Context-Free Language Induction by Evolution of Deterministic Push-Down Automata Using Genetic Programming

Afra Zomorodian

Computer Science Department
Stanford University
P. O. Box 7171
Stanford, CA 94309
afra@CS.Stanford.Edu

Abstract

The process of learning often consists of Inductive Inference, making generalizations from samples. The problem here is finding generalizations (Grammars) for Formal Languages from finite sets of positive and negative sample sentences. The focus of this paper is on Context-Free Languages (CFL's) as defined by Context-Free Grammars (CFG's), some of which are accepted by Deterministic Push-Down Automata (D-PDA). This paper describes a meta-language for constructing D-PDA's. This language is then combined with Genetic Programming to evolve D-PDA's which accept languages. The technique is illustrated with two favorite CFL's.

1 Introduction and Overview

Intelligent behavior often consists of *Inductive Inference*, making generalizations from sample incidents. The problem discussed here is finding generalizations for Formal Languages from finite sets of *positive* and *negative* sample sentences. A *positive* sentence is defined be a sentence accepted by the grammar of a language and hence included in the language. A *negative* sentence is defined accordingly.

Chomsky (Chomsky 1962) divides all languages and automata into four classes, from the most powerful (type 0 corresponding to Turing Machines and unrestricted grammars) to the least powerful (type 3 corresponding to Finite State Machines and Regular Expressions.) Context-Free Languages and Grammars (type 2) are of fundamental importance in Computer Science since high-level procedural programming languages, such as Pascal or C can be represented by CFG's and parsed by Deterministic Push-Down Automata (D-PDA). A D-PDA is a defined to be a finite-state machine coupled with a bottom-less stack. The machine is capable of reading letters off an input tape, pushing symbols onto a stack, and popping them off the stack. The machine rejects the sentence if it reads off the end of the sentence (beyond the end-of-sentence marker) or if it reaches the Reject state. It accepts the sentence if it reaches an Accept state regardless of the status of its stack.

Automatic generation of D-PDA's would be advantageous in analysis of new programming languages. An automatically generated D-PDA could educate the researcher about the idiosyncrasies or faults of the language and provide alternative unknown ways of parsing the language. Another motivation for the desirability of automatically defined D-PDA's is from a theoretical standpoint. Evolution of D-PDA's can be viewed as a step toward the evolution of Turning Machines capable of learning more complex languages.

In this paper, a language is developed capable of encoding the structure of any automata in the space of D-PDA's. This language is then successfully used along with Genetic Programming (GP) to evolve D-PDA's which accept languages.

The remainder of this paper is organized as follows: Section 2 describes some previous work on language induction. Section 3 provides some motivation for using Genetic Programming. Section 4 presents a new approach to GP language induction and describes a meta-language. Section 5 gives results of the application of the work on two CFL's. Section 6 presents a discussion of the lessons learned in this research. Section 7 provides some future directions and section 8 concludes this paper.

2 Previous Work

Much work has been carried out in language-inference since Gold (Gold 1967) initially established the possibility of inferring languages from sentences. The inference methods have included enumerative methods, hill climbing (Tomita 1982), higher order enhanced recurrent neural nets (Giles 1990), second-order recurrent neural nets (Watrous 1992), and Genetic Programming (Dunay 1994).

Most of this work consists in inferring Regular Languages and the corresponding Deterministic Finite Automata (DFA's). Tomita (Tomita 1982) was successful in evolving DFA's using hill climbing and a static number of states. He did, however, encounter a problem when his algorithm failed to find the correct machine by climbing a local hill. Dunay, Petry, and Buckles (Dunay et al. 1994) realized the usefulness of GP for inferring Regular Languages by representing DFA's as S-Expressions and allowing GP to determine the number of states needed. In their translation scheme, the program generated is never evaluated as a program but *is* the DFA (This is analogous to the DNA being the living creature.) A problem that emerged, however, was that "back pointers", pointers from

states to previous states, were hard to evolve in their translation scheme. In other words, the system was not capable of evolving D-PDA's which accept regular languages with many backpointers.

