Learning Regular Languages Using Genetic Programming

Børge Svingen
Department of Computer and Information Science
Norwegian University of Science and Technology
N-7034 Trondheim
Norway
bsvingen@idi.ntnu.no

ABSTRACT

This paper describes an experiment where genetic programming was used to evolve, given a set of positive and negative examples, regular expressions recognizing several regular languages. The Tomita benchmark languages were used to test the system, and general solutions were found for all seven languages.

1 Introduction

In (Brave 1996), genetic programming (Koza 1992, Koza 1994) is used to evolve deterministic finite automata from a set of positive and negative example strings. This is done by using the method of cellular encoding (Gruau 1994), where the program trees build the automata while being executed, along with an additional, boolean combination of the resulting automata.

There is, however, a more natural way to represent finite automata as trees — by using regular expression. Since regular expressions and finite automata represent the same set of languages — regular languages — and since algorithms exist for converting between these two representations (Sipser 1997), it should be possible to evolve a regular expression that recognizes a given language, and then convert this into a finite automaton as needed.

This is the approach taken in the work described in this paper. Genetic programming is used to evolve regular expressions from training data, and as in (Brave 1996), the seven Tomita benchmark languages (Tomita 1982) are used to test the system. The Tomita languages are shown in Table 1.

Language	*Description*
TL1	1*
TL2	(10)*
TL3	no odd 0 strings after odd 1 strings
TL4	no 000 substrings
TL5	an even number of 01's and 10's
TL6	number of 1's - number of 0's is multiple of 3
TL7	0*1*0*1*

Table 1 The Tomita Benchmark Languages

2 Regular Expressions

(Sipser 1997) gives a definition of regular expressions that can be represented by the following grammar :

$$R \rightarrow a, \text{ for some } a \in \Sigma \quad (1)$$
$$R \rightarrow \epsilon \quad (2)$$
$$R \rightarrow \emptyset \quad (3)$$
$$R \rightarrow (R \cup R) \quad (4)$$
$$R \rightarrow (R \circ R) \quad (5)$$
$$R \rightarrow (R*) \quad (6)$$

Σ is here the alphabet used, $(R_1 \cup R_2)$ matches either R_1 or R_2, and $(R_1 \circ R_2)$ matches R_1 followed by R_2. (R_1*) matches any number of occurrences of R_1.

Since no interesting regular expressions require the use of Equations 2 and 3, they will in the following be ignored.

3 Representing Regular Expressions Using Genetic Programming

As can be seen from the grammar above, a regular expression can naturally be interpreted as a tree. If the members of the alphabet Σ, which in this paper will be taken to be $\{0, 1\}$, are used as terminals, and the operations of union, concatenation and repeated matches, represented by Equations 4 to 6 above, are used as functions, a program tree can represent any regular expression.

There are now two possible interpretations of such a program tree. In the first case it would be seen as a static representation of a regular expression, and not really as a program tree at all, in the second as a computer program describing how to build a finite automaton. Since building this automaton is

Function	*Arity*	*Explanation*
+	2	Builds an automaton that accepts any string accepted by one of the two argument automata.
.	2	Builds an automaton that accepts any string that is the concatenation of two strings that are accepted by the two argument automata, respectively.
*	1	Builds an automaton that accepts any string that is the concatenation of any number of strings where each string is accepted by the argument automaton.

Table 2 Function Set

Terminal	*Explanation*
0	Returns an automaton accepting the single character 0.
1	Returns an automaton accepting the single character 1.

Table 3 Terminal Set

the most practical way to match the regular expression with a given string, the last approach will be taken.

The function and the terminal sets will then be as shown in Tables 2 and 3, respectively.

4 The Experiment

For each of the seven Tomita languages, 500 positive and 500 negative strings were created. The strings were selected randomly, and had a maximum length of 50 characters.

Genetic programming was then used to evolve regular expressions to match the examples. The population size was 10000, divided over 16 demes (Koza 1992, Wright 1932, Tanese 1989, Andre and Koza 1996, Niwa and Iba 1996) with 625 individuals in each, and the best 25 individuals were kept unchanged for each generation. Fitness was defined as the number of strings not correctly classified, and fitness proportional selection was used. The program trees had a maximum initial depth of 3, and a maximum depth after crossover of 10. The probability of crossover was 0.9. Automatically defined functions (Koza 1994) were not used.

