \Rightarrow \alpha X \beta \Rightarrow \delta X \gamma$ and $X \Rightarrow \sigma$, where $\alpha, \beta, \delta, \gamma \in (N \cup \Sigma)^*$

[3] This use of the word "recursive" should not be confused with its use to describe the languages at a particular level of the Chomsky hierarchy.

and $\sigma \in \Sigma^*$.

Regular Language Induction

The basic grammar induction methods for regular languages construct a regular grammar by starting with a "canonical definite grammar" constructed from E^+. This is a definite grammar which introduces a unique set of rules with new nonterminals for every string in E^+. Grammars are derived from the canonical definite grammar by merging nonterminals. To merge $X, Y \in N$, all instances of X and Y in the rules in R are replaced with a single symbol (this could just be X or Y instead of a new symbol). This generalizes the language of the grammar. Formally, this is developed by considering partitions of N and replacing the nonterminals in each block of a chosen partition by a single nonterminal; each partition gives a derived grammar whose language is guaranteed to contain E^+ since they are all derived from the canonical definite grammar. These are the "admissible grammars" for E^+. The grammars which also avoid including strings from E^- are then a subset of the admissible grammars. If no other constraints are placed on the form of a solution, any of the grammars in this subset is a solution to the grammar induction problem given training data E.

Since the subset of the admissible grammars for E^+ which are also consistent with E^- is usually too large to enumerate practically, we need some mechanism for constructing smaller collections of admissible grammars which can be checked against E^-. Two such methods cited by Fu and Booth construct grammars based on formal derivatives and k-tails. The formal derivative of a set of examples with respect to a terminal symbol is the subset of the examples which end in the symbol with that trailing symbol removed. The distinct formal derivatives of the positive training data are used as nonterminals, and the relationships among them lead to the formation of productions. If set X is the formal derivative with respect to t of set Y then we add a production of the form $Y \rightarrow Xt$ The k-tails formulation identifies formal derivatives when the last k symbols of their strings are the same and is a more limited version of the formal derivative approach to constructing a grammar. In both cases, a definite grammar for E^+ is constructed and it and all of its derivative grammars form a set. The risk for these methods is that the only grammar that is both in the admissible class and consistent with E^- might be the original definite grammar.

III.B.3 Context-free Grammar Induction

Context-free grammar induction is much less well understood than regular grammar induction. This is mainly because the forms of the productions in context-free grammars allow center-embedding and repetition of nonterminals. By contrast, regular grammars always add a symbol to the end of a string and leave a new nonterminal after it with each step of a derivation. In a context-free grammar, we might add symbols at the beginning and end of a string in the same derivation step. The broader set of possible productions leads to a large set of possible definite grammars, and the ability to add terminals before and after a nonterminal with a single derivation step renders formal derivatives as defined above useless.

The inference algorithms that Fu and Booth report for context-free grammars are limited to specific types of grammars and are often at least in part heuristic. One important antecedent of these methods due to Solomonoff is a method for discovering nesting of substrings and is based on the availability of a "teacher" who can tell us for any string whether it is in the target language or not. This method relies on the Pumping Lemma, which states that there are substrings, and also pairs of substrings, of strings in a context-free language which can be replaced with several replications of themselves, yielding another string in the language [61, 2]. Solomonoff's strategy is to explicitly search for such substrings using the teacher, and works as follows:

1. Delete substrings of some member of E^+ and ask the teacher about these strings.

2. For the strings that were still in the target language after substring deletions, replace the substrings that were removed with various numbers of repetitions of those substrings and ask the teacher about these new strings.

Whenever all of the new strings from the second step are identified by the teacher as in the target language, a recursive construction is formed. This basic idea is rather vague, but is noted by Fu and Booth to have inspired a great deal of later research. More importantly, it is based on the need to identify structure associated with one of the distinguishing characteristics of the generalization from regular to context-free languages: center-embedding.

An important variant of the basic induction task occurs when E^+ is a *structural sample*: a positive sample from a language which has parentheses around the substrings derived from specific rules. That is, it is a sample derived from a grammar which is built out of a grammar for the target language by taking rules of the form $A \rightarrow \alpha$ and replacing them with rules of the form $A \rightarrow [\alpha]$. Thus, the recursive structure of the strings is marked by $[,]$ pairs. Fu and Booth describe the Crespi-Reghizzi method for inferring grammars from structural information samples. This is developed for a special case in which there is a known method for placing the $[,]$ pairs in the strings of E^+ without actually knowing a grammar for the target language. This is the case where we are dealing with operator grammars and can parenthesize around the arguments of individual operators, at least in some cases. The method is also generalized for more general grammars than the special case but it is not clear how a structural sample would become available in the more general case.

Another important method for context-free inference cited by Fu and Booth is limited to the inference of *pivot* grammars. Pivot grammars contain only two kinds of productions. Using the same notation as in figure III.2, these are $N \rightarrow N\Sigma_p N$, $N \rightarrow N\Sigma_o$, $N \rightarrow \Sigma_o N$, and $N \rightarrow \Sigma_o$. Here, the set Σ is partitioned into Σ_p, the "pivot" terminals, and Σ_o. The pivot grammars are a subset of context-free grammars that contains the linear languages[4]. Thus, the pivot grammars include grammars for languages that are not regular, but also do not account for all context-free languages. Unlike the Crespi-Reghizzi method, the pivot grammar inference method is straightforward and works from a non-structural sample rather than a structural one.

[4]The linear languages are generated by linear grammars, which have productions of the forms $N \rightarrow \Sigma N \Sigma$ and $N \rightarrow \Sigma$.

III.B.4 Other Prior Work

Gold's results can be viewed as somewhat pessimistic since they say, in essence, that inducing context-free grammars from positive examples is impossible. An interesting question is then: are there subsets of the context-free grammars that contain languages that are not regular and that are efficiently identifiable?.

Burago proved that there is a subset of the context-free languages for which grammars can be learned efficiently, but in a model different from the identifiability in the limit model used by Gold[17]. The grammars learnable by Burago's method are the structurally reversible languages[17]. A grammar is structurally reversible if for two strings of nonterminals α and β, α and $\alpha\beta$ derive disjoint collections of terminal strings. That is, of all the nonterminal strings from which a given terminal string can be derived, none is a prefix of any of the others. In Burago's work, the learner is required to learn a grammar for a language in an amount of time polynomial in the size of the training data and is allowed to use membership and equivalence queries [5, 6]. Membership queries are the same as Solomonoff's teacher: for any string, a membership query labels it as being a positive or negative example for the target language. Equivalence queries are much more powerful: given a guessed grammar, an equivalence query either answers "yes" if the guess is indeed correct, and otherwise returns a counterexample: a string that is incorrectly either accepted or rejected by the guessed grammar. This result thus concerns a learning model in which the learner has a much greater source of information than in Gold's case. Since it uses equivalence queries, Burago's work is of mainly theoretical interest, but it does at least establish learnability of a subset of the context-free grammars, and that using information that previously was shown to be sufficient to learn regular languages.

Some other recent efforts in applied grammar induction in many ways are still very similar to the classic techniques cited by Fu and Booth. For regular languages, many methods are still based on the idea of deriving grammars from some particular definite grammar, or, equivalently, the idea of searching for a partition of the nonterminal set for a definite grammar constructed from the positive training data (see [29], for example). Context-free grammar induction methods are still mainly heuristic[83, 78], though there has also been some work done trying to define generalizations of the relatively well-understood regular inference methods to context-free grammars[69]. Stolcke and Omohundro's work is an interesting example of an approach based on state-space search techniques well-known to the artificial intelligence community.

A recent book by Charniak is primarily concerned with stochastic grammar induction, but is also concerned with learning nonstochastic context-free grammars to describe natural language [19]. Another recent example of an attempt to learn a grammar for a natural language uses a database of parse trees, called "TreeBank," to infer a grammar. This is somewhat like learning from a structural information sample, discussed above, but the sample is very large and parse trees contain even more information than a structural information sample since they include information about nonterminals deriving specific substrings of positive examples [20].

III.B.5 Grammar Induction as an approach to machine learning

Learning grammars is related fundamentally to learning computations since formal grammars are fundamentally related to state machines. Notably, for our purposes here, finite automata are equivalent to regular grammars in that both describe the same set of languages. Push-down automata and context-free grammars are similarly equivalent, and at the top of the Chomsky hierarchy, the Turing-acceptable languages can be described by unrestricted phrase-structure grammars. Thus, learning grammars is basically the same as learning automata.

In the case of learning, if we have a collection of strings labelled as positive and negative instances, we can imagine learning a grammar that distinguishes the positives from the negatives and use the parse trees assigned to the strings as an "explanation" of their membership or lack of membership in the language. Thus, we can use grammars to learn the syntactic relationships in a data set with the aim of ending up with a grammar to classify the strings in the data set. The higher in the Chomsky hierarchy from which we draw candidate grammars, the more complex the patterns we can capture and exploit for classification. Unfortunately, the cost of parsing also increases as we climb the Chomsky hierarchy. The parsing task is computationally infeasible at the context-sensitive level and impossible for the unconstrained grammars. The field of study that seeks to exploit this line of reasoning is known as syntactic pattern recognition [34, 28].

For most applications of grammar induction, the basic idea is to generate grammars by some heuristic method and test them against the training data, E. This means we need to be able to ask for $(\sigma, \text{label}) \in E$ and candidate grammar Γ, "Is $\sigma \in \Gamma$?" If the label and grammar agree with respect to this question of membership, we give a reward to Γ for σ, and impose a penalty on mismatches. It is also useful to know what the parse trees for σ given Γ are in order to examine the structure imposed on σ by the grammar. In the event that Γ fails to parse a positive example, we may want to give it some credit if it parses a substring.

As will be seen below, stochastic grammars define probability distributions over strings, with each distinct parse tree of σ contributing to the probability of σ. In this case, the question becomes "What is the probability of σ given Γ?" This approach to applying grammar induction corresponds to the lower-left cell of table II.1: the candidate grammars are used as recognizers of strings generated by the presumed target grammar.

The traditional machine learning concerns apply to grammar induction: we have some training data, on which we will minimize our learned grammar's training error as best we can. We will then judge the learned grammar based on how well it performs on test data on which it was not trained. If many grammars achieve equivalent performance, we will often want the simplest grammar, or the one with the shortest description. Savitch's notion of "essential infinity," introduced in section II.B, informs our expectations about what sort low description-length grammars can be found. If the

The sorting network application is a special case and falls in the upper-right cell of table II.1. Sorting networks are very simple algorithms to sort arrays of data and are given as a fixed sequence of operations of a fixed type on pairs of locations in the array; this is discussed further in a later chapter. For now, note that we can succeed in inducing a grammar to describe such algorithms to the extent that there are regularities in the class of sorting networks which allow networks on small arrays to be incorporated into networks to sort large arrays, for example. As will be seen below, such modularity of construction is an important characteristic of this class of sorting algorithms. This approach also resembles the work of Solomonoff[76] cited by Fu and Booth, in which a teacher was available to determine the membership of a given string. In our case, however, we will be generating sorting networks from grammars and using as the "teacher" a procedure of evaluating the resulting networks against test cases — by contrast, in the cited idea from Solomonoff's work, the teacher was not applied to strings generated by a candidate grammar, but rather to strings constructed by modifying the training data.

Computational learning theory[5] results give expressions for the size of a training set such that a hypothesis consistent with it is probabilistically guaranteed to have less than some error with respect to the target language. These bounds are based on either the size of the class of target languages, or a dimensional measure on the target class known as the VC-dimension, which is the size of the smallest training set with which at least one hypothesis is inconsistent. We will return to this later, in section III.D.

In Occam learning, the size of the training set is bounded by requiring that hypotheses be of bounded description length based on the number of training examples, m, and the shortest representation of the target language, r: $|\Gamma| \leq m^{\alpha} r^{\beta}$, where $|\Gamma|$ is the description length of grammar Γ in the learning algorithms representation. α must be less than 1, thus requiring some degree of data compression from a successful hypothesis, and β must be greater than or equal to 1, since, ex hypothesi, r is

[5] For a general introduction to computational learning theory and the VC-dimension, see [65] and [7]. [45] and [79] are two good original sources.

minimal. The ability to efficiently find a grammar satisfying this description length constraint can be shown to imply efficient learnability [7][6]. Generally, β must be larger than one since finding a minimal length solution is often NP-hard. The idea here is to reduce the length of the hypotheses under consideration so as to gain a useful bound on training sample size by eliminating hypothesis that have long (and thus often more complex) descriptions. This bound is like the naive one based on hypothesis class size, but is tighter, or even just finite in this case, because the hypothesis class has been reduced.

In practical cases, of course, we won't know the target language's actual definition and will have to work with training and test sets to arrive at and evaluate hypothesis grammars. In such cases, we find the best grammar we can by using the training data as the input to the learning algorithm and testing the generalization of the learned grammar using an independent sample, the "test set;" the test set is not used during learning. Even with these expedients, the training data will not usually eliminate all but one grammar. Biases built into our learning algorithms must provide the choice among equally good grammars. Thus, we will do better at learning some target grammars than others, and may not be able to learn some targets at all.

The lesson here is that being clever about searching for simpler, or at least shorter, descriptions of grammars rewards us in the form of a training set size bound, *if* we can carry out the Occam search for a short hypothesis consistent with the data efficiently. Equivalently, if we have an efficient means of fitting briefly describable hypotheses to training data, then we have some knowledge about training set size.

III.C Stochastic grammar induction

Stochastic grammars are an extension of their nonstochastic counterparts that allow grammars to be used to define probability distributions over strings instead of indicator functions over strings. For some very simple forms of regular grammars

[6] Natarajan presents a similar result in which r is the minimum length of any representation of any hypothesis consistent with the training data [65].

this yields Markov models; stochastic regular grammars as a whole are equivalent to Hidden Markov Models (HMMs). Stochastic context-free grammars are more general than HMMs and correspond to stochastic push-down automata. To begin, we need a precise definition of a stochastic grammar:

Definition 3 *A stochastic phrase-structure grammar (a "stochastic grammar" hereafter) is a 5-tuple* $\Gamma = (\Sigma, N, S, R, \rho)$ *where:*

> *The 4-tuple* $\mathrm{Ch}(\Gamma) = (\Sigma, N, S, R)$ *is a grammar as per definition 1;* $\mathrm{Ch}(\Gamma)$ *is called the* characteristic grammar *of* Γ.
>
> $\rho : R \mapsto \Re$ *is a function assigning probabilities to the rules in* R*, such that* $\forall r \in R, \rho(r) \neq 0$. *There are constraints on* ρ*, discussed below, needed to be sure that the probability distribution that* Γ *induces over* Σ^* *is well-defined.*

The precise manner in which the addition of ρ allows Γ to induce a probability mass function over strings will be described later by generalizing the examples from the previous section into examples of stochastic grammars. The basic idea behind the definition of a stochastic grammar is that we can annotate the rules of a grammar with probabilities and turn a characterization of a set into a probability distribution over strings. The grammar to which we add probabilities becomes the characteristic grammar of the new stochastic grammar, and defines the set of strings which can have probabilities greater than zero. The distinction between strings which are and are not in the language of the grammar becomes a distinction between those that are assigned probabilities greater than zero and those that are assigned probability zero.

A stochastic language is a language generated by a stochastic grammar's characteristic grammar, plus a probability mass function over strings. We shall notate the language $\mathcal{L}(\mathrm{Ch}(\Gamma))$ as simply $\mathcal{L}(\Gamma)$. The probability distribution for a language $\mathcal{L}$ will be denoted $\mathrm{Pr}_{\mathcal{L}}$. When $\mathcal{L}$ is the language of a grammar, we will often just subscript with the symbol for the grammar for convenience, and similarly if we are discussing several languages distinguished by subscripts, we will indicate the corre-

Γ'_{parity}: **Stoch. even parity**

Start symbol: S

r'_1	S	$\rightarrow$	0S	$(\frac{1}{3})$
r'_2	S	$\rightarrow$	1T	$(\frac{1}{3})$
r'_3	S	$\rightarrow$	0	$(\frac{1}{3})$
r'_4	T	$\rightarrow$	0T	$(\frac{1}{3})$
r'_5	T	$\rightarrow$	1S	$(\frac{1}{3})$
r'_6	T	$\rightarrow$	1	$(\frac{1}{3})$

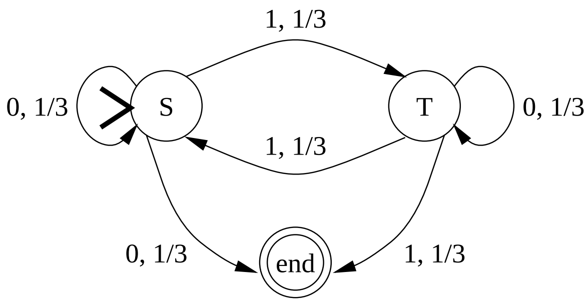
(a)

Γ'_{parens}: **Stoch. parentheses**

Start symbol: S

r'_1	S	$\rightarrow$	TS	$(\frac{2}{3})$
r'_2	S	$\rightarrow$	T	$(\frac{1}{3})$
r'_3	T	$\rightarrow$	(T)	$(\frac{3}{4})$
r'_4	T	$\rightarrow$	()	$(\frac{1}{4})$

(b)



(c)

Figure III.4: Two example stochastic grammars. (a)'s characteristic grammar is regular and (a) is thus equivalent to the HMM, (c), shown below it; (b)'s characteristic grammar is context-free.

sponding probability mass functions by writing the same subscripts on the appropriate instances of Pr.

Figure III.4 contains the grammars from figure III.3 with probabilities attached to the productions. For each rule $r \in R$ the value of $\rho(r)$ is written in parentheses to the right of r. The grammars from the previous section are thus the characteristic grammars of these two stochastic grammars, and the rule labelled r'_i in figure III.4 corresponds to the rule labelled r_i in figure III.3. The derivations and parse trees would be the same as before and are not shown here. The new information in these stochastic grammars is related to the probabilities.

To see how the probabilities of strings arise from stochastic grammars, consider again the parses shown for the two strings in figure III.3. Each parse has associated with it a probability that is the probability of the event defined by the conjunction of the events of applying each rule that goes into creating the parse. That is, whenever a given nonterminal exists in a string derived so far, the probability of deriving that string from the start symbol will be the product of the probabilities of the rules applied so far. For each distinct parse of a string, there is a probability associated which is equal to the products of the probabilities of the rules used in a derivation corresponding to that parse tree. The two strings derived from the sample grammars previously were 001010 and ()(()). The sequences of rules from our example stochastic grammars corresponding to the rules in the nonstochastic grammars which derive these are $r'_1, r'_1, r'_2, r'_4, r'_5, r'_3$ and $r'_1, r'_4, r'_2, r'_3, r'_4$, respectively. Thus, the probabilities of these strings are:

$$
\begin{aligned}
\Pr(001010|\Gamma'_{\mathrm{parity}}) &= \rho_{\mathrm{parity}}(r'_1)\rho_{\mathrm{parity}}(r'_1)\rho_{\mathrm{parity}}(r'_1)\rho_{\mathrm{parity}}(r'_2) \\
&\quad \rho_{\mathrm{parity}}(r'_4)\rho_{\mathrm{parity}}(r'_5)\rho_{\mathrm{parity}}(r'_3) \\
&= \left(\frac{1}{3}\right)^7 \\
\Pr(\ ()(())\ |\Gamma'_{\mathrm{parens}}) &= \rho_{\mathrm{parens}}(r'_1)\rho_{\mathrm{parens}}(r'_4)\rho_{\mathrm{parens}}(r'_2)\rho_{\mathrm{parens}}(r'_3)\rho_{\mathrm{parens}}(r'_4) \\
&= \left(\frac{2}{3}\right)\left(\frac{1}{4}\right)\left(\frac{1}{3}\right)\left(\frac{3}{4}\right)\left(\frac{1}{4}\right)
\end{aligned}
$$

by multiplying the rule probabilities given in figure III.4. The ρ functions belong to the grammars whose subscripted names they share and should be understood to encode the rule probabilities given in the figure. Since each string has only one parse for the grammar which derives it, these are equal to the probabilities of the strings themselves according to the grammars.

The grammar $\Gamma'_{\mathrm{parity}}$ applies the same probability to each string of a given length that it derives. $\Gamma'_{\mathrm{parens}}$, however, assigns greater probability to strings close to having an average of 3 repeated parenthesized expressions generated from any given instance of the start symbol expanded during a derivation, and an average nesting depth of 4 within each parenthesized expression; this is reflected by the probabilities on rules 1 and 2 in the first case and rules 3 and 4 in the second case. Thus, we can see from these examples that the particular probability settings can place emphasis on some features of the strings in the language of a stochastic grammar.

In general, suppose that string σ can be derived from stochastic grammar Γ. Further, suppose there are n different parses and that we know the rules in a derivation for each parse, and that the number of rules used in the i-th parse is n_i. If $r_{i,j}$ is the j-th rule in a derivation for the i-th parse, then the probability of σ under the distribution induced by Γ is:

$$\Pr(\sigma|\Gamma) = \sum_{i=1}^{n_p} \prod_{j=1}^{n_i} \rho(r_{i,j}) \tag{III.2}$$

where ρ is Γ's rule probability function. This equation gives us a definition of the mass function $\Pr(\Sigma^*|\Gamma)$. One feature of the function is that longer strings tend to be less probable. While it is not true that of two strings, the shorter always has higher probability, it is often so when the difference in the strings' lengths is large. For the grammar Γ'_{parity}, the relationship is always true.

One important consideration is the need for some way of knowing that $\Pr(\Sigma^*|\Gamma)$ is a true probability mass function. That is, we want to ensure that:

$$\sum_{\sigma \in \Sigma^*} \Pr(\sigma|\Gamma) = 1$$

for all $\sigma \in \mathcal{L}(\Gamma)$. For $\sigma \notin \mathcal{L}(\Gamma)$, $\Pr(\sigma|\Gamma) = 0$. Fu and Booth refer to Γ as *consistent* under these circumstances. This is just the situation in which $\Pr(\Sigma^*|\Gamma)$ is a probability mass function over Σ^*. If Γ has the property that each set of productions having the same left-hand side contains productions whose probabilities sum to one, then Γ is called *proper* [34]. For regular and context-free grammars, a property of derivations called path-independence assures us that if a grammar is proper than it is consistent[7]. Further references to stochastic grammars will always assume that they are no higher in the Chomsky hierarchy than context-free and that they are proper, unless specific mention is made to the contrary.

[7]Fu and Booth refer to path-independence as "unrestrictedness," but this seems too confusing since grammars with no constraints on the form of their productions are often referred to as unrestricted (see [34], section III)

The stochastic grammar induction problem is very much like the nonstochastic problem. Once again, we have a set E of training examples, each of which is a pair. Now, however, we want to assign more than a plus or minus label to each example. Each pair must now inform us about a string's probability, so we might write $(\sigma, p) \in E \subseteq (\Sigma^* \times [0,1])$ where $p = \Pr(\sigma|\Gamma)$. In fact, the probabilities themselves will not usually be available. What we will have instead is a sample of strings generated according to the distribution over strings given by the grammar. That is, a sample generated according to the distribution $\Pr(\sigma|\Gamma)$ over Σ^*; this distribution will be referred to as $\Pr(\Sigma^*|\Gamma)$. Thus, what we really have is a sample $E \subseteq (\Sigma^* \times \aleph)$ of pairs (σ_i, c_i) where c_i is the *count* of σ_i, or the number of times σ_i occurred in E, when

$$N = \sum_{|E|}^{i=1} c_i$$

strings were independently chosen according to the distribution $\Pr(\Sigma^*|\Gamma)$. Our sample E has thus become a multiset, and where in the previous section there was one notion of sample size, $|E|$, we must now distinguish between the number of distinct strings in the sample, $|E|$, and the number strings generated to produce the sample, N. We will usually be interested in N instead of $|E|$ in discussing sample size with respect to stochastic grammars.

If we actually generated E from a stochastic grammar, it would never contain any pairs $(\sigma, 0)$, where the count for a string was zero. This is because the grammar can only derive strings with nonzero probability. So any string with zero probability would never be derived. It may still be useful to include such entries in E for purposes of a learning algorithm. Even though moving from nonstochastic to stochastic grammar induction implies a shift from learning indicator functions to learning distributions, we are seeking to learn distributions whose definitions are structured by their characteristic grammars. Thus, we still wish to learn a set of rules that meets the requirements of equation III.1. In fact, our goal now has an additional requirement and becomes:

$$\text{(i)} \qquad \mathcal{L}(\Gamma_{\text{est}}) \cap E^+ = E^+$$

$$
\begin{array}{ll}
\text{(ii)} & \mathcal{L}(\Gamma_{\text{est}}) \cap E^- = \emptyset \\
\text{(iii)} & \Gamma_{\text{est}} = \arg\max_{\Gamma} \Pr(E|\Gamma)
\end{array}
\tag{III.3}
$$

where we now define the sets E^+ and E^- slightly differently:

$$
\begin{aligned}
E^+ &= \{\sigma|(\sigma, c) \in E, c > 0\}, \text{ and} \\
E^- &= \{\sigma|(\sigma, 0) \in E\}.
\end{aligned}
$$

This still preserves the basic idea behind the definitions of these sets: the strings in E^+, the positive examples, are those strings which are in the language of the characteristic grammar of the target stochastic grammar, and the strings in E^- are those strings not in that language. Thus, we can view stochastic grammar learning as being the same as the nonstochastic problem with a built-in preference over the grammars meeting our previous goal of equation III.1: those that maximize the likelihood of the data given the grammar, as is reflected in the third constraint in equation III.3.

In fact, there *is* some suggestion, based on our new learning goal, that negative examples might not be necessary[34, 19]. For a stochastic grammar Γ, in order to increase the probability under $\Pr(\Sigma^*|\Gamma)$ of some strings, we must reduce the probability of some other strings, since the sum of the probabilities of all strings in Σ^* must be 1. This suggests that some of the information in negative examples might already be present in the information about string probabilities contained in the counts of a stochastic sample E. We can only maximize the probability of the positive examples given their counts by applying zero probability to the negative examples. It does not, however, imply that some direct penalty for assigning nonzero probability to negative examples will not be useful in speeding up or shaping our search for a solution. Also, if our goal is to find a grammar to separate an E^+ and an E^- which are not complements of one another in Σ^*, the negative examples are still a necessary part of the problem definition. In this case, the stochastic problem becomes one of looking for a "separating grammar" which assigns high probability to E^+.

The examples in this section were chosen to illustrate an important point about stochastic grammars that was mentioned earlier. Refer back to figure III.4(a), which shows a stochastic regular grammar(SRG) which assigns positive probability to only those binary strings with even parity. Being a stochastic regular grammar, it corresponds to an HMM, as was noted earlier. The equivalent HMM is illustrated in figure III.4(c); each arc is labelled with a symbol and a probability. In constructing an HMM equivalent to the grammar, we make each nonterminal a state of the HMM, and each production an arc. A subset of the stochastic regular grammars corresponds to ordinary Markov models: those grammars have the same terminal on the right-hand side of all productions with a common left-hand side. Note that to construct the automaton from the grammar it was necessary to add a special extra state, "end," as an accepting state. All terminal productions in the grammar are transitions to "end." Epsilon productions would cause the states corresponding to their left-hand sides to be final states.

The grammar in figure III.4(b) *cannot* be written as an HMM since we can't simply make each nonterminal a state of an HMM. The reason is that the HMM without its probabilities is just a finite automaton, and finite automata correspond to regular grammars. Since the characteristic grammar in figure III.4(b) is context-free, the stochastic grammar of which it's a part cannot be written as an HMM.

There is, of course, a cost to working with the more complex representation of SCFGs and the cost is related to determining, for a string σ and stochastic grammar Γ, the value of $\Pr(\sigma|\Gamma)$. The added cost actually accrues to the need to determining whether $\sigma \in \mathcal{L}(\Gamma)$, which must be done to find the probability. This issue is discussed in detail in section B..1, but for now note that if σ is of length n, then answering the question when $\mathrm{Ch}(\Gamma)$ is regular takes time $O(n)$, whereas for a context-free grammar the time dependency is $O(n^3)$. Further, for a regular grammar, answering the question costs $O(n)$ space whereas it takes $O(n^2)$ space for context-free grammars. These are worst case time and space requirements and there are subsets of the context-free grammars for which the time and space are linear, given an appropriate parser with which to assign parses to strings, such as the LR(k) grammars.

The reader may ask at this point: if we're willing to spend more time, can we work with still more complex grammars? The answer is that it becomes computationally untractable to answer our basic membership question for context-sensitive grammars and impossible for fully general phrase-structure grammars[2, 61].

Though SRGs cannot capture as broad a class of distributions as SCFGs, there is a result concerning approximation given by Fu and Booth[8]:

Theorem 2 (Theorem 6.2 from [34], section VI) *Any consistent stochastic context-free or finite-state language can be strongly ϵ-approximated by a stochastic finite-state language which is generated by an unambiguous stochastic finite-state grammar.*

$\mathcal{L}_2$ strongly ϵ-approximates $\mathcal{L}_1$ if the following conditions are satisfied:

$$\sum_{\sigma \in (\mathcal{L}_1 \cap \mathcal{L}_2)} \mathrm{Pr}_1(\sigma) \geq 1 - \epsilon$$

$$\sum_{\sigma \in (\mathcal{L}_1 \cap \mathcal{L}_2)} \mathrm{Pr}_2(\sigma) \geq 1 - \epsilon$$

$$\mathrm{Pr}_1(\sigma) = \mathrm{Pr}_2(\sigma), \sigma \in (\mathcal{L}_1 \cap \mathcal{L}_2)$$

where Pr_1 and Pr_2 are the probability distributions over the strings in $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively. Strong ϵ-approximation means that the total probability on the intersection of the two languages according to either language's distribution is at least $1 - \epsilon$ and that the probabilitys *exactly* agree for every string in the intersection. Note that the third condition along with either of the first two suffices for all three to be true. The finite-state grammars which accomplish this approximation for context-free grammars still cannot capture the dependencies existing in context-free languages. Rather, the approximation is accomplished by taking a large finite subset of the language to

[8]Note that the term "finite-state" is used instead of "regular" here.

be approximated and constructing an automaton that assigns that large finite subset the probabilities assigned by the approximated language; the subset must have total probability at least $1 - \epsilon$ on the approximated language. Such a finite subset can always be found for a consistent stochastic language, basically because string probabilities, on the whole, shrink with string length for stochastic regular and context-free languages.

The problem with the approximation result is that it doesn't satisfy our desire to discover patterns in the data. We want to discover a stochastic grammar Γ for our data such that $\mathrm{Ch}(\Gamma)$ assigns parses to strings in a way that not only approximates the probability distribution from which they were drawn, but also assigns the correct syntax to each string, or at least comes close to doing so. That is, we want the characteristic grammar of our proposed solution to be a good solution to the nonstochastic grammar induction problem with respect to our data. These goals are, in fact, related. If we have data generated by a context-free grammar which cannot in turn be exactly described by a regular grammar, we will only ever find the maximum likelihood grammar for the data by searching for context-free grammars, since the distribution over strings induced by such a SCFG cannot be encoded by an SRG. The approximation result only indicates the ability to heavily overfit the data.

III.D Structurally and statistically good grammars

One issue that is central to any machine learning endeavor is the design of a reasonable figure of merit for proposed solutions to the learning problem. In grammar induction, this issue becomes: "What makes one grammar better than another, relative to a given data set?" This question needs to be answered as part of the effort of producing a fitness function for an evolutionary algorithm, as will be seen in chapter IV. In this section, we address this question by discussing two general ideas of how one grammar can be better than another: "structural goodness" and "statistical goodness." In the course of the discussion, we will need to discuss more specific quality measures which modify or help to define these two general ideas.

Recall that we mentioned the standard machine learning concerns previously in section III.B.5: a good grammar is one that, having been trained to minimize error on training data, minimized test error. As also mentioned previously, we may also prefer a low-complexity grammar over a high-complexity grammar. In this section, we will be more interested in the interplay of two different notions of a grammars goodness, the structural notion of how well the grammar models syntax and data patterns, and the statistical notion of how well the grammar does at assigning the correct strings high probabilities.

A "structurally good" grammar, Γ, is a grammar that captures any "syntax" implied by the training data. The productions capture patterns within the training data. To the extent that $\mathcal{L}(\Gamma)$ has larger cardinality than the training set, these patterns are used to generalize to a larger set of strings. When $\mathcal{L}(\Gamma)$does not contain all of the training data, we are missing some patterns implied by the data. If Γ is a stochastic grammar these conditions apply to its characteristic grammar, $\mathrm{Ch}(\Gamma)$. A more specific definition must take into account the ability of a grammar to correctly classify the training data and to generalize correctly from that data. Thus, the notions of consistency of a hypothesis with training data and generalization to strings outside of the training data comprise the definition of "structurally good." For the case of learning nonstochastic grammars, one component of the figure of merit would simply be the percentage of the training examples correctly classified; as we will see below, the stochastic case is slightly more complicated.

The desired grammar for a nonstochastic target language is the most structurally good grammar possible. Failing this, we would like the grammar consistent with the training data that is the best generalizer possible. Unfortunately, both the hypothesis space and its VC-dimension are infinite in the case of regular and context-free grammars unless some further restrictions are placed on the grammars. Limiting the total description length of the grammar, the total number of productions allowed, the total number of nonterminals allowed, or the forms of the productions (such as only allowing grammars in which productions have right-hand sides containing a single nonterminal) can make the VC-dimension finite. The infinite dimension arises because for any number of labelled examples, we can construct a grammar with start symbol S which has a production $S \rightarrow \sigma$ for every training example which we wish to label positive. This grammar accepts all and only the positive examples. Thus, for any collection of strings, we can construct a grammar which labels any subset of them positive and all other strings negative.

This argument requires more productions as the number of examples increases, and the length of the grammar increases. Restricting either will yield a class of grammars which must at some point fail to account for all labellings. Limiting the number of nonterminals allowed doesn't work unless we limit either the overall description length of the grammar or the number of productions, either of which will have the effect of limiting the number of nonterminals. It is this limitation, however, that eventually forces generalization and starts to collapse the version space. Other methods of qualifying grammars to make the version-space manageable employ the idea of grammars in reduced form[80, 36].

If we limit our class of grammars by restricting rule right-hand sides, some caution must be exercised in choosing the restrictions, since many apparent restrictions don't reduce the language class at all, even if they do reduce the grammar class; this follows from the existence of normal forms such as Chomsky's and Greibach's. Both normal forms place restrictions on the form of production right-hand sides. Any restriction that permits at least the rules either normal form permits will be allowing all context-free languages to be represented. In the case of Chomsky Normal Form, only N $\rightarrow$ NN and N $\rightarrow$ T are required, where N is any nonterminal and T is any terminal. The idea of bounding based on hypothesis class size for low description-length hypotheses bring us to "Occam" learning, discussed above in section III.B.5.

"Statistical goodness" applies only to stochastic grammars as the term is used here. It refers to the ability of a grammar to capture the probability distribution from which the training data, a stochastic sample, was drawn. A stochastic grammar assign a probability to each string $\sigma \in \Sigma^*$. Recall from section III.C that if $\Pr(\sigma|\Gamma) = 0$ then σ is not in the language of $\mathrm{Ch}(\Gamma)$; otherwise, Γ assigns some positive probability to σ and the string is in the language of the characteristic grammar.

A statistically good grammar, Γ, can be thought of as maximizing the probability of the training data given Γ, and also as minimizing the difference between the probability distribution over strings induced by Γ and the training data's probability distribution. Three figures of merit have traditionally been used to these ends: χ^2, log-probability, and cross-entropy [34, 19]. The χ^2 measure is:

$$\frac{\sum_{\sigma \in E}(|E|\Pr(\sigma|\Gamma) - c(\sigma))}{|E|}$$

where the function c counts the number of occurrences of its argument in E. The log-probability measure is just:

$$\log \Pr(E|\Gamma),$$

and the cross-entropy measure is the quantity: $\sum_{\sigma \in E} \Pr(\sigma|\Gamma_{\mathrm{target}}) \log \Pr(\sigma|\Gamma)$ which is a difference metric for the probability distributions arising from a grammar for the target language and the grammar Γ.

Since we can't know $\Pr(\sigma|\Gamma_{\mathrm{target}})$, it is usually estimated in practice using the frequencies of the training examples within E. The equivalence of this measure and log-probability in particular are discussed in [19, 50]. All three measures are essentially equivalent in the sense that they have the same basic shape as a function of grammar production probabilities when the grammar productions are held constant: all three reach their extrema at the same production probabilities and all three are bowl-like functions in which the radius fluctuates.

Structural and statistical goodness are strongly related to one another in stochastic grammars: for a stochastic grammar Γ, the structural goodness of $\mathrm{Ch}(\Gamma)$ and the statistical goodness of Γ are linked, though one doesn't determine the other absolutely. Here we compare and contrast the notions of structural and statistical goodness.

A structurally good stochastic grammar would assign zero probability to all negative examples, and this grammar would also be statistically better than one which assigned nonzero probability to some negative examples, since any probability transferred onto negative examples must be taken from positive examples (recall the discussion of negative examples from the previous section). Thus, being structurally good is a precondition for a stochastic grammar to be statistically good. A statistically good grammar might instead assign nonzero probability everywhere, but the distribution would peak around certain sets of strings — the positive examples. Note that since probability basically decays exponentially with string length, it is enough to distinguish negative from positive examples if among strings of the same length negative examples are much less probable than positive examples. Thus, in the case where a structurally good grammar is not learned, a statistically good grammar might still allow positive and negative examples from the target language to be distinguished.

Statistically good grammars will often be structurally good grammars if they achieve a reasonable degree of *compression* of the data. Imagine that we use a stochastic grammar to separate strings by log-odds scores:

$$\log \frac{\Pr(\sigma|\Gamma)}{\Pr(\sigma|\Gamma_{\text{Random}})}$$

The random grammar in the denominator generates terminal symbols independently according to a fixed distribution. To the extent that the grammar is structurally good we will get higher log-odds scores, since structurally good grammars assign more probability to strings than relatively structurally worse grammars, and this increases the numerator of our odds ratio. To the extent that the grammar is statistically good and isn't just growing in such a way that it is memorizing the data, it should be achieving higher log-odds scores by taking into account information in the sample about conditional probabilities among letter probabilities. This is described in more detail as it applies to the experiments in this thesis in chapter IV, and investigated empirically in chapters V and VII.

It is interesting to note that the ideal statistical grammar is structurally very bad. This grammar would have a distinct set of nonterminals and productions for each positive example string and its language would contain *only* the positive example strings. This grammar allows the positive training data, E^+ to be assigned probability 1.0, but is structurally very bad since it badly overfits the data[9]. Thus, there is a trade-off between structural and statistical goodness, since the statistically best possible grammar is an instance of extreme overfitting to training data. If we push for statistical goodness without taking structural goodness into account, we risk overfitting the data, and when there are several structurally *grammars*, we want one that is statistically best whenever possible.

In general, we expect learned grammars to display a balance of structural and statistical goodness, based on what training data is available and whether regularities in the data are amenable to capture in grammatical rules. There is some context-free regularity in context-sensitive languages just as there is some regular-language regularity among strings of many context-free languages. This is an important part of the expectation of a balance between statistical and structural description of the training data.

As an example, the context-sensitive language

$$\mathcal{L}_{\mathrm{abc}} = \{a^n b^n c^n | n \geq 1\}$$

contains strings which have substrings of the form $a^n b^n$ and $b^n c^n$. In fact, it is contained in the following three context-free languages, each of which it specializes:

$$\begin{aligned} \mathcal{L}_{\mathrm{ab}} &= \{a^n b^n c^k | n, k \geq 1\} \\ \mathcal{L}_{\mathrm{bc}} &= \{a^k b^n c^n | n, k \geq 1\} \\ \mathcal{L}_{\mathrm{ac}} &= \{a^n b^k c^n | n, k \geq 1\} \end{aligned}$$

and in the regular language:

$$\mathcal{L}_{\mathrm{a|b|c}} = \{a^k b^m c^n | m, n, k \geq 1\}$$

of which it is also a specialization. $\mathcal{L}_{\mathrm{abc}}$ is not context-free and thus cannot be learned as a context-free grammar, but a context-free grammar could be used to

[9] Unless the target language is finite and close in cardinality to $|E^+|$, in which case working with its extensional description is feasible.

capture some part of its syntax. A stochastic CFG could further be learned with probabilities so that on average, strings for which $k = n$ were most likely. This is a point that will be illustrated for a stochastic version of language $\mathcal{L}_{\text{abc}}$ with simulations in chapter V. In some cases we may even be able to recover the target language by intersecting context-free languages: intersecting any two of the three context-free languages above yields the target language which they generalize. The CFLs' lack of closure under intersection has been used to learn a context-sensitive language in at least one application that will be of interest in a later chapter [16].

The important point is that we can still learn useful structure even when the target language is not representable by the grammars we consider. We cannot always hope to find a structurally best grammar in such cases, but we can hope to find a grammar displaying a mixture of statistical and structural goodness. If, for example, we are learning context-free grammars, then we want the learned grammar to capture at least any available context-free structure in the data. For the $\mathcal{L}_{\text{abc}}$ example, we would prefer to learn any of the three context-free generalizations over learning the regular generalization.

III.E What makes a good sample?

In the previous section, we were concerned with judging the quality of hypotheses, in the form of grammars, with respect to how well they described a sample of strings from some formal language. In this section, we are in turn concerned with judging how well a given training sample represents the language from which it was drawn. In the next subsection, we will describe the things that we would like to be true of a training sample in order to have confidence that we will be able to use it to learn a grammar for the language from which it was drawn. The following section briefly describes some known methods for processing the training sample before it is given to the grammar induction algorithm, which are meant to make that learner's job easier.

III.E.1 Representativeness of the target

It is important to realize at the outset that if we limit ourselves to positive examples, we cannot hope to find a finite cardinality sample that will unambiguously identify the target language. This is a consequence of Gold's theorems regarding identifiability in the limit[37]. Hence we will certainly not be able to uniquely identify a language after a finite amount of time, since being able to do the latter implies being able to do the former. We must either include negative examples, or, as was mentioned in section III.C, attempt to leverage the stochasticity of the target to make up for a lack of negative examples.

A further consideration was introduced earlier in this chapter when we discussed structural completeness. Recalling definition 2, a structurally complete sample of $\mathcal{L}(\Gamma)$ is a positive sample of that language such that every production of Γ must be used in the derivation of at least one string in the sample. We can generalize this from a positive sample E^+ to a general sample E containing positive and negative examples. In this case, the positive part of E is must be structurally complete in order for E to be so designated.

Structural completeness is one of the most basic antecedents for successful grammar induction. If we didn't believe that all of the productions of some grammar for the target language were used in deriving the sample, we would certainly want to throw in more examples until this were so, or revise our assessment of the target language. Structural completeness can be qualified in two simple ways, both of which can make us more confident of our ability to learn a structurally good grammar from the training data.

First, consider adding the requirement that each production of Γ_{target} be *critical* for at least one string of E. That is, for each production there exists a positive training example that could not be derived without that production. This amounts to saying that there is a "witness" in E to the importance of each production of the target grammar. Second, we could require that if E contains negative examples, that its negative part, E^- contains one refuting negative example for every possible production not found in the target grammar (probably up to some maximum right-hand side length, of necessity). That is, E^- contains at least one witness against each production that should not be present.

It is impossible that we be able to apply both extensions to every training sample we build, since together they imply a finite sample that uniquely identifies the target grammar. Nonetheless, they encourage a general strategy of trying to include a breadth of examples in an attempt to end up with a sample including as many "witnesses" as possible. A balance must also be struck between breadth of examples and similarity of clusters of examples that will allow such context-free constructions as center-embedding to be recognized.

For a stochastic grammar, we want to impose other qualifications on the training data. Two that considerations that must be balanced are the distribution of production usage in the derivation of the sample strings, and the distribution of number of occurrences of each string in the sample. For simple grammars, it is easy to write down examples in which one of these two distributions matches that given by a target grammar or its language and the other does not. In figure III.4(b), the grammar shown admits many structurally complete samples of one string, such as ()(()). But this sample, in fact, satisfies neither distributional criterion just mentioned. A larger sample is needed to make both the number of occurrences of each positive example and the number occurrence of each in the training set commensurate with string and production probabilities, respectively. Satisfying these conditions for ambiguous languages can be difficult, since adding a string to the training data tends to add to the usage counts of many productions in order to account for all possible derivations of the added string.

An important question we would like to be able to answer is: "How many examples need to be in the training data?" In the previous section, we noted that the hypothesis space cardinality and VC-dimension for CFGs are both infinite. We also noted that the hypothesis space of grammars can have its cardinality reduced by restricting the grammars. The restrictions on grammars mentioned above, on overall length, number of productions, or right-hand side length and number of nonterminals, all tend to restrict the maximum number of productions in a grammar[10], and thus suggest a link to "Occam learning" [65, 7].

Several times so far, we have mentioned the possibility of exchanging the information we could obtain from negative examples for the information obtainable from occurrence counts of strings in a stochastic sample. In chapter V, we will discuss an example in detail for which simulations have been run to investigate the ability to trade one form of information for the other when learning stochastic grammars. For the moment, it should be mentioned that negative examples are likely to be more helpful in general with contributing to the discovery of structurally good grammars since rules that contribute to complete derivations of them lead to the grammar of which their a part being penalized. That is, a negative example σ unambiguously pronounces to some degree determined by our learning algorithm, against hypothesis grammars Γ when $\sigma \in \Gamma$. Negative examples contribute nothing to the tuning of probabilities in a stochastic grammar, however, and so for a fixed size training sample, we must decide how important structural and statistical goodness are when deciding whether and how many negative examples we need. This should be done in light of the trade-off between the precision of the statistic of a stochastic sample and the need for negative examples to guide the search for correct grammar productions.

III.E.2 Preprocessing the sample

In this final section of this chapter, we briefly mention three different strategies for manipulating the data before it is presented to a learning algorithm. These approaches are meant to either make the syntax of the language from which the data was drawn more apparent, or detect some simple context-free patterns in the data so that the learning algorithm can elaborate them rather than discover them.

The first two approaches are both designed to detect embedding relation-

[10] Limiting the number of nonterminals only accomplishes this if we also limit the maximum grammar production right-hand side length.

ships, though of different varieties. Solomonoff's approach involves deleting a substring of some positive example, and asking an oracle if the string we deleted it from is still in the target language [77]. If so, we "pump" the substring, replacing it with two copies of itself, and ask again. This approach is motivated by the "pumping lemma" for context-free grammars [61]. Fu and Booth discuss learning a special kind of grammar called a "pivot grammar" in which some of the terminals play distinctive roles in certain productions [33]. The approach here is to look for cases where one positive training example is a substring of another positive example. In this case, the larger string has the substring replaced by a new nonterminal and this process is repeated throughout the training data. A rule is eventually learned to associate the replaced substring with the new nonterminal. Thus, this early stage of the algorithm looks for cases of embedding in which one string is nested within another, since process may be repeated using a string on which a replacement was previously performed.

A different modification of the training data is based on the idea of a "bracketed corpus" or, as it was referred to previously, a structural information sample[19, 33]. Recall that in a bracketed corpus, the productions of the grammar, such as $N \rightarrow \alpha$, where α is a string of terminals and nonterminals, are replaced with productions of the form $N \rightarrow [\alpha]$. The inserted brackets serve to delineate substrings of example strings in such a way that their derivations are more apparent than they would have been otherwise. Bracketing does not always require access to a target grammar in advance — in some cases, such as that of arithmetic expressions, the bracketing is to some extent apparent without any embellishment (since there may be some parentheses to control subexpression order of evaluation).

All of three of these approaches to preprocessing training data inform our discussion of the previous subsection, regarding the features and limitations of training data for grammar induction. They all reflect the belief that if the training data has some feature that marks how the strings were derived, the learning task should be easier. This information may be added as a preprocessing step or to some extent may naturally occur, as in the example of arithmetic expressions. Thus, these "preprocessing" strategies also describe possible natural features of the training data which should give us cause for optimism.

Chapter IV

Evolving stochastic grammars

In figure II.1, a simple EA was described in outline. This chapter gives a detailed account of "EvoGrams," for "Evolving Grammars," the EA which is the primary implementation used in this thesis, except chapter VI. The inside-outside algorithm was previously described in chapter III; the procedure for unpacking a phenotypic grammar from a genotype and testing that grammar are described here, as well as some details of a parallel implementation used for some of the experiments in chapter VII.