The problem faced by Dunay and others stems from the fact that a state-machine is a directed graph and an S-expression is a tree. Dunay's solution to this problem was a translation scheme to convert DFA's to S-expressions. In this scheme, DFA's with backpointers are hard to evolve. These backpointers represent conditional finite and infinite loops which are basic building blocks of state-machines.

3 Motivation For Using Genetic Programming

In language induction, we are essentially searching the entire space of D-PDA's to find a generalized solution which has learned a language. Genetic Algorithms are effective search algorithms which do not fall into local minima, unlike hill climbing (Tomita 1982.) Conventional genetic algorithms, however, are not directly applicable to the problem at hand because their chromosomes have a constant length and the number of states in the solution D-PDA is unknown. Genetic Programs may be viewed as Genetic Algorithms with variable-sized chromosomes.

4 A New Approach

GP is not directly applicable to the generation of D-PDA's. As Dunay explains, "[u]nless a mechanism can be added to the GP to more easily account for back pointers, there will continue to be simple DFA's which are difficult to evolve" using their translation scheme (Dunay et al. 1992). Traditionally, S-expressions are used in GP because the cross-over of two S-expressions is always syntactically valid. S-expressions, however, cannot represent loops and backpointers elegantly. One possible solution to the backpointer problem is the invention of another set of languages which have closure under the cross-over operation. Another approach would be to add a level of indirection into the GP pipeline. The latter approach is presented in this paper.

4.1 APDAL: A PDA Language

The approach taken here is different from previous work done in this field in that the GP generated program is not the solution but rather the constructing program for the solution. The encoding language is a meta-language which describes the structure of a D-PDA. This adds a level of indirection to the traditional pipeline of GP. Figure 1 shows a partial representation of the GP pipeline.

In each generation, a D-PDA is constructed by running an program from the population. Once constructed, the D-PDA is asked to evaluate a number of sentences. The program's fitness depends on it's D-PDA's performance inasmuch as an animal's DNA fitness depends on the animal's survival.

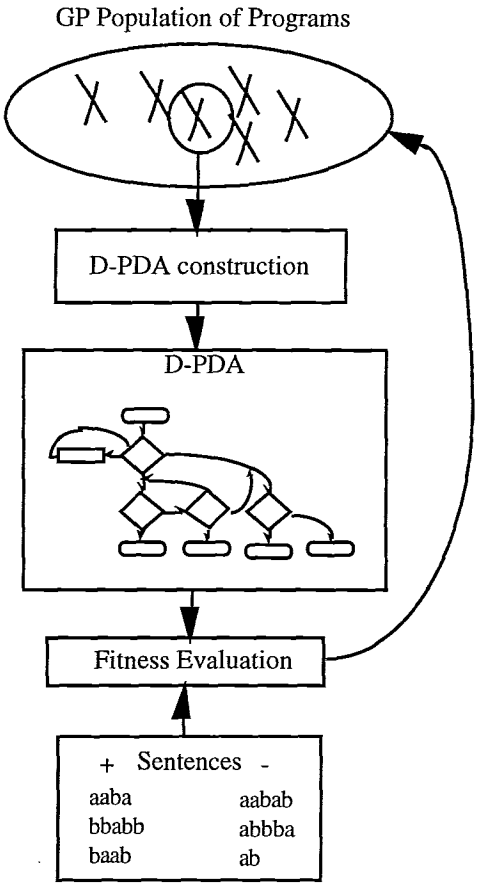

Figure 1: GP problem solving pipeline

The encoding language, APDAL, consists of a number of *macros*. A Lisp-like *macro* is executed before any of its parameters are. In APDAL, the parameters are macros themselves which create D-PDA states or *pseudo-states* (as defined below) and attach their creation to the state or pseudo-state created by the calling macro. The APDAL macros are:

