Adaptive Automata for Syntax Learning

João José Neto and Margarete Keiko Iwai

Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
e-mail: jjneto@pcs.usp.br and mkiwai@pcs.usp.br

Abstract. This article presents adaptive automata as an alternative theoretical model to formally describe recognizing devices with learning capabilities. A simplified formal definition of the notation and its semantics is given, and some analysis is made on their time and space behavioral properties, showing positive aspects that make them suitable for knowledge acquisition and for describing and processing natural languages. A simple application of adaptive automata in this field is then shown, illustrating how they may be employed in the construction of a little sample-driven syntax-learning device for regular languages. The paper concludes that adaptive automata are well-suited not only to describe context-dependent languages and to operate as a powerful learning formalism, but also as very convenient formal specification tool to describe and even automate efficient implementations too.

Keywords: adaptive automata, syntax learning, knowledge acquisition.

Introduction

Syntax Learning is a task that may be automatically performed in a number of ways. This paper is concerned with syntax learning within adaptive automata.

Adaptive automata are Turing-powerful formal devices intended to describe context-dependent languages, which operate as finite-state or as pushdown automata while accepting already-known syntax.

Once some not yet explored legal aspect of the syntax is detected in the input string, a corresponding so-called adaptive action is taken. The purpose of that action is to modify the automaton to accept further instances of the new syntax. Besides of adapting itself to follow the syntactical needs of the input string, adaptive automata may be added further self-modifications.

These changes may be designed to enable the automaton to retain information on the already acceptable set of sentences, so implementing the basics for acquiring knowledge about its input language.

In extreme cases, an initial all-accepting automaton may evolve by steps, during a learning phase, into a tree-shaped automaton that strictly accepts the set of already read sentences.

In a second turn, usual generalization procedures may easily convert this acceptor into a graph-shaped automaton describing a wider superset of the language represented by the sample employed to build the former tree.

This procedure may lead to automata that accept a syntax wider than needed.

So, specializing procedures must be applied to the automaton in order to exclude undesired syntax forms.

Obtaining the desired adaptive acceptor may be accomplished in a final specializing phase of the learning process, with the aid of an input sample of rejecting strings.

Obviously, the accuracy of the accepting device obtained in this way is highly dependent on the quality of the given samples and on the heuristics adopted in the generalizing and specializing learning procedures.

This subject is not covered in this paper for being beyond its scope. [ref.]

Formal adaptive devices have been presented in many papers as powerful tools for rigorously defining complex languages [ref.].

Adaptive automata are one of such devices, and their main features include:

- a significant part of the accepting procedure performed by an adaptive automaton is done **very efficiently**, due to their structure based on finite-state or structured pushdown automata [ref.]
- a unique feature of adaptive devices is the natural way they may be designed to learn how to **handle new syntactic constructs**
- adaptive automata are Turing-powerful, allowing them to **handle context dependencies in a strict syntactical way**, without the aid of auxiliary semantic procedures

- adaptive automata are based on structured pushdown automata, that are in turn built up from simple finite-state automata, allowing it to achieve **high time performance** in deterministic cases
- adaptive automata may be easily extended to implement transducers, allowing the **generation of parse trees** as a side effect of parsing sentences
- general adaptive automata may be restricted to simpler models according to the strict needs of each particular language, allowing to minimise unnecessary resource wasting in the formal model, so leading to **cheaper implementations**
- natural language processing is a practical problem in Artificial Intelligence that may explore adaptive formal devices as an alternative to existing usual solutions for all these features

In this paper we show that it is possible to make practical use of adaptive formal devices in the field of syntax learning, one of the hundreds of problems that arise in natural language processing research

No attempt will be made to propose efficient methodologies or to evaluate the degree of usability of the proposed solution.

Instead, we will concentrate in showing it is possible to efficiently use adaptive concepts to solve fundamental problems in syntax learning, so applying adaptive formal devices to problem solving in such an important research field.

An adaptive automaton may be viewed as an underlying evolving structured pushdown automaton, represented by an initial fixed state machine implementing the adaptive automaton, and a set of adaptive functions.

Adaptive functions are intended to perform, along the automaton's operation, appropriate dynamic changes to the current version of its underlying state machine at that moment, so leading to a new version of that state machine.

In the next sections, a simplified description of the notation and the semantics of adaptive automata is given first, then an example is shown to illustrate its application to solve a typical problem on syntax-learning of regular languages.

Adaptive automata

Finite-state and pushdown automata are known to be inadequate to represent context-dependent languages, because of their lack of resources for stating context-sensitive features.