Before describing the details of the genotypic representation and fitness function used by EvoGrams, we give some general specifications of our EA. EvoGrams is panmictic and has no generation gap, meaning that the individual in generation t are never held over into generation $g+1$, though generation $g+1$ may contain some exact copies of individuals from generation g. It uses "1-elitism," meaning that the single best individual at each generation is guaranteed a single unmodified copy in the next generation; in the event of a tie for first place, a random choice among the highest-scoring grammars is made. The selection method is tournament selection with a tournament size of 3: to select and individual, we make three random choices from the population and keep the one with highest fitness. Genotypes are variable-length strings with a hard limit set on maximum length: longer genotypes are simply clipped to the maximum length.

IV.A The EG representation

To perform grammar induction, we let each genome represent a stochastic grammar; the EA population becomes a population of candidate grammars. Each genome represents the start symbol, rules, and rule probabilities for a single grammar; the allowable nonterminal and terminal sets are fixed at the outset of the run.

Each individual genotype in the population encodes a SCFG as a variable length string over a fixed alphabet. The alphabet contains all of the possible nonterminals and terminals that are to be allowed in a grammar, set by the user, and a special punctuation symbol, '|'[1]. The '|' symbol surrounds each grammar left-hand side and right-hand side on any given genotype. There is also a '|'-delimited segment at the beginning of each genotype specifying a start symbol for each grammar.

Each genotype is a string of characters made up of nonterminals, terminals, and the two special symbol '|,' used to mark off regions which code for parts of a grammar, and '*,' used as a "wildcard" symbol. The '*' is treated as if it generates each of the other possible terminals, but cannot be used on the left-hand side of any rule. When '*' is used in place of a terminal, t, in a derivation, its as if a rule $* \rightarrow t$ (p) must be used and is available in the grammar. The values of p in these "implicit productions" are selected by the user. The rest of the genotype contains noncoding segments between these coding regions. Figure IV.1 shows an example of the genotypic encoding, the grammar on the top is encoded by the genotype under it. Note that there are many possible genotypes to encode a given grammar because the contents of the noncoding segments may vary. $\Sigma = \{(,)\}$ is the terminal set for this grammar, and the nonterminal set is $N = \{S, T\}$.

To decode this genome, we start scanning from the beginning of the string until we find a bar. If this bar is followed by a nonterminal and another bar, that nonterminal becomes the start symbol; otherwise, we scan forward to the next bar and try again. Once a start symbol is found, we scan forward from the character after

[1]The punctuation symbol should not be confused with the Backus-Naur form "alternative right-hand side" symbol.

Start symbol: S

S	$\rightarrow$	TS	(1/3)
S	$\rightarrow$	T	(2/3)
T	$\rightarrow$	(S)	(1/2)
T	$\rightarrow$	( )	(1/2)

((**|S|** ((S)(((|(())|S **|S|** TSST()) **|T|** |ST| (())(SSSSSSS **|T|**
||()))T **|(S)|** |||(S|((**|S|** ()))STTTT **|T|** STSTSTT(((((
SSSTT()))) **|T|** STSTSTS **|()|**))(SSSTTT)( )))))SS **|S|** **|TS|**
|SSSS|TT((|T| STTSSSS)))))

Figure IV.1: A grammar and its encoding showing coding regions in boldface and noncoding regions which look like coding regions underlined.

its closing bar, looking for bar-nonterminal-bar combination to serve as the left-hand side of the first grammar rule. We keep looking until we find such a combination or hit the end of the genome. If we do find a left-hand side, we similarly scan forward for a pair of bars bracketing any series of terminals and nonterminals, up to some maximum length, to serve as the corresponding right-hand side. If the right-hand side is found, we scan forward for a pair of bars bracketing a sequence of terminals. These terminals will be decoded to yield a probability value for the rule. At this point, we have decoded one complete rule and start looking again for another left-hand side. In the sample genome, we have set coding segments in bold-face. Note that some noncoding segments may look like coding segments, as is shown by the underlined sections.

Sequences of terminals are decoded into probabilities by considering half of the terminal symbols to stand for 1, and the other half for 0. Using this scheme, sequences of terminals are decoded as simple binary fractions, with the radix point always before the first bit. If the fraction contains all zeros, a small nonzero value is substituted instead; zero probabilities on the genotype are not allowed.

In the following chapters we will discuss results from two different versions of the algorithm. In one version, the probabilities are decoded as just described. More recent experiments no longer encode probabilities on the genotype, and will thus have no probability fields decoded. Rather, the interpretation process just starts looking for the next rule left-hand side. In these simulations, we set the probabilities all to the same value instead of using decoded values from the genotype.

Once we have decoded a grammar, we have a start symbol and a set of rules whose probabilities do not add up to one for each nonterminal. Furthermore, some of the rules may be such that no derivation of a string of terminals from the grammar can ever use them. Finally, there may be duplicate rules represented on the genotype. The last steps in decoding a genotype into a phenotypic grammar are to first eliminate the rules that are not useful in deriving any string, then to eliminate duplicate rules by adding up their probabilities and keeping one copy with this probability. Last, the probabilities are normalized for each nonterminal, so that the probabilities of all rules having a given left-hand side sum to 1. At this point, we have a grammar ready for fitness evaluation.

Notice that there is redundancy in this representation at three levels. First, we might the contents of introns can vary without changing the decoded grammar, at least as long as we don't introduce any punctuation characters into an intron. Even then the decoded grammar won't always be changed as long as the punctuation characters don't change the pattern of coding regions on the genotype. Thus, there are many different genomes for a given start symbol and collection of rules. The noncoding regions, or "introns," are similar to those used in EAs by other researchers[59, 60, 81]. The use of the punctuation character to delimit coding regions is reminiscent of "messy GAs"[39].

Once the productions are decoded, any productions that cannot be used as part of any complete derivation of a string of terminals from the grammar are dropped from the grammar. Thus, varying these rules on the genotype will, subject to the constraint that none them be used in any derivation, has no effect on the final grammar to be evaluated. Thus, even if the noncoding regions were fixed, there are still many genotypes for each phenotypic grammar. The eliminated rules arose from coding regions but do not function in the presence of the other rules decoded from the same genotype; they are thus similar to "pseudo-genes"[81].

The previous two forms of redundancy in the representation arise because we can very parts of the genotype which do not play a role in the final phenotypic grammar to be evaluated. One final form of redundancy is inherent in the use of grammars: it is possible to write more than one grammar for the same language. This leads to a different set of coding regions which need to be encoded genotypically. Even if normal forms are used, the grammar for a given language is usually not unique.

When we describe the fitness function, we will describe a partial credit mechanism for use when a grammar does not completely parse a training example. The representation admits grammars that do not derive any complete string from the training: the filtering of unusable productions is *not* done relative to the training data, but only relative to $\mathcal{L}(\Gamma)$ for grammar Γ. Such a grammar is given partial credit for being able to derive substrings of the data, as is described below. These grammars may be the only ones obtaining fitness early in an EvoGrams run.

IV.B Genetic operators and grammar rules

The genotypic representation in EvoGrams uses strings over a fixed alphabet, and encodes grammar components in a way designed to promote the use of "standard" forms of commonly used genetic operators, such as crossover and mutation. The representation has been designed to allow straight-forward definition of genetic operators, and these operators do not need to know that a grammar is being encoded. In fact, the operators are all blind to the semantics of the genotype: they all treat genotypes as simple strings without respect for boundaries between productions.

EvoGrams uses four genetic operators: crossover, mutation, inversion, and substitution. The crossover operator is 2-pt crossover on the variable-length genomes;

the fragments exchanged may be of different sizes. The point-mutation operator does a single-character insertion, deletion, or modification; insertions insert a randomly chosen character. Inversion reverses a substring of a genotype whose endpoints are chosen at random. The substitution operator replaces one character for another, both chosen randomly, throughout a randomly selected substring of a genotype. Note that the rates for these operators are specified per pair of genomes for crossover, per character for mutation, and per genome for substitution and inversion.

The EvoGrams genotypic representation gives the crossover operator on a strings the ability to recombine and even duplicate productions. One of the main motivations for performing grammar induction with an EA is that grammars are sets of rules which form a natural grist for the recombinative mill: crossover recombines groups of rules with other groups of rules to yield novel grammars. The coding regions for left- and right-hand sides of rules, as well as probabilities when used, make intuitive "schema" or "genes" to be recombined and mutated. This has also been found in past research on the evolution of rule-based systems, such as classifier systems[47], as well as in other work on evolving grammars[58, 83].

An important lesson learned from previous experience with sorting networks[2] was that it is important to recombine parts of rules[52]. The need for this has also been noted by others working on grammar induction with EAs[83]. If crossover is constrained to occur between rules only, then crossover's role is restricted to that of duplicating and forming sets of grammar rules. If this is the case, then we crossover is not helping to search the space of individual rules, which limits its use as a search operator. If mutation is the only other operator present, then mutation is responsible for searching through the space of possible productions by itself.

EvoGrams was designed to use a representation which allows crossover to apply to arbitrary strings which contain no knowledge of the objects encoded by those strings, but the encoding was designed with the operator crossover in mind. The other two operators which work on substring of genotypes, inversion and substitution

[2]Discussed in chapter VI.

were added later because it was suspected that they would prove useful inducing grammars. Substitution is meant to fulfill the role played by merging nonterminals in other grammar induction systems[33, 78]. Inversion is often included among an EA's genotypic operators to help maintain diversity among the population's genotypes, but was also included here to make learning palindromic relationships easier by allowing rules' right-hand sides to be reversed.

IV.C Fitness measurement

Our fitness measure rewards grammars that assign high probabilities to training data strings. For a string σ and a SCFG Γ, we write $\Pr(\sigma|\Gamma)$ for the probability assigned to σ by Γ. The fitness of a phenotypic grammar is the sum of fitness contributions from individual strings minus a penalty for grammar complexity, discussed later. For now, we discuss the part of fitness based on Γ's ability to derive each σ in the training data, referred to as the "performance" of Γ:

$$\mathrm{performance}(\Gamma) = \sum_{\sigma \in T} \mathrm{fcontrib}(\Gamma, \sigma) \tag{IV.1}$$

where T is the training data. The only exception to equation IV.1 is when the genotypic grammar cannot be decoded even so far as to provide a start symbol. In this case, an arbitrary very negative value is chosen, -10^{50}.

The contribution of each string to fitness for positive examples is given by:

$$\mathrm{fcontrib}(\Gamma, \sigma) = \begin{cases} \log \frac{\Pr(\sigma|\Gamma)}{\Pr(\sigma|\Gamma_{\mathrm{Random}})} & \text{if there is a derivation of } \sigma \\ \mathrm{partialcredit} & \text{if there is a derivation of a substring of } \sigma \\ K & \text{if no substring of } \sigma \text{ has a derivation} \end{cases} \tag{IV.2}$$

where K is chosen to be a negative number with a high absolute value, so that Γ is heavily penalized if it doesn't even partially parse σ; for all of the experiments reported in this thesis, $K = -10000$. Γ_{Random} in equation IV.2 refers to the grammar:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{Start symbol: S}} \\
\text{S} \rightarrow \text{*S} & & \text{(1-p)} \\
\text{S} \rightarrow \text{*} & & \text{(p)}
\end{array}
$$

where $p = 1/(\text{averagetrainingdatastringlength})$. "Partial credit" refers to fitness assigned to a grammar which is able to derive part of a training string. In this case, we compute a "greedy cover" of the string using entries from the chart parse of the string and σ which can be derived starting from any nonterminal and compute fitness based on a "pessimistic completion" from the cover. The cover operation looks through the entries in the chart parse and finds, among the longest entries available, the one with the largest log-odds score. The log-odds score for an individual entry is computed as if the entry represented a complete parse of the portion of the training example it covers. Once the first entry is found, the process is iterated to find additional entries for the remaining uncovered portions of the string. The iteration continues until no more entries can be found to cover still uncovered portions of the training example.

The pessimistic completion assumes that the rest of σ is generated by rules with two symbols on their right-hand sides, all having probabilities less than the smallest allowed genotypic probability. This score will always be worse than that for any grammar generating all of σ. The value of -10000 when no part of the string can be generated imposes a heavy penalty when a grammar does particularly badly on a given training example.

Figure IV.2 shows an example of a greedy cover and its attendant pessimistic completion. The grammar shown on the right of the figure has a single nonterminal, its start symbol S, and terminals $\{x, y, z\}$. The grammar derives simple palindromic strings in x and y with a z marking the center of the string. The string which we are trying to parse with this grammar, "yyxzxwxyzyxwwzw," contains a terminal which the grammar does not include in any of its rules, w. Because of this, the grammar is only able to parse a few substrings of the string: these are the three substring shown below S's in the figure.

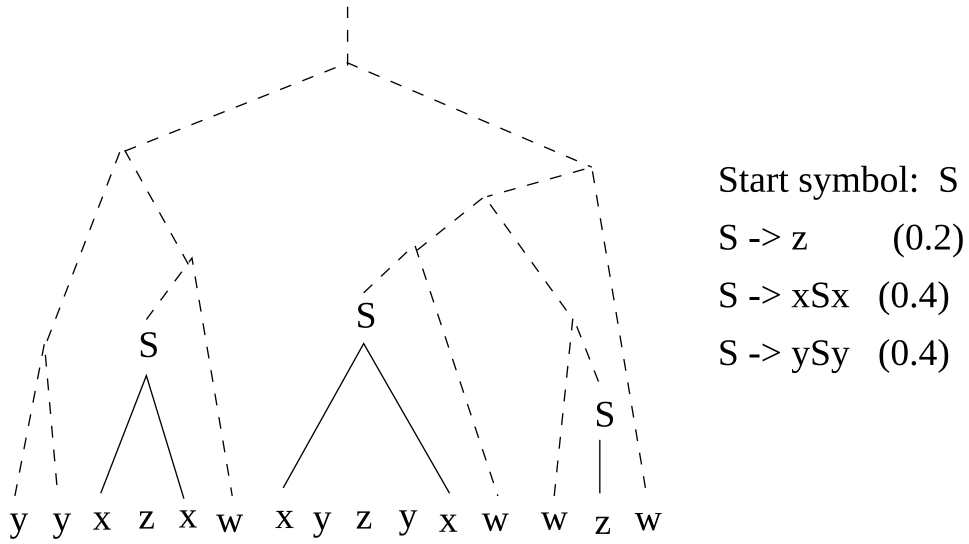


Figure IV.2: A grammar and it's greedy cover and pessimistic completion for a string.

Each S above a pair of lines represents the ability to derive the indicated substring from S. The three entries shown form a greedy cover. Note that because we prefer longer entries first in covers, we would have taken the center entry over "xyzyx" into the cover first, the left entry over "xzx" next, and the entry over the singleton "z" last. The dotted lines above the entries show the pessimistic completion of the parse tree to cover the whole sequence. Each vertex in the part of the tree composed of dotted lines represents an imagined low-probability rule firing as described above. Note that the exact shape of the completion doesn't matter as long as it is a binary tree: the number of leaves of a binary tree is always one more than the number of internal nodes. The extra vertical dotted line at the top of the tree indicates that we assume that once the parse is complete, yet one more production must be fired to rename the grammar's start symbol to the nonterminal that generates the whole string. This ensures that a greedy cover of the whole string with one entry whose nonterminal is other than the start symbol will be worse than a genuine complete parse.

The above definition of fcontrib($\cdot$) applies to positive examples. In most of the simulations below there will be no negative examples, but when there are, the contribution of negative examples to fitness is given by equation IV.3.

$$\text{fcontrib}(\Gamma, \sigma) = \begin{cases} K - \log \Pr(\sigma|\Gamma) & \text{if there is a derivation of } \sigma \\ 0 & \text{otherwise} \end{cases} \tag{IV.3}$$

where $K = -10000$ as in the definition of fcontrib(.,.) for positive examples. This heavily penalizes any acceptance of negative examples by a grammar, but allows the penalty to ameliorated to the extent that a low probability is assigned to the example.

When all of the training data can be derived from Γ, the fitness measure above is a log-odds scoring system with Γ_{Random} as the *background model.* Note that when the inside-outside algorithm is used, we assign partial credit according to the untuned grammar, since the inside-outside algorithm may set some production probabilities to zero, thus lowering partial credit.

Log-odds scoring solves a problem relate to string lengths in the training data. Since SCFGs for infinite language usually assign lower fitness to longer strings, short strings will have much higher probability and thus a much stronger contribution to fitness than relatively long strings. This is in addition to any differences owing to differences in the frequency of these strings in the training data. By taking the ratio between $\Pr(\sigma|\Gamma)$ and $\Pr(\sigma|\Gamma_{\text{Random}})$, we attempt to normalize for length thus eliminating this difference. This works best when the variation of probability of strings with increasing length is on average close for the two probabilities in the ratio.

IV.C.1 Modifying basic fitness with complexity

For various reasons, we would like EvoGrams to prefer simpler grammars over more complex ones. The main reasons for preferring simpler grammars over more complex ones are familiar in machine learning: parsimony, interpretability, and, to a lesser extent, standardization. Parsimony is a reflection of "Occam's Razor," the principle that says that if two otherwise equally good explanations of some data exist, the simpler of the two is preferable since it makes less assumptions about the world in explaining the data. The explicit definition of "simpler" is encoded by our measure of grammar complexity in EvoGrams. Interpretability refers to the ease with which evolved grammars can be read by a human and used to gain insight about the data from which they were evolved. Standardization refers to our desire to evolve the same grammar, or equivalent grammars, in separate runs on the same data. Without standardization, we don't have the sense of having found the "right" answer.

In EvoGrams, there are two pressures operating to prefer simpler grammars. In evolving *stochastic* grammars, we automatically prefer grammars which yield shorter derivations of strings in the data set. Suppose that we have two grammars, Γ_1 and Γ_2 in the population and that they both yield a single derivation of σ. If the start symbols of the grammars are S_1 and S_2, respectively, then we have:

$$D_1 \stackrel{\text{def}}{=} S_1 \rightarrow^{n_1} \sigma$$

$$D_2 \stackrel{\text{def}}{=} S_2 \rightarrow^{n_2} \sigma$$

where n_1 and n_2 are the numbers of derivation steps in the two derivations. Suppose that $n_1 \leq n_2$, so that the first derivation is the shorter one. If the rules used in the derivation steps of D_1 form a subset of those used in D_2, then it is trivially true that

$$\Pr(\sigma|\Gamma_1) < \Pr(\sigma|\Gamma_2) \qquad \text{(IV.4)}$$

since these probabilities are products of rule probabilities from the respective derivations. If $n_1 \leq n_2$, but we don't know whether D_1 uses a subset of the rules used in D_2, then the probabilities may not be so related, but if $n_1 \ll n_2$ then it is still likely that equation IV.4 holds.

Equation IV.4 tells us that a grammar that induces shorter derivations of training data will generally be preferred over a grammar that induces longer derivations. This will not always be the case, but in many cases it will hold as long as the minimum rule probability in D_1 and the maximum probability in D_2 obey certain constraints. Suppose that the maximum rule probability in D_1 is p_1 and the minimum rule probability in D_2 is p_2; then the relationship

$$p_1^{n_1} \leq p_2^{n_2}$$

suffices to guarantee that equation IV.4 holds. Within these bounds, grammars yielding shorter derivations of the strings are preferred. It is important to note, however, something that balances this tendency. If Γ_1 permits more distinct derivations of σ than Γ_2, then the sum of probabilities over Γ_1's derivations may make Γ_1's probability higher. Thus, grammars that contain more rules that are used to generate distinct parses of σ, rather than longer parses, may actually be favored.

Thus, to be sure that we are preferring simpler grammars as intended, we include an explicit penalty for "complex" grammars in the fitness function. We will always consider our complexity measure on grammars, $C(\Gamma)$ to be evaluated on a "reduced" grammar. In a reduced grammar Γ, every rule participates in at least one string in $\mathcal{L}(\Gamma)$. There are a number of possible complexity measures we might use. Among them are average derivation length to derive a set of strings, number of rules, number of nonterminals, and number of bits needed to encode the grammar.

In practice, we will use a measure that increases with the number of rules, the number of nonterminals, and the number of bits needed to encode the right-hand sides of grammar production rules. This particular choice of $C(\Gamma)$ has also been successfully applied in other grammar induction research[34, 74, 32]. We use, in particular, a nonstochastic variant that ignores the rule probabilities, since we tune these to the sample, and we are most interested in the complexity of Γ's characteristic grammar. Our definition is exactly the same as one of the previous researchers[32]. Thus, we are using $C(\Gamma)$ as a convenient notation to indicate a complexity measure that is really only using information from $\mathrm{Ch}(\Gamma)$.

The definition of $C(\Gamma)$ is a sum over rules of the $c(\mathrm{rhs}_i)$, where rhs_i is the right-hand side of the i'th rule of Γ. $c(\sigma)$ is a complexity measure on strings. Suppose σ has length n and contains m different letters, and the i'th letter occurs k_i times. Then we define c as:

$$c(\sigma) \stackrel{\mathrm{def}}{=} (n+1)\log_2(n+1) - \sum_{i=1}^{m} k_i \log_2 k_i \tag{IV.5}$$

If $(n+1)$ is replaced by n in this definition, we have the number of bits needed to encode σ in an optimal code. Using $(n+1)$ makes the expression scale with string length. Now if Γ has N productions, the definition of grammar complexity becomes:

$$C(\Gamma) \stackrel{\text{def}}{=} \sum_{i=1}^{N} c(\text{rhs}_i) \tag{IV.6}$$

This expression obviously increases with number of rules. Since we will be evaluating it on reduced grammars, it also increases with the number of nonterminals in the reduced grammar, since every added nonterminal must occur on the right-hand side of some rule in the reduced grammar, and this either increases the number of rules, the length of some rule's right-hand side, or the complexity of some right-hand side. The encoding length captured by $C(\Gamma)$ isn't the encoding length of the whole grammar, but rather the sum of the encoding lengths of the grammar's rules' right-hand sides. A larger coding length per right-hand side means that the right-hand side is longer, or contains more distinct symbols, or both. As a rule of thumb, longer or more heterogeneous right-hand sides are indicative of a more complicated explanation of training data than shorter more homogeneous ones.

Once we have a complexity measure such as $C(G)$ in hand, we need to know how it should be used to modify the fitness of individual grammars in the GA's population. Recall that our fitness measure is based on log-odds of the data given a grammar Γ: the log of the ratio of the probability of the data given Γ to the probability of the data given a "background" model implemented as a simple grammar. We define the fitness modification caused by complexity to be such that a very complicated grammar assigning high probability to the training data has net fitness equivalent to the background model. This approach is designed to introduce a need for added complexity in a grammar to be justified by increased log-probability.

We define a factor, β, used to scale grammar complexity before it is subtracted from the performance to form the overall fitness for Γ. To define the complexity-modified fitness measure, fitness, we will use the value of performance(Γ) defined in equation IV.1:

$$\text{fitness}(\Gamma) \stackrel{\text{def}}{=} \text{performance}(\Gamma) - \beta C(\Gamma) \tag{IV.7}$$

The value of β is chosen so that a definite grammar which derives just the training data and no other strings will have fitness zero. That is, if Γ_{definite} is a stochastic grammar whose start symbol is S and whose characteristic grammar has production rules $S \to \sigma$, for each $\sigma \in T$, where T is the training data, then we define β as:

$$\beta = \frac{\text{performance}(\Gamma_{\text{definite}})}{C(\Gamma_{\text{definite}})} \tag{IV.8}$$

With this definition, the fitness of Γ_{definite} according to equation IV.7 is:

$$\begin{aligned} \text{fitness}(\Gamma_{\text{definite}}) &= \text{performance}(\Gamma_{\text{definite}}) - \beta C(\Gamma_{\text{definite}}) \qquad \text{(IV.9)} \\ &= \text{performance}(\Gamma_{\text{definite}}) - \text{performance}(\Gamma_{\text{definite}}) \\ &= 0 \end{aligned}$$

Thus, the higher a grammar's complexity with respect to the complexity of Γ_{definite}, the more fitness is taken away for the grammar's complexity. Since no grammar should be able to achieve unmodified fitness greater Γ_{definite}'s, grammars approaching Γ_{definite} in complexity should receive a large fitness penalty, whereas simpler grammars, according to the measure defined by equation IV.6 should receive much smaller penaltys.

IV.C.2 Fitness evaluation by subsampling

If the training data is too large, we will not be able to evaluate each grammar in the population on every string in the training data in a reasonable amount of time. In this case, we would like to know whether it is worthwhile to perform less testing per grammar per generation by sampling the training data randomly. This introduces noise into individual evaluations in each generation, but allows much faster evaluation.

Since EAs are population-based, they are robust to the noise from such sampling. A given individual will only survive across many generations if it is successful with respect to many different samples of the training data[31, 50]. In EvoGrams, we choose a single sample at each generation when training data sampling is in effect. This keeps the evaluations consistent within generations.

Let the training data be in S. Though fitness values need to distinguish good grammars from bad ones, they need not be based on testing over all of S at each generation. In particular, if stochastic grammars Γ_i and Γ_j occur in the population at least once each at generation g, and if Γ_i generates a larger fraction of the strings in S than Γ_j, then successfully distinguishing between them requires that we draw an $R \subseteq S$ containing more strings in $\mathcal{L}(\Gamma_i)$ than strings in $\mathcal{L}(\Gamma_j)$. If selection isnt strong, this sampling process can be spread across more than one generation, so long as both grammars remain in the population across those generations.

The remainder of this section briefly analyzes the probability of misjudging the fitnesses of two grammars. While this discussion is in terms of the abilities of grammars to *generate* strings from S, it is relevant to work on stochastic grammar induction since EvoGrams heavily penalizes failure to account for example strings. Also note that because EvoGrams uses tournament selection, it is the *ordering* of fitnesses which is important, not the ratio of fitnesses. Finally, whenever specific numbers are given below, they are cited for the $|R| = 10$.

Divide S into four subsets: those strings which are derivable from exactly one, both, or neither of Γ_i and Γ_j. Figure IV.3, shows these four regions graphically, with further labels indicating S and the subsets of S derivable from each of the two grammars; these labels are used in the equations which follow. The probability of either not being able to tell the two grammars apart, or of judging the better grammar as worse is just the probably of drawing an R which contains the same number or more strings from $\mathcal{L}(\Gamma_j)$ than strings from $\mathcal{L}(\Gamma_i)$, rendering the grammars indistinguishable according to R. This occurs according to the probability of selecting samples from a multinomial distribution:

$$\sum_{m+n+o=|R|, m \leq n} \begin{pmatrix} |R| \\ m, n, o \end{pmatrix} f_{i,\sim j}^{m} f_{\sim i,j}^{n} \left(f_{i,j} + f_{\sim i,\sim j}\right)^{o}$$

where m is the number of strings drawn that fell in $f_{i,\sim j}$, n is the number of strings drawn which fell in $f_{\sim i,j}$, and o is the number of strings drawn which fell in $f_{i,j} \cup f_{\sim i,\sim j}$.

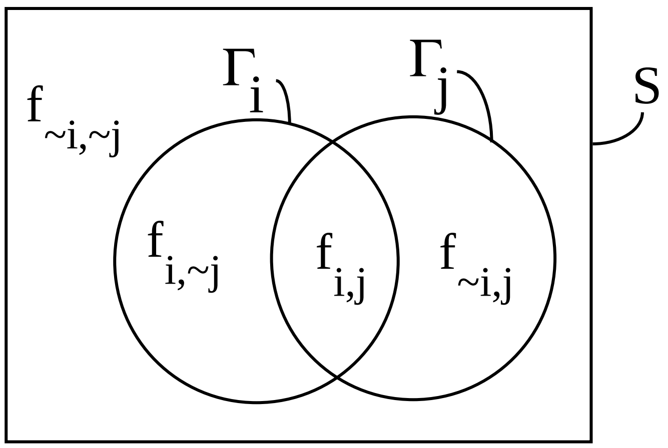


Figure IV.3: Division of training data, S, into four sets based on subsets correctly classified by two grammars Γ_i and Γ_j.

The above expression can also be written as:

$$\sum_{n=0}^{|R|} \sum_{m=0, m+n \leq |R|}^{n} \begin{pmatrix} |R| \\ n \end{pmatrix} \begin{pmatrix} |R| - n \\ m \end{pmatrix} f_{i,\sim j}^{m} f_{\sim i,j}^{n} \left(f_{i,j} + f_{\sim i,\sim j} \right)^{|R|-n-m}$$

Note that $f_{i,\sim j}^{m}$ is never zero since this would mean that Γ_i did not derive more of the strings in S than Γ_j.

When both $f_{i,\sim j}^{m}$ and $f_{\sim i,j}^{n}$ are close to zero, however, the probability above is close to one: there will be few strings which distinguish the two grammars. This means that early in a run, when most grammars will fail to account for most of the training data, it will be very hard to distinguish grammars and drift becomes likely. This motivates the use of the EvoGrams "partial credit mechanism, even when the "subsampling" strategy of choosing $R \subseteq S$ is not employed, since R can be replaced by S in the above equations to make an argument motivating partial credit when complete testing over S is used.

The worst case occurs when all strings in S are generated by exactly one of the two grammars, which means the two grammars partition the space between them and differ as little as possible in the number of strings from S each can derive. That is, the above expression reduces to the case where:

$$f_{\sim i,\sim j} = f_{i,j} = 0 \text{ and } f_{i,\sim j} - f_{\sim i,j} = \frac{1}{|S|}.$$

Setting $f_{\sim i,\sim j} = f_{i,j} = 0$ simplifies the previous double summation to:

$$\sum_{n=0}^{|R|} \sum_{m=0,m+n=|R|}^{n} \begin{pmatrix} |R| \\ n \end{pmatrix} f_{i,\sim j}^{m} f_{\sim i,j}^{n}$$

since these are the only terms of the earlier sum that will remain under the conditions given. This is the same as:

$$\sum_{m=0}^{\lfloor |R|/2 \rfloor} \begin{pmatrix} |R| \\ m \end{pmatrix} f_{i,\sim j}^{m} f_{\sim i,j}^{|R|-m}.$$

From the constraints, we know that each string in S is derivable from either Γ_i or Γ_j. Thus, we have:

$$f_{i,\sim j} - f_{\sim i,j} = \frac{1}{|S|} \text{ and } f_{i,\sim j} + f_{\sim i,j} = 1$$

which is easy to solve to get:

$$f_{i,\sim j} = \frac{1}{2} + \frac{1}{2|S|} \text{ and } f_{\sim i,j} = \frac{1}{2} - \frac{1}{2|S|}.$$

Plugging these values into the formula derived by setting $f_{\sim i,\sim j} = f_{i,j} = 0$, we get the worst-case estimate of the probability of a "fitness mistake when each string is generated by exactly one of the two grammars:

$$\sum_{m=0}^{\lfloor |R|/2 \rfloor} \begin{pmatrix} |R| \\ m \end{pmatrix} \left(\frac{1}{2} + \frac{1}{2|S|}\right)^{m} \left(\frac{1}{2} - \frac{1}{2|S|}\right)^{|R|-m}.$$

For $|S| = 1000$ and $|R| = 10$, for example, this yields a probability of about 0.62 of misjudging the ordering of the fitnesses of the two grammars. Even in this case, the probability of failing to distinguish the two grammars for two generations in a row is $(0.62)^2 = 0.38$. For three generations in a row, it is only $(0.62)^3 = 0.24$. These estimate remain reasonable if $|R|$ is fixed at 10 and $|S|$ is increased. Note that these are probabilities for the case where Γ_i and Γ_j are as hard to distinguish using S as possible: since we examined the case where $f_{\sim i,\sim j} = f_{i,j} = 0$, there are no strings from S which are simply non-distinguishing. Every string in R that doesnt support the belief that Γ_i has higher true fitness will instead support the belief that Γ_j has higher true fitness.

Generally, however, two grammars will differ in their extensions by more than just one string if they differ in the amount of S they can derive. This is especially to be expected when S, as is often the case, contains many duplicates. If the two grammars differ in their ability to generate the sample by a number of strings equal to 10% of the sample size $|S|$, then, using the same calculation as we just employed, but with the slightly different constraints:

$$f_{i,\sim j} - f_{\sim i,j} = \frac{(0.1)|S|}{|S|} = \frac{1}{10}$$

and, as before, $f_{\sim i,\sim j} = f_{i,j} = 0$, we find that the probability of a fitness mistake falls to about 0.5 for one generation and about 0.25 for two generations. Larger differences in the grammars abilities to generate the sample will lead to probabilities of less than 0.5 in one generation. If the difference between the two grammars abilities to generate S amounts to half of $|S|$, then the probability for even one generation falls below 0.1.

A final interesting case is that in which the better grammar, Γ_i, accounts for a subset of S that is a proper superset of those strings accounted for by the worse grammar, Γ_j. Still assuming that all strings in S are derived by at least one of the two grammars, the worst case for the purpose of distinguishing them is when the two differ by just one string:

$$f_{i,\sim j} - f_{i,j} = \frac{1}{|S|} \text{ and } f_{\sim i,j} = f_{\sim i,\sim j} = 0\,.$$

Note that these conditions imply that Γ_i is consistent with all of S. In this case, again with $|S| = 1000$ and $|R| = 10$, the probability of a fitness mistake falls to less than 0.001. In this case, there is no chance of actually getting the fitnesses of the two grammars backwards, only one of judging them to be equivalent in fitness. This is encouraging, since it suggests that if the genetic operators produce a grammar which is a marginal improvement over its parents, it is unlikely to be misjudged as no better than either parent.

The above discussion suggests that subsampling a small number of strings from S and testing each genome on them might suffice to evolve grammars to model the training data S. That is, testing each grammar on a set $R \subseteq S$ strings should allow induction of grammars correct as long as $|R|$ is big enough so that the fitnesses of any two grammars are more often than not correctly ordered.

When subsampling is used, R is used both as the test set and the inside-outside algorithms training set. Each element of R is drawn from a uniform distribution over S, R is drawn once per generation. The reason for freezing R within each generation instead of redrawing R for every fitness evaluation is that it keeps the fitness function consistent within each generation. Thus, if an R is drawn at some generation which is not representative of S, the grammars rewarded according to the skewed R will all receive punishment at the next generation in which R is more representative of S. $|R|$ must be large enough to insure that several generations will not pass, all with particularly unfortunate subsamples, since this might lead to the complete elimination of the current best grammars from the population. As some of the experiments in chapter V will show, even very small sampling rates can lead to induction of correct grammars.

IV.C.3 EAs with Local Search: The Inside-Outside Algorithm

The inclusion of the inside-outside algorithm in EvoGrams changes the way in which the algorithm behaves. If two genotypes encode the same grammar but the probabilities are different after decoding[3], the local search performed by the inside-outside algorithm will tend to optimize each with respect to the training data. Because the inside-outside algorithm converges quickly[19], it is likely that the two will

[3]This could be because of genotypically differing genotypic probabilities or because of differing frequencies of productions on the genotype.

have much closer fitness values than before the inside-outside algorithm was run. This effect of a local search operation on an EA has been previously studied in detail[44].

On the whole, this effect is helpful, because it allows us to detect grammars which should have the same fitness when we might not based on decoded probabilities. The inside-outside does this well enough that we were motivated to remove the probabilities from the genotypic encoding. An important consequence of this is the lack of an option to perform Lamarckian evolution.

In a Lamarckian EA, a local search or learning operation on individuals results in phenotypic changes which are recoded into individuals' genotypes before they undergo selection and are effect by genetic operators. Thus, acquired characteristics are inherited across generations. We do not perform Lamarckian evolution even for those older simulations in which the genotype contained probabilities. This is because we would have to decide how many bits to retranscribe into the genotype, and we would sometimes loose information gained by the inside-outside algorithm in doing so. Further, the inside-outside algorithm works well enough that it seems to make more sense to eliminate the production probabilities from the genotype, thus simplifying the algorithm.

IV.D Parallelization of EvoGrams Using MPI

In evaluating the fitness of a grammar in EvoGrams, the fundamental operation is parsing a string σ using a grammar Γ. This is what allows us to determine the value of $\Pr(\sigma|\Gamma)$. The cost of parsing grows, however, as the cube of σ. The cost of parsing becomes problematic for data featuring many long examples, such as the protein data which we will consider in chapter VII. Even for reasonable string lengths, data with a large number of strings can be very costly. For this reason, we use a parallelized version of EvoGrams for simulations whose cost would be prohibitive on a workstation.

The parallel version of EvoGrams was written using version 1.1 of the MPI

("Message Passing Interface") library, a portable message-passing library for parallel programming[8]. The parallel code was written and currently runs on a Cray T3E at the San Diego Supercomputer Center (SDSC), which has 256 DEC 21164 processors running at 300MHz, each with 128 megabytes of local memory. The T3E runs the UNICOS/mk operating system, a version of UNIX. Since no T3E-specific mechanisms were used, EvoGrams should run on other MPI installations with minimal or no modification.

The parallelization of EvoGrams seeks to distribute parses among a group of processors. The parallel code does not parallelize the work of the evolutionary algorithm or distribute its population. Rather, the entire population is kept at each processor. During fitness evaluation, each processor evaluates some individuals on some of the training data. The resulting partial fitness evaluations are combined across processors to get complete fitness evaluations. The processors then individually perform selection and apply genetic operators to produce the next generation. If the inside-outside algorithm is in use, each fitness evaluation will have to communicate with other processors evaluating the same individual in order to combine production counts across the entire training sample. This aggregation will be across a subset of the processors, not all of them.

The goal in parallelizing EvoGrams was to parallelize the fitness evaluation only. The GA itself is still panmictic: the population remains unstructured. Thus, each each processor is running an identical GA, but none evaluates all of its population at any generation. Rather, it performs some of the task of fitness evaluation and shares its information with the other processors. In this way, complete fitness evaluations for all individuals are obtained at each generation. Three critical facts make this distribution of labor possible and reasonable:

1. The fitness function is additive across examples for a single grammar, so fitnesses can be added across processors. A processor that does not participate in evaluating a given grammar contributes zero to the sum for that grammar.

2. The overhead associated with the GA's operations is small compared to the

cost of fitness evaluation, so duplicating the GA's computations instead of parallelizing them doesn't reduce our performance gains much in this case.

3. The random number generators at different processors can be identically seeded and kept in synchrony by guaranteeing that in parallelized code segments, the processors all call the random number generator an equal number of times; this guarantees that the populations on the different processors will remain identical across generations.

Thus the unit of labor in our parallelization if a parse of a string according to a grammar. It is possible to be even more fine-grained about the distribution of work among multiple processors by parallelizing the parser code. On the T3E, however, we have powerful enough processors to make allocating work at the level of individual parses reasonable.

During fitness evaluation in EvoGrams, each processor decides whether to evaluate each (σ, Γ) pair by parsing σ using Γ, where Γ is a member of the population and σ is a training example. If the population size is p, the training sample size is s, and the number of processors in use is n, there are sp the it must be the case that $sp \geq n$ if we are to have at least one parse to do at every processor. To guarantee k examples per processor instead, we would need $sp \geq kn$ to be true. We might wish to have more than one per processor when the examples are all relatively short, since we need to do enough work at each processor to offset interprocessor communication costs, or because the inside-outside algorithm requires that we aggregate its counts for each production across all examples.

We allocate parses to processors using a two-tiered scheme: first, we partition the processors and population using partitions with equal numbers of blocks. Each block of processors handles fitness evaluation for a block of individuals. Within each processor block, the training examples are distributed across the processors. That is, the training data are partitioned into a number of blocks equal to the number of processors in the processor block and example blocks are associated with individual processors. We now need to decide how to assign blocks of individuals to blocks of processors and blocks of examples to individual processors. To do this, we decide on a value of k, the minimum number of examples per processor and the compute the number of blocks into which to partition the population such that we have at least k examples at each processor. That is, once we know k, we want to choose c so that:

$$\frac{s}{n/c} \geq k$$

Solving for c, we arrive at the formula:

$$c \geq \left\lceil \frac{kn}{s} \right\rceil \tag{IV.10}$$

which gives us the number of equal-size blocks into which we will partition the population and processors, based on the minimum number of examples per processors that we find tolerable. When $c = 1$, we distribute only the training data and refer to the configuration as "example parallel." If $c = s$, only possible when $p \geq n$, we only distribute the population across the processors and refer to the configuration as "individual parallel." The actual configuration depends on the value of k and on whether $p \geq n$, but most often the assignment of individuals and examples to processors display a mixture of example and individual parallelism.

The question remains of choosing a value of k so that we can compute a value of c using equation IV.10. Our choice is based on two considerations. First, we have a preferred value chosen based on the training data. Second, there is a maximum possible value of k given the prevailing values of p, s, and n. Table IV.1 gives the preferred value of k based on median example length among the training data. We use the median instead of average because we want to make a decision based on a guarantee that at least half of the data are some length or more before increasing the value of k.

When the median example length is less than 20, we prefer to place as many examples as possible on one processor. We may not, however, be able to set $k = s$, placing all of the examples on a single processor. This happens if the population isn't large enough to assign the entire evaluation of at least one individual grammar to every processor, when $p < n$. In this case, there is a maximum permissible value of k

Median example length, L	Preferred $k = k_{\text{pref}}$
$L < 20$	s
$20 \leq L < 50$	16
$50 \leq L < 100$	4
$100 \leq L$	1

Table IV.1: Preferred value of k, the minimum number of examples allowed per processor, based on median example length.

and we set k to the minimum of the preferred value, k_{pref}, from table IV.1, and the maximum value, $k_{\max}$: $k = \min(k_{\text{pref}}, k_{\max})$.

Recall that in order for our parallelization to be reasonable at all, we must choose the number of processors, n, so that $ps \geq n$. This tells us that

$$s \geq \frac{n}{p}$$

so the number of examples is at least the number of processors per individual. Since s is at least this value and s is an integer, we know that

$$s \geq \left\lceil \frac{n}{p} \right\rceil \tag{IV.11}$$

as well. This tells us how many processors per individual are available if we wish to distribute the population as much as possible while keeping the load balanced. This gives us as few processors as possible per individual, and thus allows the sample to be distributed over as few processors as possible. Equation IV.11 guarantees us that in this distribution the number of examples at each processor is

$$\frac{s}{\lceil n/p \rceil} \geq 1$$

Since k must be an integer, we choose $k_{\max}$ to be the floor of this quantity, since we can *guarantee* that all the processors can be assigned so many examples. Thus the maximum possible number of examples per processor in a parallelization that tries to evenly distribute the population among blocks of processors that nearly equal-sized, and tries to evenly distribute examples within the processor blocks, is given by:

$$k_{\max} = \left\lfloor \frac{s}{\lceil n/p \rceil} \right\rfloor \tag{IV.12}$$

And we set $k = \min(k_{\text{pref}}, k_{\max})$ as mentioned above, and choose c according to equation IV.10. Given this parallelization, we need to know how to decide which individuals and examples will be evaluated at which processors.

Since we are dividing n processors into c blocks, each processor will have two ranks: its rank among all processors, r, and its rank within the block of processors in which it falls, subrank(r). Further, each processor belongs to a block whose name will be given by block(r), which yields a block number from 0 to $c-1$. Ranks are integers from 0 to $n-1$, and subranks will be integers from 0 to $n_{\text{block}(r)}-1$, where $n_{\text{block}(r)}-1$ is the number of processors in block(r). Individuals in the GA population will be assumed to be indexed in some arbitrary way from 0 to $p-1$. Similarly examples will be assumed to admit of an arbitrary indexing from 0 to $s-1$.

We compute the part of fitness for individual i deriving from example j on processor r, and thus must parse example j using individual grammar i on processor r, when both of the following equalities hold:

$$\text{block}(r) = i \text{ modulo } c \tag{IV.13}$$

$$\text{subrank}(r) = j \text{ modulo } n_{\text{block}(r)} \tag{IV.14}$$

The first inequality tells us that processor r participates in some measure in the evaluation of individual i: it tells us that individual i is in the block of the population corresponding to block(r). The second inequality tells us that example j should be evaluated by this processor: it is in the block of examples assign to processor subrank(r) within the block block(r).

This assignment of individuals and examples to processors is a striped decomposition of individuals across processor blocks, and a striped decomposition of examples across processors within each block. The latter implies that it is useful to sort the examples first by type (positive or negative) and second by length. Sorting on these two quantities is helpful in load balancing and is done automatically in the parallelized version of EvoGrams.

Sorting by type is useful for the inside-outside algorithm: it guarantees distribution of the positive examples among the processors in block(r) for each processor

r. This is important because the inside-outside algorithm doesn't use negative examples and good load-balancing depends on evenly distributing positive examples among the processors within each block. Sorting by length is helpful because it leads to a tendency to assign the longest c examples to different processors if possible. In general, the examples of a given length will be distributed across the processors of each block, which helps balance the amount of time spent parsing across the processors within each block.

Since the parallelization just outlined distributes fitness evaluation and involves identical copies of the population on all processors, we only do output from one processor. The main choice in deciding how to decompose fitness evaluation for the entire population is the choice of k according to table IV.1. The rest of the parallelization simply tries to assign nearly equal load to different processors by assigning nearly equal-sized blocks of individuals to nearly equal-sized blocks of processors, and nearly equal-sized blocks of examples to processors within a processor block. Sorting the examples is a refinement on this load-balancing strategy meant to evenly distribute examples with the same type or the same length.

Chapter V

Empirical results for some test problems

The purpose of this chapter is to explore the behavior of EvoGrams on a range of benchmark problems. These experiments are designed to accomplish two main goals: they demonstrate instances of the algorithm successfully solving test problems, and they allow us to study how the search for a solution grammar proceeds.

The benchmark problems contain both regular and context-free stochastic grammars, and include targets commonly used as test problems in the grammatical inference literature. For the simpler problems, we can run EvoGrams many times on the same data and examine the results for common features to gain insight into how the search proceeds. The next section briefly describes the problems and how each is to be used in understanding how EvoGrams works; the subsequent sections report and interpret the results for each problem.

V.A Target Languages

For all of the problems below, we fixed the parameter settings of EvoGrams in advance. These settings are given in figure V.1. The maximum number of generations and population size are reported individually for each problem. One noteworthy

Default parameter settings					
Parameter	=	Value	Parameter	=	Value
randseed	=	clock	wildcard	=	none
max lhs len	=	1	max chart edges	=	500
max rhs len	=	49	max inout iters	=	2
max prb len	=	8	inout app prob	=	1.0
max genome len	=	500	inout stop thresh	=	0
max init genome len	=	250	xover rate	=	0.4
num nonterminal	=	10	mut rate	=	0.004
num terminal	=	2	invert rate	=	0.02
nonterminal	=	ABCDEFGHIJ	subst rate	=	0.02
terminal	=	10	xover points	=	2
Alternate settings used in most recent runs					
xover rate	=	0.415	invert rate	=	0.415
subst rate	=	0.415			

Table V.1: Parameter settings for EvoGrams benchmark experiments.

exception is section V.F where the emphasis is on understanding EvoGrams's behavior for a broad spectrum of genetic operator rate settings. In more recent experiments, the substitution, inversion, and crossover operators used different rates, shown in the bottom section of table V.1. All three operators have the same rate, and the value is set so the probability of one or more of them being applied to an individual at a given generation is 0.8. Some of the results reported below are for simulations run prior to the used of the complexity measure as part of the fitness function, so the results below are reported with an indication of whether the fitness measure was "performance" or "fitness" as these two were defined in section IV.C.

V.A.1 Parity

The first target language is defined by a simple regular grammar whose start symbol generates bit strings with an even number of 1's. $\Sigma = \{0, 1\}$ is the terminal alphabet and the strings in $\mathcal{L}_{\text{parity}}$ are those generated by $\Gamma_{\text{parity}} =$

Start symbol: A

$$\begin{array}{lcll} A & \rightarrow & 0A & (1/3) \\ A & \rightarrow & 1B & (1/3) \\ A & \rightarrow & 0 & (1/3) \\ B & \rightarrow & 0B & (1/3) \\ B & \rightarrow & 1A & (1/3) \\ B & \rightarrow & 1 & (1/3) \end{array}$$

The probabilities were assigned uniformly to make obtaining a sample which uses all of the productions in proportion to their probabilities easier. If the production probabilities for nonterminal A were, for example, $0.05, 0.05$, and 0.9 respectively in the order the productions are given in the grammar, 90% of any sample would be the string "0," since A is the start symbol. Setting the probabilities uniformly for each nonterminal's alternatives eliminates this problem.