- READ (arity 3): creates a read state and attaches it to the previous state. The read state has pointers to three other states as specified by its three parameter macros. A read state reads a character off a sample sentence and depending on the character (a, b, or empty), chooses a transition to another state.
- POP (arity 3): like READ, except the created state reads a character off the D-PDA's stack.
- PUSHA, PUSHB (arity 1): creates a pseudo-state which pushes an A or B onto the stack, respectively. A *pseudo-state* is a state which can only be pointed to by one state.
- REJECT, ACCEPT (arity 0): attaches the calling macro's state to the reject or accept state. For example, if REJECT is the first argument to a READ, it will point the read state's first pointer to the reject state.

A special scheme is used to solve the backpointer problem: During the assembly of a D-PDA from an

APDAL program, the states are numbered cumulatively as they are created. The pseudo-states created by PUSHA and PUSHB are not numbered. Furthermore, a variable (let's call it LASTSTATE) keeps track of the last state created. The following two special macros allow an APDAL program to change the value of this variable:

- DEC (arity 1): DEC is a special operator which does not create a state, but decrements LASTSTATE. Note: DEC always decrements LASTSTATE regardless of whether a call will be used or NOT. If the current state is the start state, LASTSTATE is not decremented.
- TOLAST (arity 0): This macro attaches the calling macro's state to the state specified by LASTSTATE. Therefore, any state can point to any of the previously created states by using DEC to decrement LASTSTATE, and TOLAST to achieve the connection.

The value of LASTSTATE is updated to be the number of the last state created every time READ, POP, or TOLAST are called.

4.2 An Example: $a^n b^n$

The best way to become comfortable with APDAL is to see the development of a D-PDA from an APDAL program. Consider the program in Figure 2. This program develops a D-PDA for the language $a^n b^n$ (n a's followed by n b's.)

```
(READ
    (PUSHA
      (TOLAST))
    (POP
      (READ
        (REJECT)
        (DEC (TOLAST))
        (POP (REJECT) (REJECT) (ACCEPT)))
      (REJECT)
      (REJECT))
    (TOLAST))
```

Figure 2: Program for $a^n b^n$

The construction of any D-PDA begins with a default Start State (Figure 3). The first macro in the program is a READ macro which creates a read state and attaches it to the Start state (Figure 4). Note that the read state has three state-transition pointers for characters "a", "b", and nothing (end of line marker). The states created by the READ macro's three parameters will attach themselves to these pointers. The next macro is a PUSHA macro which creates a pseudo-state to push an "a" onto the D-PDA's stack if an "a" was read by the read state (Figure 5). PUSHA is a one-arity macro and its parameter is a call to TOLAST which connects the push state's pointer to the read state, the last real state created. Execution of the entire APDAL program in Figure 2 results in the D-PDA in Figure 6. The interested reader should confirm that the D-PDA is in fact a state machine accepting the CFL $a^n b^n$.