A single run of 100 generations was performed for each of the seven Tomita languages.

5 Results

For each of the Tomita languages, a perfect and general solution was found in the single run. These solutions are given, along with simplified versions in more common notation, in Table 4.

The languages TL1, TL2 and TL7 all have simple representations as regular expressions, and should therefore be easy to find. This was also the case; the first two were even found in the first generation.

The other solutions are not as intuitively correct, and will be dealt with in the following.

To start with TL3, it is clear that no strings matched by this regular expression contain an odd number of 0's after an odd number of 1's, since the only way to get an odd number of 1's is by using the 100 part of the expression — except for 1's at the end of the string — which contains a pair of 0's, and this can again be followed by either a 1 or the double 0.

To see that any string that contains no odd 0's after odd 1's is covered by this expression, it is enough to look at the subset 0*(11 | 100 | 110 | 00)*1*; the first 0* takes care of any leading 0's. There are then three possible cases which repeat throughout the rest of the string; even 1's followed by even 0's, which are handled by the 11's and the 00's, even 1's followed by odd 0's, which are handled by the 11's, 110's and 00's, and odd 1's followed by even 0's, which are handled by the 11's, 100's, and 00's. The 1* then takes care of any trailing 1's.

It is clear that no strings matched by the solution for TL4 contain the substring 000. To see that any string with no such subset can be matched by the solution, look at the subset (ϵ|00|0) (1 | 100 | 10)* — the first part handles any leading zeros, while the second part matches the rest of the string.

The language TL5 consists of all strings with an even number of 01's and 10's. The only way to achieve this, is to have a string that starts and ends with the same character, and this clearly applies to any string matched by the solution expression. To see that any string that starts and ends with the same character can be matched by the expression, it is important to notice that the expression consists of two parts; one handles strings that start and end with a 0, and the other handles strings that start and end with a 1 — each part of the expression consists of repeated patterns with just this property. For the 0's, the subset 0 | 01*0 makes this clear, while for the 1's, the subset 10*1* does the same thing.

Finally, for TL6, the solution consists of the repeated union of four parts, 1(10)*0, 0(01)*1, 11(01)*1 and 0(01)*00)*, and it is again clear that any string matched by this expression fulfills the requirement that the number of 1's subtracted the number of 0's should be a multiple of 3. It is more difficult to see that any such string can be matched by the expression — to see that this is in fact the case, it is useful to imagine going through such a string, counting modulo 3, where 1 is added for each 1, and 1 is subtracted for each 0. For most strings, there will be points in the string where the count reaches zero. Split the string at these points.

The result will be a number of substrings, where the count does not go to zero before the end. If the substring starts with a 0, then the count will go to $-1 \bmod 3 = 2$. This means that the next character must be another 0, to avoid reaching zero, getting the sum 1. The following character must then be a 1, and the count is back to 2. This pattern will repeat until the end of the string, where 0 is reached either by two 0's or two 1's in a row. This means that the substring will be matched by

Language	*Solution*	*Simplified Solution*
TL1	(* (* 1))	1*
TL2	(* (* (. 1 0)))	(10)*
TL3	(. (* (+ (. 1 (+ (+ 1 1) (. 1 0))) (* 0))) (. (* (+ (. 1 (+ (+ 1 (. 0 0)) (. 1 0))) (* (. 0 0)))) (* 1)))	(11 | 110 | 0)*(11 | 100 | 110 | 00)*1*
TL4	(+ 0 (. (+ (* (+ 1 (. 0 (+ 1 (. 0 1))))) 1) (+ (+ (. (. 0 0) (* 1)) 0) (* (+ 1 (. (+ (. 1 0) 1) 0))))))	((1 | 01 | 001)*|001*|0) (1 | 100 | 10)*
TL5	(+ (* (+ 0 (. (. 0 (* (* (. (* 0) 1)))) 0))) (* (. (. 1 (* (. (* 0) 1))) (* 1))))	(0 | 0(0*1)*0)* | (1(0*1)*1*)*
TL6	(* (+ (* (+ (. 1 (. (* (. 1 0)) 0)) (. (. 0 (* (* (. 0 1)))) 1))) (+ (* (+ (. 1 (. 1 1)) (. (. (. 1 1) (* (. 0 1))) 1))) (. (. (. 0 (* (* (. 0 1)))) 0) 0))))	(1(10)*0 | 0(01)*1 | 11(01)*1 | 0(01)*00)*
TL7	(. (. (. (* (* 0)) (* 1)) (* 0)) (* (+ 1 1)))	0*1*0*1*