Below, we will work with various training sets for $\mathcal{L}_{\text{parity}}$. The first includes all bit strings of length between 1 and 6 inclusive. This yields a training set of 126 distinct strings, half of which are positive examples. The other training sets were generated randomly from the grammar above. These will always have 126 strings overall, though the number of unique strings will be lower. In the terminology of section III.C's discussion of sample size, $|S| = 126$ and $N << 126$. Finally, we will also use a training set that incorporates only the positive examples of length between 1 and 6 inclusive, in which case $|S| = N = 63$. In this case, EvoGrams is run longer, since each generation only needs to be evaluated against half as many examples. Note that in the case of the stochastic sample, N is often less than 63, but no extra generations are allotted in this case since $|S| = 126$, even though EvoGrams avoids parsing the same string of S more than once per individual grammar by caching chart parses.

V.A.2 AtLeast(k, n)

Here we define not a single language but a collection of related languages. These are defined over the same binary alphabet as was used in the parity problem. The language AtLeast(k, n) is the set of bit strings of length n or less with at least k bits set:

$$\text{AtLeast}(k, n) = \{\sigma \in \{0,1\}^* |\ |\sigma| = n \text{ and } \text{ONES}(\sigma) \geq k\}$$

where ONES(σ) gives the number of one bits in the string σ. These languages have the property that the balance between negative and positive examples shifts as k is varied from 0 to n. When $k = 0$, there are no negative examples[1] At $k = \frac{n}{2}$, half the examples are negative. When we reach $k = n$, there is only one positive example, the string of all 1's: that is, AtLeast(n, n) is the "1-max" problem on n or fewer bits.

Evolving grammars for these languages allows us to explore the effect of availability of positive and negative examples with respect to the target language. In the parity problem, half of the strings of a given length are positive examples and half are negative examples, and we varied how many of each we made available to the algorithm. Here, the target language itself only allows a certain number of examples of each type to be available. As k approaches n, it becomes possible to write a very simple grammar for the target language, but the target language is characterized more by negative and less by positive examples. Thus, these languages allow us to explore the algorithm's performance on a continuum of languages from those specified mostly by what they are to those specified mostly by what they are not.

V.A.3 Equal numbers of two terminals

The language of strings with equal numbers of two terminals is one of the basic context-free target languages on which grammar induction systems are tested. Here, we will use 0's and 1's as the two terminals, $\Sigma = \{0, 1\}$. A stochastic grammar for this language is Γ_{01}:

[1]Strictly speaking, the empty string would be a negative example, but we are only evolving grammars that do not derive the empty string, as discussed in chapter IV.

Example type	Number in training sample	Number distinct
Positive	100	64
Negative	50	40
Both	150	104

Table V.2: Contents of training samples for inducing $\mathcal{L}(\Gamma_{01})$

Start symbol: A

A	$\rightarrow$	BA	(0.5)
A	$\rightarrow$	B	(0.5)
B	$\rightarrow$	0A1	(0.125)
B	$\rightarrow$	1A0	(0.125)
B	$\rightarrow$	01	(0.375)
B	$\rightarrow$	10	(0.375)

Below, we will be more concerned with whether the characteristic grammar of this stochastic grammar is learned, and less with the probabilities. In addition to serving as a basic test problem, this language will be used to investigate the distribution and size of noncoding regions in the genotypic representations of EvoGrams grammars. The sample from which we will learn grammars consists of 150 strings; it's properties are summarized in table V.2.

Note that unlike the training sets for the other problems in this section, these training sets contains negative examples. The positive examples were generated from the grammar above. Negative examples were generated as follows: pick a length from 1 to 10 with uniform probability, and generate a string of that many bits. Each bit is chosen with equal probability of being a 1 or 0. Each string is kept if it contains differing numbers of 1's and 0's, and discarded otherwise. This process is repeated to obtain the 50 negative examples in the training sample.

We also use the "01" language to explore the use of learning with evolution. The following grammar generates *all* strings over Σ^*, but for a given string length concentrates probability on those strings which have equal or nearly equal numbers of zeros and ones:

$\Gamma'_{01} =$

Start symbol: A					
A	$\rightarrow$	0	(0.09)	(*)	$0.3\alpha/2$
A	$\rightarrow$	1	(0.09)	(*)	$0.3\alpha/2$
A	$\rightarrow$	01	(0.21)		$0.7\alpha/2$
A	$\rightarrow$	10	(0.21)		$0.7\alpha/2$
A	$\rightarrow$	0A1	(0.14)		$0.7\beta/2$
A	$\rightarrow$	1A0	(0.14)		$0.7\beta/2$
A	$\rightarrow$	AA	(0.12)	(*)	$0.3\beta/2$

This grammar is used to demonstrate the use of the inside-outside algorithm with evolution[2].

The production probabilities for Γ'_{01} were chosen by first considering the grammar with only the productions marked "(*)," which generates all of Σ^*. These were assigned probabilities $\frac{\alpha}{2}$ each for the first two and β for the last one, where

$$\alpha + \beta = 1$$

and $\alpha = 0.6$. The α value was chosen so that the longest string in the training sample could be parsed in a reasonable amount of time given our computational resources and the need to run several repetitions of the experiment. We added the other four productions to place extra probability mass on strings with the same numbers of 0's as 1's. Next, the formulae at the far right of Γ'_{01} were used to assign probability to the non-(*) productions. Note that in each case, 70% percent of the probability mass on old productions is shifted to the new productions: this places emphasis on the structure of strings with the same numbers of 0's as 1's.

V.A.4 Palindromes

A palindrome is a string which is equal to its reverse. For any alphabet, a context-free palindrome grammar is easy to construct. The palindromes over a

[2]The results to be presented below regarding Γ'_{01} have been previously published[50].

Sample	#strings	#distinct strings
Derived from Γ_{pal}	100	34
Random selection from length-16 palindromes	25	25
Random selection from length-8 palindromes	6	6

Table V.3: Contents of training samples for inducing $\mathcal{L}(\Gamma_{\text{pal}})$

finite alphabet are another language often used as a benchmark problem for grammar induction algorithms. For $\Sigma = \{0, 1\}$ we have $\Gamma_{\text{pal}} =$

$$
\begin{array}{llll}
\multicolumn{4}{l}{\text{Start symbol: A}} \\
\text{A} & \rightarrow & \text{0A0} & (0.25) \\
\text{A} & \rightarrow & \text{1A1} & (0.25) \\
\text{A} & \rightarrow & 00 & (0.125) \\
\text{A} & \rightarrow & 11 & (0.125) \\
\text{A} & \rightarrow & 0 & (0.125) \\
\text{A} & \rightarrow & 1 & (0.125)
\end{array}
$$

Since we don't use epsilon productions, we need the last four productions to ensure that grammar derives both odd- and even-length palindromes.

The palindrome language is used below to explore the issue of the use of the wildcard character when learning from samples of fixed-length strings. The three different samples used are summarized in table V.3.

The first sample is built by deriving 100 strings from the target grammar using the production probabilities to decide which production to use at each step of the derivation. This yields 34 distinct strings out of the 100 generated, all of which are positive examples since they were generated by the target grammar. This is the sample used to determine ability to learn the stochastic language $\mathcal{L}(\Gamma_{\text{pal}})$. The ability to learn from this sample is also compared to ability to learn from the reduced samples. The experiments using this sample ran for 1000 generations with a population of size 250.

The latter two samples in table V.3 contain only positive examples. In the first, all of the examples are of length 16 and in the second all are of length 8. The palindromes in each training set were selected by making a uniform choice over the length 16 and 8 palindromes respectively. The length-16 sample contains 25 strings, one fourth the number in the derived sample.

To make the test fair, we allow the runs using this sample to run twice as long with twice as large a population. In the case of the length-8 sample, we want to multiply number of generations and population size by $\sqrt{100/6} \approx 4$. This makes the population unmanageably large, so we multiply population size by 2 and number of generations by 8 instead. In each case, the population size and number of generations are scaled so that the same number of strings are parsed as in the case with the derived sample.

V.A.5 $a^n b^n c^n$

In section III.D, we claimed that we can fall back on a more statistical description of data sets which are really samples of languages from above the context-free level of the Chomsky hierarchy. We need to test EvoGrams on a target language above the context-free level to investigate this claim.

We will use the nonstochastic context-sensitive language mentioned in section III.D:

$$\mathcal{L}_{\text{abc}} = \{a^n b^n c^n | n \geq 1\}$$

which is generated by many context-sensitive grammars. Note that this language uses a different terminal set from the previous languages, $\Sigma = \{a, b, c\}$. For the purposes of generating data for this language, it is easiest to generate samples from an unambiguous grammar:

Start symbol: A

A	$\rightarrow$	cB	(1.00)
B	$\rightarrow$	abBc	(0.66)
B	$\rightarrow$	aDbc	(0.34)
Ca	$\rightarrow$	aC	(1.00)
Cba	$\rightarrow$	aCb	(1.00)
Cbb	$\rightarrow$	bCb	(1.00)
Cbc	$\rightarrow$	Dbc	(1.00)
bD	$\rightarrow$	Db	(1.00)
aaD	$\rightarrow$	aDa	(1.00)
baD	$\rightarrow$	aCb	(1.00)
caD	$\rightarrow$	a	(1.00)

Since none of the distinct left-hand sides is a substring of any other left-hand side, the probability assignments are coherent. The data set for this language contained 126 examples generated from the above grammar, of which only 12 were unique. The longest string was 36 characters long. We will learn a stochastic context-free grammar below which captures part of the syntax of this language in a context-free grammar.

V.A.6 Motifs

Suppose that we know a very good grammar for a given training data set. We can then assemble a custom initial population in order to test the question, "Will the EA retain a good grammar in its population once such a grammar has been discovered?" The answer depends on many characteristics of the EA, but we would especially like to know what settings of the operators are best.

The data we will use consists of 41 protein sequence fragments, each of length 10. Of the 41 training examples, 19 are distinct. These were found by running the sequence analysis program MEME2.0 on a collection of five alcohol dehydrogenase protein sequences[11]. MEME returns descriptions of motifs, models that can be used to score protein sequence fragments of a fixed width. We took the MEME motif discovered for our five sequences and used the search tool MAST to find sequences with similar fragments in three databases: "GenPept90," "SwissProt31," and a database containing just the five sequences given to MEME. The MAST search resulted in 41 matching sequences from the databases and we took the subsequence from each which matched the MEME motif and included it in our data set.

The MEME motif is represented by a letter-probability matrix whose entries $m_{i,j}$ give the probability of encountering amino acid i in position j relative to the beginning of a subsequence being compared to the motif. In the actual scoring done by MAST, a log-odds matrix is used, with a background model based on the base-frequency of occurrence of amino acids in proteins generally.

Such a motif can be turned into an equivalent SCFG, so we know there is a good model for our training sample. This might not be the grammar that assigns *highest* probability to all of the data, since it was learned on five sequences and used to select the rest of the data. Thus, we have a grammar that fits the data well, since it is equivalent to the motif that was able to retrieve our data set, and we know that it may be possible to improve it by changing it so that it gives a tighter fit to the data.

We build a stochastic grammar from the motif using the motif probabilities, and the log-odds are computed automatically as part of the fitness evaluation. The grammar corresponding to the MEME motif is[3] Γ_{seed}, as shown in figure V.1.

Γ_{seed} is the model with which we will "seed" our initial populations to varying to degrees. A given run is successful to the extent that its final population contains grammars at least as fit as this grammar; thus we are actually detecting the combination of the of the EA's ability to maintain a good solution with its ability to improve on a good solution, or the EA's ability to take "advice" or "instruction."

V.B Basic grammar induction tests

In this section we will examine the performance of EvoGrams relative to other grammar induction algorithms, two of which are also EAs[58, 83, 78]. The two

[3]The probability values have been rounded off for presentation, so the alternatives for a given nonterminal might not sum to exactly 1.

Start symbol: m

$m \rightarrow abcdefghij \quad (1)$

a	→	F	(0.1377)	e	→	C	(0.9607)
a	→	I	(0.0787)	e	→	*	(0.0392)
a	→	L	(0.5393)	f	→	D	(0.8549)
a	→	M	(0.0551)	f	→	*	(0.1450)
a	→	V	(0.0472)	g	→	S	(0.6941)
a	→	*	(0.1417)	g	→	T	(0.1294)
b	→	I	(0.6181)	g	→	*	(0.1764)
b	→	L	(0.0748)	h	→	G	(0.8627)
b	→	V	(0.1732)	h	→	*	(0.1372)
b	→	*	(0.1338)	i	→	F	(0.8431)
c	→	S	(0.0944)	i	→	*	(0.1568)
c	→	T	(0.6811)	j	→	G	(0.8627)
c	→	*	(0.2244)	j	→	*	(0.1372)
d	→	G	(0.8627)				
d	→	*	(0.1372)				

Figure V.1: The seed grammar used as the known good solution for the Motifs problem.

target languages we will use for these test are very commonly reported in the grammar induction literature as test cases: the languages of the grammars Γ_{01} and Γ_{pal}. All four algorithms considered, EvoGrams and the other three, are induce grammars for these targets reliably, as we shall see below.

Since all of the algorithms can solve the test problems, distinctions between them are based on which finds the correct grammar quickest and how much each algorithm assumes about the target. As a standard measure of time taken to induce a grammar, we will count the number of parses of a string σ by a grammar Γ during the algorithm's execution. The assumptions made by the algorithms will be discussed individually.

Figure V.2 summarizes the runs performed to learn the palindromes language. Note that these runs used the complexity measure and the alternate operator rates reported earlier. The EvoGrams runs whose target was $\mathcal{L}(\Gamma_{01})$ used neither the complexity measure nor the alternate operator rates. All of the EvoGrams runs reported here were allowed to run to generation 1000 but found solutions well before that generation. Even though these runs were continued to generation 1000, all found the solution by generation 625.

The curves in figure V.2 show time course data averaged over five runs, with error bars that are one standard deviation in each direction. The top curve shows how many members of the population had the maximum fitness for each generation. The curve below shows the maximum, median, maximum pre-inside-outside algorithm fitnesses. Note that though most of the gain occurs early, the small increments in fitness that occur later are important. This is because the differences between grammars that can parse different amounts of the data tends to be large, due to pessimistic completions and the heavy penalty for failing to even get partial credit for a string. The third curve shows the maximum and average consistency with the data for the population: consistency with the data is defined as the fraction of the training examples which are correctly classified. The bottom curve shows how the population maximum and average grammar complexity change with time.

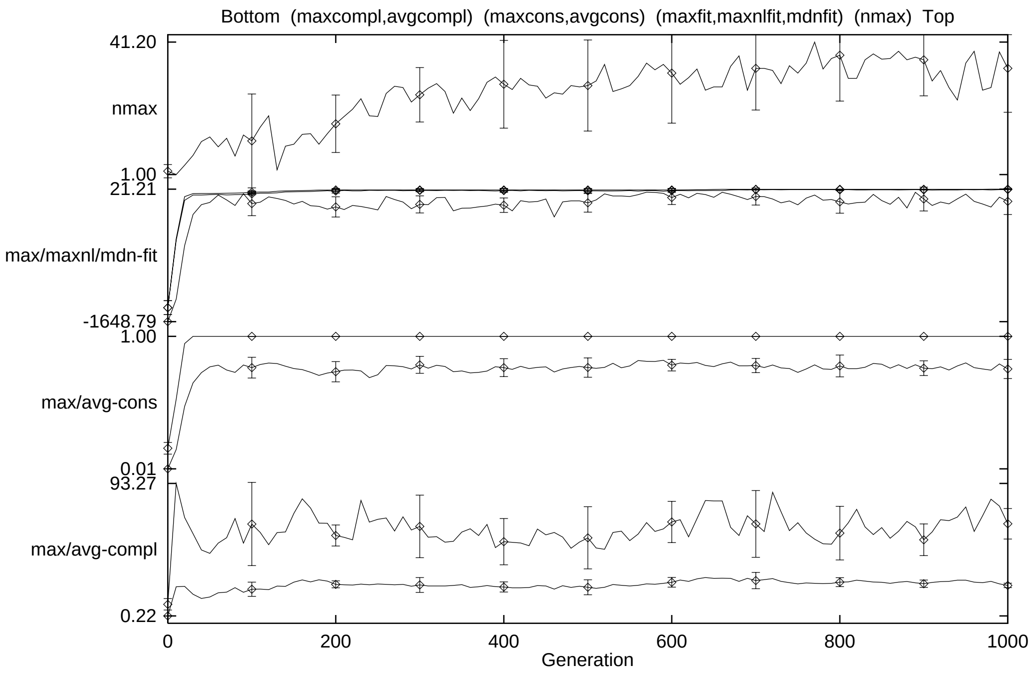


Figure V.2: A summary of five runs that learned the palindromes language.

Learning occurs in three identifiable phases in EvoGrams: the phases are easily distinguished by consistency with the training data and are marked on figure V.3[4]. In the first phase, no grammar is consistent with even one string in the training data and only the partial credit mechanism causes differences in fitness between grammars. In the second phase, at least one grammar is consistent with at least one training example, but no grammar is consistent with all of the training data: the partial credit mechanism is used to compute some values of fcontrib(Γ, σ) and full credit is applied for any pairs (Γ, σ) where Γ is consistent with σ. In the third phase, at least one grammar in the population is consistent with the entire training set. If all examples are positive, the three phases can be described using the predicates in table V.4. Not all runs have all three phases: the first two phases may be skipped or may be very short in some cases. In all of the palindrome-learning runs summarized

[4]This run used training data for the language $\mathcal{L}_{\mathrm{abc}}$ as described in section V.A.5 and was chosen instead of one of the palindromes runs because it clearly shows all three phases.

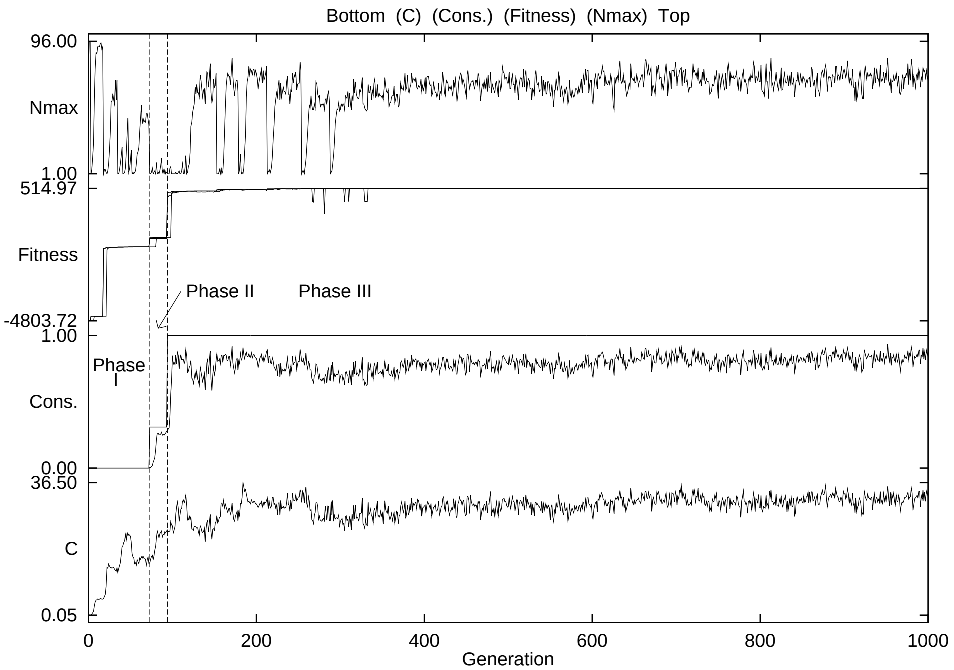


Figure V.3: A single run attempting to learn the $\mathcal{L}_{\mathrm{abc}}$ language of section V.A.5, with phases labelled.

Phase	Predicate
I	$\forall \Gamma \in P_g \forall \sigma \in T \Pr(\sigma \vert \Gamma) = 0$
II	$\exists \Gamma \in P_g \exists \sigma \in T \Pr(\sigma \vert \Gamma) > 0$
III	$\exists \Gamma \in P_g \forall \sigma \in T \Pr(\sigma \vert \Gamma) > 0$

Table V.4: Predicates describing the three phases of an EvoGrams run: P_g is the population of grammars at generation g and T is the training data, assumed to contain only positive examples.

Palindromes				01			
Start symbol: E				Start symbol: A			
E	$\rightarrow$	IEI	(0.276)	A	$\rightarrow$	AA	(0.0957)
E	$\rightarrow$	DED	(0.257)	A	$\rightarrow$	01	(0.173)
E	$\rightarrow$	11	(0.131)	A	$\rightarrow$	10	(0.173)
E	$\rightarrow$	1	(0.126)	A	$\rightarrow$	0A1	(0.171)
E	$\rightarrow$	00	(0.117)	I	$\rightarrow$	1A	(1)
E	$\rightarrow$	0	(0.0935)	A	$\rightarrow$	I0	(0.171)
I	$\rightarrow$	0	(1)	A	$\rightarrow$	10A	(0.108)
D	$\rightarrow$	1	(1)	A	$\rightarrow$	0I	(0.108)

Figure V.4: Grammars evolved by EvoGrams for the 01 and palindromes languages.

in figure V.2, for example, at least one of the first two phases was either very short or absent.

In figure V.4, we show typical grammars learned by EvoGrams for the palindrome and 01 languages. Both of these are exactly correct. Note that the palindrome grammar is eventually simplified to eliminate the last two rules, under fitness pressure from the complexity measure; the grammar provided is from generation 150 of one of the palindrome runs. The 01 grammar from generation 275 of one of the runs for that target and contains redundant productions: only the first 6 productions shown are needed, though the extra two don't harm the grammar. The grammars evolved for this target do not evolve under fitness pressure to be less complex and the later generations of the run that produced this grammar continue to show grammars with redundant productions.

Table V.5 shows a comparison of the other three grammar induction methods with EvoGrams on the two test problems. Note that the other three algorithms are not attempting to learn the probabilities on the rules. The two EA approaches are performing context-free grammar induction, but are not looking for stochastic grammars at all[58, 83]. The non-EA approach considers the estimation of probabilities as something to be done with the inside-outside algorithm *after* the rules have been

Algorithm	Target	Runs	Avg. Parses	#Examples(pos.,neg.)
EvoGrams	$\mathcal{L}(\Gamma_{01})$	10	1,500,000	(1000,50)
EvoGrams	$\mathcal{L}(\Gamma_{\text{pal}})$	5	850,000	(100,0)
Lankhorst	$\mathcal{L}(\Gamma_{01})$	4	2,601,600	(100,100)
Wyard	$\mathcal{L}(\Gamma_{01})$	1	540,000	(150,150)
Wyard	$\mathcal{L}(\Gamma_{\text{pal}})$	1	2,700,000	(150,150)
Stolcke&Omohundro	wcw^R	1	?	5

Table V.5: Comparison of EvoGrams with other methods.

learned; it thus uses statistical information from the sample in learning the correct rules, as part of its equivalent of a fitness function, but does not estimate the rules itself.

As can be seen from the table, EvoGrams does better than Lankhorst's EA on the 01 language and better than Wyard's EA on the palindromes. In the case of palindromes, EvoGrams uses no negative examples at all. In both cases, EvoGrams has many fewer training examples if only the unique examples are considered: 350 for the 01-language (and all of the negative examples are distinct), and 34 for the palindromes. The duplicated strings occur because the training data were created by deriving strings from actual target grammars, and the frequencies of strings contains information about production probabilities. Further, EvoGrams only tested grammars on a random selection of 100 strings at each generation. The average generation at which solutions were found by EvoGrams were 150 for the 01-language and 250 for the palindromes.

Note that Lankhorst's algorithm takes much longer than EvoGrams and uses more examples. Both algorithm, however, find the 01-language in every time. Lankhorst also limited the scope of his simulations to a greater extent than was done with EvoGrams. In Lankhorst's simulations, the right-hand side of each grammar rule in the population was limited to length 3; EvoGrams uses no such limit. Further, the number of rules in each individual grammar was limited to 11. EvoGrams allows as many rules as the algorithm can fit into a genotype, in excess of 30 for these experiments for both target languages.

Stolcke and Omohundro are included here because they use a novel and different approach to grammar induction. Their algorithm employ "Bayesian model merging," in which they use standard artificial intelligence search algorithms, such as best-first search and beam search, to walk through a space of grammars using various operations on grammars to take steps, such as merging nonterminals, introducing new productions, or others. They report successfully learning a variant of our palindromes language from very few examples. These examples are chosen to be a very high-probability sample with respect to the target language. Unfortunately they do not report actual numbers of parses or times to find their grammars. Further, the language learned is simpler by virtue of using the character c as a center marker between a string w and its reverse, a special case of palindromes. They are included here because they succeeded from a very small sample.

The most interesting basis for comparison is Wyard's EA, which performs nonstochastic grammar induction. Interestingly, Wyard's algorithm seems to be doing much better on the 01 language and much worse on the palindromes, though it is difficult to discern much from Wyard's singleton runs. As with Lankhorst, there are many more assumptions being made about the target languages than EvoGrams makes. First, Wyard restricts the numbers of rules in his grammars to 7, and the number of symbols on each rule's right-hand side to match the maximum in the grammar used to generate the training data. Also, the number of nonterminals is restricted to be exactly the same as the number in the target grammar. The latter two settings represent heavy-handed hints given to his algorithm, but he goes still further in including information about the target in the fitness function. Wyard's fitness function gives +1 for each correctly classified example, -1 for each incorrectly classified example, and -1 for every nonterminal above the number occurring in the *target grammar*. The stated motivation is to prefer more compact grammars, but choosing the threshold at which to introduce a penalty based on a known target grammar is unreasonable.

The data above show that EvoGrams successfully solves grammar induction problems commonly reported as benchmarks in the literature. Further, EvoGrams generally makes fewer assumptions than the other algorithms considered here, though it does not always perform better. Finally, EvoGrams performs markedly better than either EA considered here despite its lack of negative examples. This may be because EvoGrams uses samples that contain stochastic information in the sample in the form of duplicate strings. We investigate this issue in detail in the next section.

V.C Positive and negative examples

In this section we will use the Parity and AtLeast(k, n) languages defined above to investigate the effects of sample properties on the ability of EvoGrams to learn grammars for these languages. The sample properties we will be manipulating are the availability of positive and negative examples in the training data, and the use of stochastic information in the sample. By "stochastic information" we mean relative frequencies of strings in a data set generated from a correct stochastic grammar for the target language. None of the runs in this section used the complexity measure as a part of the fitness function, since all of them were performed before the complexity measure was introduced.

In the case of parity, positive and negative examples exist in equal quantities in Σ^*. In the AtLeast(k, n) languages, the availability of useful negative examples is manipulated by changing the parameter k. In all cases, we will give more time to runs that have fewer samples as a way of equalizing time spent running EvoGrams using the different data sets. Throughout this section, we will use the Mann-Whitney U test to compare results for different sets of runs; in the following subsection, comparisons between results for different collections of 10 runs will reported by giving the Mann-Whitney U_{obs} value and the corresponding significance level. The Mann-Whitney U is useful here because it uses weaker assumptions than a t-test: ranked data is acceptable instead of continuous data, and the data need not be assumed to have

Sample name	$\|S\|$	Description
All	126	All positive and negative examples from length 1 to 6
Pos	63	All positive examples from length 1 to 6
Stoch	126	Stochastic sample generated by Γ_{parity}
ShortStoch	126	Another stochastic sample generated from Γ_{parity}, but restricted to only contain strings of length from 1 to 6 when generated
Test	2046	The test set used for post-hoc comparison of results; all positive and negative examples of length from 1 to 10

Table V.6: Data sets for learning parity

been drawn from a normal distribution.

V.C.1 Learning Parity

In this section, we will use parity as the target language and try to learn it from a variety of samples. For each training sample described in table V.6, EvoGrams was run 10 times to try to learn a grammar for it.

We want to ask a particular set of questions about the performance of EvoGrams using the various samples in the table. In particular, we want to investigate the trade-off between learning using positive and negative examples and learning using stochastic information. We start by running using the **All** and **Pos** data sets and comparing the results. Next, we run using the **Stoch** data and compare it to **All** and **Pos** to see whether the stochastic sample is as good as the complete sample for short strings. These comprise our basic suite of tests.

The **Stoch** sample turns out to contain some examples longer than 6 strings and these might be responsible for any differences noticed, instead of the frequency data in the stochastic sample making a difference. To resolve this issue, we generate a special stochastic sample, **ShortStoch** of strings no longer than 6 characters. Because this sample contains 126 strings, $|S| = 126$ and the data was generated stochastically using Γ_{parity}, there turn out to be less than 126 distinct string in **ShortStoch**, $N < 63$, since all of the examples are positive and some repeats occur. This means

Comparison of results for...	U_{obs}	Significance, p=
All v. **Pos**	26	0.05 (directional)
All v. **Stoch**	54	not significant (nondirectional)
Pos v. **Stoch**	73	0.05 (directional)
All v. **ShortStoch**	60	not significant (nondirectional)
Pos v. **ShortStoch**	80	0.025 (directional)

Table V.7: Results of comparisons using Mann-Whitney U for learning parity from different data sets.

ShortStoch contains no more unique examples than **Pos**, but it contains distributional information over the examples it does contain. We compare the results using **ShortStoch** to those using **All** and **Pos** as with **Stoch**. The results of our statistical comparisons are given in table V.7.

Note that comparisons to **Pos** have been performed as directional tests and comparisons between **All** and the stochastic samples are nondirectional. This is because **Pos** clearly contains less information than any of the other samples and we should expect it to do worse, thus justifying the directional test. Since we do not know have reasonable prior expectations about one sample or another resulting in grammars more consistent with the test data, we cannot justify a directional test in these two cases.

The results show that the sample **Pos** leads to significantly poorer grammars from the point of view of consistency with test data. We cannot, however, detect differences among the two stochastic samples and the complete up to string length 6 sample. In these cases, the target language is exactly learned so often that it is hard to distinguish between the results using these samples. It is clear, however, that **ShortStoch** was the overall best sample to use because it contained the fewest distinct strings, these were of length no greater than 6, and using it as training data, the target was learned exactly in all 10 runs. Contrast this with using sample **All**, in which the strings are short, but all are distinct, requiring more parses to evaluate each grammar at each generation of the GA.

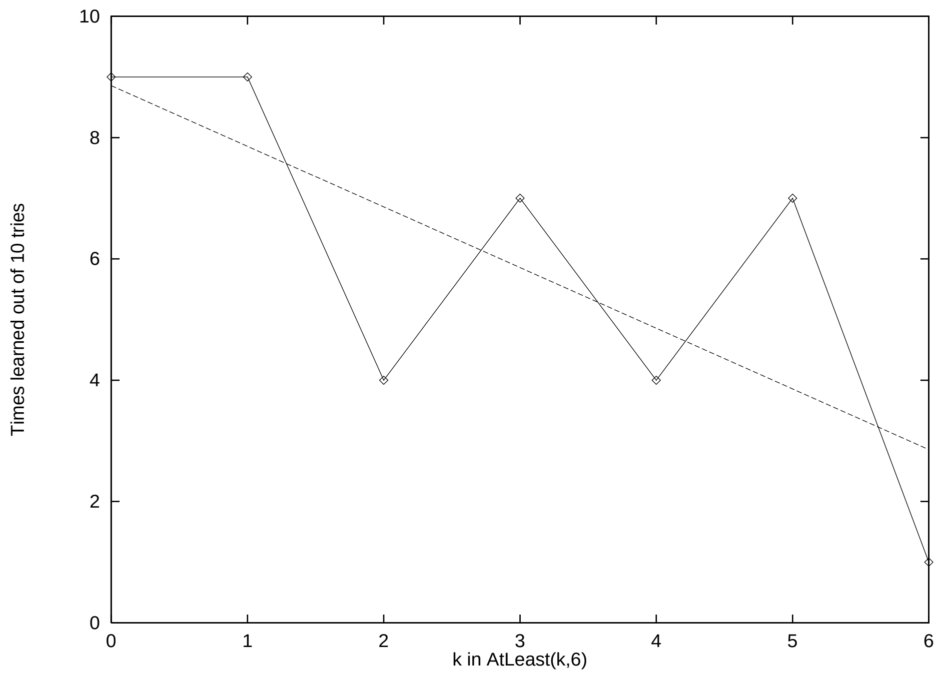


Figure V.5: Results for learning the AtLeast(k, n) languages

V.C.2 Learning AtLeast(k, n)

In the case of parity, we provided or denied positive examples as opposed to stochastic information in the sample. Here, we will explore a collection of languages for which we always provide a data set containing all strings of lengths 1 through 6, labelled as positive or negative depending on the value of k for the AtLeast$(k, 6)$ language we're considering. For each value of k run EvoGrams ten times to try to learn from this sample containing all possible examples for AtLeast$(k, 6)$. The results will tell us how hard it is for EvoGrams to learn from such a *complete* sample of a target language.

Figure V.5 shows how many times the correct grammar was induced out of 10 tries for each value of k. The solid line is a trend line through all of the data points. The straight dashed line is the regression line based on the Pearson's r correlation coefficient. The correlation coefficient is -0.7278, and is significant with $p < 0.07$.

The significant anti-correlation between k and the number of successes out of 10 runs suggests that as the target language shrinks, evolving a grammar becomes harder and harder. This may be because more and more early hypothesis grammars will be judged to be worse and worse as k increases. For example, the simple grammar which accepts every string of 1's becomes a worse and worse candidate as k increases to n.

An interesting feature of the data is that the $k = 0$ case is easy, as we would expect, since all strings in the training sample are positive examples. Also, the $k = 6$ case is much more difficult, since all 126 possible examples are present and all but 1 are negative examples. This means that most attempts to build even a fragment of a grammar are likely to fail since any partial strings of one derived will turn out to be a negative example. Because we are including strings of many lengths in the training data, the problem is harder than a 1-max problem on fixed-length strings. The cases $k = 2$ through $k = 5$ fluctuate between 4 and 7. The even values of k seem to present more of a challenge to the algorithm than the odd values.

An interesting experiment would be run EvoGrams on the same languages, providing only positive examples. Many of the grammars learned will be likely, at least early in a run, to generalize incorrectly while capturing all of the positive examples. When $k = 6$, for example, there would be a single positive example in the positive-only data set for the language $\text{AtLeast}(6, 6)$. Through the generations, grammars that assign the highest probability to this single example will be rewarded, and eventually a grammar may arise which assigns it probability 1, meaning that it is the only positive example recognized by the grammar's characteristic grammar[5]. If this happens, the problem has been solved. This may be less likely when using all possible negative examples, since grammars that overgeneralize would be strongly discouraged when they arise because they would lose fitness for assigning probability to negative examples more than they would gain credit for accepting the lone positive example. In the case of the singleton positive example, the algorithm is more likely to tolerate gross overgeneralizations that could be refined over time.

[5]Recall that a stochastic grammar Γ's characteristic grammar, $\text{Ch}(\Gamma)$ is Γ without production probabilities.

V.D Wildcards

Here we will use the three samples described in section V.A.4 to determine whether the wildcard symbol mechanism in EvoGrams, as described in section IV.C, is used when the data is restricted to strings of a single length. In this case, the target grammar, Γ_{pal}, describe palindromes over $\{0, 1\}$.

EvoGrams was run both with and without the wildcard character for the fixed-length samples, and the results of these runs were compared[6]. The fixed-length samples should encourage use of the wildcard symbol since many columns of the fixed-length data sets are about 50% 1's and 50% 0's. The parameters for these runs are identical to the other benchmark runs, except that in two sets of runs we use the wildcard symbol, '*,' and the number of generations and population size is varied to account for differences in sample sizes.

First, to make sure the language wasn't too hard to learn even from a reasonable sample of positive examples, we attempted to learn the palindromes from the data set derived from the target grammar. Out of ten runs, we successfully learned a grammar for the palindromes 9 times. The evolved grammars were tested on a large sample containing all labelled positive and negative examples of length up to 10, plus 100 randomly chosen examples for each of the lengths 21 through 30[7]. The latter examples were chosen by flipping a fair coin to decide whether each example was positive and then generating a random string until a string with the correct label, positive or negative, occurred. Each random string was generated by choosing each bit independently according to a flip of a fair coin. This leads to a large test set that can be used to accurately predict whether a given grammar is a correct palindrome grammar. The ten runs just mentioned were evaluated against the test set, and it was observed that the only grammar of the ten not scoring perfectly on the test set was the one grammar which was not a correct grammar for the palindromes.

Next, we attempted to learn palindromes from the fixed-length data sets

[6] None of the experiments in this section used the complexity measure to modify fitness.
[7] This gives a test set of 3046 distinct strings.

both with and without the wildcard symbol in use. The wildcard symbol generated a uniform distribution over the two terminals when it was used. For each of the two lengths(8 and 16), we ran EvoGrams five times with and five times without the wildcard in use. None of these runs found a perfect grammar for palindromes, but in the case in which the data was of length-16, the different ways in which the grammars failed were interesting. For the length 8 data, there was no observable significant difference in consistency with the test set between using and not using the wildcard, according to a nondirectional Mann-Whitney U test[8]. For the length-16 data, the runs with and without the wildcard were significantly different at p=0.1, again for a nondirectional Mann-Whitney U test[9].

The reason for the significant difference in the length-16 case was that two of the runs found much worse grammars than the other three. The mistakes being made by grammars in the length-16 case for grammars not containing the wildcard in any production were always owing to an inability to generate short positive examples in the test set. That is, none of the negative examples in the test set were accepted as positive by these grammars; rather, the grammars reflected the biased length of their training sets and could not generate some of the palindromes shorter than length-16. Some of these reflected the bias of the sample to even-length strings and couldn't generate odd-length palindromes. These observations apply to *all* of the grammars learned from the length-16 data set except two of the grammars learned with the wildcard allowed. All of these comparatively better grammars were consistent with $0.89 \pm \epsilon$ of the test set strings, with $\epsilon < 0.01$.

The two grammars that did less well than the others on the length-16 data were both consistent with 0.79 of the test set. The reason for their lower consistency values was that these two grammars generated some non-palindromes. A side-by-side comparison of representatives of these two varieties of solution to the problem of learning palindromes from length-16 examples is shown in figure V.6. The grammar on the left uses the uniform wildcard symbol "*." Every derivation from the grammar

[8] $U_{\text{obs}} = 4.5$, both groups have size 5
[9] $U_{\text{obs}} = 4$, both groups have size 5

Start symbol: C				Start symbol: C			
C	$\rightarrow$	IGI	(1)	C	$\rightarrow$	0C0	(0.444)
J	$\rightarrow$	0*	(0.283)	C	$\rightarrow$	10A	(0.025)
G	$\rightarrow$	11	(0.5)	C	$\rightarrow$	1FA	(0.019)
I	$\rightarrow$	J*JJ	(1)	A	$\rightarrow$	01	(1)
J	$\rightarrow$	1*	(0.566)	C	$\rightarrow$	11	(0.049)
G	$\rightarrow$	00	(0.5)	F	$\rightarrow$	000	(0.616)
J	$\rightarrow$	01	(0.009)	F	$\rightarrow$	00F	(0.384)
J	$\rightarrow$	*0	(0.142)	C	$\rightarrow$	1C1	(0.462)
				C	$\rightarrow$	F0F	(0.002)

Figure V.6: Grammars learned from a small set of length-16 palindromes, evolved with (left) and without (right) the wildcard symbol allowed.

on the left contains at least one "*" and so this grammar is too general to be a grammar for palindromes only. The grammar on the right, however, generates many palindromes and no non-palindromes, though it generates no palindromes shorter than symbols 4 except "11." Thus, we have evidence that the wildcard character can interfere with learning when its allowed but is not needed.

V.E Structure versus probability

We examine the results of 10 runs when the training data is from a context-sensitive language. Recall the following definitions from section III.D.

$$
\begin{aligned}
\mathcal{L}_{\mathrm{abc}} &= \{a^n b^n c^n | n \geq 1\} \\
\mathcal{L}_{\mathrm{ab}} &= \{a^n b^n c^k | n, k \geq 1\} \\
\mathcal{L}_{\mathrm{bc}} &= \{a^k b^n c^n | n, k \geq 1\} \\
\mathcal{L}_{\mathrm{ac}} &= \{a^n b^k c^n | n, k \geq 1\} \\
\mathcal{L}_{\mathrm{a|b|c}} &= \{a^k b^m c^n | m, n, k \geq 1\}
\end{aligned}
$$

Complexity used?	$\mathcal{L}_{\mathrm{ab}}$	$\mathcal{L}_{\mathrm{bc}}$	$\mathcal{L}_{\mathrm{ac}}$	Match b,c
Yes	0	3	1	1
No	1	2	2	0

Table V.8: Number of times various languages were learned from the data for $\mathcal{L}_{\mathrm{abc}}$.

The training sample was drawn from the grammar for $\mathcal{L}_{\mathrm{abc}}$ given above in section V.A.5. EvoGrams was run 10 times on the training data with each run stopped after 1000 generations. Five runs used the complexity measure in the fitness function and five runs did not. Table V.8 summarizes the languages derived by the best-of-run grammars for these simulations.

First, all of the grammars found were consistent with the entire training set, though they differed in how they generalized from it. Most of the runs found expected context-free generalizations. The one unusual run, occurring in the case when complexity was used to modify fitness, produced a grammar which generalized language $\mathcal{L}_{\mathrm{bc}}$. This grammar allows the 'a' terminal to appear between instances of the 'b' terminal.

A representative grammar from the runs using the complexity measure is[10]:

Start symbol: C

$$
\begin{array}{lcll}
\mathrm{C} & \rightarrow & \mathrm{aAc} & (1) \\
\mathrm{A} & \rightarrow & \mathrm{C} & (0.669) \\
\mathrm{A} & \rightarrow & \mathrm{F} & (0.331) \\
\mathrm{F} & \rightarrow & \mathrm{bF} & (0.669) \\
\mathrm{F} & \rightarrow & \mathrm{b} & (0.331)
\end{array}
$$

which corresponds to $\mathcal{L}_{\mathrm{ac}}$. This grammar is shown after tuning by the inside-outside algorithm which gives its productions probability values that ensure that the same number of each terminal is generated on average.

The grammars evolved without the complexity measure used in fitness were usually a little more complex, having 8 to 10 productions instead of 4 to 6 produc-

[10]Note that all of the probabilities on the grammars here have been rounded to three decimal places

tions. The complexity of the grammars evolved with complexity was 30.8 and that of the grammars with 41.0. The difference is significant using a one-way t-test at the significance level $p = 0.05$. A representative example of the grammars evolved without complexity is:

Start symbol: D

D	$\rightarrow$	aLc	(0.69)
D	$\rightarrow$	H	(0.31)
L	$\rightarrow$	aLc	(0.659)
L	$\rightarrow$	Ec	(0.341)
H	$\rightarrow$	Rc	(1)
E	$\rightarrow$	Eb	(0.659)
E	$\rightarrow$	Rb	(0.341)
R	$\rightarrow$	ab	(1)

which also generates the language $\mathcal{L}_{\mathrm{ac}}$.

The unusual grammar which arose when complexity was used generalizes $\mathcal{L}_{\mathrm{bc}}$:

Start symbol: B

B	$\rightarrow$	bBc	(0.506)
B	$\rightarrow$	bc	(0.173)
B	$\rightarrow$	aaaB	(0.101)
B	$\rightarrow$	aaB	(0.0962)
B	$\rightarrow$	abc	(0.0775)
B	$\rightarrow$	aaaaB	(0.0457)

This is similar to some of the solutions which generated $\mathcal{L}_{\mathrm{bc}}$, but uses only a single nonterminal. The result is that the terminals in the generated strings may not all be in correct order, though the relative numbers of the two terminals 'b' and 'c' are maintained correctly. Thus, 9 of the 10 grammars learned a proper ordering over the terminals in strings in the target language and learned to match the frequencies of two terminals, the best that can be done with a context-free grammar. One of the grammars succeeded only in matching frequencies of two terminals and missed some of the ordering information, though not all: terminal 'c' does not occur intermingled with 'a' or 'b.'

The grammars discovered by all 10 runs were grammars for infinite languages, though they aren't infinite extensions of the target language which satisfy the Savitch's definition of essential infinity[71, 70]. The grammars do have the property, however, of being less complex than definite grammars for the training set, if we use the grammar complexity measure derived in chapter IV. This is true even when complexity was not used to modify fitness during the EA run.

In fact, however, the (finite) training data language just described is likely to be essentially *finite* with respect to the context-free languages. Nonetheless, the learned grammars are reasonable and simple context-free models of data taken from a context-sensitive language. This suggests that the definition of essential infinity of a finite set of strings F could be varied for stochastic languages to take into account the probability that a grammar assigns to F, even if the characteristic grammar isn't a precise enough model to justify the usual definition.

V.F Ability to hold a known good solution

Our experiments in this section are designed to test operator rate settings to determine their effects on GA robustness[11]. The names of the parameters and the values considered for each are shown in table V.9. The maximum number of generations for all of the runs reported here was 50, and these runs did not use the complexity measure.

The analysis usedhere is an analysis of variance with repeated measures[63, 23] [12]. A repeated measures analysis applies because a collection of 4 initial populations were used, each specified by a value of "seed rate" and a value of "noncoding region insertion" in table V.9; each initial population includes not only an entire starting population, but also the random number generator state to restore when starting

[11]The results in this section have been previously published[51].

[12]Appendix A explains the relevant aspects of the analysis of variance.

Parameter name	Values
Seed rate (fraction of initial population)	$0.01, 0.1$
Noncoding region (in seeded genomes)	Between, After
2-pt Crossover rate (per individual)	0.0, 0.157, 0.4, 0.5
Inversion rate (per individual)	0.0, 0.0025, 0.01, 0.157
Substitution rate (per individual)	0.0, 0.0025, 0.01, 0.157
Mutation rate (per allele)	0.0, 0.0005
Inside-outside algorithm iterations	0, 2

Table V.9: GA parameter settings that were varied in these experiments, and their values.

from the population. Each of these four initial populations was used to start a run for every combination of settings of the other parameters; the factors whose influences we analyzing using ANOVA are the four genetic operators' rates. The repeated measures ANOVA makes use of the fact that we are measuring the same starting points under different parameter settings to reduce sampling error. Since we will analyze the cases with and without the inside-outside algorithm using two separate ANOVAs, we have two sets of 512 measurements, each of which is subdivided into 128 groups of 4.

Once we complete the simulations, we have a large collection of data to use as input to a multifactor analysis of variance. The parameters are referred to as "factors." For every subset of the factors, we have a set of groups given by the combinations of values of factors in the subset. In our case, the factors have predetermined values which we chose as characterizing an area of the GA's parameter space that interested us. These factors are "fixed" as opposed to "random." Because of this, we cannot make inferences to broader space of operator settings. To do so, we would have had to randomly sample the operator values instead of choosing them.

Initial populations were generated by first generating a completely random population and then replacing part of it with instances of the seed grammar. For a seed fraction f, the first $f \times \text{Population} - \text{size}$ individuals are replaced with Γ_{seed}. The population size was held at 100 for all of these runs. If the noncoding region insertion option is "after," a noncoding region is added to the end of the genome for

	Without Local Search		With Local Search	
Source, S	F(S) (d.f.)	Significance	F(S) (d.f.)	Significance
All	447.759 (1,3)	< 0.001	482.868 (1,3)	< 0.001
Substitution	–	–	15.861 (3,9)	0.001
Inversion	19.950 (3,9)	< 0.001	20.610 (3,9)	< 0.001
Mutation	45.600 (1,3)	0.007	296.802 (1,3)	< 0.001
Crossover	37.231 (3,9)	<0.001	37.399 (3,9)	< 0.001
Mutation, Crossover	12.932 (3,9)	0.001	29.480 (3,9)	< 0.001

Table V.10: Significant effects from ANOVAs on the data for the observations from the experiments with and without local search. Entries that say "< 0.001" were reported as 0.000 by UNIXSTAT:anova.