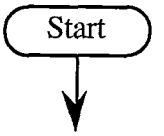

Figure 3: Start State

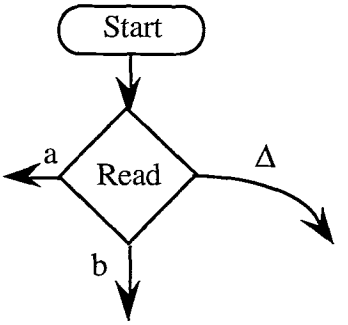

Figure 4: Start and Read states

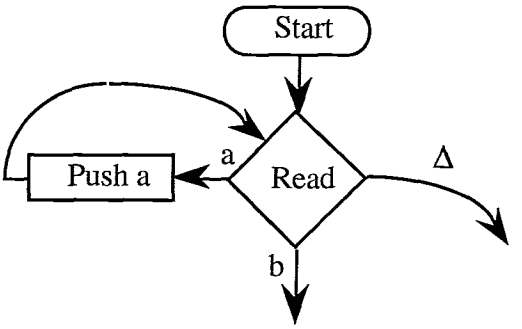

Figure 5: Special Macro TOLAST

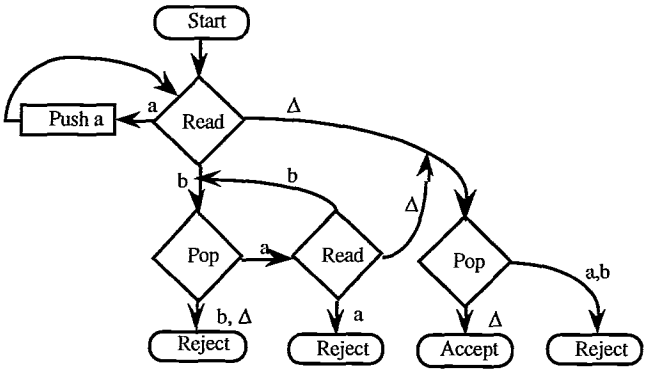

Figure 6: D-PDA for CFL $a^n b^n$ (See Figure 2)

4.3 Implementation

APDAL was implemented and debugged in Ansi-C using Symantec 7.0 on a Macintosh. The code was ported to a Sun SPARCstation 20 Model 61 and compiled using GNU's gcc. DGPC (Dave's Genetic Programming Code) by David André was used to generate and evolve APDAL programs.

Objective	Evolve a program whose output is a D-PDA which accepts a CFL.
Terminal Set	ACCEPT, REJECT, TOLAST
Function Set and arity	READ (3), POP (3), PUSHA (1), PUSHB (1), DEC (1)
Fitness Cases	For each developed D-PDA: a number of positive and negative samples from the language.
Raw Fitness	The fitness for which the D-PDA produces the right output ($Ntp + Ntn$)
Standard Fitness	$\frac{1-C}{2}$, where $C = \frac{NtpNtn - NfnNfp}{\sqrt{(Ntn+Nfn)(Ntn+Nfp)(Ntp+Nfn)(Ntp+Nfp)}}$ where: Ntp = Number of True Positives, Ntn = Number of True Negatives, Nfp = Number of False Positives, Nfn = Number of False Negatives.
Hits	Same as Raw Fitness.
Wrapper	None.
Other Prams.	M = 3000 individuals, T = 75 state transitions, G = 200 generations, Crossover(node) = 80%, Crossover(leaves) = 10%, Mutation = 0%
Success Predicate	The Standard Fitness equals zero.

Table 1: Genetic Programming Parameters

4.4 Genetic Programming Parameters

Table 1 summarizes the GP parameters that were used for the first language discussed in the Results section. Similar parameters were used for the other language.

5 Results

Two languages were chosen for this research. The languages are: a^nb^n and balanced parenthesis. Both languages are favorites in Computer Science as examples of CFL's and are frequently used to demonstrate the difference between a Regular Language and a CFL by the Pumping Lemma for Regular Languages (Hopcroft and Ullman 1979.)

5.1 a^nb^n

a^nb^n is a very small language since for any n, the fraction of strings of length up to 2n belonging to this language is:

```
(READ
  (PUSHB (TOLAST ))
  (POP
    (PUSHB
      (ACCEPT ))
    (READ
      (DEC (REJECT ))
      (DEC (TOLAST ))
      (POP
        (TOLAST )
        (REJECT )
        (ACCEPT ))
    )
    (REJECT ))
  (TOLAST ))
```