Infinite-state automata can perform such a task, but by their nature they represent a non-practical solution to that problem.

Many abstract devices have been proposed in the literature, each having their own features and limitations, and, due to their complexity, seldom leading to easy creation of efficient implementations.

Adaptive automata, as originally proposed [JJN] are devices that may be designed to operate extensively as finite-state or as pushdown automata.

An exceptional case occurs strictly at the first time syntactical constructs that have not been handled previously by the automaton.

In response, the adaptive automaton executes self-modifying *adaptive actions* that change its own shape into another one for accepting the novel construct.

By designing in such a way an adaptive automaton, the task of accepting context-dependent sentences is limited to some combination of three types of activities:

- handling a context-dependency not previously experienced by the acceptor
- handling an occurrence of a non-regular context-free construction in the sentence
- handling the remaining regular portions of the input string

Structure of adaptive automata

Structured pushdown automata are a class of pushdown automata in which we find:

- a finite set of *states*, including its
  - initial state, and
  - one or more final states;
- an input alphabet and
- a pushdown alphabet;
- a special symbol, indicating that the pushdown store is empty;
- a set of the sub-machines; and
- a set of productions, specifying all allowed transitions.

A set of productions denoting some structured pushdown automaton represents, at any instant, the current state machine implementing the adaptive automaton.

Let a configuration of the adaptive automaton be a triple, indicating the contents of the top of the pushdown store (g, g'), a state (s, s') and the next input symbol to be consumed (σ, σ'). Each production assumes the general form

$$(\gamma g , s , \sigma \alpha), \mathcal{A} : \rightarrow (\gamma g' , s' , \sigma' \alpha), \mathcal{B}$$

whose left- and right-side triples denote respectively the configurations of the adaptive automaton immediately before and immediately after the application of the production.

Symbols γ and α represent the invisible part of the pushdown store and the not yet consumed part of the input string, respectively.

Optional *adaptive actions* $\mathcal{A}$ and $\mathcal{B}$ may be attached to the productions, representing calls to adaptive functions to be executed before and after the production is applied, respectively.

By restricting the format of the productions, one may also restrict hierarchically the class of languages the automaton can describe: by eliminating adaptive actions, no context-dependencies are allowed, so context-free languages may be represented; by further eliminating references to a pushdown store, nested constructs are no more accepted, so only regular languages will be represented.

Eliminating the references to a pushdown store without forbidding the use of adaptive actions allow specifying context-dependent languages through a subclass of the adaptive automata in which the underlying state machine is restricted to be a finite-state machine.

An adaptive automaton may be stated as an evolving formal device that starts from an initial underlying structured pushdown automaton

$$E_0 = (Q_0 , SM_0 , \Sigma, \Gamma, P_0 , Z_0 , q_0 , F)$$

where

- Q_0 is its set of states, including a single initial state $q_0 \in Q_0$.
- F is the subset of final states, $F \subseteq Q_0$.
- Σ is the input alphabet,
- Γ is the pushdown store alphabet, where $Z_0 \in \Gamma$ indicates empty pushdown store
- SM_0 is the set of sub-machines constituting E_0 .
- P_0 is the adaptive automaton's initial set of productions

whose most general case has the form

$$(\gamma g, e, s\alpha), A : \rightarrow (\gamma g', e', s'\alpha), B$$

specifying a transition the adaptive automaton may execute from situation (g, e, s) .

At each instant, the transitions describing the underlying structured pushdown automaton are represented by productions of the form

$$(\gamma g , e , s \alpha) : \rightarrow (\gamma g' , e' , s' \alpha)$$

where

- (g , e , s) is the situation of the adaptive automaton before applying the production
- g is the contents of the pushdown store top before applying the production
- e is the sate of the automaton before applying the production
- s is the input symbol to be consumed next
- (g', e', s') is the situation after applying the production. Primed symbols denote the same elements above.
- Optional actions A and B specify adaptive function calls to be executed respectively before and after the transition is performed.

Different classes of languages may be properly described from this single general form of productions by imposing restrictions adequate to each language's needs:

- General context-sensitive languages will need adaptive actions. Writing onto the input string is optional, as well as the use of a pushdown store.

$$(\gamma g , e , s \alpha), A :\rightarrow (\gamma g' , e' , s' \alpha), B$$

- General context-free languages will need a pushdown store. No adaptive actions are used, nor writing onto the input string.

$$(\gamma g , e , s \alpha) : \rightarrow (\gamma g' , e' , \alpha)$$

- Regular languages do not need a pushdown store. Adaptive actions and writing onto the input string are forbidden.

$$(e\ ,\ s\ \alpha) : \rightarrow (e'\ ,\ \alpha)$$

Adaptive automata gradually evolve by departing from its initial shape and successively adding productions to or deleting productions from its own current set of productions P_i, as a result of executing adaptive actions.