Γ_{seed}. When noncoding regions are of type "between," noncoding regions are placed between the elements of each grammar production and between productions on the genome. The maximum genome length is fixed at 2000, and the maximum initial genome length is fixed at 1000. The noncoding regions are generated randomly, but in such a way that on average the size of the whole genome will be close to 1000.

The value measured for each run was the percent of the final population whose fitness was at or above the seed grammar's fitness. This is the value that each run contributes to the various groups in which it participates in the analysis. The data files, as mentioned in section V.A.6, all contain the same collection of 41 short protein fragments. We now present results for the two cases not using and using the inside-outside algorithm, respectively.

We performed two separate ANOVAs: one on observations gathered without and one on observations gathered with the local search algorithm in use. In the discussion that follows we will examine the main effects and interactions for each case. Finally, we will treat the crossover-mutation interaction separately. Before discussing the effects of varying operator rates, we note that when all operators had rate zero, all four initial populations converged to contain nothing but the seed grammar within the allotted 50 generations.

Table V.10 lists the sources displaying significant effects for both sets of observations. The substitution operator showed a main effect only when local search was in use. Each other operator showed a main effect in both ANOVAs. The “All” source contains all four operators; its significance indicates that at least two of the 128 groups arising when all factors are given a level are different. We do not perform contrasts in this paper to determine whether the other observed effects account for this interaction completely.

V.F.1 Main effects

In general, the disruptive effect of each operator increases with the operator’s rate: the fraction of the final population with fitness at or above Γ_{seed}’s fitness decreases as the operator’s rate increases. This is true for the main effects for crossover, mutation, and inversion. Substitution has no main effect unless local search is in use, in which case its effect does not follow the typical pattern above.

The main effect for substitution when local search is used occurs because of only two significant differences between pairs of group means: the differences between the groups with substitution rates 0.002 and 0.157, and between the groups with substitution rates 0.01 and 0.157. The highest substitution rate, 0.157, is significantly worse than the next two lower rates, 0.01 and 0.002. The 0.0 substitution rate, however, is worse than both 0.01 and 0.002, though not significantly. This was the only operator for which middle values were better than either extreme value. Some substitution appears to be better than none, but the highest substitution rate was too high.

V.F.2 The Crossover-Mutation interaction

Table V.11 gives the means for the groups for crossover and mutation, with and without local search. In both cases, there are 8 groups with 64 runs in each group, and the number of degrees of freedom within groups is 9. In the case of the non-local-search experiment, all but 5 of the pairwise means are significantly different,

	Mutation	
Crossover	0.0	0.0005
0.0	0.9702 / 0.9705	0.8619 / 0.8245
0.15	0.8527 / 0.8669	0.8175 / 0.7959
0.4	0.6713 / 0.6792	0.6272 / 0.6300
0.5	0.5995 / 0.5881	0.5217 / 0.5211

Table V.11: Individual means for the groups induced by crossover and mutation in ANOVA experiments for runs with and without local search. Each cell gives the mean for experiments without local search followed by the mean for experiments with local search.

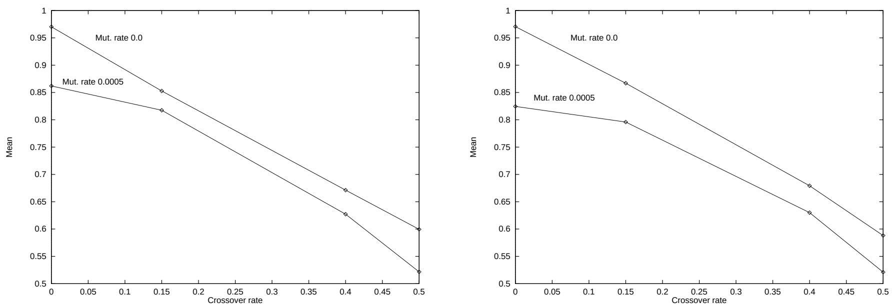
Tukey's HSD = 0.047. In the case of the experiment with local search included, Tukey's HSD = 0.039 and all but one of the pairwise comparisons is significant. The significance for all pairwise comparisons is $\alpha = 0.01$. Table V.12 shows exactly which pairs of groups were significantly difference with and without local search.

Figure V.7a and figure V.7b show how the means change for the experiments without and with local search, respectively. The means are graphed on the vertical axis and crossover rates are on the horizontal axis; each curve represents a mutation rate. Between the middle two crossover rates, a slight amount of mutation versus none has about the same effect on the amount of disruption from crossover. At the more extreme values of crossover rate, however, the difference between a little mutation and no mutation is noticeable, to the extent that at mutation rate 0.0005, the difference between crossover rates 0.0 and 0.15 is no longer significant.

The interaction discovered between crossover and mutation demonstrates that combinations of genetic operators can have combined effects in addition to the sum of their individual effects. This suggests that analyses based on separate consideration of each operator without attention to potential for "crosstalk" between operators could miss significant phenomena. Further, our experiments revealed an unexpected dependence of the main effect due to substitution on the presence of the local search mechanism. This suggests that for some operators disruptiveness will have to be considered in light of components of the EA other than the other oper-

		Significant differences with following groups	
Mutation	Crossover	Without Local Search	With Local Search
0.0	0.0	-*******	-*******
0.0	0.15	-**--**	-******
0.0	0.4	-***-*	-*****
0.0	0.5	-**-*	-****
0.0005	0.0	--**	--**
0.0005	0.15	-**	-**
0.0005	0.4	-*	-*
0.0005	0.5	-	-

Table V.12: Significant differences among pairs from the eight groups defined by setting levels for the mutation and crossover factors. Each cell shows a map of significant differences for the row compared with itself and each row below it. A '*' indicates a significant difference between a pair of groups.



(a) (b)

Figure V.7: Graphical depictions of the interaction between crossover and mutation (a) without local search and (b) with local search.

ators. The present case is surprising because the relationship between local search and substitution reaches across the boundary between the genotypic and phenotypic, demonstrating that causally distal components of an EA can modify one another's effects.

It is important to realize that these observations concern a purposefully limited experiment, which test only for our EA's robustness and not its exploratory ability. Since we measured only the percent of the final population whose fitness was at or above the seed grammar's fitness with respect to our data set, we have only measured the extent to which the algorithm didn't lose ground compared to its starting point. Further, because the operator rates were treated as fixed factors, it is not statistically valid to draw inferences from the results for the operator rates considered to other rates. A better experiment might use a randomized block design, for example, to choose operator rates randomly within fixed subranges.

V.G Learning and Evolution

The grammar Γ'_{01}, defined in section V.A.3, generates all of Σ^*, but assigns higher probability to strings $\sigma \in \Sigma^*$ with the same numbers of 0's as 1's relative to other strings $\sigma' \in \Sigma^*$ such that $|\sigma| = |\sigma'|$. In this section, we will examine attempts to learn the stochastic language generated by Γ'_{01}. The results presented in this section were previously published and the fitness function did not contain the explicit cost for complexity discussed in chapter IV[50].

Generalizing the target language over all of Σ^*, has the potential to give us an important advantage. Local search should never hurt us for this problem! Imagine comparing the target grammar's fitness with the fitness of a grammar which contains only the *'d productions from Γ'_{01}. Both grammars generate all of Σ^* with nonzero probability assigned to every string. Without local search, a particular training sample might reward the *-only grammar more than the target, if it contained a disproportionate number of strings whose numbers of 0's and 1's differed. If local search is used, however, and is “deep” enough, meaning that enough iterations of the inside-outside algorithm are allowed to allow both grammars to fully tune to the data, then the target grammar should do no worse than the *-only grammar, since at worst it could simply tune the probabilities of the non-* productions arbitrarily close to zero if this made the training sample as likely as possible.

Conversely, assume a training sample which more accurately reflects the target language. Then the target grammar should be favored, since it contains productions allowing it to capture the correct nonuniform distribution of probability over Σ^*. Thus, the correct grammar should be favored when found, and grammars like the *-only grammar but with a subset of the targets extra productions thrown in should also be favored for reasons similar to those above. Thus, we expect that the GA with local search will, on the whole, do as well or better than the GA without local search on this problem.

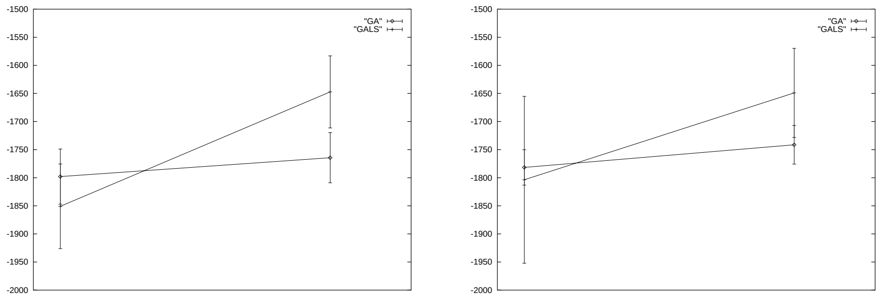
Initially, we ran a set of simulations to demonstrate the need for frequency information in the sample. These stripped all frequency information by including only one copy of each distinct string from the original learning sample. When this reduction was done, the size of the sample was only $|S| = 152$. None of these runs learned the correct grammar or anything close with or without local search.

Consequently, all subsequent runs made use of the full $|S| = 1000$ sample containing replicated strings. These results are summarized in figure V.8a and figure V.8b. These show that populations that had benefit of local search throughout the GA run are more successful than those without such a benefit. It is interesting to note, however, that individuals in these final populations are not immediate, genetic successes but only succeed after they are subjected to the same training that shaped their ancestors. Conversely, the final generation of individuals from the runs which did not enjoy the benefit of local search, improve only marginally from just one post-hoc exposure to training.

Unlike the other results in this chapter, we didn’t evaluate each grammar at each generation on all strings in S. Rather, we “subsampled” S every generation to get a set of 10 strings called R. These 10 strings were used as the test set and were rechosen from S every generation. The simulations were run to 1250 generations for the GA-only (GA) runs, and 625 generations for the GA-plus-local-search (GALS) search runs. The number 1250 was an arbitrary choice based on how long the entire suite of runs could be allowed to consume; the runs with the inside-outside algorithm consumed on average about twice as much CPU time per generation as the GA-only runs, and thus were allowed half as many generations. The maximum number of inside-outside algorithm iterations was 10. The maximum genome length was 512, and the length of the initial populations genotypes was 128 characters. The maximum length of production right-hand sides was restricted to 3, and the maximum length of genotypic encodings of probabilities was 5. The genome was able to accommodate up to 32 productions. The terminal set was $\Sigma = \{a, b\}$, and the nonterminal set $N = \{A, B, C, D, E, F, G, H, I, J\}$. Assuming that the search space included grammars with only up to 16 productions, the number of grammars in the search space is about 1.4×10^{79}. Ignoring probability fields, the number of grammars is 1.1×10^{55}.

We ran 10 of the GA simulations and 10 of the GALS simulations. To measure each run's performance, we evaluated each final population on a test set of 1000 strings, which was the same for all runs and was generated separately from the training sample S. Evaluation on the test set did not use subsampling, but computed fitness over the entire collection of 1000 strings to achieve a good estimate of true fitness. Our performance measure for all runs is the best fitness on the test set over all individuals in the final population. This measurement is made both with and without local search, even for runs which used no local search, since it would be unfair to compare two final populations of grammars when only one was tuned using the inside-outside algorithm on the test sample.

The difference between the GA and GALS runs in figure V.8a, when no local search was performed in final evaluation (left-hand side of figure) was not significant (Mann-Whitney U test, Uobs=28). When local search was performed, the runs trained with local search did significantly better than those not trained with local



(a) (b)

Figure V.8: Results for learning the stochastic language $\mathcal{L}(\Gamma'_{01})$ with and without learning during evolution

search (Uobs=93, significance level 0.01, nondirectional test). Final testing seems to have made some difference for the GA runs in figure V.8a, since the difference between the fitness scores before and after local search is significant (Uobs=79, significance level 0.05). Local search, as one might guess from the figure, was used much more effectively by those runs trained with it, and the difference was significant at the 0.01 level (Uobs = 98).

The above results indicate that the difference between GA and GALS runs was based on the use of the local search operator during simulations, as opposed to only during final testing. An obvious question that arises is: Did we need to do ten iterations of the inside-outside algorithm per individual per generation, or would fewer have sufficed to produce the effect in figure V.8a?

We ran the GA and GALS runs again, but this time we only allowed 1 iteration of the inside-outside algorithm per individual per generation in the GALS runs. The results are summarized in figure V.8. In addition, the number of generations was higher and was the same for both sets of runs, 2000. One iteration of local search turns out to result in negligible overhead; the CPU times of GA and GALS runs in this case showed no significant difference using a t-test even though the number of generations did not differ between the two. Once again, the non-local-search final evaluations were not statistically different across the two sets of runs (left side of the figure). With local search in the final evaluation, however, the GALS runs performed better than the GA runs (Uobs=85.5, significance level 0.01).

Both sets of runs improved significantly at the 0.01 level this time when their final evaluations used local search (Uobs=91 for GA, Uobs=86 for GALS). Note that the only difference between the GA runs in this case and those in the previous case was that these runs were 750 generations longer. Finally, figure V.8b, and figure V.8a to a lesser extent, shows more variation in the GALS runs than in the GA runs, as seen in the size of the error bars. Note that this is the variation in the fitness of the best of final population over ten runs when evaluated on the 1000-string test set, not the amount of variation in the final population's fitness values during the run. In both cases, the range of variation of outcomes is wider for GALS runs, but these runs are almost always better than the GA runs. For the runs corresponding to figure V.8a, we performed a Mann-Whitney test to check whether the standard deviations of the whole final populations were different between the GA and GALS runs both with and without local search in the final testing. No statistically significant results were found. We also performed such a test for the right-hand side of figure V.8b, and found weak significance (at the 0.1 level). We lacked the data to do this test for the left-hand side of figure V.8b, the case where final testing does not use local search. The GALS runs show a wider dispersion in the figures, but this does not indicate a larger ending population variance for those runs.

The lesson we can learn from figure V.8 is that evolving grammars with the learning algorithm, the inside-outside algorithm, in place is more effective than evolving without it. In particular, grammars evolved with the inside-outside algorithm are significantly better than those evolved without it, even when all of the grammars involved are allowed to benefit from a large number of iterations of the inside-outside algorithm after the GA has finished. Though there is not a specific component of the fitness evaluation to support the ability to make effective use of the inside-outside algorithm, grammars that are able to do so have a seem to have a selective advantage. That is, we are at least indirectly selecting for grammars that effectively use the inside-outside algorithm and these tend to be successful, as evidenced by the results reported above.

Chapter VI

Evolving Sorting Networks

This chapter will describe the use of a grammar representation as the basis for an evolutionary search for simple sorting algorithms called "sorting networks"[1]. Basic characteristics of the EA used for the work in this chapter are discussed first, and then its application to evolving sorting and merging networks.

The previously discussed EA, EvoGrams, was used in a way that was consistent with the paradigm of the lower-left cell of table II.1. In this chapter, the fitness evaluation regime of the upper-right table cell is explored: evaluating strings derived from grammars according to some figure of merit which defines fitness. In this case, the strings derived from the grammars are formally simple algorithms to merge two lists already in ascending order into a single list in ascending order.

The work presented in this chapter also makes contact with the material discussed in section II.D. The programs represented by the grammars are not the merging algorithms, however, but the developmental programs which build the merging algorithms. These developmental programs are tested by sampling strings derivable from the grammars, which are tested to see how well they perform the merging task. As will be seen below, the grammars are being used as programs which recursively build up merging algorithms for lists of length N out of merge algorithms for lists of size $N/2$. Some of these programs will by a purposely applied representational bias

[1] This work has been previously published in the journal "Artificial Life"[52].

in the grammars, encode correct sorting algorithms as well.

VI.A Overview and Definitions

A compare-exchange network, or CMPX-net[2], is a sequence of operations, each written $[i : j]$, which operate on an array, D, of length N. The network is said to have "width" N. The "length" of the network is the number of CMPX's in the network. For each $[i : j]$, $i < j$ and $i, j \in [0, N-1]$. To apply a CMPX-net to an array, swap $D[i]$ and $D[j]$ if $D[i] > D[j]$ for each $[i : j]$ in the sequence of CMPX's defining the network. A "sorting network"(SNet) is a CMPX-net which will sort D's contents into nondecreasing order no matter how D's contents are ordered initially. The simplicity of the control structure for sorting networks (one pass through, executing each instruction in sequence) does have a cost in that some redundant CMPX operations will be done for some inputs. Knuth explains this well in volume 3 of The Art of Computer Programming (note that n is the number of inputs, m the number of operations):

> "... (Note that the same number of comparisons is made on each level, so there are 2^m outcomes after m comparisons have been made; since $n!$ is not a power of 2, it follows that some of the comparisons will be redundant, in the sense that one of their subtrees can never arise in practice. In other words, some branches of the tree must make more comparisons than necessary in order to ensure that all of the corresponding branches of the tree will sort properly.)" [55, p. 220]

Knuth notes soon thereafter, however, that the simplicity of sorting networks is also their strength:

> "Our previous studies of optimal sorting have concentrated on minimizing the number of comparisons with little or no regard for any underlying data movement or for the complexity of the decision structure that may be necessary. In this respect sorting networks have some advantages, since the data can be maintained in n locations and the decision structure is 'straight line;' there is no need to remember the results of previous

[2] Throughout this chapter, "CMPX" is used to abbreviate "compare-exchange."

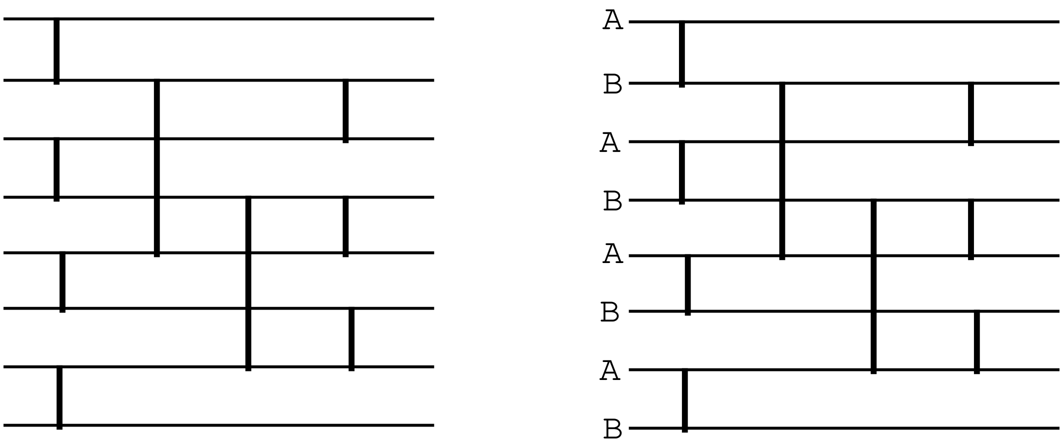


Figure VI.1: An 8-input CMPX network presented with and without a merge partition. The network on the left is a log-sequential sorting network. On the right, the same network is presented as a merging network.

> comparisons – the plan is immutably fixed in advance. Another important advantage of sorting networks is that it may be possible to overlap several of the operations, performing them simultaneously (on a suitable machine)." [55, pp. 222–223]

By "it may be possible to overlap several of the operations...," Knuth means that sequences of CMPX's such that no two of them have an index in common, can in theory be applied to the array in parallel. The minimum number of parallel steps in which a network can be executed is called that network's "depth" or its "delay."

Sorting networks gain simplicity by sacrificing the use of information about the outcomes of CMPX operations during the sort to decide which elements of the input to examine next. The sacrifice is that they will, for some input sequences, perform extraneous operations. Thus to search for sorting networks is to search for sorting algorithms subject to the constraint that they have the requisite simple control structure.

A "merging network" (MNet) is a pair containing a CMPX-net of width N, and a partition of the indices into two equal-size sets or "sides." In the following simulations, N is always a power of two and the partition divides the inputs into two sets with equal cardinalities. The partition is called the "merge partition" of the network, and may be construed simply as a labelling of the inputs with two symbols. In the picture below, the inputs are labelled with A's and B's to indicate the merge partition. If the data on each side of the partition are sorted initially then the output will be sorted.

The space of CMPX-nets with n_c CMPX's and width N is large, of size

$$\begin{pmatrix} N \\ 2 \end{pmatrix}^{n_c}.$$

Since MNets have associated merge partitions, they occur in a still bigger space whose size is the number of networks times the number of merge partitions,

$$\begin{pmatrix} N \\ N/2 \end{pmatrix}.$$

An EA is used to search for CMPX-nets which are MNets. The EA repeatedly samples the space of potential solutions to the merging problem in a series of generations, each using the relative fitness of the previous generation's samples to apportion more samples in promising regions. Mutation and especially cross-over operators are applied to generate similar but novel new sample points; this process is iterated until some stopping criterion is achieved.

This work concentrates on searching for merging networks, but with a bias that leads to the discovery of MNets which are also log-sequential sorting networks. Log-sequential sorting networks are CMPX networks which are guaranteed to sort their inputs when the input is passed through the network $\log_2 N$ times, where N is the width of the network. Equivalently, a log sequential sorting network is a network such that the network built by concatenating $\log_2 N$ copies of it is a sorting network.

Figure VI.1 shows a CMPX network drawn according to standard conventions: each horizontal line represents an index into the input array, D. If the lines are considered to be labelled from 0 to $N-1$ starting at the top, then a vertical bar between lines i and j stands for the CMPX $[i:j]$. The network on the left is a log-sequential sorting network. On the right, the same CMPX's with an appropriate merge partition function as an MNet.

VI.B Classic history

Volume 3 of The Art of Computer Programming discusses the state of the art for compare-exchange networks as of its writing and collects much of the theory which was then available[55, pp. 220–246]. Knuth discusses compare-exchange networks which sort, merge, or select on a list of numbers. He also gives the known bounds for the minimum number of compare-exchange operations needed for each task, and discusses the delay of sorting networks. The bounds known as of Knuth's writing were tight for less than or equal to eight inputs.

The best recursively-defined sorting networks as of Knuth's writing were discovered by K.E. Batcher and are built recursively out of merging networks[12, 55]. These merging networks are defined for any number of inputs and are called the Batcher odd-even merging networks. The Batcher sorting network on N inputs is made of two networks which sort $\frac{N}{2}$ inputs and a merging network that merges the output of the two smaller sorting networks. This scheme corresponds to the basic mergesort algorithm, with the Batcher merge used as the merging algorithm. A similar scheme to implement sorting networks based on quicksort could be accomplished using selection networks instead of merging networks.

A useful property of sorting and merging networks is that testing either kind of network on all possible sequences of zeros and ones is sufficient to establish that the network is correct. Knuth's book contains a proof of this principle for sorting networks, and a similar result holds for merging networks. The advantage of this principle from a machine-learning point of view is that sorting networks need "only" be tested on and exponential number of inputs, 2^N for an N-input network, and merging networks need only be tested on a polynomial number of inputs, $(\frac{N}{2}+1)^2$ for a network which merges two sequences of length $\frac{N}{2}$, or $(N_1+1)(N_2+1)$ for a network which merges two sequences of lengths N_1 and N_2.

VI.C Sorting networks since Knuth

Since the 1960s, interest in SNets has shifted away from particularly good solutions for SNets of fixed width, to generalized solutions which allow construction of networks of arbitrary width. Also, as VLSI fabrication of SNets become more and more feasible, SNet characteristics affecting implementation have also become more important. One obvious example is SNet depth or delay: all CMPX's within the same delay can be executed in parallel. It is less obvious that the fewer unique CMPX lengths used (the smaller the Net's "spectrum"), the more compact the network's topological structure, and thus the more compactly it can be laid out in VLSI.

The question of whether Batcher's recursive techniques were asymptotically optimal, however, remained open for fifteen years. In 1983, Ajtai, Komls, and Szemerdi answered this question with a resounding "no"[3]. The AKS construction makes use of randomized expander graphs. Constant degree graphs are randomly constructed to form SNets of constant depth. While this construction answered a question of long standing theoretical importance by proving the existence of SNets with optimal $O(log_2N)$ delay, the constant hidden under the asymptotic notation is so large as to make constructing such networks impractical.

A second milestone was achieved by Dowd, Perl, Rudolph, and Saks with their "periodic balanced sequential" SNet[27]. These SNets are log-sequential networks as described above. They have the additional property that they serve as merging networks in one pass, with the merge partition that divides the inputs into those with even and odd indices. They also proved an important fact about such blocks, that a sequential sorting network can only be constructed if the block contains all adjacent comparators[3].

The third important, recent result returns to Batcher's recursive odd-even merge design, and exploits the fact that the merging function can be defined with respect to an arbitrary partitioning of input lines into two sets. The "modified Batcher" networks, discovered by Canfield and Williamson, are defined as the Batcher merges, but employ a different merge partition[18]. Thus, the modified Batcher networks are constructed from the Batcher merges by renumbering the inputs. The result is a log-sequential sorting network.

It is important to note that the periodic balanced sequential and modified Batcher networks are not each a single network. Rather, they are families of networks defined for any number of inputs which is a power of two. The definitions of these networks tell how to build a $2N$-input network from two N-input networks. To do this, a partition of the inputs to the network being constructed, the *call partition* is used to specify how the inputs are to be divided among two N-input networks. Essentially, each side of the call partition is used to renumber the CMPXs in an N-input network before they are incorporated into the $2N$-input network. The $2N$-input network ends with a formulaic "patch" layer: a group of CMPXs which are added to the network after the renumbered sequences from the smaller networks. The idea of a call partition is fundamental to the genotypic representation, as will be seen below.

VI.D Grammars as a genotypic representation for evolutionary algorithms

The primary focus here is on an EA which incorporates a process analogous to biological development. If local search is performed on each individual in the population a representation with good neighborhood structure with respect to local search might be very bad from the point of view of the evolutionary algorithm's operators. Neural network/EA hybrids provide a clear example of this situation.

[3]A CMPX is termed adjacent if the difference between it's indices is 1.

Neural network learning algorithms work well if the neural network is given as a weight matrix or a list of weights from each layer of nodes to the next. The result of applying the evolutionary algorithm's crossover to such a network is known to be often disastrous, since two good networks that compute the same function may have different representations arrived at by permuting the network weights in blocks. As a result, a large amount of time and effort has been spent in efforts to alleviate this difficulty[72]. In this case, development allows the representation in which the EA searches, the "genotypic space," to be transformed into a representation amenable to local search, the "phenotypic space."

In cases in which no local search is to be performed, development may still serve to transform a representation amenable to search by a evolutionary algorithm into an object for testing. That is, good building blocks for a given problem may not correspond to obvious features of objects in the search domain, motivating a transformation from some representation which allows for good building blocks to a representation which is efficiently testable. Alternately, it may be the case that a good genotypic representation turns out to be one which requires a nontrivial, possibly stochastic, "unpacking" step in order to arrive at something which can be evaluated at all under the EA's fitness measure. If sufficiently complex, such a transformation is best viewed as development.

This chapter concentrates on the benefits of having a developmental step which relates to the problem of searching for merging networks. Since local search does not play a role in the simulations, the latter reasons for desiring the use of development in an EA come into play. In particular, we desire a representation of CMPX networks amenable to genetic search which can be simply transformed into a network for testing. We also would like our encoding to specify not just one sorting network, but instead a whole family of recursively defined networks, each member of which is built out of smaller members of the family. Phrase-structure grammars provide a natural representation for capturing interactions and relationships between different parts of a structure which is encoded as a string derived from the grammar.

Further, grammars can be used to describe an infinite collection of objects, each derived from the grammar given a certain start symbol and depth of derivation. For these reasons the genotypic representation consists of simple grammars, with each grammar production a gene. Section VI.E contains the details of the process by which these grammars derive CMPX networks. The order in which the productions occur on the genotype plays an important role in this derivation process, since at any given stage of the derivation the earliest applicable production on the genome is always applied. Crossover occurs only *between* productions. Thus, crossover seeks small collections of productions for building sorting networks. Mutation serves to modify individual productions, but happens with a much lower frequency than does crossover.

VI.D.1 Other EA use of developmental grammars

Evolving Polynomials

In earlier, related work Belew describes a system in which polynomial networks are grown according to a set of grammar rules and then trained and tested to perform a time-series prediction task[13]. The networks grow into a discrete space, starting from a single "cell." Simple pattern-action rules specify the local behavior of each cell based on its surroundings.

The polynomials in these simulations were being judged on their goodness-of-fit to a chaotic time series. Each polynomial was interpreted as a function which computed the value of a series at time index $t+1$ given the value at time index t. The simple, low-order polynomials eventually captured the lowest-frequency regularities of the environment, effectively occupying a "niche." In order for other polynomials to perform better, they had to account for the same regularities and some higher-frequency regularities as well. Thus, these other polynomials were competing for the "residual fitness" which was not already available to simpler (monomial) solutions. This lead to the eventual advent of higher-order polynomials. A particularly interesting aspect of these experiments is the wide extent of the exploration of higher-order polynomials once anything beyond a simple monomial meets with success.

One of the results of this work was the tendency of the population to maintain simple, low-order solutions for a lengthy period and then suddenly and rapidly "discover" higher-order polynomials. Following this "discovery," the solution was generally found quickly. Belew describes the pressure toward more complicated polynomial networks via the notion of residual fitness.

Another issue explored by Belew is the issue of genome length. There is a definite pressure toward longer genomes (containing more productions). Belew demonstrated this by running a "neutralist" simulation in which all individuals were assigned the same fitness and convergence was ignored. The average genome length didn't continue to grow as it did when the actual fitness measure was used; instead, the average length tended to stay near a fixed value.

The Cellular Development of Neural Nets

Though he did not use an explicit grammar formalism, Gruau used an equivalent system in his approach to evolving neural networks[40]. His use of a EA with development was motivated by the previously noted trouble with weight matrix and list representations. In essence, Gruau's genotypes represent the input to a machine which is very much like a simple tape-based automaton, with the exception that it's tape is a tree, and nodes with more than one child cause nodes in the developing network to split. Basically, each cell in the developing network has a read-head positioned on some part of the tree and reads from that position downward. Thus the tree really encodes several input tapes for several finite- state machines in an overlapping fashion, and with the addition that some instructions on the tape (nodes in the tree) cause the FSM's to split, with each child of the current node in the tree inheriting the read-head of a node resulting from the split. The languages computable by these automata are the context-free languages.

One could in theory turn one of Gruau's genomes into a series of production rules by turning each node in the tree and its children into a single production rule, and using each state of a cell as a nonterminal.

VI.E Grammars for Evolving Merging Networks

An EA is used to search for CMPX networks: each genotype contains a "start gene," a limit on the number of rules applied in a derivation, and a list of grammar productions. The start gene is a start symbol for the grammar productions and the genotype "develops" into a network by a series of rule applications. Thus, "development" corresponds to a derivation of a string in the language of the grammar specified on the genotype. The limit on the number of rule applications takes the form of a bound on the depth of the parse tree of the resulting phenotype along any one path from a root to a leaf.

VI.E.1 Nonmorphological information on the genome

As mentioned above, each genome is a list. The first element of this list is itself a list, $\left\langle N_S^W, D\right\rangle$ consisting of a nonterminal and a limit on derivation depth. This information is called "nonmorphological" not to indicate that it doesn't effect the structure of the resulting network, but rather to distinguish it from the structure-transforming genes which encode production rules.

The nonterminal encoded in a given genotype's start gene is used as the start symbol in the derivation of a CMPX network using the productions on that genotype. The resulting CMPX network is the genotype's associated phenotype. The start gene's nonterminal, like all others in the grammar, is parameterized by two values: W, the log of the number of inputs of the network into which the nonterminal expands, and S, a single bit of information with no extrinsic meaning[4].

The bit associated with each nonterminal, S, allows the existence of two

[4]Those familiar with unification grammars will notice that the grammars disucssed here are a simple form of them, in that they use parameterized nonterminals.

distinct rules which build networks of the same size; both of which may be used in building a larger network. The necessity of this parameter will be discussed below, as will the precise role of the time limit which is also encoded in the nonmorphological gene. A nonterminal which expands to a net of size 2^W with the second parameter set to S is written N_S^W. When $W > 2$, the only way to expand the nonterminal creates two new nonterminals $N_{S'}^{W-1}$ and $N_{S''}^{W-1}$ which derive networks with half as many inputs; the production expanding N_S^W also describes how the two smaller networks are combined to form one network with 2^W inputs. When $W = 2$, the nonterminal can only be expanded by a production which replaces it with a complete four-input network[5].

Note that each individual has its size encoded in its genome in the value of W encoded in the start gene's nonterminal. Thus, different individuals of the population may build networks of different sizes, potentially from the same set of rules, as the EA proceeds from generation to generation. The fitness measure encourages and makes use of this property, as will be described below.

VI.E.2 Productions allowed in the grammar

Following the start gene on the genome is a list of grammar productions. Each production falls into one of the two broad categories "base" and "recursive." Base productions specify a four-input CMPX network completely. Recursive productions encode a rule for building a $2N$-input network out of two N- input networks.

The basic form of a nonterminal was discussed above in section VI.E.1. Nonterminals are slightly more general than was indicated, when they occur in productions. On the left-hand side of a production, either W or S may be replaced by a wildcard, or "don't care," symbol, denoted by "\$" below. Thus, a nonterminal which matches[6] any other nonterminal with $S = 1$ and any W at all would be written $N_S^{\$}$.

[5]The case $W = 1$ is not allowed.

[6]Or "unifies with" in the sense of unification grammars.

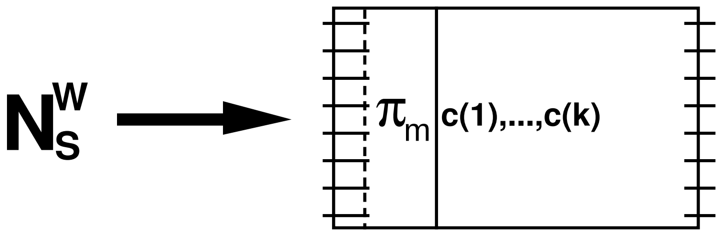
Base network productions

As mentioned above, one variety of production specifies a "base-case" for the recursive definition of a family of sorting networks to be encode by the grammar as a whole. The right-hand side of such a production specifies 3 to 5 CMPX's. The left-hand side can be any nonterminal. Since we are searching for merging networks, the right-hand side of the rule must also specify the merge partition for the resulting network. The general form of base network rules is:

$$N_S^W \to \langle \pi_m, \langle c_1, \ldots, c_k \rangle \rangle\,, \quad 3 \le k \le 5$$

where N_S^W is the nonterminal which this rule expands, the $c_i, 1 \le i \le k$ are the CMPX operations in the network specified by the rule, and $\pi_m = (\mathrm{mp}_A, \mathrm{mp}_B)$ is the merge partition specified as two sets, one specifying the input indices labelled "A" and one specifying the indices labelled "B." Note that $\mathrm{mp}_A \cup \mathrm{mp}_B$ is always the set of all input indices, $\{0, 1, \ldots, 2^W - 1\}$, and $|\mathrm{mp}_A| = |\mathrm{mp}_B| = 2^{W-1} = \frac{N}{2}$[7].

For clarity, rules are also shown in a more pictorial fashion below when demonstrating the derivation of a network from a grammar. The general form of a base network production in this pictographic scheme is:



where the elements of the picture have been labelled to correspond to the textual representation of these rules given above. The right hand side of the rule depicted contains a merge partition and a sequence of CMPXs superimposed on a network "frame." Some examples of rules of this kind in textual form are:

(b1) $N_0^2 \to \langle (\{0, 2\}, \{1, 3\})\,, \langle [0:1][2:3][1:2] \rangle \rangle$

[7] N denotes the number of inputs here. In general, N only denotes a nonterminal when it appears with super- and sub-scripts and in boldface.

(b2) $N_1^2 \to \langle (\{0,3\},\{1,2\}), \langle [0:1][2:3][1:2] \rangle \rangle$

(b3) $N_0^\$ \to \langle (\{0,1\},\{2,3\}), \langle [0:2][1:3][1:2] \rangle \rangle$

(b4) $N_1^\$ \to \langle (\{0,2\},\{1,3\}), \langle [0:3][1:2][0:1][2:3] \rangle \rangle$

(b5) $N_0^3 \to \langle (\{0,1\},\{2,3\}), \langle [0:3][1:2][0:1][2:3] \rangle \rangle$

(b6) $N_1^4 \to \langle (\{0,3\},\{1,2\}), \langle [0:2][1:3][0:1][2:3] \rangle \rangle$

Note that the right-hand side of each of these rules specifies a correct merging network on four inputs. The left-hand sides of these rules, however, have significant consequences for each rule's usefulness. **(b1)** and **(b2)** are used only when expanding a nonterminal with $W = 2$, the only cases in which they work correctly. **(b3)** and **(b4)** have a wildcard symbol for W; this means that it is possible that they will be used when trying to expand a nonterminal with $W > 2$. If this happens, the network resulting from this expansion will contain no CMPX's which operate on any inputs with indices beyond three, since the right-hand sides of these rules contain no such CMPX's. Finally, **(b5)** and **(b6)** are likely to be problematic if used in building a network for the reasons just cited, since they both contain nonterminals specifying $W > 2$.

The base network productions provide the base cases for another class of productions that define networks recursively in terms of smaller networks. Thus, the base network productions are what cause the recursion to "bottom out."

Recursion productions

Recursion productions always specify a method for combining two networks with $\frac{N}{2}$ inputs each into a single N-input network. This is accomplished by a "call partition." In addition, recursion productions specify CMPX's which are added to the network built out of the two smaller networks. These form what is called a "patch" or "patch layer" in the resulting network, and always form a suffix of the network which together comprise a single delay cycle consisting solely of adjacent CMPXs.

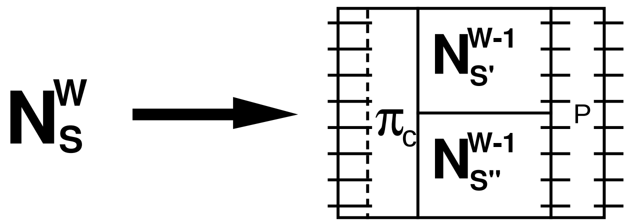
Call partitions, as merge partitions, divide the set of input indices into two equal-cardinality sets such that the union of these two sets is all input indices. Call partitions label inputs with C's and D's instead of the A's and B's used for merge partitions. The difference between the two kinds of partition is in their uses. Merge partitions describe how the two sorted lists which serve as inputs two a merge network are to be placed in the input array, D, on which the network operates. Call partitions describe which inputs in an N-input network are to be assigned to each of the two $\frac{N}{2}$-input networks from which it will be built. Call-partitions are written in a way that is analogous to the way wrote merge partitions were written above: $\pi_c = (\mathrm{cp}_C, \mathrm{cp}_D)$. The general form of recursion rules is:

$$N_S^W \rightarrow \left\langle \pi_c, \frac{N_{S'}^{W-1}}{N_{S''}^{W-1}}, P \right\rangle$$

where N_S^W is the nonterminal which this rule expands, π_c is the call partition specified using a representation which is discussed in section VI.E.2., $N_{S'}^{W-1}$ and $N_{S''}^{W-1}$ are the nonterminals which must be expanded in order to build the two smaller networks out of which the network resulting from the application of this rule will be built, and P is a patch type. There are two kinds of "patch" used: even and odd. Both patches are built using the same basic method, but are started differently. An even patch is built by starting with input index 0 and stepping through the input indices by adding two to i. Whenever indices i and $i+1$ are assigned different labels by the merge partition, the CMPX $[i : i+1]$ is added to the end of network[8]; note that no two CMPX's thus added will have an index in common; odd patches are built in the same way, but starting with the index 1 instead of 0. The two patch types are named according to whether the first index in any CMPX added when applying them is always even or always odd. $N_{S'}^{W-1}$ and $N_{S''}^{W-1}$ are always constrained to have W values one less than the value in the nonterminal on the production's left-hand side. If the left-hand side has $W = \$$, then the value of the nonterminal which the production matches when it is fired is decremented and used on the right-hand side in both nonterminals.

These productions are represented using pictures like the following:

[8] i.e., at the right-hand end of the network's CMPX sequence.



As with base net productions, a diagrammatic form is given here indicating where particular information will be filled-in when demonstrating a network derivation below. The call partition may be thought of as "routing" the inputs from the left-hand side of the network to one of the two subnetworks drawn as small boxes in the diagram above. Below, lines are drawn to indicate such routing. As above, we shall discuss some examples represented textually:

(r1) $N_0^3 \rightarrow \left\langle ((2),(2))\,, \frac{N_0^2}{N_0^2}, \text{odd} \right\rangle$

(r2) $N_1^4 \rightarrow \left\langle ((1),(1))\,, \frac{N_1^3}{N_1^3}, \text{even} \right\rangle$

(r3) $N_0^\$ \rightarrow \left\langle ((2),(2))\,, \frac{N_0^{\$-1}}{N_0^{\$-1}}, \text{odd} \right\rangle$

(r4) $N_1^\$ \rightarrow \left\langle ((1\ 1),(2\ 0))\,, \frac{N_0^{\$-1}}{N_1^{\$-1}}, \text{even} \right\rangle$

(r1) and **(r2)** specify networks of specific sizes, and **(r3)** and **(r4)** specify networks for any size. All of these productions depend on the existence of base network rules to provide a base-case for the recursion that occurs when the nonterminals on their right-hand sides are expanded.

Representing call partitions

The initial attempt at representing call partitions specified them in the same way that base network rules specify merge partitions: the indices labelled with each letter by the partition were simply enumerated. This representation proved unworkable mainly because it fails to work properly with wildcards. That is, specifying a call-parition in way that makes the call partition expressed specific to a given number of inputs obviates the function of wildcards, since a production whose left-hand side is $N_S^\$$ won't apply to all possible matches of its left- hand side if something on the right-hand side is only useful for some particular W value. That is, e.g., if a production whose left-hand side is $N_0^\$$ fires to expand a nonterminal N_0^4, the production's right-hand side must contain an appropriate partition of $\{0, 1, \ldots, 15\}$ for the call partition; in the absence of an appropriate partition, we must either insert something random or kill the individual whose grammar we are expressing by assigning it a very low fitness value. Thus, using the simple representation of a partition, it wasn't possible to find grammars for infinite languages which built families of correct merging networks for arbitrarily large numbers of inputs. At most, we might find grammars that worked up to the maximum allowed width in the simulation, and in which the recursion rules were all of the same kind (i.e., all of their call partitions were in the extension of some simple predicate).

Ideally, a call partition representation should be independent of the number of inputs to the network being built. That is, we would like a representation in which we could, for example, specify, independently of the W value given by the nonterminal on the production's left-hand side, that all odd indices are labelled C and all even indices are labelled D by the call partition.

In the representation used here, the call partition is specified by two lists of numbers which are interpreted to yield a call-partition for any number of inputs. The representation has the virtue that shorter representations result in more regular call-partitions than longer ones. Call partition specifications in general look like the following:

$$\big((c_1\ c_2\ \ldots\ c_{N/2}), (d_1\ d_2\ \ldots\ d_{N/2})\big)$$

$(c_1\ c_2\ \ldots\ c_{N/2})$ and $(d_1\ d_2\ \ldots\ d_{N/2})$ are called the "run lists" for this partition and the numbers c_i and d_i determine the lengths of alternating C's and D's in the list of C's and D's that result from reading of the labels of the inputs in increasing index order. That is, if the call partition itself is:

$$(\{0, 2, 4, 6\}, \{1, 3, 5, 7\})$$

then even inputs have label C and odd inputs have label D. The list of C's and D's alluded to above would be:

$$C\ D\ C\ D\ C\ D\ C\ D$$

and this can be generated using the call partition specification:

$$((1\ 1\ 1\ 1), (1\ 1\ 1\ 1))$$

which specifies one C followed by one D followed by one C, and so forth.

More generally, the partition is built as follows. Imagine that we've written down the input indices for the network under construction in increasing order. Starting at 0, we first write down a C under the first c_1 indices, provided that this involves labelling half of them C. We then label the next d_1 indices with D subject to a similar constraint. We continue reading numbers alternately from the two run lists until all of the indices are labelled. If we run out of numbers at the end of the list, we start over from the beginning of that run list. If we come to a number in a run list larger than the number of indices remaining to be labelled with the appropriate letter, we label as many indices as remain for that letter and label the rest with the other letter and stop. Thus, a partition specification equivalent to the one given above is:

$$((1), (1))$$

because both run lists in the specification above contained only 1's.

There are two important details of call partition specification that require further comment. First, the numbers in a specification's run lists must be in the interval $[1, k]$ for some constant, with the exception that the *last* number in either list may be 0. This allows such patterns as "$C\ D\ D\ C\ C\ D\ D\ C$" to be generated. Note that 0's in the middle of the run lists could be removed easily, and 0's at the beginning would only serve to potentially allow a sequence to begin with D, which is unnecessary since any such partition could be arrived at by swapping the run lists. Figure VI.2 depicts a call parition specification being interpreted and shows the partition which results.

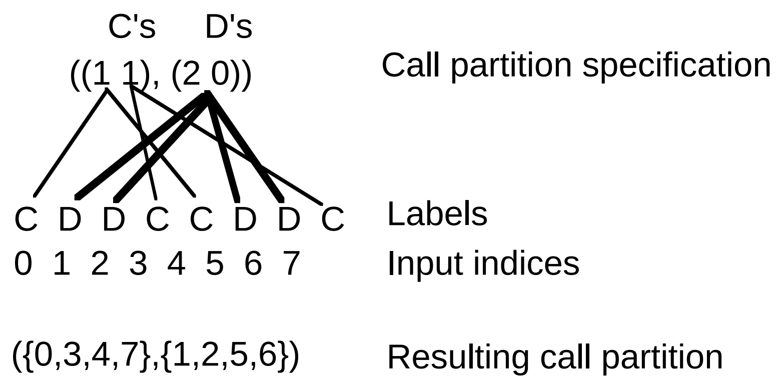


Figure VI.2: How genotypic call partition specifications become call partitions.

VI.E.3 The interpretation process

We are now ready discuss the process by which grammars are used to derive networks. Basically, a left-most first derivation from the grammar is performed, always applying the rule closest to the beginning of the genome which is applicable to the string so far at any given derivation step. This section gives a detailed example of the workings of a particular grammar, found in figure VI.3.

(e0) is the start gene, and **(e1)** and **(e2)** are grammar productions. To derive a network from this grammar, start with the nonterminal in the nonmorphological gene and derive a string by expanding the leftmost nonterminal in the string at any given step using the first matching production on the genome. Each production contains specific information about the network. This information is summarized table VI.1.

The start gene specifies the initial nonterminal from which the entire network is derived. **(e1)** is a base production, the only one in this grammar, and full specifies a 4-input CMPX network and a merge partition. **(e2)** is a recursion production. It explains how to make a large network from two smaller ones, and specifies a "patch layer" to be included with the result.

For **(e2)**, the left-hand side of the picture shows the call partition by showing

Gene	Information about phenotype
(e0)	Number of inputs in completed network, maximum depth of derivation/recursion along any branch
(e1)	Base case for the recursive definition, a complete network on four inputs
(e2)	Method by which a network is built from two networks each with half as many inputs

Table VI.1: The information provided by each gene in the grammar in figure VI.3.

the assignment of lines to subnetworks. In this grammar, the call partition assigns alternating pairs of inputs to each subnetwork. The patch layer is similarly shown schematically, with a CMPX starting on input number one. Note that only those adjacent CMPXs beginning on an odd index which also cross the network's merge partition will be included in this patch. The merge partition is derived from the merge partitions of the two subnetworks using the input assignments given by the call partition. Figure VI.4 illustrates the derivation of a network from the genome given by genes **(e0)**, **(e1)**, and **(e2)**. Shown are the string derived from the grammar above at each step of the derivation, a sequence of pictures illustrating the development of the network, and the rule applied at each step to arrive at the next row of the table.

The final string may appear somewhat complex and lengthy, but it can be easily read and translated into a network[9]. It is helpful, however, to use a form of the grammars which places parentheses around each nonterminal on a production's right-hand side, to make interpreting the string easier. They are omitted here because they clutter the derivations. The grammars are implemented using LISP S-expressions which contain the same information in the same order as it appears in the grammars as written above, with the exception of the aforementioned parentheses.

