Learning Restarting Automata by Genetic Algorithms[*]

Petr Hoffmann

Charles University, Department of Computer Science, Malostranské nám. 25, 118 00 PRAHA 1, Czech Republic, petr.hoffmann@matfyz.cz

Abstract. Restarting automaton is a special type of a linear bounded automaton designed for modelling the so-called analysis by reduction. We use genetic algorithms to learn restarting automata to recognize languages according to input consisting of sets of positive and negative examples of words from the language together with positive and negative examples of simplifications.

1 Introduction

The checking of the syntactical correctness of a sentence may be based on the so-called analysis by reduction. Its principle consists in a stepwise simplification of a given extended sentence until a simple sentence is got or an error is found. To model the analysis by reduction we can use restarting automata [1].

The problem discussed in this article is the following. We want to do the analysis by reduction of a given language L. At the beginning we have a finite subset of L, a finite subset of its complement and finite sets of pairs of words describing preferred and prohibited simplifications. The goal is to find a restarting automaton which does the analysis by reduction of the given language.

This is the first attempt at learning restarting automata. With respect to the complexity of this problem, we decided to use genetic algorithms [2] to solve it.

In the next section we present definitions for restarting automata. Section 3 introduces genetic algorithms and main problems which must be solved for applying them to the above stated problem. Section 4 presents the solution. Section 5 describes experiments done. The last section discusses the achieved results and presents ideas for further research.

2 Restarting Automata

A *restarting automaton* [1] is a 7-tuple $M = (Q, \Sigma, k, I, q_0, q_A, q_R)$, where

- Q is a finite set of *states*,
- Σ is a finite set of *symbols*, $\Sigma \cap \{¢, \$\} = \emptyset$

[*] This work was supported by grants 300/2002/A INF/MFF of Grant Agency of the Charles University and 201/02/1456 of Grant Agency of the Czech Republic.

- k is a non-negative integer called the *size of lookahead*,
- $q_0 \in Q$ is the *initial state*,
- $q_\mathrm{A} \in Q$ is the *accepting state*,
- $q_\mathrm{R} \in Q$ is the *rejecting state*,
- I is a finite set of *instructions* of the following two types $(q, q' \in Q, a \in \Sigma \cup \{¢, \$\}, u \in \Sigma^* \cdot \{\$, \lambda\}, v \in \{¢, \lambda\} \cdot \Sigma^* \cdot \{\$, \lambda\}, k + 1 \geq |au| > |v| \geq 0)$:
  (1) $(q, au) \rightarrow (q', \mathrm{MVR})$,
  (2) $(q, au) \rightarrow \mathrm{RESTART}(v)$,

The *restarting automaton (RW-automaton) M* is a device with a finite state control unit, and a head moving on a finite linear (doubly linked) list of items containing the word of the form $¢ \cdot \Sigma^* \cdot \$$. The head has a lookahead of the size $k \geq 0$ – in addition to the item scanned by the head, M also scans the next k right neighbouring items (or the end of the word when the distance to \$ is less than k).

We suppose that Q is divided into the set of *nonhalting states* $Q - \{q_\mathrm{A}, q_\mathrm{R}\}$ (when the state control unit is in such a state, then there is at least one instruction which is applicable) and the set of *halting states* $\{q_\mathrm{A}, q_\mathrm{R}\}$.

The *computation* of M is controlled by a set of instructions I. The left-hand side (q, au) of an instruction determines when it is applicable – q means the current state, a the symbol being scanned by the head, and u means the contents of the lookahead.

The right-hand side of an instruction describes the activity to be performed. In the case (1), M changes the current state to q' and moves the head to the right neighbour item of the item containing a. If a is equal to \$ then q' must be a halting state. In the case (2), the activity consists of deleting some items (at least one) of the just scanned part of the list, rewriting some (possibly also none) of the non-deleted scanned items (in other words au is replaced with v), entering the initial state and placing the head on the first item of the list. Symbols ¢ and \$ cannot be deleted or rewritten. In this article we will deal with a restricted version of restarting automata – so called R-automata [1]. For R-automata v is the (proper) subsequence of au for all restart instructions.

A computation of M can be divided into certain phases. A phase called cycle starts when the control unit is in the initial state and the head is attached to the item containing ¢. The head moves to the right along the input list until a restart instruction is performed. Then the head is moved to the item containing ¢, state of the control unit is set to the initial state and a new phase starts. A phase called tail differs from cycle in that the head moves right until a halting state is reached. A word w is accepted if there is a computation in which the first phase starts on the word $¢w\$$ and which ends in the accepting state.

A cycle corresponds to one simplification in the analysis by reduction and a tail corresponds to accepting a simple sentence or rejecting an incorrect one.