Table 4 Results

the expression 0(01)*1 | 0(01)*00. Similarly, the other parts of the solution matches any substring that starts with a 1.

All the evolved solutions are therefore correct.

6 Conclusion

Based on the results described in the previous section, evolving regular expressions seems to be a useful method for recognizing regular languages. It has several advantages compared to the method used in (Brave 1996). First and most obvious, it seems to give better results — all the seven Tomita languages were recognized successfully here, while (Brave 1996) did not recognize TL6. Second, regular expressions are a more natural and elegant way of specifying a regular language in the form of a tree, and it is immediately understandable to the user. The extra steps with additional boolean combinations, used in (Brave 1996), are not necessary. Third, regular expressions can specify any regular language — this is not the case with the cellular encoding used in (Brave 1996).

Acknowledgements

This work has received support from The Research Council of Norway (Program for Supercomputing) through a grant of computing time. The GP++ genetic programming library (Svingen 1997) was used to do the genetic programming.

References

Andre, David and John R. Koza (1996). Parallel genetic programming: A scalable implementation using the transputer network architecture. In: *Advances in Genetic Programming 2* (Peter J. Angeline and K. E. Kinnear, Jr., Eds.). Chap. 16, pp. 317–338. MIT Press. Cambridge, MA, USA.

Brave, Scott (1996). Evolving deterministic finite automata using cellular encoding. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (John R. Koza, David E. Goldberg, David B. Fogel and Rick L. Riolo, Eds.). MIT Press. Stanford University, CA, USA. pp. 39–44.

Gruau, Frederic (1994). Genetic micro programming of neural networks. In: *Advances in Genetic Programming* (Kenneth E. Kinnear, Jr., Ed.). Chap. 24, pp. 495–518. MIT Press.

Koza, John R. (1992). *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press. Cambridge, MA, USA.

Koza, John R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press. Cambridge Massachusetts.

Niwa, Tatsuya and Hitoshi Iba (1996). Distributed genetic programming: Empirical study and analysis. In: *Genetic Programming 1996: Proceedings of the First Annual Conference* (John R. Koza, David E. Goldberg, David B. Fogel and Rick L. Riolo, Eds.). MIT Press. Stanford University, CA, USA. pp. 339–344.

Sipser, Michael (1997). *Introduction to the Theory of Computation.* PWS Publishing Company.

Svingen, Børge (1997). GP++ An introduction. In: *Late Breaking Papers at the 1997 Genetic Programming Conference* (John R. Koza, Ed.). Stanford Bookstore. Stanford University, CA, USA. pp. 231–239.