Note that the final row of the above table shows the network with the inputs rerouted to subnetworks (to display the call partition). This was done to make the recursive network-derivation process apparent, but is not the best way to view the

[9]Note that several of the "nonterminal fractions" used in the earlier textual representations have been changed to a form that uses slashes rather than horizontal lines.

final network. Below, figure VI.5 shows the final version of this network redrawn so that all of the input lines are straight. Imagine that the vertical (CMPX-denoting) lines are elastic and that the horizontal (input-location-denoting) lines are rigid and have their merge partition labels attached. Then imagine picking up the final network and "shaking it out" to straighten the input lines. If the resulting CMPX network is redrawn so that CMPX's that can be applied in parallel are grouped, figure VI.5 is the result.

The network in figure VI.5 turns out to be an important and interesting log-sequential sorting network, found by the evolutionary algorithm, and will merit further discussion below. For now, note that by choosing the value of W and the time limit in gene **(e0)** to be at least $1 + log_2 N$, a correct log- sequential sorter for any N which is a power of two may be derived.

The grammar used for demonstration purposes above has the feature that no start symbol N_S^W will ever lead to an "illegal" or inevaluable phenotype. In the course of deriving a network description from a grammar, however, it is possible for the interpretation process to become stuck, when no production in the grammar matches the nonterminal which currently needs to be expanded, or to encounter inconsistent information about the network being built, such as a reference to input index 5 in a 4-input network, caused by a mutation to the gene containing the CMPX in question.

Repair

Often, such flawed genomes contain nonetheless useful genes and are close to being correct grammars, or at least grammars which build testable networks. To exploit the information available in such "near misses," a "repair" mechanism is used; the unusual features which need to be repaired are called "anomalies." Repair allows limited difficulties in interpreting a grammar to be fixed: the cost of these repairs is not *directly* fitness-related. Rather, a certain amount of repair is allowed all individuals, and each repair has an associated cost. Individuals accumulating more

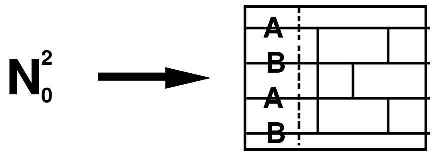
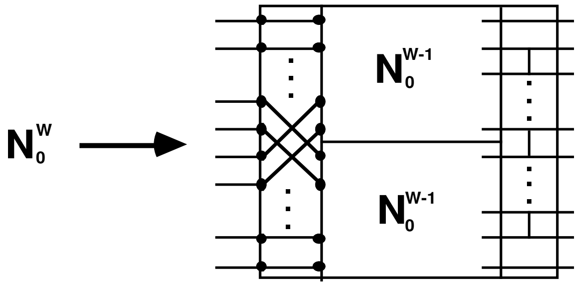
Label	Textual Representation	Graphic Representation
(e0)	$\langle N_0^4, 4\rangle$	(None)
(e1)	$N_0^2 \to \langle (\{0,2\},\{1,3\})\,,$ $\langle [0:3][1:2][0:1][2:3]\rangle\rangle$	
(e2)	$N_0^\$ \to \langle ((2\ 2),(2\ 2))\,,$ $\frac{N_0^{\$-1}}{N_0^{\$-1}}, \text{odd}\rangle$	

Figure VI.3: Grammar productions for an example derivation

than a certain fixed repair cost are considered nonviable and assigned a fitness worse than any other individual in the generation in which they occur.

An important consideration once repair is incorporated into the grammar expression process is that of selecting reparable flaws and assigning their costs. Repairs are never performed by inserting random material into the developing network at a site at which flawed or inconsistent information exists, or by inserting random material where information is missing. This is because repair is meant to exploit the information available in the genome: if there is little enough information in the genome to require random insertion during derivation of a network from a grammar, it is preferable to simply give up on the genome in question. Table VI.2 summarizes the anomalies possible during development. Note that networks with a total repair cost larger than 15 are judged nonviable. A network which is judged nonviable receives the worst possible for the generation in which it occurs.

As an example of a situation in which repair is necessary, consider the grammar used in the derivation example, but with gene **(e1)** modified to read:

$$\textbf{(e1')}\quad N_0^2 \to \langle (\{0,2\},\{1,3\})\,, \langle [3:0][1:2][0:1][2:3]\rangle\rangle$$

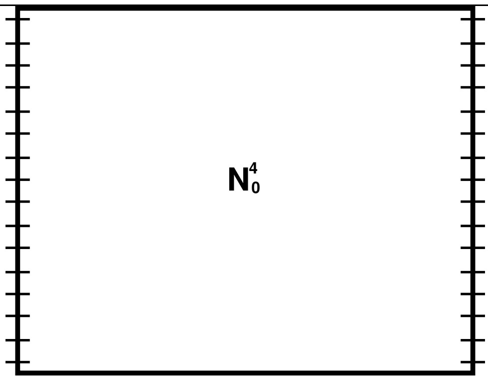
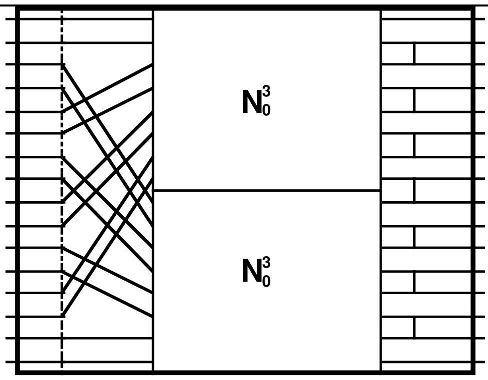
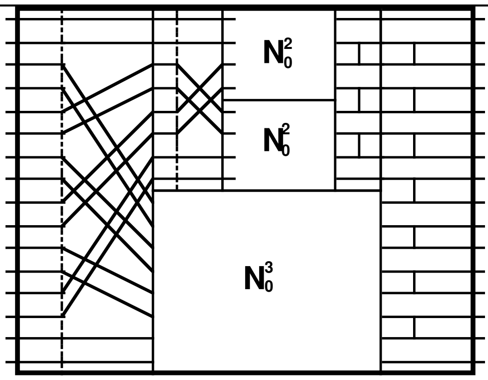
Derived String	Pictorial Representation	Rule
N_0^4		**(e2)**
$\left\langle ((2\ \ 2),(2\ \ 2)), \dfrac{N_0^3}{N_0^3}, odd \right\rangle$		**(e2)**
$\left\langle ((2\ \ 2),(2\ \ 2)), \dfrac{\left\langle ((2\ \ 2),(2\ \ 2)), \frac{N_0^2}{N_0^2}, odd \right\rangle}{N_0^3}, odd \right\rangle$		**(e1)**

Figure VI.4: A derivation from the grammar in figure VI.3.

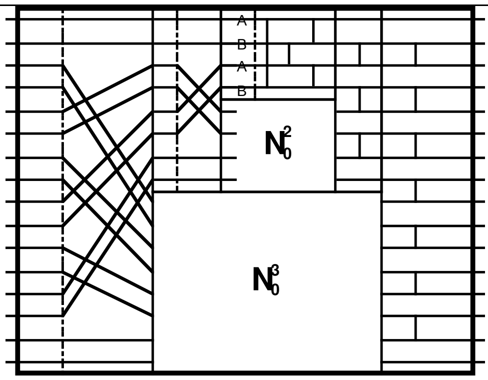
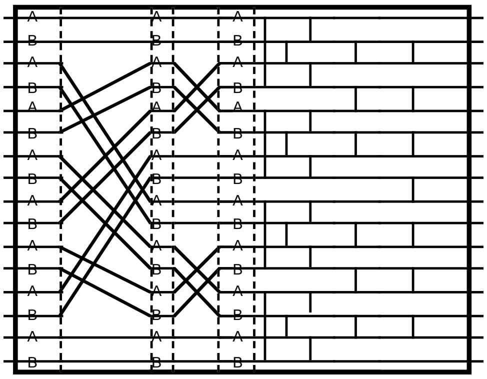
Derived String	Pictorial Representation	Rule
$\left\langle \begin{array}{l} ((2\ \ 2),(2\ \ 2)), \\ \left\langle \begin{array}{l} ((2\ \ 2),(2\ \ 2)), \\ \left\langle (\{0,2\},\{1,3\}), \left\langle [0{:}3][1{:}2][0{:}1][2{:}3] \right\rangle \right\rangle \big/ N_0^2, \\ odd \end{array} \right\rangle \Big/ N_0^3\ , \\ odd \end{array} \right\rangle$		**(e1)**
At this point, several steps are omitted in order to show the final network, after all subnetworks are completed. The process for filling in the remaining width four and width eight subnetworks is similar to the steps already shown.		
$\left\langle \begin{array}{l} ((2\ \ 2),(2\ \ 2)), \\ \left\langle \begin{array}{l} ((2\ \ 2),(2\ \ 2)), \\ \left\langle (\{0,2\},\{1,3\}), \left\langle [0{:}3][1{:}2][0{:}1][2{:}3] \right\rangle \right\rangle \big/ \\ \left\langle (\{0,2\},\{1,3\}), \left\langle [0{:}3][1{:}2][0{:}1][2{:}3] \right\rangle \right\rangle, \\ odd \end{array} \right\rangle \Big/ \\ \left\langle \begin{array}{l} ((2\ \ 2),(2\ \ 2)), \\ \left\langle (\{0,2\},\{1,3\}), \left\langle [0{:}3][1{:}2][0{:}1][2{:}3] \right\rangle \right\rangle \big/ \\ \left\langle (\{0,2\},\{1,3\}), \left\langle [0{:}3][1{:}2][0{:}1][2{:}3] \right\rangle \right\rangle, \\ odd \end{array} \right\rangle, \\ odd \end{array} \right\rangle$		**none**

Figure VI.4: (continued)

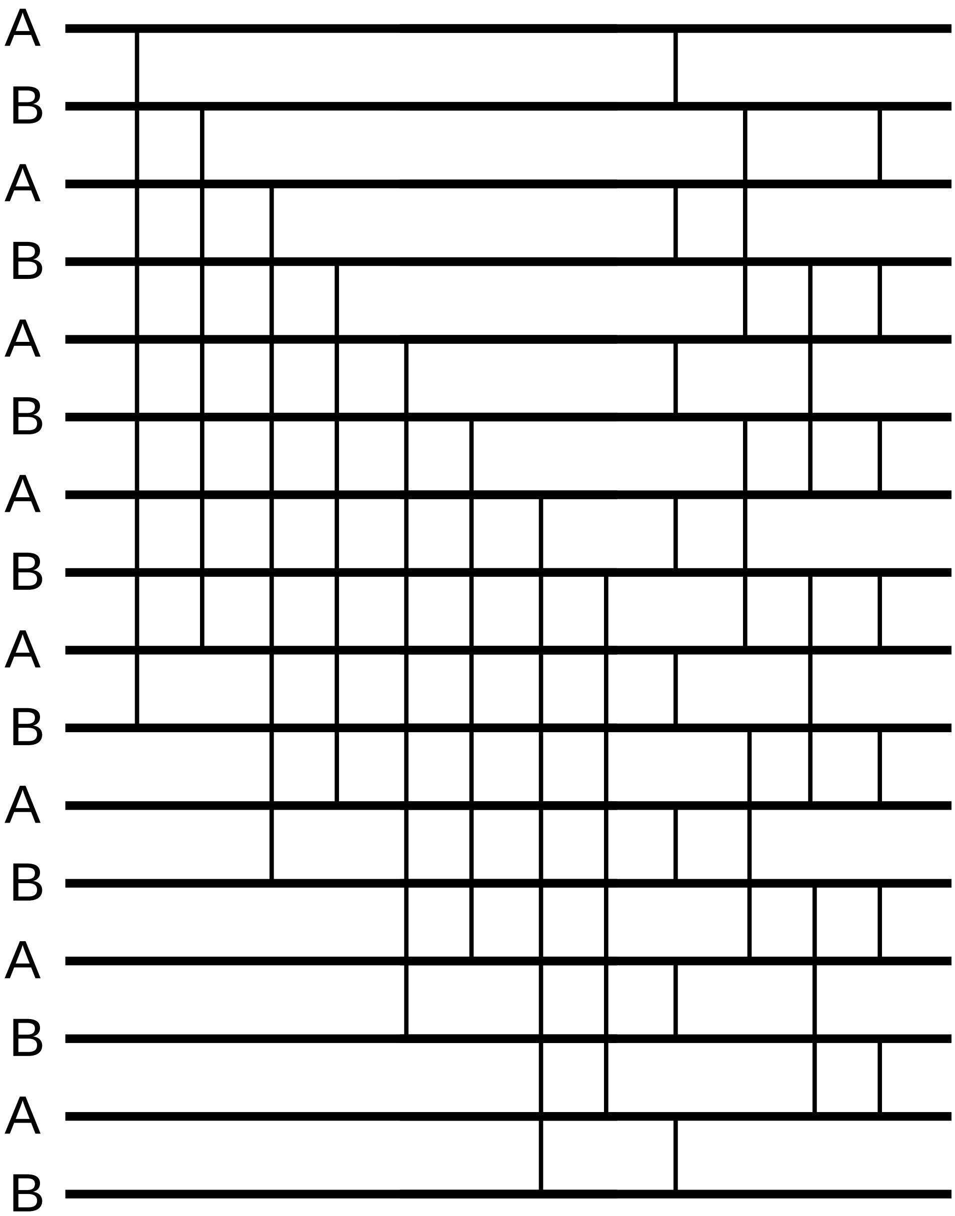


Figure VI.5: The network resulting from the derivation in figure VI.4.

In this version of the gene, the indices in the first CMPX have been swapped. This can happen as the result of a single mutation. If this is the only anomaly in the network, it may well be survivable if this network isn't reused too often during derivation. Since anomalies don't affect fitness, the CMPX in question is effectively "switched off" since it will be eliminated by repair. An example of an anomaly whose effect can depend on other information in the genome might result from a mutation to the S parameter in gene **(e2)** of the same grammar:

(e2') $N_0^{\$} \rightarrow \left\langle ((2\ 2),(2\ 2))\,, \frac{N_0^{\$-1}}{N_1^{\$-1}}, \mathrm{odd} \right\rangle$

This modified gene contains a nonterminal with S set to 1 instead of 0. There is no gene in the genome with a left-hand side which will match this nonterminal and the derivation process will thus have no action to take on this nonterminal. In this situation, the S parameter is relaxed to a wildcard and the search for a production happens again from the top of the genome. This action, however, counts as the firing of a production called a "tick" and thus consumes one point of "depth" in the recursion[10]. Since the grammar's nonmorphological gene allows derivations of depth 4 or less, some branches of the parse tree will lead to an "Out of time" anomaly and the nonterminals will be replaced with empty networks. This is illustrated in figure VI.6, which shows parse trees for derivations from the grammars using the original gene **(e2)**, on the left, and the modified gene, **(e2')**, on the right. Each line denotes the firing of a production. Thus, counting the number of edges from a root to a leaf tells how many production had to be fired along that branch of the derivation. On the left-hand side is a derivation using the original grammar. The leaf nodes all expand to base networks using gene **(e1)** as shown, but no information about call partitions has been included in the diagram. In the right-hand diagram is a derivation using **(e2')** instead of **(e2)**. In this case, one of the branches runs out of time and the "out of time" repair is initiated, leading to its replacement by an empty network. The lines corresponding to "tick" productions are the three dashed lines.

[10] This feature of the derivation process is discussed further in the next section.

Anomaly Description	Remedy	Cost
Bad CMPX: $[i : j]$ occurs in a base network with $i > j$, or one of $i, j \notin [0, 3]$.	The offending CMPX is dropped from the base network before it is incorporated into the larger network.	4
Out of time: Some branch of the recursive derivation process is deeper than is allowed by the nonmorphological gene's time limit.	Any nonterminals too deep to be expanded expand to empty networks with merge partition "$A\ B\ A\ B\ A\ B \ldots$".	5
None: Either a recursion production was invoked to build a base network, or a base network production was invoked to build a larger network. Also posted automatically whenever total repair costs exceed the maximum allowed.	None. The individual in question is nonviable. This anomaly must always be assigned a cost greater than the maximum allowed total repair cost.	16
No empty subnetwork: No subnetwork had to be expanded to the empty network. This means "out of time" didn't happen along *any* branch of the derivation process.	None needed. This isn't a "bad" anomaly as the above, it's an "interesting" anomaly. Individuals with this property are marked in the dump file. This anomaly has no effect on fitness or viability.	0

Table VI.2: Repairs possible during development, and their costs.

Note that **(e0)**'s "time parameter" is important here. If it were only 3, only one subnetwork would be nonempty in the right-hand diagram above, and there would be three "out of time" anomalies during development. If the time parameter were 5, there would be *no* such anomalies. The situation would be much more serious if, after the first mutation changed **(e2)** to **(e2')**, another mutation subsequently made the same change to the other nonterminal on the rule's right-hand side, resulting in the gene:

$$\textbf{(e2")} \quad N_0^{\$} \rightarrow \left\langle ((2\ 2), (2\ 2))\,, \frac{N_1^{\$-1}}{N_1^{\$-1}}, \mathrm{odd} \right\rangle$$

Since "out of time" anomalies cost 5 repair units each, three of them may occur in the absence of other anomalies without resulting in a nonviable network. If **(e2")** were used, however, there would be four such anomalies and the individual would be nonviable.

As noted above, the presence of a repair mechanism is meant to allow genomes which are "partially correct" to be tested so that any useful genes they contain may have a chance to propagate. The cost for this mechanism is worth noting: it "dilutes" the fitness measure by allowing individuals which would normally be assigned a very bad fitness value to receive better fitness values. This in particular is why it doesn't make sense to effect repairs by inserting random information into the network during development. Such a move would simply inject random noise into the fitness measure, and there are better ways of doing so when it is intended.

Other constraints on interpretation

There are some constraints on the interpretation of grammars that are not listed as repairs. In particular, there are two aspects of the derivation of networks from grammars which are noteworthy as constraints or "patches" to the interpretive process.

The first concerns merge partitions. As discussed above, these are decoded from a pair of lists in such a way that half of the inputs are labelled each with C

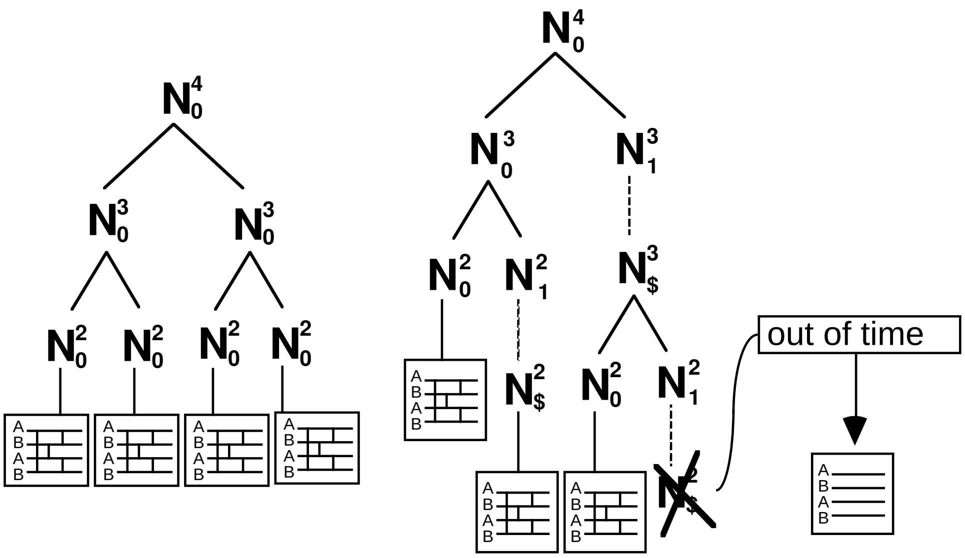


Figure VI.6: Parse trees for derivations without (on the left) and with (on the right) the modified form of gene **(e2)**.

and D. In particular, when one of the lists is empty, or contains a number greater than the number of indices remaining to be assigned a given label, the number of indices assigned the appropriate label is artificially (and arbitrarily) constrained to be exactly half of the inputs. The implication of this is that there are very many ways to represent a single call partition. This was noted above where call partitions were discussed and we saw that the descriptions $((1\ 1\ 1\ 1), (1\ 1\ 1\ 1))$ and $((1), (1))$ are equivalent.

The second derivation "patch" concerns the behavior of the interpreter when a production matching a given nonterminal is not found in the grammar from which a string is being derived. When such a match is not found, a "No-Op" or "empty" production is fired, which doesn't change the network being specified, but takes a single unit of time, and then tries again to find a match, but ignores the S parameter when looking for this match. Thus, at the cost of a unit of time, the S parameter on a nonterminal can be circumvented when no other matching productions exist. As mentioned above, this implicit production is referred to as a "tick."

VI.F Details regarding the EA

The initial population of 100 grammars was chosen from allowing W_{start}, the W value for the nonterminal in the start gene, from 2 to 6, and thus generates its initial population from a large space, a point which is elaborated in the next section. The initial population has k, the number of production rule genes per individual, fixed at 5, but the grammars are allowed to grow as the simulation progresses. There are, however, some limitations placed on this growth, as will be discussed below. The simulation stops if the population converges, in the sense that the fitness measure returns the same value for all individuals, or when a specified number of generations have been executed. Stochastic remainder selection and a fixed population size are used. In each generation, one of the best individuals is guaranteed at least one offspring in the following generation. The choice is arbitrary when more than one individual achieves the best fitness in a given generation.

The EA uses five operators: the two classic operators, crossover and mutation, are supplemented by a trimming operator and a permuting operator, and a doubling operator. Crossover occurs only between genes, never within productions or within the nonmorphological gene. Single-point crossover with a rate of 0.7 is used. Crossover points are chosen for each parent. Mutation operates on individual genes. On the nonmorphological gene, it simply replaces the W, S, or the time limit with a random legal value. On production rules, mutation has an equal probability of effecting the left-hand side, by randomly replacing one of the two parameters there, changing the right-hand side, or changing the a base-net production to a recursion production or vice versa by random replacement. Mutations to a rule's right-hand side change one of the values there. On base-network productions, this can mean inserting, altering, or deleting a CMPX or moving indices between the sides of the merge partition. In recursion production, the call partition representation may be modified by performing insertions, alterations, or deletions on either of the two lists it contains, with the caveat that a zero may never occur anywhere other than on the ends of these lists. Any of the other values on the right-hand side of a recursion production is mutated simply by replacing it with another random value. Note that only one of the possible mutations on the right-hand side of a production is performed each time a mutation hits the production's right-hand side. The per-gene mutation rate is 0.066.

The other three operators are motivated by the grammar representation and the linear ordering imposed on the genome and made use of during development. Permutation picks a random production within the genome and promotes it a random distance toward the front of the genome. The nonmorphological genome is always first, and this constraint is maintained when a production is moved. The per-genome permutation rate is 0.1. This is implicitly subject to the constraint that permutation never occurs if doubling has occurred. Doubling is very much like a permutation, but makes a duplicate random production and inserts it into the genome at a location determined by taking a random walk away from the original's location. There is a bias in this random walk towards the end of the genome. Doubling occurs with probability 0.02 per genome. Note that these two operators have rates defined per-*genome* rather than per-gene.

The final operator keeps the genomes from growing too large by imposing a soft limit on genome length. If the genome length exceeds a limit set as a simulation parameter (usually 10), a random number of genes between 0 and the number of genes over the limit, inclusive, is eliminated from the tail of the genome. This counters a natural pressure for the genomes to grow without limit. Crossovers producing one very long and one very short child will usually result in the death of one or both parents (both if the cut occurs near the front of both genomes). The trimming operator has no limit per se; it simply performs its random truncation on any genome over the soft limit in each generation.

The fitness function is a "staircase:" lower fitness is better[11]. Individuals which develop into small networks have a penalty added to their fitnesses so that their

[11]Note the difference from EvoGrams, which maximizes fitness.

best fitness, if their error is zero, is the same as the worst fitness for a network with twice as many inputs (the next possible size, since the networks have widths which are powers of two). A simulation parameter determines the widest possible network in a simulation, and only individuals which develop into networks of that width may attain the best(lowest) possible score. Empty networks (networks containing no CMPXs) are given the largest possible penalty, though their fitness will still usually be better than nonviable individuals.

A final aspect of the simulations, related to the fitness measure, is that the ceiling on the fitness measure is lowered as the simulation proceeds. That is, the highest (worst possible) fitness measure is lowered to be 1.5 times the average fitness whenever the average fitness falls low enough to necessitate the change. At any time during the simulation, there are at least a few nonviable individuals in the population. If the worst fitness value weren't adapted to stay within a constant factor of the average, the average would be maintained at an artificially high level and the selection scheme (which is essentially fitness-proportional) would suffer from a failure to distinguish among the current best individuals. Adapting the "fitness ceiling" downward helps with this problem and helps to maintain selective pressure as the population average falls.

VI.G Results

This section discusses experimental results, including the generation of the individual used to demonstrate development in section VI.E. Priors to these, some random sampling experiments and the details of the evolutionary algorithm implementation are discussed.

VI.G.1 Difficulty of search

Several experiments to determine the difficulty of searching for merging networks were run. The first of these experiments was performed in a "naive" representation which generated random sequences of CMPX operations and a randomly chosen merge partition. These experiments are referred to as occurring "in phenotypic space" because the randomly generated structures are CMPX networks with merge partitions, exactly the objects which form the space of phenotypes derived from genotypes as described in section VI.E. The second group of experiments involved generating random grammars and testing them by attempting to derive and test a 32-input network from each. These latter experiments are said to occur "in grammar space" since grammars are what was randomly generated in them. In the latter experiments, the random generation of grammars used the same method which was used to set the evolutionary algorithm's initial population in the EA results described below.

Random generation in phenotypic space

As mentioned earlier, the space of merge partition-CMPX network pairs with n_c CMPX's and width N is large, of size $\begin{pmatrix} N \\ N/2 \end{pmatrix}\begin{pmatrix} N \\ 2 \end{pmatrix}^{n_c}$. 1000 32-input networks were randomly generated for different values of n_c ranging from 10 to 700. Each CMPX was chosen uniformly from the set of all CMPXs on 32 inputs. A merge partition was randomly generated for each net by choosing a random permutation of the input indices and labelling the first 16 indices with A and the latter 16 with B. These experiments were aimed at determining how likely it is that choosing networks randomly, even unacceptably large networks, will yield correct mergers. Figure VI.7 shows the average error over 1000 networks for each value of n_c.

A similar set of experiments is summarized in figure VI.8. These experiments also involved generation 1000 32-input networks for various widths. For the results of figure VI.8, however, there was a single fixed merge partition for each collection of 1000 networks. The average error is very similar to the average error for the first collection of runs, as can be seen from the two figures. This is intuitively correct, since any given correct MNet can be easily rewritten, by renumbering inputs, to use

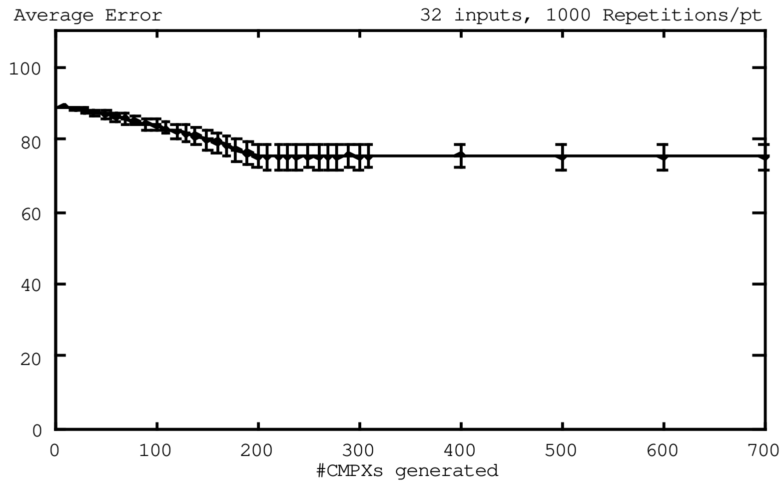


Figure VI.7: Average error for random networks. Error bars are 2 standard deviations high.

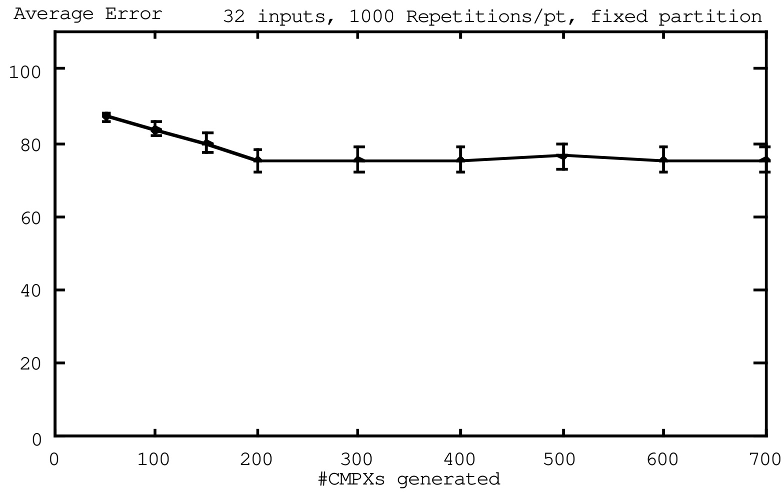


Figure VI.8: Average error for random networks with a fixed merge partition for each data point. Error bars are 2 standard deviations high.

any merge partition. None of the experiments summarized above found a correct merger on 32 inputs for any value of n_c tested.

Many readers will find it somewhat surprising that merging networks seem so difficult to discover, since merging two sequences of sorted data is a very simple problem, requiring in general only $2n - 1$ comparisons where both sequences to be merged have length n. This bound applies to algorithms which are allowed to use feedback, however, and not to merging *networks*. Knuth relates a proof[55, pp. 230–231] that merging two sequences, each of length n, using a CMPX network requires at least $\frac{1}{2} n \log_2 n + O(n)$. This is asymptotically the same as the bound for sorting with a CMPX network, $O(N \log_2 N)$ when a total of N inputs are involved.

Random generation in grammar space

Another set of random generation experiments are aimed at checking whether the evolutionary algorithm is doing useful search on the genotypic representation. A fixed number of genomes were generated at a variety of lengths and the fitness of networks derived the grammars these genomes encoded recorded. There are a large number of grammars possible with a given length, but there are many ways of encoding the same networks. Also, many grammars don't encode any network at all; these are the grammars which would be judged nonviable if they arose in the population. These random experiments used the same development procedure as that used for evolutionary algorithm simulations and the random genome generation process was identical to that used to initialize the evolutionary algorithms initial population, except that for random generation a predetermined W value of 5 was used in the nonmorphological gene. Any other constraints on random generation parameters will be listed below with the experiments in question.

The left-hand sides of randomly generated productions were chosen by making a uniform choice for S, and choosing W to be a wildcard with probability 0.5 and otherwise making a uniformly distributed choice between 2 and the maximum nonmorphological W value allowed, which was 5. To choose random base network productions, a network from a collection of 60 known to be correct 4-input merge networks was randomly chosen. These were found by exhaustively testing all 4-input networks containing 3, 4, or 5 CMPXs using each of the 3 possible merge partitions. Of those that merged correctly, most were eliminated as "redundant" since they were identical to smaller CMPX-count networks with extra CMPXs inserted. The recursion productions were generated by making uniform choices of S for the right-hand side nonterminals. The patch type was generated with equal probability assigned to either option. Finally, the each entry in a call partition specification was generated uniformly to be an integer between 1 and 4 inclusive, and a 0 was added to the end of either list in the representation with probability 0.5 for each list. The length of these lists was given by a random number between 1 and the nonmorphological gene's W value, fixed at 5.

In order to gain insight about the number of possible grammars that could be written down in the genotypic representation, we derive an expression for the number of grammars with k productions. This will be the number of nonmorphological genes times the number of sequences of k productions. Since there are 4 values of W, 2 of S, and the time parameter is a number from 1 to 10 inclusive, there are 80 nonmorphological genes possible. The number of possible sequences of k productions is:

$$k! \begin{pmatrix} n_{\text{bnp}} + n_{\text{rp}} \\ k \end{pmatrix}$$

where n_{bnp} is the number of possible base-net productions and n_{rp} is the number of possible recursion productions. Simple counting arguments based on the information above give:

$$n_{\text{bnp}} = 5 \cdot 2 \cdot 60 = 600$$

and if W_{start} is the start gene's W value, then we have:

$$\begin{aligned} n_{\text{rp}} &= (5 \cdot 2) \cdot \left(4 \cdot \sum_{i=1}^{W_{\text{NM}}} 4^i \right) \cdot (2 \cdot 2) \cdot 2 \\ &= 320 \cdot \sum_{i=1}^{W_{\text{NM}}} 4^i \end{aligned}$$

Number of	%error on merging task		Fraction w/0% error	
rules	Average	Standard Deviation	Average	Standard Deviation
5	98.7503	5.6074	0.0000	0.0000
10	98.2016	6.8537	0.0000	0.0000
20	97.5034	7.7753	0.0000	0.0000
30	97.7357	7.7603	0.0000	0.0000
40	97.4939	7.8103	0.0000	0.0000
50	97.1643	8.6829	0.0010	0.0316
60	97.5730	7.4501	0.0000	0.0000
70	97.9556	7.2447	0.0010	0.0316
80	97.9594	6.5737	0.0000	0.0000
90	97.7688	7.8299	0.0010	0.0316
100	97.6849	7.8100	0.0010	0.0316

Table VI.3: Results for random grammar generation experiments including only viable individuals.

and this gives a total number of grammars with k productions equal to:

$$n_{\text{grammars}}(k) = 80 \cdot k! \binom{600 + 320 \cdot \sum_{i=1}^{W_{\text{NM}}} 4^i}{k}$$

where summing over all allowable W_{start} values yields the total number of grammars possible. The random generation experiments described below build grammars with W_{start} fixed at 5 and vary k over many values. Even for small numbers of productions, this is a very large number. With $W_{\text{start}} = 5$ and $k = 5$, the number of grammars is already extremely large. It's base-10 logarithm is 30.105885. Summing over all W_{start} from 2 to 5, the number is not yet a full order of magnitude different, and has base-10 logarithm equal to 30.106312.

Thus, the random generation experiments and the evolutionary algorithm both explored very large spaces of grammars. It should be noted, however, that the random grammar generation process doesn't cover this space isotropically because of the way left-hand sides of productions are selected. The nonterminal on the left-hand side of a production has a 50% chance of being generated with a wildcard value for W. This results in a strong bias in favor of productions with wildcards.

The first collection of random experiments generated 1000 viable individuals for various numbers of productions. That is, genomes were generated until 1000 viable individuals were generated, throwing away nonviables. These experiments actually found some individuals which were also found with the evolutionary algorithm[12]. These experiments generated genomes with D in the range 1 to 7, and had a maximum for the numbers in the call partition representation of 2. The results are summarized in table VI.3.

Four correctly merging individuals were found in these experiments, and the individual found after generating 1000 individuals with 100 productions each, generated a version of the example network from section VI.E with some redundant CMPXs included. The base network contained an extra CMPX in this case, and the grammar generated a series of networks that were precursors of the network in figure VI.5 in the EA run which produced it.

Another collection of experiments was run with the call partition representation containing numbers from 1 to 4, as was used with the evolutionary algorithm. Genomes were generated for the same numbers of productions as for the first collection of experiments. No correct MNets were found in this set of random generation experiments, and the average error was always between 97% and 99.5%, with the maximum standard deviation less than 7.

The simulations above are not directly comparable to the evolutionary algorithm, because the number of nonviable genomes being thrown away as viable genomes are generated is not known. In fact, the random generation experiments just described often took much longer to run than an equivalent number of generations of a evolutionary algorithm run. This isn't surprising since the evolutionary algorithm doesn't get to "throw away" nonviable genomes. Further, the EA runs describe here allowed up to 64-input sorting networks in their populations[13].

The last set of random genome generation experiments didn't discard non-

[12]In these cases, the EA found an equivalent grammar, not exactly the same one.

[13]It is important to note, however, that subsequent runs, many of which found the best known solution as well as a new novel solution, both discussed below, used an upper limit of 32-inputs for the networks in their populations.

viable genomes. These experiments involved generating 5000 instead of 1000 random genomes for each number of productions used in the previous two experiments. In general, these experiments ran very quickly. The reasons is that for every number of productions, more than 99.8% of the genomes for every number of productions were nonviable, and no perfect mergers were found. Since the computational cost of developing a nonviable genome is on average cheaper than that for a viable genome[14], these runs usually ran faster than the evolutionary algorithm.

The data above, though they concern only random generation experiments, suggest some guesses at how the evolutionary algorithm is operating for this problem. The last experiments demonstrate that most of the EA's initial population is incapable of deriving a 32-input network at all. Many individuals in the initial population, however, are derive correct 4-input networks, since this only requires a single correct base production and an appropriate start symbol. In many runs, there are also a few individuals capable of building 8 or even 16 input networks, though these usually run out of time along some paths of the derivation as indicated in figure VI.6 of section VI.E.

The EA tends to require more time than random experiments that kept nonviables, and less time than the random experiments that discarded nonviables. This suggests that EA is, in part, serving to keep the search mainly within the space of viable genotypes. During typical runs using a population size of 100, the number of nonviables starts very high and then drops off to consistently account for roughly 15% of the population. The number of nonviable individuals fluctuates about this figure for the remainder of the run. The first collection of results suggests as well that the EA is doing more than just randomly searching the space of viable genotypes, and given the fitness function, and the heterogeneous nature of the population with respect to the nonterminal in the start symbol, it certainly has information available which would allow it to do better than random search among viable phenotypes. Even

[14]Cost of development for nonviable individuals isn't uniformly zero, since the nonviability can be determined after part of the derivation of phenotype from genotype is completed, as seen in figure 7 in section 6.

so, restricting the search to deal mainly with viable genotypes could be an important part of what the EA is accomplishing in the implementation discussed here.

VI.G.2 Basic Evolutionary Algorithm Results

The evolutionary algorithm as described in section VI.F looks for MNets. The most interesting results, however, relate to log-sequential SNets. In particular, the EA using the grammatical representation described in section VI.E often finds correct MNets, and sometimes finds MNets that are also log-sequential sorting networks. The EA isn't actually selecting for merging networks which are also log-sequential sorting networks, but the representational bias imposed by the grammars tends to lead to the discovery of networks that are log-sequential sorters or close to log-sequential sorters. It is important to note the consequence that discovering a grammar which encodes a family of log-sequential SNets using the EA proceeds as follows: first, run the EA to solve the merging problem and cull the individuals from the final population which build correct MNets at all sizes allowed during the simulation[15]. Next, take each individual that passes the tests of the first step, and test the networks it generates for log- sequential sorting up to 16 inputs[16]. Some individuals might not need testing if, for example, they can be eliminated on criteria that can be verified more quickly. For example, no network which is missing any of the CMPX operations of the from $[i : i+1]$ cannot be a log- sequential sorting network. The remaining networks generate correct MNets up to width 64 or 128 according to the maximum tested width. They also generate correct log-sequential SNets up 16 inputs. At this point, they are candidates for generators of families of log-sequential sorting networks; establishing them to be correct for all widths would have to be established by some form of inductive proof using the regularity of construction afforded by the

[15]In fact, since merging is cheap to verify, it can make sense to also test to see whether individuals that were evolved to build MNets on up to 32 inputs also build correct MNets for 64 and 128 inputs. This helps to eliminate grammars with constructions that don't generalize to any number of inputs which is a power of two.

[16]Testing for sorting on 65536 bit sequences and appealing to the 0–1 principle is feasible. Testing 2^{32} bit sequences isn't feasible for even a small number of networks.

grammars.

Most runs find correct MNets within a few hundred generations, and the simulations almost never run to the preset 1000 generation time limit because they converge (in the sense that fitness variance becomes zero) first. Figure 10 shows a fitness curve for one typical EA run. For this simulation, networks of width up to 64 were allowed in the population.

The best known recursively-defined log-sequential sorting networks are currently Canfield and Williamson's modified Batcher networks. Many of the evolutionary algorithm runs "rediscovered" these networks, with various encodings of the correct call partitions. Subsequent runs in which networks no wider than 32 inputs were allowed also often rediscovered the modified Batcher networks.

The evolutionary algorithm also, however, discovered an entirely new family of sorting networks. In the run whose fitness curves are plotted in figure VI.9, an individual arose whose genome encoded a grammar which employed a novel regularity to find a family of MNets which are also log-sequential sorters. These networks have proved to be of theoretical interest because they exploit features of both the modified Batcher networks and the Periodic Balanced Sequential networks discussed earlier.

VI.G.3 A New and Interesting Merging Network

The most interesting network found to date is a hybridization of the Periodic Balanced Sequential and Modified Batcher networks described earlier. Specifically, this individual's genome use the base network from Periodic Balanced Sequential Network and the recursive rule from the Modified Batcher Network. This network, which was dubbed "Dino" in the course of e-mail exchanges about it, occurred after 103 generations in a run with 100 individuals in the population. The run converged at generation 142.

The genome for Dino taken from the dump file which generated it is:

(d0) $\langle N_0^6, 8\rangle$

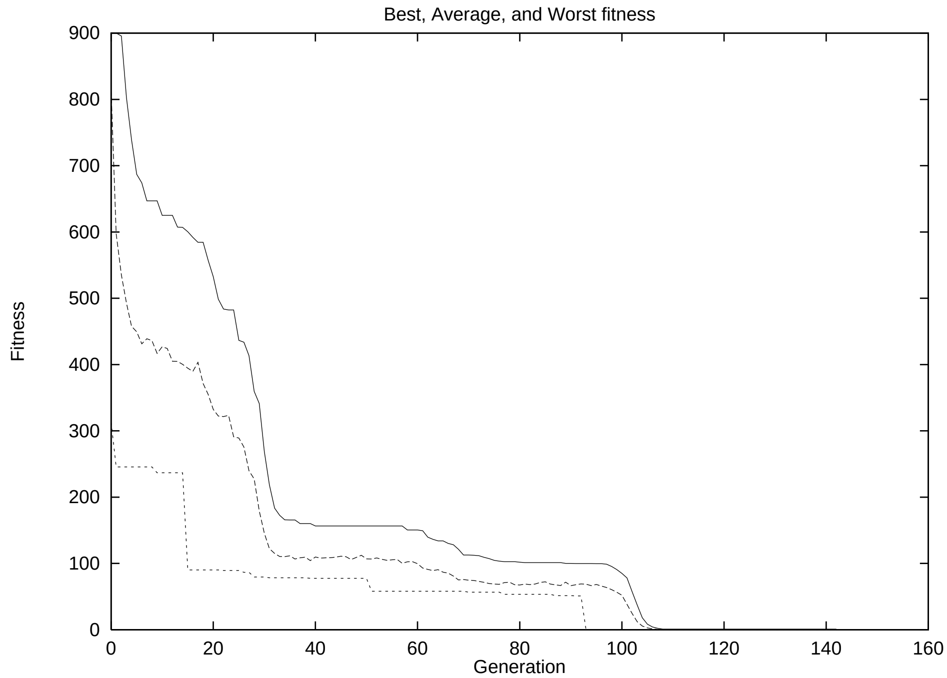


Figure VI.9: Worst, average, and best fitness per generation for a typical EA run.

(d1) $N_0^2 \rightarrow \langle(\{0,2\},\{1,3\}),\langle[0:3][1:2][0:1][2:3]\rangle\rangle$

(d2) $N_0^2 \rightarrow \langle(\{0,2\},\{1,3\}),\langle[0:3][1:2][0:1][0:2][2:3]\rangle\rangle$

(d3) $N_0^{\$} \rightarrow \left\langle((2\;2),(2\;2)),\frac{N_0^{\$-1}}{N_0^{\$-1}},\mathrm{odd}\right\rangle$

This is equivalent to the example grammar used in section VI.E, but contains a production, **(d2)**, which can never be used because it is masked by **(d1)**, which occurs earlier on the genome and has the same left-hand side[17]. It is worth noting that the first perfect individual detected in this run was slightly larger than Dino. In generation 93, the following genome was listed as the best-of-population individual.

(a0) $\langle N_0^6, 5\rangle$

(a1) $N_0^2 \rightarrow \langle(\{0,2\},\{1,3\}),\langle[0:3][1:2][0:1][0:2][2:3]\rangle\rangle$

[17] A single mutation, however, would serve to make **(d1)** unusable and "activate" **(d2)**. The mutation in question would change the S parameter on the nonterminal on **(d1)**'s left-hand side from 0 to 1.

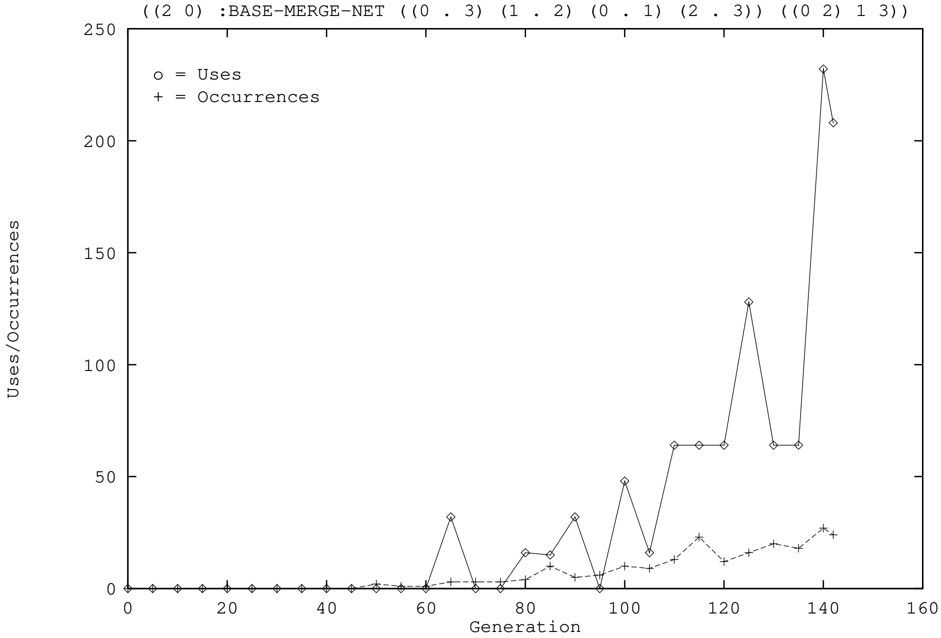


Figure VI.10: Use and occurrence curves for gene **(d1)**.

(a2) $N_0^{\$} \rightarrow \left\langle ((2\ 2), (2\ 2))\,, \frac{N_0^{\$-1}}{N_0^{\$-1}}, \mathrm{odd} \right\rangle$

Since the EA used for here reports an arbitrary choice among the best individuals for best-of-population at a given generation, we don't know whether or not Dino's genome occurred at generation 93. The final population did, however, contain both individuals.

The prevalence of some genes relevant to Dino's evolution are traced in the following graph. In each graph, the occurrence and usage frequency in all members of the population at each generation are plotted for a single production. The occurrence frequency counts how many times the gene occurred in all genomes during each generation. The usage frequency gives the number of times that the production was actually applied during development during each generation, again across all genomes. Figure VI.10 shows this information for the base network gene **(d1)**.

There are two aspects of this graph which are of particular interest. First, the gene was present well before the discovery of Dino, but the presence was somewhat furtive as evidenced by the fluctuations in the usage curve. Second, the usage and occurrence curves cross several times, indicating that there were (admittedly brief) periods of time during which this gene behaved as if it were "recessive" - present but not expressed. In addition to these two considerations, the occurrence graph for Dino was never as high as the population size. Recall that there is no pressure on networks (in terms of fitness) to encourage minimality. Thus, the EA did not distinguish between Dino and its close relative which arose in generation 93, shown above. Figure VI.11 gives data equivalent to that of figure VI.10, but for the recursive gene **(d3)**.

Looking at both graphs, it appears that gene **(d3)** was decisive in the simulation. Prior to its discovery, the simulation had not found a perfect individual (recall that the first was in generation 93). **(d1)** was present but not expressed at that time (see the preceding graph), so it appears that Dino's relative was "discovered" prior to Dino, but that all of the materials for building Dino were nonetheless present. Soon thereafter, Dino's base network gene became much more prominently represented in its usage curve, even though its occurrence is not more than about once per four genomes. This can again probably be attributed to its coexistence with another gene, **(a1)**, which exists in conjunction with **(a2)** which is the same as **(d3)**[18].