Figure 7: Solution at generation 11 for a^nb^n

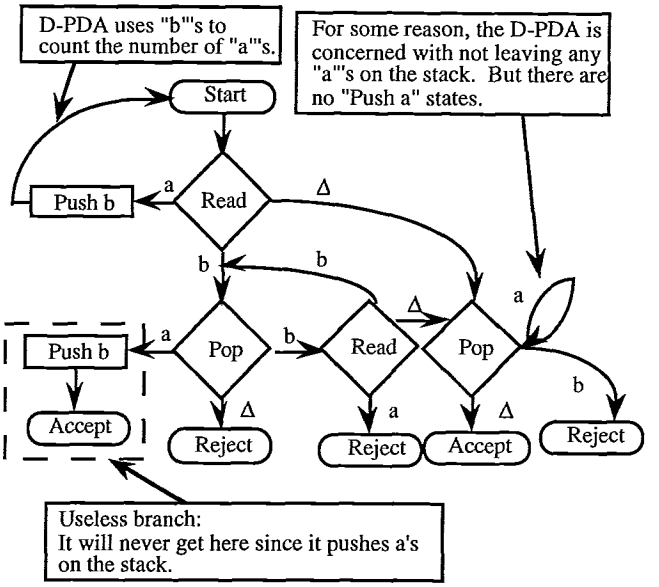

Figure 8: An exact solution for a^nb^n

$$\frac{n+1}{2^{2n+1}-1},$$

since the language only accepts one string from every schema of even-length and none from schema of odd-length. A set of ten positive and nineteen negative sample sentences were chosen for the language. The number of negative samples is higher because the language is very restrictive (most sentences do not belong to the language.) In a sample run, the best-of-generation for generation 0 had a fitness of 0.33515 with 16 Hits. The solution emerged in generation 11 and is shown in Figure 7. The D-PDA generated by the program is shown in Figure 8.

Note the following: The D-PDA uses b's to count a's. While this is bizarre to a human, it is only so because humans associate counting a's with the symbol "a". The machine does not differentiate between symbols and uses one for counting purposes. Also note that the solution in Figure 7 is not only an exact solution, it is almost the *same* as the human-generated one in Figure 6!

5.2 Balanced Parentheses

For balanced parentheses, the symbol "a" was used to denote "(" and the symbol "b" was used to denote ")". There were 11 positive samples and 12 negative samples. The best-of-generation for generation 0 had a fitness of 0.28035 with 15 hits. At generation 24, a solution emerged. The program is shown in Figure 9 and the generated D-PDA in Figure 10.

The solution generated for the balanced parentheses is rather clever. It has the first pop state in an unfortunate position (right after the start state.) This pop state is checking to see if there are enough open parentheses in the sentence by popping the stack. When the machine is activated, however, the stack is empty. The D-PDA's solution is to push a "b" on the stack in order to reach the read state. Another interesting aspect of this D-PDA is that it pushes a "b" and an "a" for every open parenthesis onto the stack. It uses the symbol "b" to count the number of open parentheses, so before making any decisions in a pop state, it always pops off all the "a"s. Other than the misplaced pop state, the solution has exactly the number of *active* states as the human-generated solution and it has the same structure too!

6 Discussion

The successful generation of D-PDA's using APDAL and GP was the fruit of the lessons learned during the design of APDAL and the many unsuccessful runs which led to design and parameter changes. I will now briefly discuss some of the issues raised by the APDAL-GP system.

A fast implementation of APDAL is an important concern. The memory for the states in the program is therefore not dynamically allocated but is taken off an static global array of states. The *same* memory is then used for all the individuals and for all the generations. Thus, the system can be used on computers with a small amount of memory and is also faster since it is not allocating and de-allocating memory hundreds of thousands of times per run. Another interesting implementation issue is how the states are connected. The final implementation uses a stack: every state is responsible for attaching itself to the last state by popping the stack and in turn allowing other states to be connected to it by pushing its successor pointers onto the stack. The stack is not needed by DGPC but is included so that the language can be used to generate D-PDA's for other purposes.