In general, an R-automaton is *nondeterministic*, i.e., there can occur two or more instructions with the same left-hand side (q, au) in its set of instructions. If this is not the case, the automaton is *deterministic*.

3 Genetic Algorithms

Genetic algorithms [2] are search algorithms based on the mechanics of natural evolution. At the beginning genetic algorithm has a fixed number of strings (called generation) of the same fixed length coding some points of the searched space. It works in a cycle – it takes the previous generation and generates a new one using simple genetic operators (reproduction, crossover and mutation) or returns the generation as the output.

Reproduction is a process in which individual strings are copied according to their fitness function values. The fitness function measures suitability of the point (coded in the given string) for our purposes. Strings with a higher value have a higher probability of contributing one or more offspring in the next generation.

After reproduction crossover proceeds in two steps. First, the newly reproduced strings are mated at random. Second, each pair of strings undergoes crossing over with some probability. If the pair does not undergo crossing over, it is copied to the new generation directly. Crossing over means combining information from two strings into the new ones and putting them into the new generation.

Mutation is needed to insert some useful information randomly (a probability is given with which parts of new strings are changed).

4 Using Genetic Algorithms

4.1 String Representation of an Automaton

We suppose that the number of states, the length of lookahead and the alphabet are fixed before the searching starts. The only thing coded into the string representation of a nondeterministic R-automaton is the set of instructions. The set of all possible instructions contains more than $(|Q|-2)|Q||\Sigma|^{k+1}$ elements, so it may be too big to fit into the memory. Also the searched space may be too large. For this reason we decided to limit the number of instructions coded in the string representation using some fixed number.

String coding a particular automaton is a sequence of codes of its instructions[1]. An instruction is coded as a triplet of numbers representing the state and the string from the left-hand side of this instruction and the action from its right-hand side.

There occured a problem in using of this simple coding. If the code of a restart instruction contains the word v from its right-hand side, the operator of mutation could change the word au in its left-hand side and make this instruction incorrect. To solve this problem the number coding the right-hand side of a restart instruction codes indices of the deleted symbols of the word au from its left-hand side. The operator of mutation must be implemented in such a way that the generated instructions are correct.

[1] So one automaton may have more than one string representation.

4.2 Genetic Operators

The used operator of crossover chooses randomly the number $1 \leq n \leq |I| - 1$. Then codes of n instructions are got randomly from the first string, copied to the first output string and this string is completed using randomly chosen codes of instructions from the second string. The analogical way (using the same n) is used to generate the second output string.

The next operator needed is the operator of mutation. It is based on a simple change of an item of a triplet coding an instruction, but also assures that the output string is a valid code of an automaton.

Reproduction is not described here because of space limitations (it is done using a standard method [2]).

4.3 Fitness Function

For a successful application of genetic algorithms, it is important to have a good fitness function. Fitness function should measure the quality of an automaton with respect to the input requirements (positive and negative examples of words and simplifications). The trivial fitness function could be the number of requirements satisfied by the given automaton. This function has the following drawback: We often got an automaton recognizing the given subset of the language, but do not recognizing any element of the given subset of its complement or respectively. We think this automaton is worse than the one which recognizes some elements from both given subsets. A better measure of the quality of a given automaton can be the function $\frac{sp}{p}\frac{sn}{n}$ (inspired by [3]), where p (n, resp.) is the number of positive (negative, resp.) examples of words from the language, sp (sn, resp.) is the number of correctly classified[2] words from the positive (negative, resp.) examples. In the similar way we measure the quality with respect to the positive and negative examples of simplifications.

Let $pwr = \frac{sp}{p}$ and $nwr = \frac{sn}{n}$. To further constrain the problem described above we suggest to handicap automata with very small pwr relatively to nwr or respectively: If $pwr < \frac{nwr}{d}$ then we multiply the value of the previously given fitness function by $\frac{pwr}{\frac{nwr}{d}}$ and respectively. In experiments we used [3] $d = 5$.

Using this fitness function may have following problem: During the work of the genetic algorithm there may be a lot of randomly generated strings coding automata without the ability to end any computation in a halting state (thus incorrect). Therefore we force every computation to end in a halting state. The way to do it may consist in adding some *default instructions* – every time the automaton has no applicable instruction we apply some special instruction – for example the instruction leading the automaton into the rejecting state[4].

[2] To avoid time consuming computations we limit the maximal number of operations that can be done during simulation of a work of an automaton. If the limit is achieved, the given automaton is said not to classify the given word correctly too.

[3] This value was obtained by doing several experiments, but more statistic testing must be done to choose the best one.

[4] Note that this is a correct extension of the set of instructions.