There were probably other genes in the final population which were similar in effect to **(d3)** in that they were had the same patch type and nonterminals and had call partition representations which unpacked to the same partition as Dino's call partition. The data here do *not* indicate the prevalence of those genes and so we have really only measured one possible representation for Dino which might have existed in the final population. Also not shown here are any genes for other networks different from Dino which might have coexisted with it in the final population.

Dino's recursion rule is the same as that used by the Modified Batcher

[18] See the immediately preceding grammars.

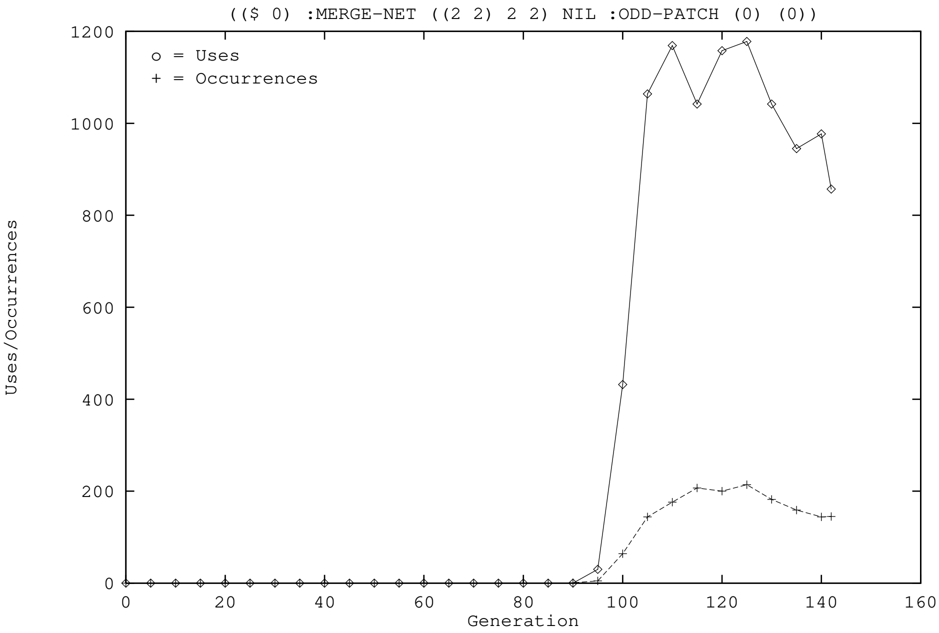


Figure VI.11: Use and occurrence curves for gene **(d3)**.

Network	#CMPXs	Delay	Spectral#
Batcher Odd-Even	N	$\frac{1}{2}(\log_2 N)^2$	1
AKS	$k \log_2 N$	$k \log_2 N$	unknown
Periodic Balanced Sequential	$\frac{N}{2} \log_2 N$	$\log_2 N$	$\frac{N}{2}$
Modified Batcher	$(\frac{1}{2} \log_2 N - \frac{1}{2})N + 1$	$\log_2 N$	$\log_2 N - 1$
Dino	$(\frac{1}{2} \log_2 N - \frac{1}{4})N + 1$	$\log_2 N$	$\log_2 N$

Table VI.4: Number of operations, delay, and spectral number for Dino versus several other theoretically-derived networks. N denotes the width of the network in question.

network, and the base-network production for that network was heavily represented early in the run, well before gene **(d3)** arose. It disappeared completely before that gene entered the gene pool and thus this simulation never discovered the Modified Batcher network.

Other simulations have, however, discovered the modified Batcher network. Various genomes which build this network for any size have been discovered, with considerable variety in how the call partition is represented on the genome, since there are many ways of writing down a given call partition using the representation described in section VI.E. To date, no EA run has found the periodic balanced sequential network. This may be because the grammar representation would require it to be represented by a fragile grammar, in which four productions interact in a very specific manner. It also seems to be the case that Dino's recursion production is a very general and powerful method for making merging networks.

Many of the networks found have used Dino's call partition. Some genomes in some simulations *have* contained call partition specifications which yield the periodic balanced sequential network's call partition, but that partition has never been used successfully. Table VI.4 summarizes the current status of endeavors to derive log-sequential sorting networks, and shows where Dino stands with respect to previous endeavors.

VI.H Discussion

One set of interesting results of this research has been the the collection of observations arrived at by manually working with Dino's description offline. As mentioned above, some of this manual tinkering lead to the discovery that it may be possible to reduce Dino's description by eliminating some of the delay cycles during intermediate passes of the use of the Dino as a log-sequential sorter.

VI.H.1 Onid, a "cousin" to Dino

Observing the structure of Dino's genome has lead to another interesting discovery. Namely, the observation that Dino is the result of using the base 4-input network from the periodic balanced sequential network and the recursive scheme for building the modified batcher network. The opposite trick can also be tried, and it appears also to lead to another log-sequential sorting network. This network has been dubbed "Onid" and was *not* found by the EA, but became apparent upon examination of Dino's genome and hand-constructed genome's for the known families of networks discussed above. Table VI.5 gives some interesting data about Dino, Onid, and the networks from which they derive.

In table VI.5, each row shows what fraction of all of the inputs were sorted after that number of passes. All of the networks had 16 inputs. Note especially the differing fractions correctly sorted after the third pass. Also, note that all of the networks sort at least 90% of test cases after three passes. This could indicate that much of the final pass is unnecessary, reflecting a lack of optimality in these networks when built on 16 inputs.

In fact, it may be possible to reduce Dino's size somewhat. When Dino is built for a number of inputs which is an even power of 2, the first delay cycle may be "skipped" during half of the iterations of the input through the network during sorting. If we were to build a network of $\log_2 N$ concatenations of Dino built on N inputs, we could remove from the network the second, fourth, sixth, etc... occurrences

Pass#	Modified Batcher	Dino	Onid	Periodic Balanced Sequential
0	0.0003	0.0003	0.0003	0.0003
1	0.0411	0.0901	0.0594	0.1158
2	0.4622	0.6221	0.7220	0.7775
3	0.9095	0.9766	0.9916	0.9980
4	1.0000	1.0000	1.0000	1.0000

Table VI.5: Percent bit strings sorted after each pass through four networks.

in that network of Dino's first delay cycle (the series of long CMPXs at the front of Dino). This operation removes enough CMPXs from the network that it executes the same number of CMPXs overall as the modified batcher network, though it still contains more delay cycles. For odd powers of 2, there will be an odd number of occurrences of the first delay cycle. Namely, there will be one in the last iteration of the execution of Dino that may, in addition to the even numbered occurrences, be skipped. If this occurs, the actual number of CMPXs executed is *less* than the overall count for them modified Batcher network, though the delay is still higher. These observations are still rooted in very few observations and are interesting conjectures, but as yet unproven.

VI.H.2 Proof for Dino

It is important to realize that the EA was not guaranteed to give a valid sorting network. Rather, merging was tested up to 64 inputs during the EA run. Later, offline, the sorting property was checked for 4, 8, and 16 inputs. The 32-input network could not be tested for sorting. Even using optimized C code on a Sun SPARC10 workstation, this would have required about a month of real time if the process received a large amount of CPU time. The worth of such testing is also somewhat dubious, since what is really desired is to know whether all of the networks derivable from Dino's genome, given the necessary start symbol and sufficient time for the derivation, will be correct log-sequential sorters.

Gill Williamson of the computer science department at U.C.S.D. has proven that the networks given by derivations from Dino's grammar are always correct log-sequential sorters. The proof makes use of the fact that these networks are merging networks. He has also shown Onid's grammar to specify a family log-sequential sorters, and in ongoing research has been investigating the prevalence of families of sorting networks with delay-1 patch networks.

Chapter VII

Data modeling for molecular biology

This chapter deals with inducing stochastic grammars to describe syntactic regularities in protein sequences. The training data for the experiments described below will always be composed of amino acid sequences taken from or comprising entire proteins. The terminal alphabet will always be the 20 amino acids found in proteins in nature, plus the wildcard character, as shown in table VII.1. The letters used to represent the 20 amino acids are standard. In what follows, a "grammar" should always be understood to be a SCFG unless otherwise noted.

The results here were arrived at using data suggested or provided by Dr. Michael Baker of the UCSD School of Medicine. The data sets are all concerned with the globin family of proteins. The training data are used to learn grammars which are in turn used to search a nonredundant subset of the Brookhaven Protein Data

A	Alanine	E	Glutamate	M	Methionine	Y	Tyrosine
R	Arganine	G	Glycine	F	Phenylalanine	V	Valine
N	Asparagine	H	Histidine	P	Proline	*	Wildcard
D	Aspartate	I	Isoleucine	S	Serine		
C	Cysteine	L	Leucine	T	Threonine		
Q	Glutamine	K	Lysine	W	Tryptophan		

Table VII.1: Terminal alphabet for grammar induction on proteins in EvoGrams.

Bank (PDB). This nonredundant subset of the PDB is called "OBSTRUCT" and was retrieved from the world-wide web[22]. The learned grammars are evaluated to see how well they retrieve all of the proteins in their respective families having been trained on a few such sequences.

Different search techniques are presented below for grammars learned from different forms of training data. We present a taxonomy of different approaches to database search using grammars and relate different search techniques using grammars to different search techniques explored by other researchers.

VII.A Modeling Proteins

Attempts to model proteins can have many different goals. We might asking what shape the protein takes when it is folded, what subregions of the protein are most important to its biochemical function, which regions are most likely to be conserved evolutionarily, what other proteins are related to it, or still other questions. For our purposes here, the task is homology detection: determining relatedness between proteins in the sense of having a common evolutionary ancestor. Within the framework of seeking to detect homology, we then want to find models that explain relationships between proteins, rather than just stating them.

VII.A.1 Prior Work

Finding a model of a collection of proteins in the form of a stochastic grammar is related to other forms of probabilistic modeling already well known. The "HMMer" system trains a hidden Markov model(HMM) to a set of sequences[30]. HMMer can tune its HMM to be a maximally discriminative("MD") or maximal a-posterior ("MAP") model of the data:

$$\mathrm{MAP}: \sum_i \log \Pr(S_i|M)$$

$$\mathrm{MD}: \prod_i \frac{\Pr(S_i|M)}{\Pr(S_i|M) + \Pr(S_i|R)}$$

where the S_i are the training sequences and M is the HMM; the probabilities in M are tuned to maximize the quantity given in each case. The model R is the random or "background" model and provides a probability that S_i occurs because of random chance. The tuned model can be used to retrieve similar sequences from a sequence database. Since HMMs are isomorphic to stochastic regular grammars (SRGs), this form of modeling is very similar to that pursued here.

The MEME system tries to identify highly informative fixed-width regions within a protein and model these using simple HMMs[11]; the resulting models are called "motifs." MEME motifs may be concatenated for different informative regions to form a "motif-based HMM," as in the metaMEME system[42]. The HMM built by metaMEME can be tuned to be optimal in the MAP or MD sense, just as with HMMer. MEME uses MAP to tune its motifs. Each MEME motif is a matrix that gives a series of columns representing positions in the motif. Each column is a probability distribution over the 20 amino acids; MEME also use a regularizer to ameliorate effects of uninformative features due to sampling effects in particular data sets[1].

Finally, the simple alignment tool BLAST allows a similarity score for a pair of proteins to be computed. This allows retrieval of similar sequences from a database. A technique called "Family Pairwise Search" (FPS) can be used to search from a "training set" of proteins by averaging the pairwise scores between each sequence in a database and the members of the training set; this essentially uses the training set as an extensional model of the data[41].

Searching a database with HMMer, MEME, or FPS involves scoring each sequence in the database using a model derived from a training set of proteins. The three scoring techniques can be distinguished according to how rigorous they try to be about the distance between sequences. Edit distance is the minimal number of insertions, deletions, or substitutions required to change one sequence into another.

[1] These are Dirichlet mixtures.

In HMMer's case, the HMM is an estimated model of edit operations between the database sequence and the training set as whole. FPS averages BLAST scores, which were developed to approximate edit distance. The models in MEME and metaMEME represent a compromise: the motifs represent an attempt to be precise about substitutions over short regions; searches based on distinct motifs can be combined. The combination is done rather straightforwardly in MEME using an assumption that distinct motifs are independent of one another. In metaMEME the motifs are incorporated in an HMM which can encode more information about co-occurrence of motifs. metaMEME, HMMer, and MEME all use log-odds scores of the kind described below as their method of scoring sequences during a database search.

A major difference between HMMer and EvoGrams is that HMMer fixes the basic form of the model to account for edit distance and concentrates on tuning its probabilities. The advantage of this approach is that it is possible to write database search tools to take advantage of the structure of the model. The disadvantage is that HMMer requires a large amount of training data to arrive at a model that reliably retrieves other instances of the family from which the training data were derived. This is because HMMer's models contain a large number of probabilities to tune, about 10 per sequence location beyond the 20 needed to specify a distribution over the 20 amino acids. MEME, by contrast, encode a distribution over the 20 amino acids at each of w contiguous locations. metaMEME uses more parameters to fit together motif models and still more to encode simpler models regions between motif occurrences. Note that though it has basically no parameters to tune, FPS doesn't give us a model of our training set of proteins. Rather, it allows us to expand the set reliably with proteins from the same family as those in the training set. There is no attempt to codify the essence of membership in the family.

VII.A.2 Evolving SCFG Models

By evolving grammars we hope to learn models with less tunable parameters. For example, if only some of the 20 possible amino acids are present at a given sequence position, then a grammar that contains only productions to generate those five amino acids at that position is preferable to a grammar that allows any amino acid at the given location, since the latter contains 15 tunable parameters that could be eliminated. Further, some productions will usually specify multiple amino acids; if the alternatives of a nonterminal, N, are all of the form $N \rightarrow a_1 a_2\ (\rho)$, where the a_i are amino acids, then we have a large reduction in potential numbers of parameters as long as the number of alternatives of N is less than 400. Thus, we expect that evolved grammars will demonstrate some degree of parsimony, and one advantage we will look for is a reduction in the number of tunable parameters in grammars tuned with the inside-outside algorithm relative to the number of parameters in models tuned by other algorithms such as HMMer or MEME.

Another important difference between EvoGrams and either HMMer or MEME is that EvoGrams allows context-free constructs, and thus allows models from a more general class than the other two. Since we are learning stochastic grammars, the models we will find have both a syntactic and statistical aspect. It is known that there are subsequences in proteins whose syntax cannot be generally modeled with SRGs (equivalently, HMMs)[75, 1]. These include some beta-sheets and palindromic binding regions in some homodimers. In these cases, there are syntactic regularities in the protein sequences that can be captured by context-free rules but not by regular rules, and thus a SCFG will allow the introduction of a parameter capturing the regularity and an SRG or HMM will not. There are also, however, subsequences in proteins whose syntax cannot be captured using context-free grammars. These include some parallel beta sheet constructions[75, 1].

As in the case of the context-sensitive language $\mathcal{L}_{\mathrm{abc}}$ from chapter V, we still try to learn a structural generalization and account for other aspects of the data statistically. As in the results for $\mathcal{L}_{\mathrm{abc}}$ from chapter V, we expect that different runs will find different generalizations of the actual syntax of the language from which the data were drawn.

Alternatively, a group of researchers at UCSC has exploited the lack of closure under intersection for CFLs to model explicitly model pseudo-knots in tRNA molecules, which are above context-free, using SCFGs. Though these models were hand-constructed and then tuned, they point out the viability of intersections of SCFGs as models of non-context-free syntax[16]. The intersection of context-free languages(CFLs) may be context-sensitive, so one strategy if we suspected our data of being more complex than context-free grammars is to find distinct grammars and intersect their search results from a given database.

As a more formal example, the intersection of the languages of the grammars learned from the training data for $\mathcal{L}_{\text{abc}}$ gives the correct target language, $\mathcal{L}_{\text{abc}}$. One important issue that we do not explore here, since we do not explore this line experimentally, is how to combine the probabilities assigned by the grammars involved in the intersection.

EvoGrams has a concept that plays the same role as motifs play in MEME: "grammatical motifs."

Definition 4 *A grammatical motif of grammar* $\Gamma = (\Sigma, N, S, R, \rho)$[2] *is the set of strings derivable from a nonterminal* $N_i \in N$*, called* N_i*'s* extension*:*

$$\text{extension}(N_i) = \{\sigma \in \Sigma^* | N_i \Rightarrow \sigma\} \tag{VII.1}$$

If every string in extension(N_i) *is a substring of some string in* $\mathcal{L}(\Gamma)$ *then we call* extension(N_i) proper*. If* Γ *is* **reduced***, then* extension(N_i) *is proper for all* $N_i \in N$*, since if* Γ *is reduced every rule in* R *must be used at least once in a derivation of some* $\sigma \in \mathcal{L}(\Gamma)$ *by definition.*

Grammatical motifs are defined syntactically by collections of productions, and statistically by the probabilities on those productions. Syntactic motifs are the EvoGrams equivalent of MEME motifs, though they are not necessarily fixed width. Informally, we will refer to Γ as a "motif grammar" whenever Γ's start symbol defines the only grammatical motif of interest, especially when Γ only models particular regions of

[2] Recall definition 3.

sequences and not whole sequences. Below, we will learn motif grammars from small data sets of subsequences of proteins.

VII.A.3 Training Regime

In each of our experiments, we draw training sequences from a protein family, F, and learn a grammar, Γ_F from it. In some cases we will divide the training data into sets of subsequences, s_i, learn several Γ_{s_iF}, and combine these models of subsequences to arrive at a model of F. We will judge the success of these grammars as models by seeing how well they retrieve other members of F not occurring in the training data to the exclusion of sequences that are not members of F. We have several options in preparing our data; those discussed here fall into three categories, based on how much of each training sequence is used in the training data:

whole proteins: In this case, we're trying to discover a grammar that captures entire instances of a given protein family. Any restriction of attention to informative or biologically interesting regions of the sequence must arise during learning.

important regions: The s_i can be chosen automatically for informativeness as in MEME[11], or can be chosen based on biological significance, as in our experiments below in which we learn from helices extracted from globin sequences.

corresponding substrings: This is an attempt to compromise when the sequences are too long for parsing them in their entirety to be within what we consider to be reasonable computational costs. The exact meaning of the word "corresponding" is important, but here we mean subsequences defined by a range of indices into a set of aligned sequences. The sequence alignment matches up conserved regions across the sequences, inserting gaps as needed to account for different sequence lengths. We would prefer to cut the subsequences off at an index that falls in a gap for most sequences.

Thus, we are either seeking a grammar, Γ_{F}, that gives complete or partial parses as did the "greedy covers" in section IV.C. Greedy covers and "pessimistic completion" will play an important role in how we score database sequences using a particular Γ_F. Our training set will be purged in the case in which we work with full sequences; purging gives a set of sequences guaranteed to to have pairwise similarity smaller than some pre-decided value, relative to a chosen similarity scoring system, often BLAST[4].

VII.A.4 Grammatical Log-odds Scores for Proteins

Recall that in EvoGrams, fitness is defined using log-odds scores, based on greedy covers and pessimistic completions, as discussed in chapter IV. Here we briefly discuss what this means for proteins sequences and recapitulate the log-odds formula. Also, the use of a similarity metric over amino acids complicates the computation of inside and outside probabilities and is discussed in the following subsection.

When scoring sequences according to grammars, it's important to consider the probability assigned to a sequence, σ, by a grammar, Γ, relative to the probability that σ would be generated by chance. This is encoded by the log-odds for σ given Γ:

$$\mathrm{LO}(\sigma|\Gamma) = \log P(\sigma|\Gamma) - \log P(\sigma|\Gamma_{\mathrm{Random}}) \tag{VII.2}$$

$\mathrm{LO}(\sigma|\Gamma)$ is equal to $\mathrm{fcontrib}(\Gamma, \sigma)$, σ's contribution to Γ's fitness as defined in section IV.C, when there is at least one derivation of σ from Γ. Γ_{Random} is as in chapter IV:

$$\begin{array}{llll} \multicolumn{4}{l}{\text{Start symbol: S}} \\ \mathrm{S} & \rightarrow & {}^*\mathrm{S} & (1\text{-p}) \\ \mathrm{S} & \rightarrow & * & (\mathrm{p}) \end{array}$$

Γ_{Random} accounts for any given sequence using the hypothesis that the amino acid at each sequence position is chosen randomly according to the wildcard distribution, and independent of the other amino acids in the sequence. The wildcard distribution is one of the following three choices:

uniform The wildcard generates each amino acid with the same probability, 0.05.

data The probability of each amino acid according to the wildcard is its frequency in the training data.

protein The probability of each amino acid according to the wildcard is its frequency in proteins generally, as measured over a large number of proteins. We use Jones's 1992 update to Doolittle's 1986 frequencies[26, 49].

$\mathrm{LO}(\sigma|\Gamma)$ is used in distinct roles; when σ is part of the training data from which Γ was learned, and when using Γ to score σ during a database search. In both cases, we are only interested in Γ as a model which accounts for σ when Γ assigns more probability to σ than would the hypothesis that σ occurred at random, as captured by Γ_{Random}.

From section IV.C another motivation for using log-odds is that doing so helps to eliminate the inequity among sequence probabilities owing to differences in length. As was noted, this is important for learning because it serves as a method of scaling fitness contributions to avoid giving short strings too much influence over fitness values. In the case of database searches, this is also an important issue, since we do not want sequence σ from some database to be scored higher than other sequences merely because σ is shorter than those other sequences. If length is an important characteristic on which to base retrieval of sequences from a database, we expect it to be captured in the learned grammar, and especially in that grammar's tuned probabilities. Note that when we perform searches, Γ_{Random} has its parameter p set to maximize the the probability of each sequence σ: $p = |\sigma|$; the length-adjustment is done per-sequence in the computation of log-odds scores.

VII.A.5 Amino Acid Similarity ("cooking with PAM")

An important aspect of protein data is that many members of a family may be different from other members in one or two positions even in highly conserved regions. If the training data were identical in some positions, this may cause us to miss proteins with a single amino acid mutation when we perform a database search using the learned grammar. Thus, we would like to modify $\Pr(\sigma|\Gamma)$ using the probability that a given individual amino acid difference represents a mutation event. When this probability is high, $\Pr(\sigma|\Gamma)$ should undergo little modification. In terms of SCFGs, we want to impose a similarity metric over nonterminals to account for cases where one has been substituted for another.

To decide whether a given replacement is a matter of chance, we use the "PAM" matrices[24, 49]. These give probabilities and log-odds of replacement for pairs of amino acids, based on a simple model of evolution as a Markovian process of replacements, insertions, and deletions of individual amino acids. For amino acid j replacing amino acid i, the log-odds give the log of the ratio of the probability that the observed occurrence of j is due to a mutation from i, to the probability that j occurred independently by chance. Both probabilities are computed using observed data from proteins. The replacement probabilities are computed using counts of replacements between aligned sequences of proteins known to be related. The chance probabilities are given by frequencies of occurrences of amino acids observed over a larger number of proteins. Log-odds greater than zero for amino acids i and j indicate that finding j when i was expected is more likely due to a mutation than to chance occurrence of j.

PAM probabilities and log-odds are presented in a 20x20 matrix showing the probability that amino acid j, labelling a column, replaces amino acid i, labelling a row. We will refer to this probability as M_{ij}. The symmetric log-odds matrix tells us whether the replacement occurs at greater than chance odds. We will refer to entries in the log-odds matrix as L_{ij}. We call two residues similar whenever they have positive log-odds, $L_{ij} > 0$.

PAM matrices assume a given evolutionary distance between sequences. The PAM(1) matrix assumes 1 mutation event per 100 amino acid residues. Since the probabilities and log-odds in the PAM matrices measure the number of point mutations which were *accepted*, and are thus seen in surviving proteins, the probabilities are very small. It is more common to use a PAM matrix for a larger distance, arrived at by exponentiating PAM(1): PAM(D) = PAM(1)^D. D is chosen to reflect how distant a relationship we wish to detect.

Once we have the PAM(D) matrices, we can use them to modify the behavior of the EvoGrams parser so that sequences similar to the sequence in question are assigned complete parses. To do this, we allow parsing to occur around amino acids other than those mentioned in a given rule, provided that the amino acid in the sequence, i, and the amino acid expected by the rule, j, satisfy $L_{ij} > 0$.

As an illustration, suppose we have some sequence containing a Tryptophan, "...W..." where a dot denotes an arbitrary amino acid, and suppose grammar Γ parses the sequence. If we are using PAM(250) then (W,F) and (W,R) have log-odds scores above zero, and we would also be able to parse "...F..." and "...R...". This is accomplished by allowing a dotted rule such as $0 \rightarrow 0 \cdot WGK$ to expand around an entry for W, F, or R to form $0 \rightarrow 0W \cdot GK$.

The mutation probabilities M_{ij} are used in computing sequence probabilities when using PAM information during parsing. If we are parsing a sequence σ with σ_i = W, the probability of seeing W at location i in σ is normally 1. If we are using PAM matrices, however, this probability is replaced with the diagonal entry of M for amino acid W: M_{WW}. If the sequence contained an F where the W was expected, we would use M_{FW}, the probability that the amino acid mitigated by the grammar, W, *was* present and mutated into an F over time. This probability is normalized over probabilities $M_{i\mathrm{W}}$ for amino acids satisfying $L_{i\mathrm{W}} > 0$. The normalization is over amino acids with positive log-odds with respect to W to avoid using mutation probabilities for substitutions less likely than chance occurrences of amino acids other than W where a W was expected. The idea is that the grammar is a model of some collection of sequences expected to contain a W at some point, and when we find a PAM-related amino acid, we multiply the derivation probability by the probability that the W was mutated into it.

PAM information is not used during learning in the experiments reported here, because the computational costs are too high to evaluate every grammar in a population with it. Rather, we use it to allow a grammar to retrieve homologs which a database search which might miss due to variation at a single amino acid from what the grammar learned to expect from the training data. Some of the sequences removed in a purge may be nearly identical to other sequences kept in the data set, but with one or two amino acids changed in an otherwise conserved region. Modifying the parser with PAM information is meant to allow us to find distant homologs without needing to throw the entire unpurged collection into the training data.

The main drawback to using PAM information is its expense. As a very simple example, if the sequence to be parsed is "WGKV" and we are using PAM(60), then we are doing the equivalent of parsing all sequences which can be formed by replacing any of the four amino acids with an amino acid similar to it in the sense that the two have positive log-odds. For PAM(60), the sets of possible values at each position are: { W }, { G }, { K, R, N }, and { V, M, I } respectively. Thus we are simultaneously parsing nine different sequences given the variability in the last two positions. This number increases when we consider longer sequences and especially higher PAM distances, since the number of sequences being parsed simultaneously is the product of the numbers of amino acids possible at each position given the PAM distance used.

VII.B Database searches using grammars

An important part of the work in this chapter is the idea of searching a database using a grammar. A database is understood to be a triple: $D = (I, S, v)$. I is a set of sequence identifiers, $S \subseteq \Sigma^*$, is a set of sequences of amino acids, and v is a function $v : I \rightarrow S$ which uniquely maps each identifier to a sequence. The database search will return a subset of I. The intuitive question we ask for each sequence in the database is: "How much more or less likely is the sequence to occur according to the learned grammar Γ_F than it would be likely to occur according to Γ_{Random}?" The following subsections discuss nonredundant databases and different forms of search, covering variants based on whether the whole sequence is compared to Γ_F and on whether partial matches to Γ_F are of interest.

VII.B.1 Nonredundant databases

Suppose we learn a grammar from some sequences and we then score each entry in a protein sequence database, such as the PDB. The scoring is done in the usual way, by evaluating $\mathrm{LO}(\sigma|\Gamma_F)$ as defined in equation VII.2.

A common feature of most biological sequence databases will usually cause our search to be appear to be doing much better at retrieving similar sequences than is the case. Due to sequence similarity conservation during the evolution of the proteins, most sequence databases contain many sequences that are very similar to one another. Such conservation often results because of functional constraints on the proteins and because of relatively recent divergence from common ancestors. Some pairs of sequences may be more than 90% identical and many pairs of sequences that seem dissimilar at a glance appear more similar after sequence alignment techniques are used to account for replacements, deletions and insertions.

If a search is carried out in a database that contains many clusters of highly similar sequences, the search is likely to retrieve these clusters in groups, so that being able to successfully retrieve one member of the group (i.e., by assigning it a high score relative to other sequences) means the grammar is likely to retrieve the rest of the cluster. If a few of the clusters are inordinately large, this can have a profound effect on apparent performance. As an example, the version of the PDB used here contains about 10,000 entries[9]. The reduced subset of the PDB that we use for our searches[22] in the results reported below contains approximately one-tenth as many sequences, after sequences are eliminated to guarantee that no two sequences are more than 50% identical.

VII.B.2 Performing database searches

The mechanics of performing a database search are relatively simple: compute a score for each sequence in the database and return the identifiers of high-scoring sequences. In fact, there are many methods for performing the search based on choices regarding partial matches between sequences and grammars, and, for motif grammars, how many motif occurrences per sequence are allowed.

We will classify different methods for searching a database by considering answers to four questions. As we consider these questions below, we relate them to similar forms of search using HMMs and MEME motifs. The four questions are, for sequence σ and grammar Γ_F:

Q1 do we require a complete or allow a partial match to Γ_F?

Q2 do we require a complete or allow a partial match of σ?

Q3 if **Q2** was answered "partial," how many matches between σ and Γ_F are used in scoring σ?

Q4 will we use amino acid similarity information during the database search?

A complete match to Γ_F means that we are only concerned with derivations of σ or substrings of σ from Γ_F's start symbol; partial match means that we consider other nonterminals. A complete match of σ means that we require that some nonterminal of Γ_F generates σ in its entirety; a partial match means that we consider nonterminals of Γ_F which generate substrings of σ.

We can deal with **Q4** separately since it is an orthogonal consideration to the other three questions. If amino acid similarity is used, then the parser operates in the modified fashion discussed in section VII.A.5. Since all of the strategies below use a score based on log-odds, all of them can be used with or without the information from PAM matrices. The principal advantage of using mutation probabilities is that it allow parses to be completed for more sequences, as was mentioned above, and this makes the grammar less brittle with respect to accounting for homologs. The principal disadvantages are that using PAM(n) for too large an n might make the grammar too accepting, and that the cost of parsing is markedly increased. If we know in advance the average pairwise similarity within the family we are attempting to model, we are more likely to make a good choice for n and reap the benefits of using amino acid similarity in parsing sequences during database searches.

We can further eliminate two of the four possibilities given by answering **Q1** and **Q2**. If we demand a complete derivation of σ from Γ's start symbol, then we are interested in Γ_F as a model of whole proteins, and we are only concerned with matches to Γ_F's start symbol. This situation does not arise in practice in this thesis because we will always want to assign scores to partial matches between sequences and grammars. We want to score partial matches in this manner because we do not want to reject out of hand a match between a grammar's start symbol and all but a few amino acids of a sequence. In the opposite case, in which we allow partial matches to Γ of parts of σ, we will end up with a large number of scored fragments per sequence. The question then becomes one of how to combine these or choose one to represent the match between Γ and σ. These issues seem to lead naturally to seeking an account of all of σ using an incomplete parse according to Γ, since they are questions of combining fragment matches into a meaningful account of the degree of overall match between Γ and σ. Thus, we are left with the two cases in which **Q1** and **Q2** are answered differently. We considering these below, and consider **Q3** in the relevant case.

Partial matches to Γ_F with complete matches of σ: In this method of database search, the grammar usually will have been learned from whole protein sequences or from very large subsequences of whole proteins; if this is not the case, Γ_F will not have much chance of containing information relevant to seeking an account of any sequence σ encountered during the search. This corresponds to the strategy of matching a whole sequence to part of an HMM in the HMMer model.

If Γ_F does yield a complete derivation of a given σ, we can score σ straightforwardly by using the log-odds score:

$$\mathrm{score}(\sigma|\Gamma) = \mathrm{LO}(\sigma|\Gamma).$$

Even in this case, there might be sequences in the database which have a great deal in common with our training set but which are also much longer and perhaps can't be parsed in an acceptable amount of time: our nonredundant subset of the PDB contains sequences whose length is up to about 1023 amino acid residues. Unless the learned grammar allows fairly efficient parsing, it is unlikely that we'll be able to evaluate complete parses for all sequences in the database. This probably means no worse than quadratic in the length, n, of the sequence to be parsed. Context-free parsing has a worst case bound of $O(n^3)$. Some grammars, however, parse in time $O(n^2)$ and still others are as efficient as $O(n)$. We do not currently bias EvoGrams towards grammars for which parsing is cheap, since we do not know how a search would be affected by such limits on the search space; that issue is a possible subject of future investigation.

To be able to evaluate sequences for which complete parses are unavailable we return to the ideas of chapter IV concerning partial credit and greedy covers of sequences according to chart parses. When evaluating the fitness contribution for a grammar from a sequence which it didn't recognize[3], the method used below is to run the chart parse and then find a collection of chart entries which came as close as possible to covering the sequence. This was described in section IV.C and illustrated in figure IV.2. Here, as before, the covering is found in a way that prefers entries covering longer subsequences over those covering shorter subsequences, and among equal-length entries chooses the one with the highest log-odds score.

Once this collection of entries is found, we imagine completing the parse using the background model, Γ_{Random}. The nonterminal for each entry in the greedy cover is substituted into σ in place of the substring of σ covered by the entry. Γ_{Random} is used to provide a probability for this sentential form to serve as the denominator of our odds. The only exceptional condition is when the sentential form contains a

[3]This means a sequence for which the grammar finds no complete parse.

nonterminal, which Γ_{Random} cannot generate. We use the probability of the entry giving rise to the nonterminal occurrence in the sentential form as the probability of the nonterminal. This amounts to using the background model to fill in the gaps left by our greedy cover to arrive at a complete parse tree.

The same mechanism can be applied here to recognize whatever substructures exist to be found in a sequence that a grammar's productions can recognize and then assign a log-probability to the sequence that, as before, corresponds to a "pessimistic completion" of a parse of the whole sequence. For this reason, we use:

$$\text{score}(\sigma|\Gamma) = \text{LO}_{\text{partial}}(\sigma|\Gamma) = \log P_{\text{partial}}(\sigma|\Gamma) - \log P_{\text{Random}}(\sigma)$$

when no complete parse of σ by Γ exists, where $\log P_{\text{partial}}(\sigma|\Gamma)$ gives the probability of a parse pessimistically completed from a greedy cover.

Complete matches to Γ with partial matches of σ: If the training data was made from structural elements extracted from sequences or other short subsequences, then we need to score subsequences of database entries to determine whether they match the grammar. Since the grammar will be learned from fragments of proteins we do not expect to arrive at a grammar that describes proteins in their entirety. In this case, we are looking for a detector for features inherent in the short subsequences from which we learn Γ_F: we are trying to learn what we earlier referred to as a motif grammar. During the database search, we are not trying to use Γ_F to account for entire sequences as in the previous case. Rather, we wish to use the grammar to decide whether a subsequence resembling those from which it was learned is present.

It is possible that we will find more than one subsequence of σ which is in $\mathcal{L}(\Gamma_F)$. In this case, we must answer **Q3**: how many matches will we use in computing a score for σ? In the MEME motif discovery system, it is possible to select whether one wishes to discover motifs that occur once per sequence, zero or one times per sequence, or multiple times per sequence. Since we know that each member of the data set is a single genuine instance of the motif we wish to learn, we don't need these controls during learning[4]; we do however retain a related option during database searches for the purpose of restricting focus of attention to the few best matches between subsequences of σ and Γ_F.

Since the training data consist only of subsequences of actual proteins, it is inappropriate to expect that the learned grammar will find a complete parse for a whole protein[5]. Rather, the grammar should be expected to find complete parses for *fragments* of proteins that contain the structural elements or, more generally, subsequences from which it was learned.

To score sequences from the database, we perform a greedy cover of them using the grammar and limit the length of the entries used in the cover to lengths close to the lengths of the training data. We further restrict how many entries are allowed in the cover according to our answer to **Q3**. If the answer is m, then the cover will contain at most m entries with the highest log-odds scores. This allows us to limit how many of the entries count for purposes of computing score$(\sigma|\Gamma)$ based on information about how many matches should be expected. In many cases below, we will set $m = 1$ since we will learn from instance of α-helices that are known to occur at most once in each sequence of the family.

Finally, an important special case occurs if m is a very small value. In this case, we risk penalizing so heavily when we use pessimistic completions that we obviate any normalization over sequence length accomplished by scoring with log-odds. In the following, whenever $m = 1$, we directly score the single match to the sequence and do not perform pessimistic completions.

VII.C Data sets

In the following section, we will show stochastic grammars learned from data sets for two protein families. We then use the learned stochastic grammars to

[4]MEME needs them because it selects its own data sets and trains simplified HMMs, its motifs, on them.

[5]Or at least we would not expect any such complete parse to be particularly informative

retrieve sequences from a nonredundant subset of the PDB. This nonredundant subset of the PDB, called "OBSTRUCT," was retrieved from the web site at the European Molecular Biology Lab (EMBL) and is guaranteed to have pairwise sequence identity of no more than 50%. The OBSTRUCT database is about one-tenth the size of the full PDB. For each protein family below, one data set includes either whole protein sequences or long subsequences given after Dr. Baker clipped the ends. The other data set for each family contains structural elements, helices and strands, or corresponding subsequences from a multiple sequence alignment using BLAST[4].

The most familiar members of the globin family are probably the two distinct strands occurring in human hemoglobin. The globins divide into hemoglobins and myoglobins, the latter older evolutionarily. Many of the hemoglobins are made of two distinct sequences labelled 'A' and 'B.' We will use data in two forms from the globin family: instances of helices trimmed by hand from six sequences, and complete sequences.

Helix data sets from 6 globins

Our research collaborator, Dr. Michael Baker provided us with data from the following six globins[6]:

mhu Human myoglobin

mch Chicken myoglobin

hhu_A Human hemoglobin, chain A

hch_A Chicken hemoglobin, chain A

hhu_B Human hemoglobin, chain B

hch_B Chicken hemoglobin, chain B

For each sequence, we have the subsequence corresponding to each of the 6 α-helices A, B, E, F, G, and H. Helices C and D are too short to be of use, and helix D is not present in every sequence. For each of the other six, we have a data set containing

[6]The sequence labels were assigned arbitrarily by the author for mnemonic value; they are not PDB labels.

six examples. The training examples range from about 10 to about 20 residues. All of the helix boundaries were determined by Dr. Baker.

Two more data sets allow us to explore the effects of choosing the training data to be slightly more than just the helix in question. The two data sets Ac and Bc contain the instances of helices A and B but each example also contains a few amino acids occurring to each side of the helix instance in the protein from which the instance was taken. In the case of data set Ac, we use three amino acids of context before and after the helix. In the case of data set Bc, we use six amino acids to each side. These data sets allow us to observe the difference between data sets cut at and beyond the end of the helices.

An additional data set allows us to study the effects of learning from subsequences spanning two helices. The data set for the FG pairing was created by taking all residues from the first of F to the last of G from each of the six sequences. The training sequences in this set were each 35 or 36 amino acid residues long.

For all of these data sets, the aim is to learn a motif grammar for the helix represented by that data set. We attempt this under various circumstances and then perform database searches with individual and groups of helices to check our ability to retrieve globins using combinations of grammatical motifs. As will be seen below, we will combine searches either by combining the results from searches using individual grammars, or by combining the grammars into one grammar used to search for whole sequences.

Whole sequences of 14 dissimilar globins

In addition to learning motif grammars for individual fragments of globins, we also attempt to directly learn from whole sequences. The training set in this case consists of the 14 globins found in the OBSTRUCT subset of the PDB. The sequences' lengths vary from about 135 to 160, and have pairwise identity of less than 50% since they are from the OBSTRUCT database. Note that there are many more globins in the full PDB.

This data set is meant to provide us with a reliable spanning set of globins from which to learn a SCFG for the whole family. In the next section we will report experiments on this data set and describe the results in detail. Most of these experiments were expensive enough to require running on the Cray T3E.

VII.D Experiments and results

The experiments below all have a common goal: for $F =$ globins we want to learn a Γ_F that scores members of F higher than other sequences. In the case of learning from globin helix data sets, we are satisfied if combining the grammars $\Gamma_{\text{globin},h}$ for $h \in H \subseteq \{\text{A}, \text{Ac}, \text{B}, \text{Bc}, \text{E}, \text{F}, \text{G}, \text{H}, \text{FG}\}$ reliably separates the globins from other proteins. For the most part our standard practice will be to learn a grammar from the training data and then search the OBSTRUCT database.

Section VII.A, reiterated the distinction between statistical and structural description introduced in chapter III and explored experimentally in chapter V. Here, it is important to bear in mind this distinction since we do not necessarily expect to be able to extract all of the globins from a database based only on syntactic considerations. Rather, we want $\mathcal{L}(\Gamma_F)$ to capture as much context-free syntax as is available and then tune the grammar and distinguish any proteins not in F based on comparing log-odds scores with actual members of F. This is the practical side of mixing syntactic and statistical description.

As was noted at the beginning of the chapter, Γ_F is not likely to give us the fastest available method for retrieving sequences from a database. Family-pairwise search, MEME, and even HMMer will usually be faster. If grammatical modeling is to be useful, it must do more than retrieve the correct sequences in order to justify the extra time spent evolving grammars. Thus, for the results that follow we will typically provide further analysis of the data than just retrieval statistics. In particular, we explore methods of using the grammatical model to explain membership in F rather than strictly to detect membership. These will include analyses of production usage by sequences in the family of interest, as well as proposed measures of similarity using the results of parses. This is something we can only do to a small extent with HMMer, MEME, and metaMEME because all three work by tuning models of a fixed form to the training data. It is because EvoGrams is learning the form of its models and not just the parameters that it is worthwhile to attempt this sort of analysis.

VII.D.1 Learning from helices

In this section, we present results of using EvoGrams to learn grammars for individual helices extracted from globin sequences. Though we will be learning motif grammars for small subsequences of globins, the task is markedly different from that of MEME. MEME makes an informed selection of subsequences of proteins, automatically, on which to train a simple HMM which represents a motif. Thus, the training sets for MEME's HMMs are selected to yield informative motif models. MEME's question is essentially, "Can we find relatively short, informative regions in a set of proteins?" Here we ask a different question: "Can we learn to describe a set of proteins by learning models of small regions which we know are important because they have structural significance for the protein?"

Recall that we learned grammars for nine different data sets, each containing six short sequences. These grammars are presented in table VII.2. All of these EvoGrams runs were 2000 generations long with a population size of 100. The data sets contained two of each training example: this is a way to reduced the influence of the grammar complexity measure(see chapter IV) on grammar fitness by a factor of two. If we place every training example in the training set twice, it contributes twice as much to fitness and thus the performance of grammars will be twice as high relative to the complexity component of fitness. If we allow very large numbers of duplicates, the complexity measures influence will become negligible. In this subsection, we will present the results of searching the OBSTRUCT database with these grammars individually and in combination. Finally, we will analyze the way in which one of the grammars arose and discuss how some of the grammars model their training data.

A

Start symbol: 6

6	→	46	(0.751)
6	→	V	(0.207)
6	→	1I	(0.0415)
4	→	*9	(0.573)
4	→	A9	(0.141)
4	→	*9Q	(0.14)
4	→	D	(0.0922)
4	→	*	(0.0536)
1	→	K	(0.527)
1	→	WG1	(0.263)
1	→	W	(0.105)
1	→	EK	(0.104)
9	→	31	(0.65)
9	→	4T	(0.15)
9	→	VL9	(0.0982)
9	→	V	(0.0517)
9	→	F	(0.0499)
3	→	E	(0.308)
3	→	NV	(0.231)
3	→	L	(0.154)
3	→	AD	(0.154)
3	→	A	(0.0769)
3	→	TI	(0.0769)

Ac

Start symbol: 4

4	→	*0	(1)
0	→	4	(0.656)
0	→	*8	(0.227)
0	→	LS0	(0.0577)
0	→	WQ0	(0.0299)
0	→	88	(0.0297)
8	→	K4	(0.46)
8	→	WG8	(0.208)
8	→	AD	(0.152)
8	→	H	(0.0832)
8	→	A	(0.0524)
8	→	D	(0.0444)

B

Start symbol: 9

9	→	29	(0.839)
9	→	3*	(0.161)
2	→	E	(0.244)
2	→	A	(0.181)
2	→	*	(0.125)
2	→	V	(0.088)
2	→	9RL	(0.0768)
2	→	GH	(0.0576)
2	→	Y7	(0.0384)
2	→	77	(0.0382)
2	→	M	(0.0366)
2	→	I	(0.0338)
2	→	R	(0.0334)
2	→	L	(0.028)
2	→	7Q	(0.0192)
3	→	L	(0.6)
3	→	F	(0.4)
7	→	G	(0.858)
7	→	C	(0.142)

Bc

Start symbol: 5

5	→	*5	(0.884)
5	→	EAL1	(0.0557)
5	→	EA1	(0.0308)
5	→	IA1	(0.0297)
1	→	*1	(0.923)
1	→	P38	(0.0774)
3	→	*	(0.226)
3	→	W	(0.164)
3	→	Q	(0.158)
3	→	E	(0.155)
3	→	K	(0.148)
3	→	L	(0.147)
8	→	T3	(1)

E

Start symbol: 6

6	→	65	(0.656)
6	→	16	(0.278)
6	→	8G	(0.0663)
5	→	*	(0.409)
5	→	A	(0.148)
5	→	K	(0.117)
5	→	V	(0.0808)
5	→	G	(0.0678)
5	→	T	(0.0543)
5	→	VL	(0.0484)
5	→	IL	(0.0255)
5	→	AQ	(0.0242)
5	→	PM	(0.0129)
5	→	QL	(0.0125)
1	→	5	(0.705)
1	→	K	(0.216)
1	→	DL	(0.0793)
8	→	H	(1)

F

Start symbol: 0

0	→	S0	(0.296)
0	→	H	(0.222)
0	→	*5A	(0.148)
0	→	A5	(0.141)
0	→	5C	(0.0741)
0	→	E5	(0.0717)
0	→	*5	(0.0467)
5	→	20	(0.867)
5	→	P5	(0.133)
2	→	L	(0.692)
2	→	Q	(0.0774)
2	→	T	(0.0769)
2	→	QT	(0.0769)
2	→	QL	(0.0764)

G

Start symbol: 6

6	→	6*	(0.538)
6	→	6L	(0.173)
6	→	6I	(0.106)
6	→	6V	(0.0914)
6	→	9*	(0.0689)
6	→	6FR	(0.0225)
9	→	VK	(0.333)
9	→	PVN	(0.333)
9	→	PE	(0.333)

H

Start symbol: 6

6	→	85	(0.751)
6	→	P	(0.108)
6	→	8T	(0.0874)
6	→	A	(0.0538)
8	→	655	(0.866)
8	→	899K	(0.134)
5	→	*	(0.423)
5	→	A	(0.225)
5	→	V	(0.114)
5	→	L	(0.0947)
5	→	9	(0.0826)
5	→	D	(0.0349)
5	→	F	(0.026)
9	→	*	(0.526)
9	→	Q	(0.228)
9	→	M	(0.193)
9	→	W	(0.0531)

Table VII.2: Grammars learned from helix data; each grammar has been tuned on its training data using the inside-outside algorithm. Note that digits are nonterminals.

We used each of the grammars from table VII.2 to search the OBSTRUCT database for members of the globin family. The best single matching subsequence of each sequence in the database is reported and the searches are performed as described above for matching a whole grammar to part of a sequence.

To describe the quality of the results, we use the receiver-operator characteristic(ROC) which is the area under the receiver operator curve (RO curve). The area under the RO curve between $x = 0$ and $x = k$ summarizes the ability of the grammar to score globins above other sequences as we increase our tolerance to having nonglobins intermingled with globins in the search results. The RO curve has an x-axis labelled with numbers of false positives(nonglobins from OBSTRUCT in this case) and a y-axis labelled with numbers of true positives. The value of the RO curve at a x tells us how many globins the grammar scored above all but at most x false positives. Thus, a perfect RO curve is the line $y = p$, if p is the number of true positives in the database, so that no false positives are greater than any true positives. The area under the curve is then kp, and ROC values are typically scaled by dividing by this quantity so that the perfect value is 1.0.