At first, the standard fitness of an individual was defined to be the maximum number for hits minus the individual's number of hits. Because of this definition, however, the number of positive and negative sample sentences had to be the same (ten sample sentences were used for each category.) In addition, a population of 1000 was used for 100 Generations with an allotted time of 100 ticks per string (M = 1000, G= 100, T= 100.) The first successful individual for the language $a^n b^n$ was evolved on the second run on generation 17 as shown in Figure 11.

```
(PUSHB
  (POP
    (TOLAST
    (READ
      (PUSHB (PUSHA (TOLAST )))
      (DEC (TOLAST ))
      (POP
        (POP
          (TOLAST )
          (REJECT )
          (REJECT ))
        (READ
          (TOLAST )
          (TOLAST )
          (ACCEPT ))
        (PUSHB
          (ACCEPT ))
      )
    )
    (REJECT )))
```

Figure 9: Solution at generation 24 for Balanced Parentheses

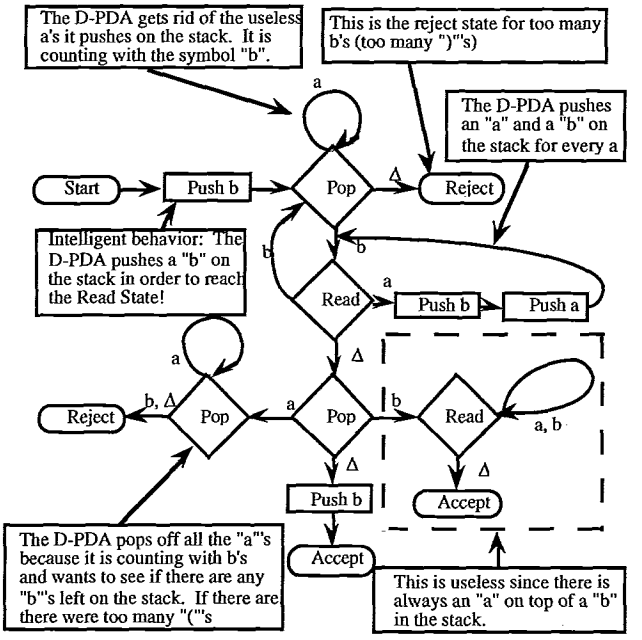

Figure 10: Clarifying Diagram of solution for Balanced Parentheses

```
(PUSHA(READ(POP(READ(PUSHA(TOLAST)) (READ(PUSHA(PU
SHB(TOLAST)) ) (POP(DEC(TOLAST)) (POP(REJECT) (ACCEPT
) (ACCEPT)) (READ(REJECT) (ACCEPT) (REJECT)) ) (POP(PUS
HB(REJECT)) (POP(REJECT) (TOLAST) (ACCEPT)) (PUSHB(TO
LAST)) ) ) (READ(REJECT) (ACCEPT) (TOLAST)) ) (PUSHA(REJ
ECT)) (DEC(TOLAST)) ) (POP(PUSHA(PUSHA(REJECT)) ) (PUS
HB(REJECT)) (DEC(TOLAST)) ) (DEC(PUSHA(ACCEPT)) ) ) ) )
```

Figure 11: First successful individual, Generation 17, Second Run

The individual is not only long and composed of many more states than needed, but also is defective. The discerning user can observe that it accepts the string "abbb" which is clearly not in the language $a^n b^n$. The following list of *probable* causes for the evolution of this defective individual were identified:

- The bottomless stack (a stack which always returns an empty-stack-symbol when popped while empty) allowed the individual to get large and allow for defects.
- The allotted time of 100 times allowed large individuals to take a long time and successfully process the sample sentences (the longest time the "human" solution takes for a string is 34 ticks.)
- The individuals should be punished more for running out of time, hence increasing the selective pressure to remain small.
- The language is very restrictive and more negative samples are needed.

The last cause seemed more probable and the lack of enough negative samples had always been a disturbing factor. The author noted that the reason the string "abbb" was accepted by the D-PDA was the lack of enough negative samples with more b's than a's and that the D-PDA had *learned* the samples. The standard fitness was changed to involve correlation which allows for different numbers of positive and negative samples. Nine more negative sentences with more b's than a's and sentences starting with b's were added.