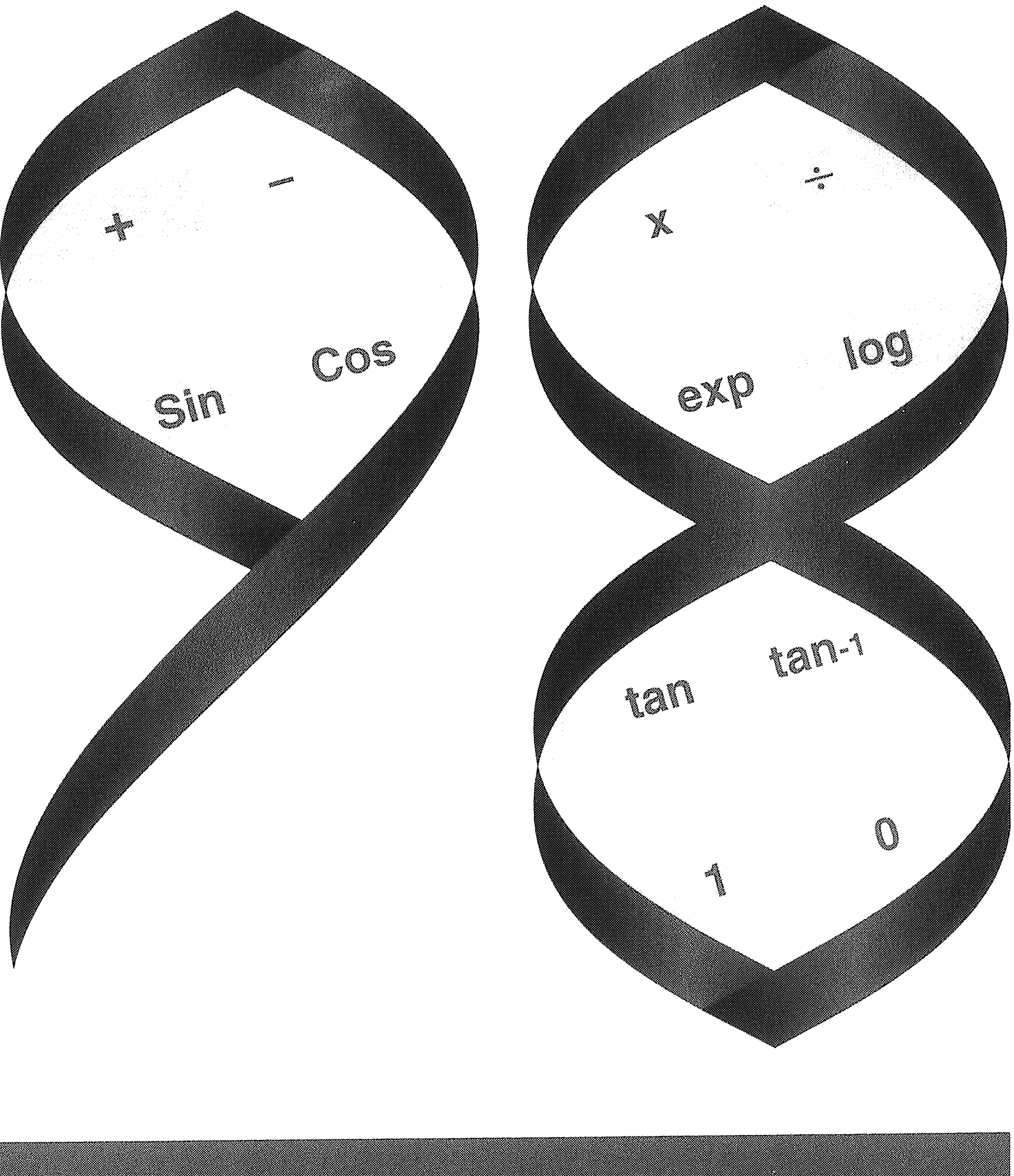
Tanese, R. (1989). Distributed genetic algorithms. In: *Proceedings of the 3rd International Conference on Genetic Algorithms.*

Tomita, M. (1982). Dynamic construction of finite-state automata from examples using hill climbing. In: *Proceedings of the Fourth Annual Cognitive Science Conference.* Ann Arbor, MI. pp. 105–108.

Wright, Sewall (1932). The roles of mutation, inbreeding, crossbreeding and selection in evolution. In: *Proceedings of the Sixth International Congress of Genetics.*

CONFERENCE PROCEEDINGS

Genetic Programming



edited by
John R. Koza
Wolfgang Banzhaf
Kumar Chellapilla
Kalyanmoy Deb
Marco Dorigo
David B. Fogel
Max H. Garzon
David E. Goldberg
Hitoshi Iba
Rick L. Riolo

Proceedings of the Third Annual Genetic Programming Conference

July 22–25, 1998
University of Wisconsin, Madison, Wisconsin