Table VII.3 lists the ROC(50) and ROC(14) values for each helix grammar. The table also shows the value of the receiver operator curve 50 and 14 so that the reader can see how many true positives (out of 14 possible in the OBSTRUCT database) were reported within 50 and 14 false positives.

It is somewhat surprising that the variant Ac and Bc did worse than their counterparts without added context. The numbers in table VII.3 seem to point to the versions without context being more reliable than their counterparts with context. Note that despite the low RO values, all of the true positives were always retrieved, but many were buried below more than the first 50 false positives.

For comparison, we show the results of running MEME on the same data sets. MEME was allowed to find one motif that had one occurrence per sequence. The results are reported in table VII.4. Many of these are better than the EvoGrams results, especially in data sets Bc, E, F, and G. There are some exceptions, however,

Data	ROC(50)	RO(50)	ROC(14)	RO(14)
A	0.27	6	0.19	3
Ac	0.10	2	0.01	1
B	0.24	4	0.23	3
Bc	0	0	0	0
E	0	0	0	0
F	0	0	0	0
G	0	0	0	0
H	0.15	3	0.02	1

Table VII.3: ROC and RO values at 50 and 14 false positives for each of the helix data sets' grammars used alone to search the OBSTRUCT database.

Data	ROC(50)	RO(50)	ROC(14)	RO(14)
A	0.21	4	0.05	2
Ac	0.19	4	0	0
B	0.04	1	0	0
Bc	0.05	2	0	0
E	0.13	2	0.09	2
F	0.11	2	0.04	1
G	0.12	2	0.05	2
H	0.15	1	0.02	0

Table VII.4: ROC and RO values at 50 and 14 false positives for each of the helix data sets' MEME motifs used alone to search the OBSTRUCT database.

in helices A, B, and H: on these helices, EvoGrams has better ROC values. All of the motifs had 20 parameters per position to tune. Since all of the motifs were at least 8 positions wide, this implies 160 probabilities per motif. Looking at figure VII.2, we can see that the grammars have far fewer parameters. In fairness, however, not all of MEME's parameters are important to the behavior of the motif. For example, the helix A grammar has 23 probabilities (23 productions) and the helix A motif has 27 probabilities with values that show in the simplified matrix printed by MEME[7]

The individual ROC values are bad in terms of the retrieval ability they

[7]MEME's simplified matrix shows probabilities larger than 0.05.

represent for both algorithms. Each was learned from a very limited amount of information from each globin used in the training data. It is difficult to say much comparative about the results but it does seem to be the case that MEME is more consistent across different helices: this may be due to its use of Dirichlet mixture priors. We would like to use information from more than one helix at a time. We explore three options for doing so:

1. combining information from searches using the individual grammars,
2. combining the grammars and searching using the combined grammar, and
3. learning from more than one helix at a time.

These options combine data across helices at different times, ranging from very late, as when we combine database search results, to very early, as when we form an integrated data set from which to learn.

The latest time to combine data among our three options is to do the database searches with the individual grammars, and then combine the search scores. For the following, we combine search scores across helices by simply summing the scores returned for each sequence according to the different grammar. In cases where a given grammar Γ's search results did not contain a sequence, σ, occurring in some other grammar's search results, we add the lowest returned score of any sequence returned by Γ as Γ's contribution to σ's combined score. This ensures that unreported sequences for a given search are never better than the worst reported sequence, which may have negative log-odds.

We formed two sets of combined scores using the results of the database searches for which we have already reported ROC values. The results are shown in table VII.5. The first combination simply combined across search results for all of the grammars learned on individual helices, including Ac and Bc. This drives both ROC and both RO values to zero. On the theory that part of the problem was the inclusion of the search results reported with ROC values of zero, we modified the combination to contain only results for A, B, and H, and the scores for these were weighted by the

Scores combined	ROC(50)	RO(50)	ROC(14)	RO(14)
All Helices	0	0	0	0
A, B, H	0.31	6	0.22	4
All Helices: MEME	0.07	2	0	0
A, B, H: MEME	0.14	4	0.01	1

Table VII.5: ROC and RO values aggregation over database search results for separate helices

appropriate RO(50) scores: 6, 4, and 3, respectively. This resulted in an increase in the ROC values, as seen in the table. These second row of is higher than the values for any individual helix. Note that the the second value RO(14) has seen a small improvement, but the RO(50) value is no better than helix A alone. Note that using this approach to combining information across helices can never increase the overall number of true positives reported: only sequences present in one of the individual searches being combined will be reported.

The last two rows of the table show the results using the motifs MEME learned from the same data sets. These are both better than EvoGrams was able to do: MEME is sophisticated about combining motifs in a ways that EvoGrams is not in this research. One reason is that MEME uses a simple for of HMM as the basis for each motif, with the advantage that statistics can be computed for expected numbers of occurrences of those motifs in random databases. For an arbitrary context-free grammar, this is not possible. All of these results are worse than MEME can do if allowed to select its own data sets to define its motifs, from entire sequences. This leaves us with a curious suggestion: the well-conserved helices in the globin family are not as informative as other regions, which may overlap helices but will include amino acids between helices.

In an attempt to more intelligently combine information present in separate grammars, we can construct a single large grammar from the individual motif grammars. This was done for grammars A, B, E, F, G, and H from table VII.2. In order to avoid interference between rules in the individual grammars, each had its nonterminals modified to a set of characters distinct from those used in any other grammar. The start symbols were modified to be the lower case letter corresponding to the name of the helix for which the grammar was learned.

This yields a grammar with 103 productions, including all of the productions from the individual helix grammars used and four more productions:

Start symbol: 0

0	$\rightarrow$	zazbzezfzgzhz	(1)
z	$\rightarrow$	*z	(0.5)
z	$\rightarrow$	*	(0.5)

where a, b, e, f, g, and h are the symbols to which the individual helix grammar start symbols were mapped. The productions shown above were added to make a complete grammar with start symbol '0.' The nonterminal 'z' has only two productions and is responsible for deriving sequences of wildcard symbols to cover regions between helices.

The constructed grammar represents a simple form of motif grammar combination analogous to what is done in metaMEME[42]. The search results for this grammar were very poor, with both ROC at zero. Part of the problem seems to be that using this combination method with an arbitrary group of learned grammar creates a grammar which is very ambiguous. As a result, most parses were not completed during database search. Note that when parsing with this grammar, we have no way of limiting the length of a region which matches the grammar for any given single helix. This is something we were able to do, as discussed previously, when doing individual searches with motif grammars.

The earliest we can combine information across helices is to build combined data sets for multiple helices that give a single training instance containing more than one helix from each of our six globins. We did this for two helices with the FG data set, which contains all of the amino acids from each of our six globins between the start of helix F and the end of helix G, inclusive. Each of these training data strings occurred four times, as opposed to twice as was the case in each of the individual helix

Data	ROC(50)	RO(50)	ROC(14)	RO(14)
FG	0.23	3	0.14	4

Table VII.6: ROC and RO values for a database search using the grammar learned from the FG data set.

data sets. We ran EvoGrams for 2000 generations on this data set. The resulting ROC and RO values are shown in table VII.6. The values in the table show that the combined data set did worse than the best individual helix but better than either F or G individually.

In all of the database searches discussed so far, all of the true positives were present in the output. None of the combination methods, however, raised all of the true positive scores enough relative to non-globin sequences for the globins to be clearly distinguished. In the next subsection we will see various methods for trying to analyze the way in which a grammar models a group of sequences, by looking at some of the grammars used in the searches above in detail. The final subsection discusses search results using whole globin proteins as training data.

VII.D.2 Explaining relationships between grammars and sequences

To understand how a learned grammar models a data set, we can begin by examining which productions occur in derivations of which training sequences, and also which positions in sequences make an important difference to the algorithm. We say a sequence "uses" a production of a grammar if the production occurs in at least one derivation of the sequence. The first of these can be easily shown in a table such as table VII.7, which shows for each production in the grammar for data set Ac whether it occurs in at least one derivation of each training sequence. An asterisk means "yes" and a dash mean "no." The columns are labelled with sequence names, and the grammar was learned from data set Ac. The two myoglobins sequences each

	mhu	mch	hhu_A	hhu_B	hch_A	hch_B
0→*8	*	*	*	*	*	*
0→4	*	*	*	*	*	*
4→*0	*	*	*	*	*	*
8→K4	*	*	*	*	*	*
8→D	*	*	*	*	*	-
8→WG8	*	*	*	*	-	*
0→88	*	*	*	-	*	-
0→LS0	*	*	*	-	*	-
8→A	*	*	*	-	-	*
8→AD	*	*	*	-	*	-
0→WQ0	*	*	-	-	-	-
8→H	-	-	*	-	*	-

Table VII.7: Usage of productions by individual sequences in the data set for the Ac run.

use almost all of the rules, as does mhu_A. mhu_B and mch_B, use fewer rules than the other sequences. This gives us some idea of which members of the training data support the existence of which grammar productions.

The above view does not tell us, however, whether a given rule plays an important role or not in accounting for a given sequence. In particular, the grammar productions shown on the right of table VII.7 allow sequences to be derived almost entirely as a string of wildcard symbols. If this happens for a given sequence, σ, then the grammar is doing no better accounting for σ than does the random model, Γ_{Random}, which was introduced in sectionVII.A.4. What we would like to know is which positions in each training sequence are being used or not. We estimate this by displaying strings in figure VII.1 which are made up of the leaves of the *most likely parse tree* for each training data string when parsed by a learned grammar. For a learned grammar $\Gamma_{F,i}$, these are the highest-probability strings which can be derived from the grammar and which agree with the training sequences, in the sense that they exactly match or match using the wildcard symbol at every location.

In figure VII.1, we have shown such strings for four of the data sets: A,

A	Ac
AADK*NVK**FTKI	*LS*ADK***K****K***H
*EEKQ**T*LWGKV	******K******WGK***A
D*EWQ*VLNVWGKV	*LS***WQ*****WGK**AD
D*EWQ*VLTIWGKV	*LS***WQ*****WGK**AD
*ADK*NVKAAWGKV	*LS*ADK***K**WGK***H
*EEK*AVTALWGKV	******K******WGK***D
B	Bc
AEEYGAE*LERMF*	**IA******************P*TK
A*EYGAEALERMF*	************EAL*******P*TK
IAGHGHEVL*RLF*	******IA**************PETL
I*GHGQEVL*RLF*	***EA*****************PETL
VAECGAEAL*RLL*	************EAL*******PWTQ
V*EVGGEAL*RLL*	************EAL*******PWTQ

Figure VII.1: Leaves of the most likely parse trees for the training instances in four of the data sets parsed using the grammar learned each.

Ac, B, and Bc. The prevalence of wildcards in the two sets of strings on the right demonstrates that the most likely accounts of these data sets using the grammars learned from them account for most of each sequence by assuming the amino acids are independently and identically distributed, according to Γ_{Random}. Note that in the figure, each row contains corresponding training instances (training instances in the two data sets which were drawn from the same string). We can now see why the Ac and Bc ROC values were worse than those for A and B: the grammars learned from the Ac and Bc data sets are not "paying attention" to the data as much as their counterparts are paying attention to the A and B data sets.

Plots like those above are useful but they do not show us exactly which nonterminals in the grammar derive which substrings of each training sequence. Consider the helix A data from the A and B strands of human hemoglobins and from chicken myoglobin. If we use the helix A grammar from table VII.2 to parse these three sequences, we get the parses summarized in figure VII.2. Each number in the figure is a nonterminal, and the diagram shows exactly which substrings of the sequence each nonterminal of the grammar for helix A shown in figure VII.2 can derive using the

```
hhu_A, helix A           hhu_B, helix A           mch, helix A
                                                  <-----6------>
                                                   <-----6----->
                                                    <----6----->
                                                     <----6---->
                                                      <---6---->
<-----6------>           <-----6------>                <---6--->
 <-----6----->            <-----6----->                 <--6--->
  <----6----->             <----6----->                <--4--->
   <----6---->              <----6---->                  <--6-->
    <---6---->               <---6---->                 <--9-->
     <---6--->                <---6--->                   <-6-->
      <--6--->                 <--6--->                  <-4-->
       <--6-->                  <--6-->                <-4-->
        <-6-->                   <-6-->                    <-6->
         <-6->                    <-6->                   <-9->
        <-4->                    <-4->                  <-9->
<4-><4->  <6->           <4-><4->  <6->            <4->     <6->
         <9->                     <9->                   <4->
  <4><9><4><6>            <4> <4><4><6>            <4>    <9><6>
 <9>      <1>             <9> <9>  <1>                  <4> <1>
 3_9_4_  9_ 6_             9_  9_ 9_ 6_              9_ 4_ 3_  6_
     3_                    1_ 4_                           9_
44444444444446           44444444444446           44444444444446
   1  91 31 1             331  9  31 1              31  9   1 1
*************V           *************V           *************V
 ADKTNVKAAWGK             EEK AVTALWGK            D EWQ VLTIWGK
PADKTNVKAAWGKV           PEEKSAVTALWGKV           DQEWQQVLTIWGKV
```

Figure VII.2: Chart parses of helix A from human hemoglobin chains A and B and chicken myoglobin using the helix A grammar.

rules of A. The last row gives the full training instance in each case for reference. The left and center diagrams are very similar, which is to be expected for data instances drawn from such strongly related sequences. The right diagram is less closely related to the other two and its diagram shows obvious differences on a visual inspection, apparent among the entries of lengths about four through eight for nonterminals 4, 9, and 6.

A quantitative measure of such similarity could form the basis for an alternate means of scoring similarity between database sequences and grammars: we use the learned grammar to form such diagrams for all of the training data and form a composite of them; then to score a sequence from a database, we parse it using the same learned grammar and compare its chart to the composite. This would represent an attempt to score sequences based on syntactic similarity.

However we score sequences during a database search, we will often be interested in how our grammatical model scores them relative to one another. Dendrograms give us a useful means of visually how sequences returned as the result of a database search cluster. The dendrograms in figure VII.3 show simple agglomerative clustering of database search scores for the sequences used to train the grammar. That is, a database was made of the training data, the grammar learned on that data was used to search the database, and the resulting scores were run through a clustering algorithm that draws Dendrograms[8].

Thus, we have a picture of how a grammar learned on some training data clusters its own data, at least according to the log-odds scores it assigns; this is of interest because these log-odds scores will be the same as the fitness contributions for each training string unless partial credit was involved. The dendrograms groups and subgroup the data. The myoglobins are recognized as similar in each case, and the separation among the hemoglobins is more questionable, though slightly better in the Ac case.

The above discussion demonstrated four different ways to go about analyzing

[8]SYSTAT was used to do the clustering and dendrogram drawing.

```
Dendrogram for data set A:

mhu_   ------
            +----
mch_   ------   |
                +----------
hhu_A  ----------         |
                          +----
hhu_B  --------------     |
                    +-    |
hch_A  --------------|    |
                     +-----
hch_B  ---------------
```

```
Dendrogram for data set Ac:

mch_   --
        +----------
mhu_   --         |
                  +--------
hhu_A  ------------       |
                          +---
hch_A  --------           |
              +------------
hhu_B  ----   |
          +----
hch_B  ----
```

Figure VII.3: Dendrograms for based scoring of training data with a learned grammar.

a grammar as a model of some training data. The second form of these is the easiest to create and easiest to interpret at a glance. The first and third methods represent attempts to specify which rules or nonterminals are important to accounting for each member of the training data. Quantitative measures of the qualitative features we discussed above would make an interesting basis for further attempts to cluster the data as demonstrated with the dendrogram output above.

VII.D.3 Learning from whole proteins

Our final round of experiments used a data consisting of the entire sequences of the 14 globins found in the OBSTRUCT database. We ran EvoGrams for 4960 generations[9] on these training instances, which occurred once each in the training data in this case. The population contained 100 grammars. Note that the data ranged in length from 136 to 159 amino acids, and that parsing these sequences was expensive. For this reason, we ran the algorithm on the San Diego Supercomputer Center's Cray T3E parallel computer. The run used up roughly the equivalent of one month of CPU time for a high-end desktop workstation. The run of EvoGrams just

[9]We had scheduled 5000 generations; the job ran out of time before generation 5000

mentioned did not employ the complexity measure to modify fitness. We will also summarize some results for another run which did use the complexity measure, but which only ran 2000 generations.

The best grammar from the final generation of this run contained 53 productions and is:

Start symbol: c

c	$\rightarrow$	h	(0.788)
c	$\rightarrow$	98	(0.212)
h	$\rightarrow$	8	(0.735)
h	$\rightarrow$	*8H	(0.173)
h	$\rightarrow$	V8	(0.092)
9	$\rightarrow$	6*	(0.555)
9	$\rightarrow$	A7	(0.31)
9	$\rightarrow$	7D	(0.0733)
9	$\rightarrow$	L4	(0.0621)
8	$\rightarrow$	8*	(0.96)
8	$\rightarrow$	8j	(0.0161)
8	$\rightarrow$	8f*	(0.00541)
8	$\rightarrow$	80	(0.00393)
8	$\rightarrow$	cFS	(0.00266)
8	$\rightarrow$	9Q	(0.00254)
8	$\rightarrow$	LS8G	(0.00209)
8	$\rightarrow$	9*	(0.00188)
8	$\rightarrow$	9D	(0.00148)
8	$\rightarrow$	8a	(0.00131)
8	$\rightarrow$	G8	(0.00104)
8	$\rightarrow$	K*A	(0.000528)
8	$\rightarrow$	S8dS	(0.000528)
8	$\rightarrow$	dD	(0.000528)
8	$\rightarrow$	7D	(0.000433)
6	$\rightarrow$	*L	(0.664)
6	$\rightarrow$	b	(0.224)
6	$\rightarrow$	bGG	(0.112)
7	$\rightarrow$	A	(0.829)
7	$\rightarrow$	SA	(0.108)
7	$\rightarrow$	AYb	(0.0327)
7	$\rightarrow$	SP9	(0.0304)
4	$\rightarrow$	AI	(1)
j	$\rightarrow$	7D	(0.378)
j	$\rightarrow$	7d	(0.186)
j	$\rightarrow$	DP	(0.168)
j	$\rightarrow$	RM	(0.0741)
j	$\rightarrow$	NP	(0.0569)
j	$\rightarrow$	H0	(0.052)
j	$\rightarrow$	DIL0	(0.0326)
j	$\rightarrow$	Cd	(0.0274)
j	$\rightarrow$	DIL	(0.0243)
f	$\rightarrow$	LS	(1)
0	$\rightarrow$	P2	(0.638)
0	$\rightarrow$	DCI	(0.0928)
0	$\rightarrow$	PNI	(0.0911)
0	$\rightarrow$	FRN	(0.0909)
0	$\rightarrow$	RHK	(0.0874)
a	$\rightarrow$	DAd	(1)
d	$\rightarrow$	W*	(0.672)
d	$\rightarrow$	7d	(0.328)
b	$\rightarrow$	C	(0.504)
b	$\rightarrow$	P*	(0.496)
2	$\rightarrow$	E	(1)

Digits and lowercase letters in left- and right-hand sides of rules are nonterminals. This grammar can be used to search the OBSTRUCT database as discussed previously regarding using a grammar to score complete sequences. We allow partial matches to the grammar during our search via the pessimistic completion operation

Database searched	Complexity Used?	Generation	ROC(50)	ROC(14)
OBSTRUCT	No	4960	0.901	0.832
	No	2000	0.543	0.337
	Yes	2000	0.213	0.031
PDB	No	4960	0.360	0.442
PDB at PAM60	No	4960	0.139	0.360

Table VII.8: ROC values from searches using grammars learned on whole sequences.

described earlier as well. An interesting feature of this grammar is that it contains a closed, mutually recursive pair of nonterminals: the symbols '7' and '9' derive each other in at least one production each, and otherwise generate only nonterminals that nonrecursively generate terminal sequences. Further, nonterminal '0' has four alternatives, all of which derive a fixed short string.

The ROC numbers are summarized in table VII.8. The table data show that the grammar evolved without complexity constraints did better than the grammar evolved with complexity constraints. We show the best grammar from the non-complexity run at the generation at which the with-complexity run was stopped. The data show that even at this generation the ROC values are much better when complexity is not counted in fitness.

It should be noted, however, that the grammar learned under pressure to reduce complexity is much simpler than the above grammar:

Start symbol: 4

4	$\rightarrow$	d*	(1)
d	$\rightarrow$	4	(0.991)
d	$\rightarrow$	L	(0.00314)
d	$\rightarrow$	GdW	(0.00189)
d	$\rightarrow$	V	(0.00189)
d	$\rightarrow$	C	(0.00126)
d	$\rightarrow$	P	(0.000629)
d	$\rightarrow$	S4	(0.000629)

This grammar contains only eight productions and operates by deriving wildcard symbols over much of the sequence, using the other productions rarely to account for amino acids common enough to warrant a separate production, though in one case there is a genuinely context-free production: d $\rightarrow$ GdW (0.00189).

For the individual helix grammars, we showed a map of production usage by each sequence. Table VII.9 does the same for the larger grammar shown above. The columns are labelled by numbers from the following list, which identifies the sequences by the names provided in the PDB index file:

1 MYOGLOBIN (MET) (PH 7.0)

2 MYOGLOBIN (MET)

3 MYOGLOBIN (HORSE HEART) MUTANT WITH HIS 64 REPLACED BY TYR (H64Y)

4 HEMOGLOBIN A (HORSE,DEOXY)

5 HEMOGLOBIN A (CYANOMET)

6 HEMOGLOBIN B (HORSE,DEOXY)

7 HEMOGLOBIN B (DEOXY)

8 HEMOGLOBIN B; HETEROGEN: CARBON MONOXIDE

9 LEGHEMOGLOBIN (CARBON MONOXY)

10 HEMOGLOBIN (CARBON MONOXY)

11 HEMOGLOBIN (SEA CUCUMBER)

12 HEMOGLOBIN I (MONOMERIC) (FERRIC)

13 HEMOGLOBIN (ERYTHROCRUORIN, AQUO MET)

14 HEMOGLOBIN V (CYANO,MET)

The table shows a core of 9 productions at the top, each used by at least 12 of the sequences in at least one parse. As we move down toward the bottom of the table, the productions we find are used more and more sparsely until we arrive at productions only helpful in accounting for one or two sequences. Most of the productions near the bottom of the table are non-recursive: they derive finite sets of usually short amino acid sequences. By contrast, an infinite number of sequences can be derived from nonterminal '8' and 'c,' for example.

The experiments on whole sequences resulted in a grammar that does a good job of retrieving the globins from the OBSTRUCT database. Since the training set was chosen to be the globins from the OBSTRUCT database, this only tests the ability of the grammar to retrieve its training sequences from the reduced database from which they were taken. The last two rows of table VII.8 show ROC values for searches in the entire PDB, a test of the ability of our grammar to generalize to homologous sequences not included among the globins in the OBSTRUCT database. The performance here is mediocre and the use of the amino acid similarity matrices at a PAM distance of 60 actually decreased the ROC numbers. This suggests that our method for applying the similarity metric is flawed. One possible culprit is our allowing only amino acids with positive log-odds to be substituted according to a mutation probability. This may mean we are only permitting the similarity metric to increase true positive scores and hampering its ability to affect false positive scores.

Production	1	2	3	4	5	6	7	8	9	10	11	12	13	14
8→8*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
8→8j	*	*	*	*	*	*	*	*	*	*	*	*	*	*
c→h	*	*	*	*	*	*	*	*	*	*	*	*	*	*
7→A	*	*	*	*	*	-	*	*	*	*	*	*	*	*
h→8	*	*	*	*	*	-	*	*	*	*	*	*	*	*
8→9*	*	*	*	*	*	*	-	-	*	*	*	*	*	*
d→W*	*	*	-	*	*	-	*	*	*	*	*	*	*	*
j→7D	*	*	*	*	*	-	*	*	*	*	*	-	*	*
8→80	-	*	*	-	-	*	*	-	*	*	*	*	*	*
9→6*	*	-	*	-	*	*	-	-	*	*	*	*	-	*
j→7d	*	-	-	*	*	-	-	-	*	*	*	*	*	*
6→*L	*	-	*	-	*	*	-	-	*	*	*	*	-	-
8→8f*	*	-	-	*	-	*	*	*	-	*	-	*	-	*
8→cFS	-	*	-	-	*	-	*	-	*	*	*	*	*	-
f→LS	*	-	-	*	-	*	*	*	-	*	-	*	-	*
j→DP	-	-	-	*	*	*	*	-	-	*	*	-	*	*
0→P2	-	*	*	-	-	*	-	-	*	-	*	*	-	*
2→E	-	*	*	-	-	*	-	-	*	-	*	*	-	*
d→7d	*	-	-	*	*	-	-	*	-	-	-	*	*	-
7→SA	-	-	-	-	*	-	-	*	*	-	-	*	-	*
8→9Q	-	-	-	-	*	-	-	-	-	*	*	*	*	-
9→A7	*	-	-	*	*	-	-	-	-	*	-	*	-	-
c→98	*	-	-	-	*	-	-	-	-	*	*	*	-	-
8→LS8G	*	-	-	*	-	-	-	-	-	*	-	-	*	-
6→b	*	-	-	-	-	-	-	-	-	-	*	-	-	*
8→8a	*	-	-	-	-	-	-	-	-	*	*	-	-	-
8→9D	-	-	*	*	-	-	-	-	-	-	-	-	-	*

Table VII.9: A map showing production usage for each of the sequences used to train the grammar learned on whole sequences.

8→G8	-	-	-	-	-	-	-	-	*	*	*	-	-	-
a→DAd	*	-	-	-	-	-	-	-	-	*	*	-	-	-
h→V8	-	-	-	*	-	*	*	-	-	-	-	-	-	-
j→NP	-	-	-	-	*	*	-	-	*	-	-	-	-	-
8→7D	-	*	-	-	-	-	-	-	-	-	-	-	*	-
9→7D	-	*	-	-	-	-	-	-	-	-	-	-	*	-
b→C	*	-	-	-	-	-	-	-	-	-	*	-	-	-
b→P*	-	-	-	-	*	-	-	-	-	-	-	-	-	*
h→*8H	-	-	-	-	-	*	*	-	-	-	-	-	-	-
j→DIL	-	-	-	-	-	-	-	-	-	-	*	-	-	*
j→H0	-	*	*	-	-	-	-	-	-	-	-	-	-	-
j→RM	-	-	-	*	-	-	-	*	-	-	-	-	-	-
0→DCI	-	-	-	-	-	-	*	-	-	-	-	-	-	-
0→FRN	-	-	*	-	-	-	-	-	-	-	-	-	-	-
0→PNI	-	-	-	-	-	-	-	-	-	-	-	-	*	-
0→RHK	-	-	-	-	-	-	-	-	-	*	-	-	-	-
4→AI	-	-	-	-	-	-	-	-	-	-	*	-	-	-
6→bGG	-	-	-	-	-	-	-	-	-	-	*	-	-	-
7→AYb	-	-	-	-	*	-	-	-	-	-	-	-	-	-
7→SP9	*	-	-	-	-	-	-	-	-	-	-	-	-	-
8→K*A	-	-	-	-	-	-	-	*	-	-	-	-	-	-
8→S8dS	-	-	-	-	-	-	-	*	-	-	-	-	-	-
8→dD	-	-	-	-	-	-	*	-	-	-	-	-	-	-
9→L4	-	-	-	-	-	-	-	-	-	-	*	-	-	-
j→Cd	-	*	-	-	-	-	-	-	-	-	-	-	-	-
j→DIL0	-	-	-	-	-	-	-	-	-	-	*	-	-	-

Table VII.9: (continued)

Chapter VIII

Conclusion

Chapter II presents the evolution of grammars as a special case of the general problem of evolving behaving things. Two ways of evaluating grammars are presented in section II.C, and both are argued to suggest a metaphor between derivations from grammars and ontogenesis. Table II.1 presents a space of options for evaluating grammar fitness; the lower-left and upper-right cells of the table describe the two EAs in chapters IV and VI. These two EAs differ in their fitness evaluation functions for a grammar Γ: one derives strings from Γ and evaluates Γ based on the strings. The other parses strings using Γ and evaluates Γ based on the parses.

Parsing in order to perform fitness evaluation leads to an EA, EvoGrams, that performs grammar induction: given a set of strings, EvoGrams searches for a grammar which assigns high probability to those strings. The basic fitness function in EvoGrams is based on this idea of maximizing the probability of the training data given a grammar, and in some cases is modified by a penalty imposed on grammars based on their complexity, as defined in section IV.C.1. The complexity measure is meant to provide pressure to evolve simpler grammars with fewer productions and to alleviate the difficulties of overfitting the training data. In the case of the EA to find sorting algorithms, the fitness function is by contrast a problem-specific measure evaluated on derived strings describing merging networks.

VIII.A Recapitulation of Results

The results divide into three sets, in chapters V, VI, and VII. The sorting networks results are discussed in chapter VI. This is the sole case in which we evaluate fitness by deriving strings from grammars and testing these strings. In this case, the strings represent simple merging algorithms. The grammars in this case are biased using expert advice so that the evolved grammars are more likely than would have usually been the case to create log-sequential sorters. A previously-unknown log-sequential sorter is found which is based on a novel combination of features from two known log-sequential sorters.

Chapter V presents tests of EvoGrams for grammar induction, including a comparison with some other grammar induction systems. EvoGrams found the target languages faster or with fewer assumptions than the other EA approaches. The comparisons suffer, however, from the lack of thorough reporting of results in the grammar induction literature. The results do demonstrate, however, that EvoGrams is able to successfully induce grammars for the test problems: the EA is able to use the representation presented in chapter IV of grammars as variable length strings, with coding and noncoding regions, to induce stochastic grammars equivalent to those that generated the data sets.

Using two simple languages, section V.C addresses the importance of availability of negative examples in the training data and in the set of all strings. It is found there that negative examples and a stochastic sample each improve the ability to learn a grammar for the parity language but that the effects of the two are statistically distinguishable. This provides evidence that sample statistics in a stochastic sample *can* substitute for the information that negative examples provide, a question of interest in grammar induction. Also, to determine whether EvoGrams finds reasonable generalizations of languages which its grammars would never correctly describe, an attempt is made in section V.E to evolve a context-free grammar to describe data from a context-sensitive grammar's language. It is found that both with and without the complexity measure in use as part of the fitness function, EvoGrams find reasonable generalizations from the context-sensitive language. EvoGrams found grammars which captured as much of the definition of the target as was possible given its restriction to context-free grammars, as opposed to finding a grammar for an even simpler language.

Other results from this chapter include the observation that EvoGrams is able to maintain a good solution in its population once that solution exists, and that the operators interact in their effects on EvoGrams's ability to do so. The latter point is of particular interest in genetic algorithms since it provides an observed operator interaction, suggesting that analyzing operators' effects separately is misleading. When the genotype includes probabilities, the ability of the grammars evolved to be improved by the inside-outside algorithm is noted to depend on the use of that algorithm during evolution. Finally, an investigation EvoGrams's use of the wildcard symbol reveals the tendency of EvoGrams to evolve grammars which use the wildcard symbol when trying to learn from small training sets: this suggests that the wildcard symbol should be used with caution, since its use may inhibit the EA's ability to induce structurally correct grammars in the sense of section III.D.

Finally, chapter VII addresses the applications of EvoGrams to the problem of evolving grammars to describe protein sequence data. The application of grammars to describing such data is discussed, including the use of grammars to perform database searches. EvoGrams was used to evolve grammars for individual helices and for whole globins, and various methods for examining the results are presented. The ROC numbers reported are better than MEME on identical data in a few cases, but on the whole MEME proves the better algorithm for homology detection, especially where combining information across motifs for individual helices was concerned. EvoGrams clearly will require further to work if it is to eventually prove useful as a sequence analysis tool.

VIII.B Parsing versus Deriving

The two grammar induction systems described in chapters IV and VI are both successful, though EvoGrams's success is on formal problems usually used in the grammar induction literature and not yet on the more realistic problem of analyzing sequence data. For EvoGrams, the target languages are well-defined stochastic languages. In the work on sorting networks, however, the target language is implicit: it is the language of algorithms which pass the test as correct merging algorithms. The application of grammar induction to protein data falls somewhere in between: we assume that some stochastic grammar will provide us with a good model of our data as a prerequisite to trying to find such a grammar, and yet we are aware that the data was not generated by a grammar as such. Section VIII.C.1 discusses the possibility of evaluating grammars to model protein data by deriving strings from them instead of by parsing.

A further linkage between the two EAs is provided by the complexity measure and the question of overfitting to a data set. In the case of the sorting networks grammars, the grammars are restricted so that they find simple recursive definitions. The lack of such a restriction in EvoGrams leads it to discover grammars which are not in any particular standard form, and which differ markedly for different simulations on the same data. The complexity measure in EvoGrams's fitness function was introduced primarily for two reasons: to simplify the evolved grammars and to standardize the grammars learned to some extent. It was hoped that the standardization would occur as a by-product of simplification. This was effective with the palindrome language and with the $a^n b^n c^n$ language, but the effect of the scaling of the complexity measure is difficult to determine. The scaling factor β from chapter V leads to the modification of the genetic operator rates to the alternate set in table V.1, and the scaling's effect on the results of chapter VII is not fully understood. Even in the case of the $a^n b^n c^n$ language, different runs find different grammars, though only four distinct grammars are found by the runs in section V.E.

An important result above in section V.C concerns the ability to use a stochastic sample instead of including negative examples. This is not an issue in the sorting networks work, since the evaluation of networks for correct merging gives a figure which essentially summarizes the extent to which they are correct as merging algorithms. This quantity seems to play the same role for that algorithm that $\Pr(\sigma|\Gamma)$ plays in EvoGrams. In both cases, there is a case corresponding to negative examples, a network that never correctly merges and a string with probability zero.

The partial credit mechanism in EvoGrams allows the assignment of reasonable scores to grammars even when they allow no complete derivations of strings in the training data. This is important especially early in run because many early grammars will only derive short substrings of the training data, and these must be differentiated from grammars which derive nothing at all in order for any progress to be made. The corresponding notion in the evolution of sorting networks in the repair mechanism, which acts during a derivation. The repair mechanism acts top-down to correct minor inconsistencies in the grammar during a derivation and to complete derivations for nonterminals which are not on the left-hand side of any rule or which can't be replaced because of time constraints on the derivation. Partial credit's greedy covers and pessimistic completions fulfill a similar goal for the bottom-up case of parsing: the parse tree is completed above the fragments we can arrive at by attempting to parse a string with a grammar. Thus, the partial credit of EvoGrams can be seen as a "repair" mechanism for incomplete parses.

VIII.C Future work

The above discussion concerns the two evolutionary algorithms used to arrive at the results of this thesis and how they relate to one another. This final section discusses possible avenues for further research.

VIII.C.1 "Derivational" EvoGrams

The discussion above tried to relate fitness based on evaluation of derivations to fitness based on evaluation of parses. EvoGrams is an example of the latter, but using a parser is costly: the worst case parsing cost for context-free grammars is proportional to the cube of the length of the string parsed. Worse, it proved difficult to integrate a similarity metric over terminals into the parser without increasing the cost of parsing dramatically. The advantage of the parser is that it allows working in terms of true probabilities and provides the information which forms the basis for the inside-outside algorithm. This subsection discusses a way in which EvoGrams might be modified to evaluate derivations instead of using a parser. This possibility is important and of interest for its potential time savings, for the way in which it draws together the two styles of fitness evaluation used here, and because it presents a potential novel use of sequence analysis techniques from computational biology.

Recall equation II.1, which defines the probability, p_σ, of a string σ given a grammar in terms of its frequency as a result of a derivations from the grammar:

$$p_\sigma = \lim_{N_D \to \infty} \frac{c_\sigma}{N_D}.$$

Here c_σ is the number of derivations resulting in σ, and N_D is the number of derivations performed. If we were willing to perform large numbers of derivations, we could use p_σ as the basis for fitness evaluation where we previously used $\Pr(\sigma|\Gamma)$. σ would be one of the training data strings in this case; the needed value of N_D is probably very high for many grammars that would arise, if the target grammar is ambiguous or the training data contains many long strings.

We could try to alleviate this problem by influencing the length of strings derived or the kinds of derivations preferred, but this is likely to influence p_σ's usefulness as a probability estimate. Rather, imagine instead that our probability surrogate is the result of a more general similarity estimate between training data and strings derived from the grammar. In particular, assume that $\mathrm{sim}(\sigma_1, \sigma_2)$ is a similarity metric over strings which is scaled to take values in the unit interval. Then we can define our heuristic probability replacement for training data string σ as:

$$\mathrm{HPr}(\sigma|\Gamma) = \frac{1}{N_D}\sum_{i=1}^{N_D} \mathrm{sim}(\sigma, \delta_i) \tag{VIII.1}$$

where δ_i is the i'th string derived from Γ. If $\mathrm{sim}(\cdot,\cdot)$ is defined as 0 when its arguments do not exactly match, and $N_D \cdot \mathrm{fcontrib}(\Gamma, \sigma)$ when they do match, this collapses into the original EvoGrams algorithm, where parsing is used to determine the probability $\Pr(\sigma|\Gamma)$ used in $\mathrm{fcontrib}(\cdot,\cdot)$, defined in section IV.C.

Suppose we derive a string σ of length n from grammar Γ. In many cases, σ's derivation will use no more than about $n-1$ rules. This will always be true if Γ contains no ϵ-productions and no renaming productions, of the form $N_1 \rightarrow N_2\ (\rho)$ for N_1 and N_2 nonterminals. This is because the derivation tree will have a branching factor of at least two at every node and will be no deeper than a binary tree with the symbols of σ at its leaves. We can even allow $O(n)$ renaming or epsilon productions in derivations and still achieve an asymptotically linear time in n, the length of the derived string, for the derivation.

If we use a similarity function with known time complexity $f(n)$ depending on the maximum string length of the arguments to $\mathrm{sim}(\cdot,\cdot)$ in equation VIII.1, we can compute the cost of evaluating a grammar with heuristic probabilities. Assume that we will do N_D derivations once per grammar and use the same strings to evaluate $\mathrm{HPr}(\cdot,\cdot)$ for all of the training data. Then the cost to evaluate Γ is the cost of doing N_D derivations plus the cost to evaluate sim on $|E|$ times N_D[1]. This gives a total cost of:

$$nN_D + f(n)N_D|E|$$

As long as $f(n)$ is at least linear in n, this expression grows asymptotically as $f(n)$. As long as $f(n)$ is sub-cubic we will do better asymptotically than parsing in terms of evaluation costs. We can add a similarity metric over individual terminal symbols at no additional cost, asymptotically, as long as determining similarity for a given pair

[1] Recall from chapter III that if the training data is E, $|E|$ denotes the number of distinct strings in E.

of terminals is a constant-time operation. The integration of PAM information into this framework, if it were used to evolve grammars for protein sequences, would be relatively inexpensive compared with its impact in the existing version of EvoGrams.

An important part of the work presented above is the inside-outside algorithm. Recall from section B..2 that this algorithm uses probability information computed by the chart parser to arrive at counts for each production in a grammar Γ. Each production receives a count of 1, prorated by the probability of the parse in which it occurs, and the counts are normalized into new probabilities each cycle; in the case of an unambiguous grammar, the algorithm does the equivalent of counting production usages and normalizing them to get new probabilities.

As long as our grammars are unambiguous, we can estimate the inside-outside algorithm by literally counting production usages in derivations and updating based on these. Moreover, we only have to compute these counts once at the beginning of the algorithm. If the grammars are ambiguous, we can scale the influence of each production usage heuristically using the probabilities of the productions in the derivation. This will not give exactly the same value as the inside-outside algorithm, in which the scaling combines information from all parses, but would probably produce reasonable results under assumptions of bounded degree of ambiguity.

Thus, the use of a heuristic replacement for string probabilities leads us to an algorithm with lower cost and which creates the obligation to come up with a reasonable estimate of similarity between strings. This similarity metric, $\mathrm{sim}(\cdot, \cdot)$ would need to be carefully selected to give well-motivated values across strings of differing lengths, and should peak when the strings are identical. We have already mentioned, in chapter VII, a potential definition for sim: BLAST scores. These scores are cheaply computable and take into account differences in string lengths by introducing gaps with associated penalties. The scoring method would have to be scaled and generalized for an arbitrary alphabet. If this were done, equation VIII.1 becomes the averaged similarity used in family-pairwise search[41]. More generally, a score based on or estimating edit distances between sequences using some fixed set of edit operations would be appropriate, and the affects of the choice of operations on the learned grammars would be an interesting subject for further investigation.

VIII.C.2 Improving Performance on Protein Sequence Data

The previous section suggests how we could modify EvoGrams to eliminate the parser. Here we concentrate on changes which would be targeted specifically at improving the results from chapter VII. Eliminating the parser would already have some benefits in this regard, including lower cost over all and in applying PAM probabilities. This section concentrates on other possible changes.

Systems such as MEME and HMMer can be seen as performing a heavily biased form of grammar induction in which the a predetermined grammar has its probabilities tuned using a training set. In MEME's case, this data is selected from whole sequences using an algorithm which locally aligns the whole sequences and uses the local alignment to choose data sets. An interesting exercise would be to use classic techniques from grammar induction, such as those discussed in chapter III, to induce grammars for protein data and compare them to evolved grammars. This would contribute to our understanding of grammar induction as it applies to protein sequence data.

An important reason for MEME's restriction on the form of its models is that the motifs it finds can are analytically tractable: for example, the probability of a chance match to such a motif is evaluable. This is not the case for EvoGrams's unrestricted grammars at present. There are several possible ways in which we might restrict the genotypic representation for grammars to improve performance or analytic tractability. For example, the monotonic context-free grammars have on nonterminal at most on the right-hand side of each rule, and allow linear-time parsing. Since they allow productions such as $N_1 \rightarrow t_1 N_2 t_2$ (ρ), they can capture dependencies between nonadjacent amino acids. Another restriction would be to disallow grammars deriving infinite sets of strings, preventing some very high-cost grammars and making the models more tractable to analysis at the expense of expressive power. The tractability to analysis derives from the absence of nonterminals which derive themselves: $N \Rightarrow \delta N \gamma$, where δ and γ are possibly empty strings of terminals and nonterminals, is never possible in such "definite" grammars. A consequence is that any given rule is used at most once along a path from the root to a leaf of a parse tree induced by the grammar.

In the results for the protein data, we noted that PAM probabilities seemed to actually decrease retrieval and carried a high cost. Dr. Baker has suggested in the past that the main helpfulness of using such a similarity metric is more in the depression of scores for false positives than for the inflation of scores for true positives. We limited our algorithm, however, to the use of mutation probabilities only for pairs of amino acids with positive log-odds: other amino acid pairs have their mutation probabilities truncated to 0. Marginalizing the mutation probabilities in this manner eliminates the information about the overall probability of the set of amino acids with positive log-odds with respect to another amino acid, and this loss of information inflates the mutation probabilities differently for mutation from different amino acids.

Finally, we noted the MAP and MD figures of merit for HMMer in equations VII.1 and VII.1. The latter was developed to help incorporate negative instances into HMMer, to perform a task more like the "separating grammar" discussed in section III.B. In EvoGrams, the fcontrib function represents our attempt to deal with negative examples. An interesting variant of EvoGrams would use the MD discrimination metric instead of the currently-used MAP metric. This would necessitate modifying the inside-outside algorithm to optimize a SCFG for MD instead of MAP, which could prove quite difficult. The relationship between EvoGrams's method for incorporating negative examples and the MD metric should in any event inform the use of EvoGrams as a grammar induction system.

VIII.C.3 Algorithmic improvements and EA-specific investigations

This section discusses various other possible extensions of the work of this thesis not specifically directed at either of the major extensions of the previous two sections. Among these are some basic improvements in the software: a more efficient parser would allow the algorithm to be run on training data containing longer strings. A possibility for a new, more efficient parser is Earley's algorithm, which is guaranteed to be linear on all LR(k) grammars, if it is implemented for lookahead k. The current parser can be generalized to allow ϵ-productions, making the representable set of grammars more complete with respect to the context-free grammars. Also, the number of productions needed to describe some languages would decrease. Also, EvoGrams represents probabilities directly and takes their logarithms in fitness evaluation. The precision of floating-point numbers on most computers makes it necessary to compute in terms of logarithms of probabilities directly if long protein sequences, for examples, are to be analyzed.

The inside-outside algorithm is currently not usable unless a grammar completely parses at least some of the training data. We could, however, use the idea of greedy cover and pessimistic completion to complete parses to allow training over strings for which no complete derivations are possible. This would at least increase fitness by resetting to zero the probabilities of productions that don't contribute to partial credit. The impact of this change on the learning algorithm as a whole, however, would have to be investigated carefully to be sure that the inside-outside algorithm still locally optimized fitness.

The ANOVA experiments in chapter V investigated only one side of a question: for what operator rate settings will a known, good solution be maintained. The complementary question is: "for what operator rate settings will new improved grammars tend continue to be discovered?" Experiments designed to explore both issues are important to deciding on a set of parameter settings likely to lead to good performance by EvoGrams.

Finally, we earlier saw a case in which work from computational biology could be used as part of a grammar induction system: the use of BLAST scores to help estimate probabilities. In general, there exist a wide corpus of genome analysis methods which could be helpful in analyzing the final population of an EvoGrams run and lead us to a better understanding of how grammars evolve. For example, the representation for EvoGrams allows intron and variable-length coding regions. Sequence alignment scores could help us decide when the population had converged by allowing the detection of similarity between genotypes in the population.

Appendix A

A Brief Introduction to the Analysis of Variance

The statistical analysis undertaken in section V.F as an "analysis of variance" (ANOVA). The presentation in that section follows the notation introduced here. ANOVA is used to determine whether each of several factors influences a measurable quantity. Each factor is a parameter or condition which is varied across measurements. ANOVA is used either when there are more than two factors to be considered or when a lone factor has more than two values. If a single factor with two values is being considered, simpler statistical procedures for testing the differences between two means, such as t-tests, apply. The basic results from an ANOVA tell us whether each combination of factors has a statistically significant influence on the measurable quantity. Further tests must be used to determine the nature of the influence.

A.A Designs and Assumptions

The experimental design associated with an ANOVA is determined by the nature of the factors involved, how the values for these factors are chosen, and the combinations of factor values for which observations of the measurable quantity exist.

In the work presented here, we will use a simple fixed-effects factorial design. In this case, all of the factors are used to divide the data into groups, as opposed to subdividing groups established by other factors for example, and the values of the factors are chosen by the researcher rather than being determined randomly. That the design is *factorial* means that there is at least one observation for every combination of factor values. An important consequence of using fixed rather than random factors is that we can only draw inferences to the factor values for which measurements were taken; if the factor values are randomly chosen, we have a random-effects model, which allows inference to factor values other than those for which measurements were taken. There are a variety of other designs: in particular, some are meant to minimize the number of observations needed to detect significant results and some accommodate the use of random factors while arranging for factor values to span some range[68, 23]. Here, the simple factorial design sufficed.

As with all statistical procedures, the analysis of variance is founded on certain assumptions about the observations which it used to explain. In particular, there are three primary assumptions which must be reasonable if an ANOVA is to be deemed to be trustworthy: the groups must be drawn from normal populations, these populations must all have the same variance, and observations must be independent. The second assumption, the "homogeneity of variance" is by far the most important of these assumptions. Even it is not critical if all of the groups are the same size, as they are in the analysis undertaken below. In general, ANOVA is held to be robust under minor violations of these assumptions.

A special feature of the analysis we will undertake below is that the third assumption for ANOVA is violated: observations are not independent. This is true because our analysis uses a design involving "repeated measures." In repeated measures ANOVA, the same "subjects" are measured under different values of the factors. This lowers the variance of observations compared to the case where some number of random samples are taken for each setting of factor values. In our case, the "subjects" will be initial populations; more generally, a repeated measures ANOVA for an EA could run the EA for the same set of random seeds for different values of the factors.

Instead of independence of observations, repeated measures ANOVA requires the assumption that effects on measurements owing to differences across subjects and effects due to changing factor values do not interact. This means that changes in the values of the factors effect the different subjects similarly. Repeated measures ANOVA is also robust in the face of minor violations of its assumptions. In fact, the criterion stated to replace independence of observations is more than is needed; a weaker condition known as a "sphericity condition" on the observations suffices[68]. We do not investigate this condition here, since we believe the stronger assumption to be reasonable in our case.