1. Initialize i = 0.
2. In situation $(g, e, s)_i$, search P_i for some $(\gamma g, e, s\alpha)_i$, $A_i : \rightarrow (\gamma g', e', s'\alpha)_i$, B_i .
3. If no such a production is found, reject the input string and stop.
4. If only one production is found, apply it by proceeding at step 6.
5. If n>1 such productions are found, a non-determinism has been identified:
   5.1. Apply all found productions in parallel.
   5.2. Accept the input string if at least one of the trials succeeds, otherwise reject it.
   5.3. Stop.
6. If A_i is present, perform it, obtaining a new production set P_i' .
7. Search P_i' for the currently executing production.
8. If the currently executing production has been removed by A_i , go back to step 2.
9. Otherwise, update the current situation, as stated in the current production.
10. If B_i is present, perform it, obtaining P'_{i+1}.
11. If a final situation has been reached, accept the input string and stop.
12. Otherwise, increment i and proceed the recognition in step 2 .

The algorithm above may be interpreted as follows:

An adaptive automaton starts executing from an initial situation (Z_0, e_0, ω), meaning that the pushdown store is empty, the automaton is being fed with the complete input string ω, and the underlying state machine is E_0, at its initial state e_0.

At any instant, being the automaton in situation $(\gamma\pi, e, \sigma\alpha)$, the set of productions is searched for some matching production.

A production matches some situation when the current situation of the automaton corresponds exactly to the situation described in the left hand side of the production.

While searching for such a production one of the following conditions may occur:

- A single productions matches the current situation. In this case, the application of the matching production will be unconditional and deterministic.
- There are more than one matching production. In this case, matching productions must be classified, and all possibilities are tried non-deterministically, in parallel
  - transitions internal to a submachine take precedence over the others.
  - token-consuming productions take precedence over the others.
- No productions are found. Accept the input string if the current state is a final state and the input string has been exhausted leaving empty the pushdown store; reject it if none of the non-deterministic trials accepts the input string.

The following text describes how to use the productions in order to make transitions in the adaptive automaton.

The procedure below is performed if the current situation is matched by a production:

$$(\gamma g\ ,\ e\ ,\ s\ \alpha),\ A : \rightarrow (\gamma g'\ ,\ e'\ ,\ s'\ \alpha)\ ,\ B$$

- If A is present, the corresponding adaptive action will take place first.
- If the execution of A has deleted the current production, abort and start over the process for a new current production.
- If g is present, pop it from the pushdown store
- If g' is present, push it onto the pushdown store
- If s is present, consume it and let its right neighbor in the input string be the new current input symbol
- If s' is present, insert it just at the left of the current input symbol, and make it the new current input symbol
- Assign e' to the current state
- If B is present, the corresponding adaptive action will take place last

We complete this description with the syntax and semantics of adaptive actions.

Adaptive actions are interpreted following the definition of a corresponding adaptive function F, which are declared as tuples with components:

- F the name of the function
- τ_1 , τ_2 , ... , τ_p a list of formal parameters
- an optional list of variables
- an optional list of generators
- an optional call to a (anterior) adaptive action
- a list of basic actions defining inspections, inclusions, deletions of productions
- an optional call to a (posterior) adaptive action

Variables, parameters and generators are given symbolical names representing values to be used in the productions.

Each of these elements are initially undefined, taking at most a single read-only value for each execution of the function.

Variables allow referencing some non-constant value. Variables are assigned values as a result of the execution of inspecting or deleting basic adaptive actions.

Generators are similar to variables, but each time they are automatically assigned unique values at the start of the execution of the adaptive function they belong to.

Parameters start undefined when the adaptive functions starts execution, and are always input parameters, staying undefined until some value is attached to them, afterwards they remain read-only throughout the execution of the function.

Parameter-passing mechanism may also assign argument values, taken from the current function call, to the corresponding parameters.

Basic adaptive actions (inspection, deletion, inclusion) assume the general forms:

$$? [(\gamma g , e , s \alpha), A : \rightarrow (\gamma g' , e' , s' \alpha), B] \quad (1)$$

$$- [(\gamma g , e , s \alpha), A : \rightarrow (\gamma g' , e' , s' \alpha), B] \quad (2)$$

and

$$+ [(\gamma g , e , s \alpha), A : \rightarrow (\gamma g' , e' , s' \alpha), B] \quad (3)$$

where the expressions in brackets represent production templates to be respectively searched, deleted or added to the current set of productions of the automaton.