5 Experiments

We applied the described method in learning several languages using the following scenario. First we suggested requirements used by the fitness function. We chose these requirements manually, more time is needed to look out some useful heuristics. Then we run the genetic algorithm (repeatedly) until an automaton satisfying all requirements was found. Then we analyzed the found automaton and possibly got this as a result of the experiment. In the case of any problems we modified requirements (by hand) and run the genetic algorithm again.

In all experiments we worked with generations of 100 members, the probability of applying the operator of crossover to given strings was 0.9, the probability of applying the operator of mutation to a part (item of a triplet coding an instruction) of a string was 0.04. To avoid loss of some very good strings during the computation of the genetic algorithm we used so-called elitism [2] – a fixed number (we used 14) of the best strings was copied to the new generation without using any operator. The maximum number of generations of one run of the genetic algorithm was 1000. Used default instructions always move the head right and preserve the current state (there is one exception – when the head scans \$, the new state is the rejecting state). These parameters were obtained by doing several experiments, but more statistic testing must be done to choose the best one's. We let the genetic algorithm work with codes of nondeterministic automata. We did not use any positive simplifications in requirements.

At first we tried some simple regular languages. Inspired by finite automata we searched for automata with zero lookahead. The first language was $L_0 = \{a^{3n}; n \in \{0,1,\ldots\}\}$. We looked for an automaton having at most 10 states and 10 non-default instructions. The same holds for experiments with languages $L_1 = \{a^{4n}; n \in \{0,1,\ldots\}\}$ and $L_2 = \{a^{5n}; n \in \{0,1,\ldots\}\}$. Automata recognizing L_0 and L_1 were found without any change of the initial requirements. In the case of L_2 we were not able to find any suitable automaton.

The next language was $L_3 = \{a^{2m}b^{3n}; m,n \in \{0,1,\ldots\}\}$. We looked for an automaton having at most 20 states and 20 non-default instructions. Automata satisfying all given requirements were not found.

We tried also a context-free language $L_4 = \{a^n b^{2n}; n \in \{0,1,\ldots\}\}$. Here the situation is simplified since the wanted automaton obviously need not use many states. We wanted an automaton using only 3 states, having 3 symbols in lookahead and using at most 10 non-default instructions. About 10 extensions of initial requirements were done until a suitable automaton was found.

At last we tried the language L_5 of expressions consisting of one symbol for variables, one symbol for operations and left and right parentheses (for example $a+(a+a+a+(a+a)+a)$ belongs to this language). We looked for an automaton using only 3 states, having 2 symbols in lookahead and using at most 10 non-default instructions. About 5 extensions of initial requirements were done until a suitable automaton was found.

6 Conclusions and Future Work

The achieved results show that genetic algorithms can be used for learning a certain type of restarting automata. However, the learning was not successful even for some simple languages. Hence, further research is needed. Some ideas are presented in the following sections.

6.1 A Problem with Default Instructions

Since default instructions are not represented explicitly in the string coding the set of instructions, the described genetic operators cannot deal with them. Suppose having the automaton satisfying some of our requirements and using the *default* instruction I_1 with the left-hand side L. Further let us have other automaton satisfying other part of requirements and using the (not default) instruction I_2 with the same left-hand side L and not having I_1 in its set of instructions. Applying of crossover onto these automata we cannot yield an automaton with both I_1 and I_2 in its set of instructions since presence of I_2 means there is no reason for using the default instruction with the left-hand side L. Similar problem exists in the case of the operator of mutation since default instructions cannot be mutated. An idea that may help is to change dealing with default instructions.

6.2 Dealing with Problematic Languages

Section 5 described experiments with languages L_1 and L_2. Here the first difference between these languages may be the higher probability of getting needed set of instructions for the automaton recognizing L_1 than for the automaton recognizing L_2. Our next conjecture is that it is easier to find an automaton recognizing L_1 since it is a subset of $L_6 = \{a^{2n}; n \in \{0, 1, \ldots\}\}$. If the genetic algorithm find an automaton recognizing L_6 first, it will obtain a higher fitness value and will influence members of the new generation. Then there may be higher probability of getting an automaton recognizing L_1 by using genetic operators.

An idea of dealing with problematic languages is to search for an automaton satisfying set of requirements (maybe for a different but in some sense similar language) other than initially suggested requirements first. If an automaton satisfying this set of requirements is found, we will use automata from the last generation as the initial generation for the computation of the genetic algorithm using the initial requirements.

References

1. František Mráz: Forgetting and restarting automata, Ph.D. thesis, Charles University (May 2001)
2. David E. Goldberg: Genetic Algorithms in Search, Optimization & Machine Learning, Addison-Wesley Publishing Company, Inc. (1989)
3. Marc M. Lankhorst: A Genetic Algorithm for the Induction of Nondeterministic Pushdown Automata, Computing Science Report CS-R 9502