A.B Notation and basic terminology

The above overview of ANOVA's purposes and assumptions was kept informal. Here we will briefly develop a more formal notation for the description of the ANOVA procedure which follows. This exposition is intended to suffice for the reader to understand the results reported in chapter VII. It will be necessary to refer to a statistics text, especially regarding experimental design, before attempting such an analysis.

Let n denote the number of factors in an ANOVA, then we will denote the set of factors by:

$$F = \{f_1, f_2, \ldots, f_n\} \tag{A.1}$$

It will be useful below to assume an ordering over factors that reflects any arbitrary indexing of factors in F: $f_i < f_j$ if and only if $i < j$. Each factor f_i has a associated with it a finite set of values:

$$L_i = \{l_{i1}, l_{i2}, \ldots, l_{in_i}\} \tag{A.2}$$

L_i is the set of values of f_i for which at least one measurement exists. The n_i distinct values of f_i are referred to as the *levels* of factor i. Since we are considering only fixed factors, the values in L_i for every f_i are determined by the experimenter and not by random choice.

An observation, o, is a measurement of some quantity under conditions defined by specifying a level for every factor. Let M be the domain of the measurable quantity and O be the set of observations. Then o may be thought of as a tuple:

$$o \in O \subseteq L_1 \times L_2 \times \ldots \times L_n \times M \tag{A.3}$$

For an observation o, we will use subscript to refer to individual elements of the tuple: o_i is the level of factor f_i for o. The measurement value for o is then o_{n+1}, but we will use m to indicate this value instead: $m = o_{n+1}$. Equation A.3 defines the form our data will take in ANOVA.

Recall that the basic purpose of ANOVA is to determine how levels of the factors in F effect the measurable quantity. In fact, since some factors may have effects which interact with other those of other factors, we would like to know the effects of each subset of F on values of m. A *source*, S, is a subset of F:

$$S \subseteq F, S \neq \emptyset \tag{A.4}$$

Every source induces a partition of the observations O; the blocks of this partition are called the *groups* of O with respect to S.

The groups for source S are defined by setting levels of all of the factors in S. We will write $(O|S)$ for the partition of O induced by S, a set of sets. We will further write:

$$(O|S)(l_{i_1}, l_{i_2}, \ldots, l_{i_{|S|}}) = \{o \in O | o_{i_k} = l_{i_k}, 1 \leq i_k \leq |S|\} \tag{A.5}$$

for the group in $(O|S)$ given by choosing the values $l_{i_1}, l_{i_2}, \ldots, l_{i_{|S|}}$ for the factors $f_{i_1}, f_{i_2}, \ldots, f_{i_{|S|}}$. Using the ordering over factors, we assume that $i_1 < i_2 < \ldots < i_|S|$. That is, the values in equation A.5 are always given in order of increasing factor index. This allows us to uniquely associate levels with factors in our notation. The mean of the group $G = (O|S)(l_{i_1}, l_{i_2}, \ldots, l_{i_{|S|}}) \in (O|S)$ will be denoted μ_G.

With the above notation available, we may now discuss the ANOVA procedure in further detail.

A.C The ANOVA Procedure

Even though it's called the "analysis of variance," ANOVA is used to determine whether differences exist between group means for each source $S \subseteq F$. Formally, we have one null hypothesis per source S:

$$H_0(S) : \forall G \in (O|S), \mu_G = \mu_0(S) \tag{A.6}$$

for some constant $\mu_0(S)$ which may depend only on S. That is, the null hypothesis is that all of the groups for source S are drawn from populations with identical means. Rejection of $H_0(S)$ means there exist $G, G' \in (O|S)$ such that $\mu_G \neq \mu_{G'}$. Note that we don't know how many or which pairs of group means from $(O|S)$ are distinct, only that at least one pair is different.

The test used to decide whether to reject $H_0(S)$, called an "F-test," is based on a ratio of two different estimates of the common variance of the groups $(O|S)$. These estimates are referred to as "mean-squares" in ANOVA. The "within groups" and "between groups" mean-squares for source S are written $\mathrm{MS}_{\mathrm{within}}(S)$ and $\mathrm{MS}_{\mathrm{between}}(S)$. The basis for ANOVA is that $\mathrm{MS}_{\mathrm{within}}(S)$ is sensitive to variation among individual observations, sampling error, and $\mathrm{MS}_{\mathrm{between}}(S)$ is sensitive to variations between group means. The "F-ratio:"

$$F(S) = \frac{\mathrm{MS}_{\mathrm{between}}(S)}{\mathrm{MS}_{\mathrm{within}}(S)} \tag{A.7}$$

is distributed according to the F distribution, from which it derives its name. Higher values of $F(S)$ indicate a higher likelihood that at least two of the groups have distinct means, and thus that we should reject $H_0(S)$. This is because one of the assumptions of ANOVA stated that the differences in the estimates cannot be due to differences in variances across groups. For each source S, the F-test consists of computing $F(S)$ and the determining its significance via a table or program for a given significance level, p. If the observed $F(S)$ is less probably then p with respect to the F distribution, then $H_0(S)$ is rejected.

For each source S for which we reject $H_0(S)$, we say there is an effect owing to S. If S is singleton, the effect is a *main effect*. If $|S| > 1$, then the effect is called an *interaction* among the factors in S. If $|S| > 1$ and $F(S)$ is significant for our chosen value of p, then the factors in S interact in some way: their total effect on m values is not simply additive across the factors' effects. Rather, changing levels of some factors in S changes the effect of varying the levels of other factors in S.

Once F-tests have determined which sources, if any, have significant effects on observed m values, we would like to investigate the nature of the effects. *Contrasts* are examinations of differences in group means for a given source. A contrast may be between a pair of means or may involve looking for differences between several means and a single distinguished mean of interest. It is particularly important to distinguish *a priori* contrasts from *a posteriori* contrasts. The former are contrasts decided upon and motivated before F-tests were performed. The latter are contrasts being performed in light of a significant F-test for some source. Since the statistical procedures are different in particular for a priori versus a posteriori comparisons, it is important to be careful about the distinction.

Here we will consider a simple procedure that allows all a posteriori pairwise contrasts: Tukey's "Honestly Significant Difference" ("HSD" hereafter). In this approach, one computes a threshold on absolute differences between means which is used to decide whether pairs of group means are significantly different. The threshold is given by:

$$\mathrm{HSD}(S) = q\sqrt{\frac{\mathrm{MS}_{\mathrm{within}}(S)}{g}} \tag{A.8}$$

where q is the "studentized range statistic," whose value depends on the number of degrees of freedom for $\mathrm{MS}_{\mathrm{within}}$, which should be reported by the ANOVA program performing the F-test, the number of groups induced by source S, and a significance level. All groups $G \in (O|S)$ must have the same cardinality, $|G| = g$. For $G, G' \in (O|S)$, the difference in means is significant if $|\mu_G - \mu'_G| > \mathrm{HSD}(S)$, in which case we reject the null hypothesis, that $\mu_G = \mu'_G$. The Tukey HSD method is a very useful method when we don't know which contrasts we will want to examine before performing F-tests.

Analysis of variance is a complex subject. We recommend that readers unfamiliar with ANOVA start by reading intuitive accounts of one-way and two-way ANOVA from an elementary text[63]. Below, we present analyses performed using the UNIXSTAT program "anova," available on many UNIX installations. There are many commercially available statistics packages that perform a wider variety of analyses. SAS and SPSS are two popular programs. BMDP, though sometimes confusing to use, is capable of performing almost any statistical procedure which might be desired.

Appendix B

Parsing and production probability tuning costs

In chapter III, it was noted that evaluating $\Pr(\sigma|\Gamma)$ requires answering the membership question $\sigma \stackrel{?}{\in} \mathcal{L}(\Gamma)$. In chapter IV it was further seen that evaluating $\log_{10} \Pr(\sigma|\Gamma)$ needs to be done for each training data string, σ, in order to evaluate stochastic grammar Γ. In addition, it will be useful to have information concerning all parse trees, or parts thereof, assigned to σ by Γ. This is the difference between recognition (i.e., answering the membership question) and parsing (i.e., answering the membership question by giving a parse tree or a compact description of all parse trees for σ). In this section, the concern is with how parsing is accomplished, how string probabilities are computed, and how we can tune the probabilities on Γ to locally maximize the probability of whatever members of E^+ are parsed by Γ.

B..1 The parser and its time complexity

The parser used here is a form of dynamic programming algorithm known as a chart parser [19, 35]. Chart parsers are a particular class of parsing algorithms, which determine for a grammar Γ and a string ω whether $\omega \in \mathcal{L}(\Gamma)$. In particular, a chart parser finds all possible parse trees for ω using the productions of Γ.

In most of what follows, I will need to assume that the reader knows a few things about the operation of a chart parser. All of these have to do with the data structure on which such a parser works, called an "active chart," or just a "chart." In particular, we will need to refer to some of the constituent structure of a chart after a parse has been completed in it. Here, we fill in just the details necessary for an understanding of what's to come. Charniak gives a fuller exposition of the rudiments of chart parsing[19]. Note that a chart summarizes parsing information for a *single* string.

Following Charniak's notation, the set of nonterminals is:

$$N = \{N^1, N^2, \ldots, N^{|N|}\}$$

and the set of terminals is:

$$T = \{t^1, t^2, \ldots, t^{|T|}\}$$

. It is assumed that the indexing of nonterminals is always done such that N^1 is the start symbol. The individual letters of ω are written as w_k for the k'th letter and substrings are written $\omega k, l$ for the k'th through l'th letters, inclusive. Subscripting ω (as opposed to w) refers to distinct strings instead of the letters of a string.

Below, we will use N_{nt} to refer to the number of nonterminals in Γ and N_{prod} to refer to the number of productions in Γ.

Charts are made up of "entries" and "edges." Entries specify completed subparses and are given as $N^j_{k,l}$, meaning that nonterminal N^j covers the substring $w_{k,l}$ of ω; that is, $N^j \Rightarrow w_{k,l}$. Edges specify partial or full completions of rules. Each edge contains a production from Γ annotated with a dot on the right-hand side (RHS). The part of the RHS to the left of the dot is a string of terminals and nonterminals from which some substring of ω may be derived using productions of Γ. Thus, we can write edges as:

$$N^j_{k,l} \rightarrow \alpha_1 \cdot \alpha_2 \ldots \alpha_r$$

Each α_i can be a terminal or nonterminal and the left-hand side is *not* an entry but a statement that the part of the right-hand side currently completed (just α_1 in this case) generates $w_{k,l}$. The dot could occur anywhere on the right-hand side of the rule. If it occurs before the first symbol on the right-hand side then $k = l$.

A special subset of edges consists of those that have the dot after the last symbol on the rule right-hand side, as in:

$$N^j_{k,l} \rightarrow \alpha_1 \alpha_2 \ldots \alpha_r \cdot$$

Such an edge is referred to as a "completed" edge, and warrants the edition of a new entry to the chart, if $N^j k, l$ is not already in the chart, or the addition of a new "completion" to an existing entry. Each entry contains a list of its completions which are just references to its completing edges.

The basic activity of parsing consists in adding entries to a chart and expanding edges around those entries, which implies adding further entries. The initial entries come from the letters of ω which are added as entries without any completing edges. To refer to such an "initial entry," we just refer to letter of ω, as in w_i for the i'th letter. If the following edge is in the chart:

$$N^j_{k,l} \rightarrow \alpha_1 \alpha_2 \ldots \alpha_{r-1} \cdot N^o$$

then adding $N^o_{l+1,m}$ causes the following completed edge to be added:

$$N^j_{k,m} \rightarrow \alpha_1 \alpha_2 \ldots \alpha_{r-1} N^o \cdot$$

and this completion of N^j leads to the addition of entry $N^j_{k,m}$ to the chart, or to the addition of this edge as a new completing edge for that entry. We associate several simple functions with edges for notational convenience, again following Charniak's notation[19]. If e is the edge:

$$N^j_{k,l} \rightarrow \alpha_1 \cdot \alpha_2 \ldots \alpha_r$$

then we can define the following:

1. start(e) = k
2. end(e) = l

3. rule(e) = the production of Γ associated with edge e

4. lhs(e) = the nonterminal on the left-hand side of rule(e)

5. rhs(e) = the sequence of entries used to complete the right-hand side of rule(e). *Not* to be confused with rhs(rule(e)), which we might prefer as a notation for the right-hand side of e's rule, which is a sequence of *symbols.*

In the case of stochastic grammars, every production has an associated probability. These probabilities are used to compute probabilities for entries in a chart, and from these we can determine that probability of a string given a stochastic grammar or the most likely parse of a string given a stochastic grammar. Recall that the probabilities for collections of productions with the same left-hand side must sum to one. Thus, the production probabilities give distributions over rules expanding each nonterminal.

Every entry in a chart has associated with it *two* probabilities: an inside probability and an outside probability. The inside probability of $N^j_{k,l}$, written $\beta_j(k,l)$, is the probability of deriving $w_{k,l}$ from N^j. The outside probability of $N^j_{k,l}$, written $\alpha_j(k,l)$ is the sum of the probabilities of all derivations of the string

$$w_{1,k-1}N^j w_{l+1,n}$$

where n is the length of ω. The inside probability of $N^1 1,n$ is then the probability of ω according to Γ; that is,

$$\Pr(\omega|\Gamma) = \beta_1(1,n).$$

The inside and outside probabilities are also used in the inside-outside algorithm, which modifies the production probabilities to make a collection of training strings as likely as possible given the grammar.

As was noted previously, the cost of parsing is $O(n^3)$ for strings of length n parsed using context-free grammars, but for some classes of grammars the cost is instead linear. Instead of the basic parser used above, I use a modified parser that avoids adding edges under two circumstances. The edge:

$$N^j_{k,l} \rightarrow \delta \cdot \gamma,$$

where δ and γ are strings of terminals and nonterminals, will not be added to the chart unless w_{l+1} begins some string of terminals derived from γ. The completed edge:

$$N^j_{k,l} \rightarrow \delta \cdot,$$

where again δ is a string of terminals and nonterminals, is not added unless there is some sentential form $\gamma_1 N^j \gamma_2$ containing N^j which derives a strings of terminals and w_{l+1} begins some string of terminals derived from γ_2. With these two constraints on parsing activity, the parser behaves like a simple LR (i.e., SLR or LR(0)) parser on LR(0) grammars. This cuts down on parsing effort at least for simple grammars.

B..2 The inside-outside algorithm and calculation error

The inside-outside algorithm's purpose is to update the probabilities of a stochastic grammar's productions to maximize the probability of a given set of strings[19]. The maximization is done via expectation-maximization (EM), which finds a local maximum. Here, I describe this algorithm, which is used in conjunction with the evolutionary algorithm in the work reported in subsequent chapters, and cover some basic arguments about the amount of expected error introduced by its calculations.

The error introduced by the inside-outside algorithm is important because we need to be sure that introducing it into the algorithm won't render fitness measurement so noisy as to be meaningless due to numerical error. This error can be estimated by computing the total number of floating point operations done by the inside-outside algorithm, N_{op}, or an upper bound on this quantity, and then multiplying by machine precision[1] by the square-root of the number of operations. This

[1] A number ϵ_{M} defined as the smallest representable floating-point value such that $1+\epsilon_{\mathrm{M}} > 1$ [67]. Every floating-point operation should be assumed to introduce this much error into the calculation.

corresponds to the assumption that the error fluctuates over time as a random walk. This is a reasonable assertion for the inside-outside algorithm, since it doesn't operate on pairs of quantities of widely disparate orders of magnitude. Thus, the quantity $\epsilon_{\mathrm{M}}\sqrt{N_{\mathrm{op}}}$ tells us how many significant digits we should trust from the mantissa of a given floating-point probability resulting from a run of the inside-outside algorithm.

In what follows we will consider three values of ϵ_{M}, taken from three different machines. In all cases we are using double-precision arithmetic. The first machine is a SPARCstation10 running SunOS 4.1.3, the second a Pentium machine running Linux 1.2.13, and the third a DEC alpha running Digital UNIX V3.2D-1 (Rev. 41). All three machines support double-precision floating point values 8 bytes long and all three have an ϵ_{M} of $2.22045 \cdot 10^{-16}$.

The inside-outside algorithm use the information stored in a chart parse for a string ω given a grammar Γ to compute inside and outside probabilities for the chart entries. These probabilities are then used to compute production "counts" which count how many times each production is used, but with usages prorated by the probabilities of the parses in which they occur. The definitions of the inside and outside probabilities are:

$$\beta_m(k,k) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } w_k = t^m \\ 0 & \text{otherwise} \end{cases} \tag{B.1}$$

$$\beta_j(k,l) \stackrel{\text{def}}{=} P(w_{k,l}|N^j_{k,l}) \tag{B.2}$$

$$\alpha_1(1,n) \stackrel{\text{def}}{=} 1 \quad \text{(Note: } N^1 \text{ is the start symbol.)} \tag{B.3}$$

$$\alpha_j(k,l) \stackrel{\text{def}}{=} P(w_{1,k-1}, N^j_{k,l}, w_{l+1,n}) \tag{B.4}$$

The method by which each is actually computed is given below. Charniak gives derivations that connect the definitions with the equations discussed in the following subsections[19].

The inside-outside algorithm does the following work each iteration:

1. Compute the inside probabilities for the strings in the training set, using the current grammar's productions' probabilities.

2. Compute the outside probabilities for the strings in the training set, again using the current grammar's productions' probabilities.

3. Compute the "counts" for each production. For each production this is the sum over all its occurrences in all parse trees for all strings in the training set of the probability of the parse.

4. Normalize the counts for each collection of productions with the same right-hand side to get the updated production probabilities.

This is done either until some convergence criterion is reached (the probabilities don't change by more than some threshold overall) or simply for some fixed number of iterations. We express the total number of operations needed to do this as:

$$N_{\mathrm{op}} = N_{\mathrm{iter}}\left(\mathrm{FLOPs}_{\alpha,\beta} + \mathrm{FLOPs}_{\mathrm{update}}\right) \tag{B.5}$$

where $\mathrm{FLOPs}_{\alpha,\beta}$ is the number of "FLOPs"[2] needed to compute inside and outside probabilities in the first two steps, $\mathrm{FLOPs}_{\mathrm{update}}$ is the number of FLOPs needed to compute the counts and renormalize them into new probabilities, and N_{iter} is the number of iterations for which the algorithm runs.

As mentioned above, a chart parser is used in practice to keep track of all parses (using the grammar being trained) of each string in the training set. The chart for training instance ω_i stores ω_i's parses such that a parse of a given substring of ω_i (a possible subtree of some full parse tree) is stored only once. Thus, chart parsers are dynamic programming algorithms that fill in charts so that they satisfy this requirement and this "non-redundant" storage of parses is used by the inside-outside algorithm to avoid recomputing probabilities for parses of the same substring of ω_i.

In the following subsections, we will calculate the maximum number of floating-point operations which could possible be performed in a single iteration of the inside-outside algorithm.

[2]Floating point operations

Inside probabilities

Equations B.1 and B.2 give definitions for the inside probabilities of entries, but the computation is generally done using the information stored in a chart resulting from a parse of ω using Γ, as was mentioned above. The equations actually used in calculating the quantities $\beta_j(k,l)$ are:

$$\begin{aligned}\beta_j(k,l) &= \sum_{e \in \mathcal{E}^j_{k,l}} P(\text{rule}(e)) \prod_{X^m_{n,o} \in \text{rhs}(e)} \beta_m(n,o) \\ &= \sum_{e \in \mathcal{E}^j_{k,l}} \beta(e) \qquad\qquad \text{(B.6)}\end{aligned}$$

$$\beta(e) \stackrel{\text{def}}{=} P(\text{rule}(e)) \prod_{X^m_{n,o} \in \text{rhs}(e)} \beta_m(n,o) \qquad\qquad \text{(B.7)}$$

Here, the set $\mathcal{E}^j_{k,l}$ is the set of all completions of $N^j_{k,l}$. That is, $\mathcal{E}^j_{k,l}$ contains all completed edges for which start(e) = k, end(e) = l, and lhs(e) = N^j. For each completion, we multiply the rule probability for production rule(e) by the probability of the entries used to complete edge e. Thus, the product is over entries in rhs(e). We use $X^m_{n,o}$ to denote these instead of $N^m_{n,o}$ because some of them may be entries for terminals (that is, X could be a terminal or nonterminal, since the edge may directly use the aforementioned initial chart entries). Equation B.7 is used below and is a convenient abbreviation for a quantity which we will want when we discuss the cost of computing outside probabilities.

For a given entry, determining the cost of computing its inside probability involves determining how many FLOPs are used for the term added to $\beta_j(k,l)$ by each completion and adding these up. We can bound this by determining the maximum number of operations per completion and the maximum number of completions of entries in any chart and multiplying these.

Each completion of $N^j_{k,l}$ arises from some production of Γ whose right-hand side consists of a sequence of symbols, each of which generates some nonempty substring of $w_{k,l}$[3]. Thus, the longest right-hand side of a production which generates

[3]Nonempty because we don't allow epsilon productions (productions which generate the empty string)

substring $w_{k,l}$ is of length no more than $l-k+1$, the length of $w_{k,l}$. The value is less if we have placed some a priori limit on the length of a right-hand side and this value is less than $l-k+1$.

Using equation B.6, for each completion $e \in \mathcal{E}^j_{k,l}$ we must multiply together the inside probabilities of the entries given in rhs(e). This costs us no more than $(l-k+1)-1$ multiplications, using our upper bound on the size of a production right-hand side. We must also multiply this by the production probability $P(\text{rule}(e))$, which gives us an overall number of operations of no more than $l-k+1$ per completion of $N^j_{k,l}$. More generally, if the number of entries in rhs(e) is always less than or equal to m, then we need no more than m multiplications for that completion. This observation is only of interest for completions of entries for which $m < l-k+1$.

The number of completions for $N^j_{k,l}$ is a difficult issue because it depends so much on the form of the grammar. We can make a few observations, however, that are helpful in certain special cases, and give an upper bound generally, though the upper bound is not informative for very long strings. If Γ is unambiguous, all entries participating in a complete parse tree have only a single completion (else the grammar is ambiguous by definition). If the grammar has bounded degree of ambiguity of degree b, then each entry which is part of a full parse tree can have at most b completions, again by definition. The entries which are not part of any complete parse will all have outside probability zero and will contribute nothing to the overall results of the inside-outside algorithm. So as long as adding zero costs us nothing in precision on the machine we're using, these operations can be safely ignored[4].

In general, if grammar Γ has N_{prod} productions then the number of completions for an entry $N^j_{k,l}$ can never be more than the number corresponding to all productions completing all partitions of the string $w_{k,l}$ into any number of nonoverlapping, nonempty substrings. Thus, we can bound the number of completions, $|\mathcal{E}^j_{k,l}|$ of $N^j_{k,l}$ by the product of the number of such partitions of $w_{k,l}$ and the number of

[4]This also counts on the floating arithmetic implementation to guarantee that multiplication by zero yields zero; we do not deem this to be a troublesome assumption.

productions in the grammar:

$$
\begin{aligned}
|\mathcal{E}^j_{k,l}| &\leq \sum_{m=1}^{l-k+1} N_{\text{prod}} \begin{pmatrix} (l-k+1)-1 \\ m-1 \end{pmatrix} \\
&= \sum_{m=1}^{l-k+1} \begin{pmatrix} l-k \\ m-1 \end{pmatrix}
\end{aligned}
\tag{B.8}
$$

This is a rather pessimistic overestimate, since it assumes that all partitions of $w_{k,l}$ into any possible number of nonoverlapping, nonempty substrings is generated by every production, which can't happen in practice since the right-hand sides of productions and the number of productions in the grammar would have to grow with the length of ω. We consider the possibility here because it is possible to construct grammars for which there are not N_{prod}, but instead at least one completion for $N^j_{k,l}$ for each appropriate partition of $w_{k,l}$ for arbitrarily large finite subsets of certain languages. Asymptotically, parsing is cubic in the length of ω, but it can be made to appear exponential (since equation B.8 simplifies to $2^{n-1} - n - 1$) for finite subsets of infinite languages, or for finite languages[5].

To add up the inside probabilities for the completions of $N^j_{k,l}$, as needed in equation B.6, we require one addition per completion to add up the inside probability over all completions. We can also bound the overall number of multiplications needed for $N^j_{k,l}$ by multiplying the number of completions for this entry by the number of multiplications needed per completion. This is never more than m multiplications if the maximum production right-hand side length in Γ is m, or n if there is no such limit in Γ.

Finally, we must determine the total number of operations required in processing the inside probabilities. This is just the sum of the number of FLOPs needed for all completions of all entries in the chart for ω according to Γ. Thus, we sum the

[5]Such grammars must have nonterminals that generate strings consisting of c copies of themselves for c from 1 up to the longest string for which the parsing cost is to be expensive; constructing such a grammar amounts to arranging for the constant hidden by the $O(n^3)$ bound to be particularly large for the grammar in question, since the expense can only be forced for a finite set of strings, since we are not considering grammars with infinite numbers of productions!

number of completions for each entry over all entries and multiply the resulting number of total completions of entries by the maximum number of FLOPs per entry, m multiplications and 1 addition from the above discussion, to obtain the total number of FLOPs needed.

$$\left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} \left(N_{\mathrm{prod}} \sum_{m=1}^{l-k+1} \left(\begin{array}{c} l-k \\ m-1 \end{array} \right) \right) \right] (m+1) \tag{B.9}$$

as the total number of FLOPs in computing inside probabilities for the string ω, with length n, according to Γ. If we replace the number of completions for an entry covering $w_{k,l}$ with $C_{k,l}$, then we can rewrite this as:

$$\left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l} \right] (m+1) \tag{B.10}$$

We will return to this form later when we consider bounds on the number of completions per entry for unambiguous, bounded ambiguity, and unbounded ambiguity grammars, and when combining the number of FLOPs needed for the inside and outside probability computations.

Outside probabilities

The definition for the outside probabilities was given in equations B.3 and B.4. The first of these simply specifies that the probability of generating the just the start symbol is by definition 1. The second is the equation that actually requires us to do computation. The form actually used to compute the outside probabilities is again different and again uses the information stored in a chart for ω according to Γ:

$$\alpha_j(k,l) = \sum_{e \in \overline{\mathcal{E}}^{j}_{k,l}} \alpha_{\mathrm{lhs}(e)}(\mathrm{start}(e), \mathrm{end}(e)) \frac{\beta(e)}{\beta_j(k,l)} \tag{B.11}$$

Equation B.11 makes use of the inside probabilities, but we don't need to consider their costs again, since they are calculated before the outside probabilities; thus, we treat the inside probabilities as given at this stage and compute the addition number of FLOPs needed to compute the outside probabilities. The sum in equation

B.11 is again over completions, but the set over which we're summing is now $\overline{\mathcal{E}}^j_{k,l}$. The addition of the overline indicates that the set contains all completed edges for which $N^j_{k,l}$ is one of the entries in $\text{rhs}(e)$, *not* the entry being completed by edge e.

We are still, however, doing a specific amount of work for each completion that we process in computing outside probabilities, even though we are encountering completed edges in a different order than we did in the case of inside probabilities, and we can again bound the number of operations per completed edge in the chart for ω and multiply by the number of completions as before. One difference is that whereas completed edge e can only complete a single entry, it can participate in up to m, the maximum right-hand side length, outside probabilities since it will be in $\overline{\mathcal{E}}^j_{k,l}$ for each entry on its right-hand side. Thus, the number of additions per completion is m in this case, instead of one as it was in the inside probability case.

Consider the number of multiplications for each completion of an entry on whose right-hand side $N^j_{k,l}$ occurs[6]. In practice, instead of doing the division indicated, we do a computation much like the computation of $\beta(e)$ given in equation B.7 except that we leave $\beta^j_{k,l}$ out of the product when we encounter it, which we must since edge e is known to be in $\overline{\mathcal{E}}^j_{k,l}$ in equation B.11. Thus, we pay $m-1$ FLOPs instead of m to compute the quotient by this method, and one more multiplication to multiply by $\alpha_{\text{lhs}(e)}(\text{start}(e), \text{end}(e))$. This total cost of m multiplications is, like the cost of an addition, paid for all m entries which make use of completion e. Thus, the overall cost per completion for outside probabilities will be $m(m+1)$. Using $C_{k,l}$ as the number of completions per entry as in the previous section, we arrive at an overall cost of:

$$\left[N_{\text{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l}\right] m(m+1) \tag{B.12}$$

for the outside probabilities. Adding equations B.10 and B.12, we finally arrive at our expression for $\text{FLOPs}_{\alpha,\beta}$:

$$\text{FLOPs}_{\alpha,\beta} = \left[N_{\text{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l}\right] (m+1)^2 \tag{B.13}$$

[6]Remember that we're talking about the right-hand side of an entry, not a production!

where $C_{k,l}$ is as defined previously, since we can factor this to get the number of completions times the sum $(m+1)+m(m+1)=(m+1)^2$. We now need to compute $\text{FLOPs}_{\text{update}}$ to be able to complete our analysis.

Production counts and new probabilities

The last two phases of the inside-outside algorithm compute production usage counts, prorated as mentioned previously, and normalizes these to form new probabilities which are used as the basis for the computation of inside and outside probabilities in the next iteration. The production counts are computed using the following formula:

$$\begin{aligned}\text{count}(N^j \to \Omega^i) &= \frac{1}{P(w_{1,n})}\sum_{k,l}\alpha_j(k,l)\sum_{e\in\mathcal{E}^j_{k,l}, rule(e)=N^j\to\Omega^i}\beta(e) \\ &= \sum_{k,l}\sum_{e\in\mathcal{E}^j_{k,l}, rule(e)=N^j\to\Omega^i}\frac{1}{P(w_{1,n})}\alpha_j(k,l)\beta(e) \qquad \text{(B.14)}\end{aligned}$$

Here, we have used the notation $N^j \to \Omega^i$ to stand for any production with N^j on the left and any string of terminals and nonterminals, which must be nonempty, on the right. Note that once again we are summing over completions. In our code, the division by $P(w_{1,n})$ is done under the summation, so in our case each completed edge is processed once, since all productions have their counts computed and each completed edge e is associated with a production rule(e). Thus, we have one division, a multiplication for the second term, and m multiplications for the third term for a total of $m+2$ FLOPs per completion.

Normalization of counts in production probabilities is a simple matter depending only on the number of productions in the grammar, not on the number of completions in a chart parse:

$$P_e(N^j \to \Omega^i) = \frac{\text{count}(N^j \to \Omega^i)}{\sum_k \text{count}(N^j \to \Omega^k)} \qquad \text{(B.15)}$$

Each production in equation B.15, $N^j \to \Omega^i$, is added into a sum for productions whose left-hand side is N^j (the denominator under the sum), then all such productions are divided by this sum. This costs us an addition and a division per production, for a total of $2 \cdot N_{\mathrm{prod}}$.

We can now add up the total FLOPs needed to compute counts and normalize them into probabilities:

$$\mathrm{FLOPs}_{\mathrm{update}} = \left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l} \right] (m+2) + 2 \cdot N_{\mathrm{prod}} \tag{B.16}$$

We now have all of the components necessary to arrive at an equation defining the number of operations needed by the inside-outside algorithm.

Total number of operations and expected error

Using equation B.5, we can total up our previous formulae as follows:

$$\begin{aligned} N_{\mathrm{op}} &= N_{\mathrm{iter}} \left(\mathrm{FLOPs}_{\alpha,\beta} + \mathrm{FLOPs}_{\mathrm{update}} \right) \\ &= N_{\mathrm{iter}} \left\{ \left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l} \right] (m+1)^2 + \left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l} \right] (m+2) + 2 \cdot N_{\mathrm{prod}} \right\} \\ &= N_{\mathrm{iter}} \left\{ \left[N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} C_{k,l} \right] \left((m+1)^2 + (m+2) \right) + 2 \cdot N_{\mathrm{prod}} \right\} \end{aligned} \tag{B.17}$$

The most critical factor in equation B.17 is the number of completed edges in a chart parse of ω according to Γ. As noted previously, the overestimate we suggested initially for $C_{k,l}$ was unrealistically high. Even in this case, however, it is instructive to look we can expect reasonable results as long as the size of ω, n, doesn't become too large. Using $N_{\mathrm{iter}} = 100$, many more iterations than is typical for EM-based methods such as the inside-outside algorithm, $N_{\mathrm{prod}} = 100$, $N_{\mathrm{nt}} = 100$, and $m = 100$, $\epsilon_{\mathrm{M}} \sqrt{N_{\mathrm{op}}}$ has the values $0.0005, 9.2317 \cdot 10^{-8}, 5.0993 \cdot 10^{-10}$ for $n = 50, 25, 10$.

In general, as noted earlier, the expression for N_{op} exhibits an exponential dependence on n in this case. Consider the other cases which we also mentioned earlier. For unambiguous grammars, $C_{k,l} = 1$ in all cases, and the summation over entries becomes:

$$N_{\mathrm{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} 1 = N_{\mathrm{nt}} \frac{n(n+1)}{2} \tag{B.18}$$

This is only a quadratic dependence on n, which becomes linear under the square root. For an unambiguous grammar, we can reasonably expect to accurately compute probabilities using the inside-outside algorithms for very long strings: with $N_{\text{iter}} = 100$, $N_{\text{prod}} = 100$, $N_{\text{nt}} = 100$, $n = 100$, and $m = 100$, the value of $\epsilon_{\text{M}}\sqrt{N_{\text{op}}}$ has the value $1.6017 \cdot 10^{-10}$. Further, we can increase these values to 1000 with the exception of n, which is increased to 10,000 and still have an expected error of $1.5725 \cdot 10^{-6}$. This in spite of the fact that we are still significantly overestimating the true cost by assuming that all nonterminals are completed for all substrings, the maximum number of completions possible in an unambiguous grammar. A very similar case to the unambiguous one arises with ambiguity bounded by a constant, in which case $C_{k,l} = b$, where b does not depend on n. In this case, the modifications to the overall number of entries looks much the same up to the constant b:

$$N_{\text{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} b = b \cdot N_{\text{nt}} \frac{n(n+1)}{2} \tag{B.19}$$

The results are then the same up to a factor of roughly b (as long as the number of productions is not too large compared to n).

Finally, we consider two more possible expressions for $C_{k,l}$ which are more realistic. Since we know that parsing context-free grammars is done in time cubic in n, it's reasonable to ask about the case in which $C_{k,l} = c \cdot n$ for c some constant not depending on n, l, or k. In this case, we arrive at:

$$N_{\text{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} (c \cdot n) = N_{\text{nt}} \cdot c \cdot n \frac{n(n+1)}{2} \tag{B.20}$$

which gives a cubic dependence on n before taking the square root. This is closer to a true worst case for finite grammars. We conclude by examining a commonly exhibited grammar with unbounded ambiguity and noting how it fits into the above definitions:

$$\begin{array}{lcll} N^1 & \rightarrow & N^1 N^1 & (p) \\ N^1 & \rightarrow & t^1 & (1-p) \end{array}$$

The productions' probabilities are not relevant to determining the number of completions and have thus been left unspecified. This grammar finds a distinct completion for all substrings of length two of any string parsed according to it. Note that it generates only strings of a single character but generates a large number of parse trees for each. The number of completions, according to this grammar, in a chart for ω equal to a string of t^1's is just our initial pessimistic formula for the number of completions per entry, equation B.8, with the sum over m limited to the case $m = 2$:

$$\sum_{m=2}^{2} N_{\text{prod}} \begin{pmatrix} l-k \\ m-1 \end{pmatrix} = N_{\text{prod}}(l-k) \leq N_{\text{prod}} \cdot n \tag{B.21}$$

which can be plugged into the formula for the total number of completions to yield:

$$N_{\text{nt}} \sum_{k=1}^{n} \sum_{l=k}^{n} (N_{\text{prod}} \cdot n) \tag{B.22}$$

which is an instance of the dependence of N_{op} on n we've just discussed.

Bibliography

[1] Naoki Abe and Hiroshi Mamitsuka. A new method for predicting protein secondary structures based on stochastic tree grammars. In *Machine Learning: Proceedings of the Eleventh International Conference*, pages 3–11, San Francisco, CA, 1994. Morgan Kaufmann Publishers, Inc.

[2] Alfred V. Aho and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, Reading, MA., 1979.

[3] M. Ajtai, J. Komlós, and E. Szemerédi. An o(n log n) sorting network. In *Proceedings of the 15th ACM Symposium on the Theory of Computing (STOC83)*, pages 1–9, New York, NY, 1983. Association for Computing Machinery.

[4] S. F. Altschul, W. Gish, W. Miller, E. Myers, and Lipmann. Basic local alignment search tool. *Journal of Molecular Biology*, 215(403), 1990.

[5] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[6] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[7] M. Anthony and N. Biggs. *Computational Learning Theory: an introduction.* Cambridge University Press, New York, 1992.

[8] Mathematics Argonne National Laboratory and Computer Science Division. The mpi home page: http://www.mcs.anl.gov/mpi.

[9] Protein Data Bank (PDB) at Brookhaven National Labs. http://pdb.pdb.bnl.gov.

[10] Thomas Back. *Evolutionary algorithms in theory and practice : evolution strategies evolutionary programming, genetic algorithms.* Oxford University Press, New York, 1996.

[11] Timothy L. Bailey and Charles Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 28–36, Menlo Park, CA, 1994. AAAI Press.

[12] K. E. Batcher. Sorting networks and their applications. In A. S. Hoagland, editor, *Proceedings of the AFIPS Sprint Joint Computer Conference*, pages 307–314, Washington, D.C., 1968. Thompson Book Co.

[13] Richard K. Belew. Interposing an ontogenic model between genetic algorithms and neural networks. In J. Cowan, editor, *Advances in Neural Information Processing Systems (NIPS5)*, San Mateo, CA, 1993. Morgan Kaufmann.

[14] Richard K. Belew and editors Melanie Mitchell. *Adaptive Individuals in Evolving Populations: Models and Algorithms*. Addison-Wesley Publishing Company, Inc., Reading, Mass., 1996.

[15] Scott Brave. Evolving deterministic finite automata using cellular encoding. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming: Proceedings of the First Annual Conference*, pages 39–44, Cambridge, Mass., 1996. A Bradford Book: The MIT Press.

[16] M. Brown and C. Wilson. Rna pseudoknot modeling using intersections of stochastic context free grammars with applications to database search. In *Pacific Symposium on Biocomputing '96: Proceedings of Biocomputing '96*, pages 109–125, 1996.

[17] A. Burago. Learning structurally reversible context-free grammars from queries and counterexamples in polynomial time. In *Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory, COLT 94*, New York, NY, 1994. ACM.

[18] E. Rodney Canfield and S. Gill Williamson. A sequential sorting network analogous to the batcher merge. *Linear and Multilinear Algebra*, 29:43–51, 1991. Patent applied for March 1991, University of California as Assignee.

[19] Eugene Charniak. *Statistical Language Learning*. The MIT Press, Cambridge, MA, 1993.

[20] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, Menlo Park, CA, 1997. AAAI Press/MIT Press.

[21] N. Chomsky. *Syntactic Structures*. Mouton, The Hague, 1957.

[22] OBSTRUCT database at the European Molecular Biology Lab. http://www.embl-heidelberg.de/obstruct/obstruct_info.html.

[23] Beth Dawson-Saunders and Robert G. Trapp. *Basic and Clinical Biostatistics*. Appleton and Lange, Norwalk, Connecticut, second edition, 1994.

[24] Margaret O. Dayhoff. *Atlas of Protein Sequence and Structure, Volume 5, Supplement 3*, chapter 22–23, pages 345–357. National Biomedical Research Foundation, Washington, D.C., 1978.

[25] Michael Devitt and Kim Sterelny. *Language and reality: an introduction to the philosophy of language*. MIT Press, Cambridge, Mass, 1987.

[26] Russell F. Doolittle. *Of urfs and orfs : a primer on how to analyze derived amino acid sequences*. University Science Books, Mill Valley, CA, 1986.

[27] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *Journal of the Association for Computing Machinery*, 36(4), 1989.

[28] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*, chapter 12. Wiley, New York, 1973.

[29] P. Dupont. Regular grammatical inference from positive and negative examples by genetic search: the gig method. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications (LNAI862)*, pages 236–245, Berlin, 1994. Springer-Verlag.

[30] Sean R. Eddy, Graeme Mitchison, and Richard Durbin. Maximum discrimination hidden markov models of sequence consensus. *Journal of Computational Biology*, 1995.

[31] J.M. Fitzpatrick and J. J. Grefenstette. Genetic algorithms in noisy environments. *Machine Learning*, 3(2):101–120, 1988.

[32] A. L. N. Fred. Clustering of sequences using a minimum grammar complexity criterion. In *Grammatical Inference: Learning Syntax from Sentences, ICGI-96*, pages 107–116, Berlin, Germany, 1996. Springer-Verlag.

[33] K. Fu and T. Booth. Grammatical inference: introduction and survey, part 1. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-5:95–111, 1975.

[34] K. Fu and T. Booth. Grammatical inference: introduction and survey, part 2. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-5:409–423, 1975.

[35] D. Gazdar and C. Mellish. *Natural Language Processing in LISP: An Introduction to Computational Linguistics*. Addison-Wesley Publishing Company, Reading, MA, 1989.

[36] J.-Y. Giordano. Inference of context-free grammars by enumeration: Structural containment as an ordering bias. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications (LNAI862)*, pages 212–221, Berlin, 1994. Springer-Verlag.

[37] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[38] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Mass., 1989.

[39] David E. Goldberg, Kalyanmoy Deb, and Bradley Korb. Don't worry, be messy. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 24–30, San Mateo, California, 1991. Morgan Kaufmann Publishers, Inc.

[40] Frederic Gruau. Genetic synthesis of modular neural networks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc., 1993.

[41] W. N. Grundy. Family-based homology detection via pairwise sequence comparison. In *Proceedings of the Second Annual International Conference on Computational Molecular Biology*, 1998.

[42] W. N. Grundy, T. L. Bailey, C. P. Elkan, and M. E. Baker. Meta-MEME: Motif-based hidden Markov models of protein families. *Computer Applications in the Biosciences*, 13(4):397–406, 1997.

[43] W. Hart, T. Kammeyer, and R. Belew. The role of development in genetic algorithms. In *Foundations of Genetic Algorithms 3*, pages 315–322, San Francisco, CA, 1995. Morgan Kaufmann, Inc.

[44] William E. Hart. *Adaptive Global Optimization with Local Search*. PhD thesis, University of California, San Diego, May 1994.

[45] D. Haussler. Quantifying inductive bias: Ai learning algorithms and valiant's learn ing framework. *Artificial Intelligence*, 36(2):177–221, 1988.

[46] John H. Holland. *Adaptation in Natural and Artificial Systems*. A Bradford Book, The MIT Press, Cambridge, Mass., 1992.

[47] John H. Holland, Keith J. Holyoak, Richard E. Nisbett, and Paul R. Thagard. *Induction: Processes of Inference, Learning, and Discovery*. The MIT Press, Cambridge, Mass., 1986.

[48] Christian Jacob. Evolving evolution programs: Genetic programming and l-systems. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming: Proceedings of the First Annual Conference*, pages 107–115, Cambridge, Mass., 1996. A Bradford Book: The MIT Press.

[49] David T. Jones, William R. Taylor, and Janet M. Thornton. The rapid generation of mutation data matrices from protein sequences. *Computer Applications in the Biosciences (CABIOS)*, 8(3 pages = 275–282), 1992.

[50] Thomas E. Kammeyer and Richard K. Belew. Stochastic context-free grammar induction with a genetic algorithm using local search. In *Foundations of Genetic Algorithms 4*, San Francisco, CA, 1996. Morgan Kaufmann Publishers, Inc.

[51] Thomas E. Kammeyer and Richard K. Belew. An empirical investigation of an evolutionary algorithm's ability to maintain a known good solution. In *Evolutionary Programming VII*, 1998.

[52] Thomas E. Kammeyer, Richard K. Belew, and S. Gill Williamson. Evolving compare-exchange networks using grammars. *Artificial Life*, 2(2), 1995.

[53] Kenneth E. Kinnear, Jr. *Advances in genetic programming*. MIT Press, Cambridge, Mass., 1996.

[54] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4(4):461–476, 1990.

[55] Donald E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison-Wesley Series in Computer and Information Processing. Addison-Wesley Publishing Company, 1973.

[56] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. A Bradford Book: The MIT Press, Cambridge, Mass., 1992.

[57] John R. Koza. Simultaneous discovery of reusable detectors and subroutines using genetic programming. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 295–302, San Mateo, California, 1993. Morgan Kaufmann Publishers, Inc.

[58] M. M. Lankhorst. Breeding grammars: Grammatical inference with a genetic algorithm (computing science report cs-r 9401). Technical report, Department of Computing Science, University of Groningen, The Netherlands, 1994.

[59] James R. Levenick. Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Francisco, California, 1991. Morgan Kaufmann Publishers, Inc.

[60] James R. Levenick. Metabits: Generic endogenous crossover control. In Larry J. Eshelman, editor, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 88–95, San Francisco, California, 1995. Morgan Kaufmann Publishers, Inc.

[61] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 1998.

[62] Aristid Lindenmayer and Przemyslaw Prusinkiewicz. Developmental models of multicellular organisms: A computer graphics perspective. In C. Langton, editor, *Artificial Life*. Addison-Wesley Publishing Co., 1988.

[63] Robert B. McCall. *Fundamental Statistics for Behavioral Sciences*. Harcout Brace Jovanovich, Publishers, San Diego, CA, fourth edition, 1986.

[64] F. Menczer and R.K. Belew. Evolving sensors in environments of controlled complexity. In *Artificial Life IV. Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 210–221, Cambridge, MA, 1994. MIT Press.

[65] Balas K. Natarajan. *Machine Learning: A Theoretical Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1991.

[66] S. Nolfi and D. Parisi. Learning to adapt to changing environments in evolving neural networks. *Adaptive Behavior*, 5(1):75–98, 1996.

[67] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*, pages 24–27. Cambridge University Press, Cambridge, first edition, 1988.

[68] John A. Rice. *Mathematical Statistics and Data Analysis*. Brooks/Cole Publishing Co., Monterey, CA, 1988.

[69] J. Ruiz and P. Garcia. The algorithms rt and k-tti: A first comparison. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications (LNAI862)*, pages 180–188, Berlin, 1994. Springer-Verlag.

[70] Walter J. Savitch. Infinity is in the eye of the beholder: Technical report cs89-152. Technical report, Department of Computer Science, University of California at San Diego, 1989.

[71] Walter J. Savitch. Why it might pay to assume that languages are infinite. *Annals of Mathematics*, 8:17–25, 1993.

[72] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In L. D. Whitley and J. D. Schaffer, editors, *International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pages 1–37. IEEE Computer Society Press, 1992.

[73] Nicolai N. Schraudolph, John McInerney, and Richard K. Belew. Evolving networks: Using the genetic algorithm with connectionist learning. In J. D. Farmer, S. Rasmussen C. Langton, and C. Taylor, editors, *Artificial Life II*. Addison-Wesley, 1991.

[74] M. Schwehm and A. Ost. Inference of stochastic regular grammars by massively parallel genetic algorithms. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, San Francisco, CA, 1995. Morgan Kaufmann Publishers, Inc.

[75] David B. Searls. The computational linguistics of biological sequences. In *Artificial Intelligence and Molecular Biology*. AAAI Press, Menlo Park, CA, 1993.

[76] R. J. Solomonoff. A new method for discovering the grammars of phrase structure languages. *Information Processing*, 1959.

[77] R. J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22, 224–254, 1964.

[78] A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the International Conference on Grammatical Inference*, 1994.

[79] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, Nov 1984.

[80] K. VanLehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74, 1987.

[81] A.S. Wu and R.K. Lindsay. Empirical studies of the genetic algorithm with noncoding segments. *Evolutionary Computation*, 2(3):121–147, 1995.

[82] Peter Wyard. Context free grammar induction using genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Francisco, CA, 1991, 1991. Morgan Kaufmann Publishers, Inc.

[83] Peter Wyard. Representational issues for context-free grammar induction using genetic algorithms. Technical report, British Telecom, 1995.