After making these changes, the runs became unsuccessful. Many of the runs would have best-of-generation individuals with up to 23 hits, but none of the runs were able to solve the problem. The number of generations was increased to 200 with no result and then up to 250 without any change. The author observed, however, that there were usually no improvements in fitness after about generation 150. This meant the population size was too small. The population size was increased to 2000 and fitness improved remarkably. A solution was found when population was raised to 3000.

Having a large population means that a solution could presumably be found by random chance (essentially changing our search algorithm from the *Genetic Algorithm* to *Random Search*.) Fitness was set to be the same for all the individuals in the population (fitness = 1) for several runs. No solution emerged from this random search. Further, no improvements were seen in the number of hits of individuals in the generations. Therefore, the Genetic Algorithm was responsible for generating solutions.

The allowed time for processing a string was then reduced from 100 to 75 and a solution was still found, except it was smaller. The number of possible nodes in the Result-Producing Branch (RPB) was also lowered from 50 to 30 and the result was even smaller programs.

The lessons learned here were:

- A small number of nodes in the RPB along with a small allotted time for string processing would apply enough selective pressure to keep the programs small.
- The problem space is too big to be solved with a population of 1000. Further, 100 generations is enough to solve the problems.

The search space is extremely large (a very rough estimate of 30^8) and a population of 3000 will not help the random search algorithm to find a solution to the D-PDA problem.

7 Future Work

Future directions include:

- Testing with a variety of CFL's
- Analysis of dependability of the APDAL-GP system to develop D-PDA's for deterministic CFL's.
- Extension of APDAL to handle languages with more than 2 symbols (with eventual extension to handle any language regardless of number of symbols.)
- Evolution of more complex state machines from the Chomsky Hierarchy

8 Conclusions

Genetic Programming with APDAL encoding language is a powerful system to develop D-PDA's which learn a CFL from a small set of sample sentences from the language. The contribution of this research is the encoding of D-PDA's into a Lisp-like language and the application of GP to inferring CFL's.

Acknowledgments

The idea for APDAL was induced to me by Frédéric Gruau's Genetic Micro Programming of Neural Networks (Gruau 1994). I would like to thank Professor Koza persuading me that this project topic was superior to my other ones, and for his many helpful comments. I would also like to thank David André and Scott Brave for DGPC. Finally, my thanks go to my advisor Maggie Johnson for introducing me to CS Theory and for her incredible notes on the subject.

References

Chomsky, Noam. 1962. Context Free Grammar and Pushdown Storage. Quarterly Progress Report 65. The MIT Research Laboratory in Electronics. MIT

Dunay, B. D., Petry, F. E., and Buckles, W. P. 1994. Regular language induction with genetic programming. In Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE Press. Volume I. Pages 396-400.

Giles, C. L., Miller, C. B., Chen D., Chen, H. H., Sun, G. Z., and Lee, Y. C. 1992. Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks. Neural Computation 4, 393-405.

Gold, E. M. 1967. "Language Identification in the Limit," Inform. Contr. 10, pp. 447-474.

Gruau, Frédéric. 1995. Genetic micro programming of neural networks. In Kinnear, Kenneth E. Jr. (editor). Advances in Genetic Programming. Cambridge, MA: The MIT Press. Pages 495-518.

Holland, John H. 1975. Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press.

Hopcroft J., Ullman J. 1979. Introduction to Automata Theory, Languages, and Computation. Reading. MA: Addison Wesley.

Koza, John R. 1992. Genetic Programming: On The Programming of Computers by Natural Selection. Cambridge, MA: MIT Press.

Tomita, Masaru. 1982. Dynamic Construction Of Finite-State Automata From Examples Using Hill-Climbing. Proceedings to the Fourth Annual Cognitive Sciences Conference. Pages 105-108.

Waltrous, Ramond L. and Kuhn, Gary M. 1992. Induction of Finite-State Languages Using Second- Order Recurrent Networks. Neural Computation 4. Pages 404-414.