In the particular case of inspection actions, the set of productions will be searched for productions matching the given template.

Variables in the template will be filled with the corresponding values indicated in the production eventually found.

If no matching production exists, all variables used will remain undefined.

If more than one match exist (non-determinism) all matching instances are handled in parallel.

Basic inspection actions do not modify the set of productions in the adaptive automaton.

For basic deleting actions, proceed usually like in the case of basic inspection actions, and delete all matching productions from the set of productions.

If no matching occurs, do not remove any production from the set.

Basic adding actions add the indicated production to the production set if it is not yet there, otherwise do nothing.

The general form for adaptive actions A or B is

$$F (\varphi_1, \varphi_2, ... , \varphi_p) \quad (4)$$

where $\varphi_1, \varphi_2, ... , \varphi_p$ are p arguments passed to an adaptive function named F.

In parametric adaptive function calls the following conditions must be followed:

- use the same name F to refer to the function in an adaptive action A or B
- arguments $\varphi_1, \varphi_2, ... , \varphi_p$ correspond positionally to parameters $\tau_1, \tau_2, ... , \tau_p$
- each of the arguments $\varphi_1, \varphi_2, ... , \varphi_p$ may be either the name of any element of the adaptive automaton or a symbol of its input alphabet.

Anterior and posterior adaptive actions are usually denoted by calls to (often parametric) adaptive functions

Operation of adaptive automata

Adaptive automata are formal devices whose shape evolve, while accepting input sentences, by starting from a fixed initial form, and changing its set of productions by executing in sequence the adaptive actions attached to the transitions performed.

The evolution of an adaptive automaton is performed by steps, through the execution of *elementary adaptive actions* that

- search the current set of productions for productions matching a given template.
- change the set of productions by adding a new production to the set
- remove a specified production from the set of productions

The operation of deterministic adaptive automata are sketched in the following steps:

1. Search the current set of productions of the automaton for a production matching the current situation. If there is no such a production, reject the input string and stop execution.
2. From that production, determine the next situation of the automaton and the adaptive actions to be performed before and after the specified transition.
3. If there is an adaptive action to be performed before changing the situation of the automaton, perform it first.
4. If the adaptive action performed in step 3 erased the currently executing production, go back to step 1.
5. Change the current situation of the adaptive automaton according to the next situation extracted in step 2.
6. If there is an adaptive action to be performed after changing the situation of the automaton, perform it now.
7. If the new current situation is not a final situation, go back to step 1, otherwise accept the input string and stop.

Adaptive automata may be interpreted as evolving state machines with an underlying state-machine that start from a given fixed initial form, operates conventionally by performing either

- a sequence of non-adaptive transitions until reaching the end of the analysis or
- some adaptive transition that may change the adaptive automaton's set of productions by deleting existent transitions and adding new ones as needed.

So, an adaptive automaton may evolve to a new shape whenever adaptive actions are executed, afterwards resuming its operation by proceeding from an adequate restarting state in the new state machine.

Therefore, the recognition of an input string will be viewed as a recognition path, representing so many intermediate steps as the number of adaptive transitions performed by the adaptive automaton.

Each of these intermediate steps is concerned to the recognition of some input sub-string by means of

- a sequence of non-adaptive transitions, followed by
- a structural self-modification of the automaton, by executing an adaptive action.

Time-complexity analysis

A brief analysis is made in the following discussion, concerning some aspects of the time- and space-behavior of deterministic adaptive automata, which accept the sentences of their input language without any backtracking.

Non-deterministic automata, because of their characteristics, are of little interest as models that lead to efficient implementations, and are not considered in this discussion.

Deterministic adaptive automata accept any n-length sentence by means of:

- one transition for each token in the input sentence (n token-consuming transitions, each taking α units of time)
- one empty transition that push a symbol onto the pushdown store for each start of a nested construct in the input sentence (maximum $(n + 1) / 2$ transitions, each taking β units of time)
- one empty transition that pop off the pushdown store a previously pushed symbol for each end of a nested construct in the input sentence (maximum $(n + 1) / 2$ transitions, each taking γ units of time)
- one self-modifying call to an adaptive function for each yet inexperienced context-dependency detected in the input sentence (maximum n calls to adaptive functions, one per token).
- Assuming that, in the worst case, δ is the time response of the most lengthy adaptive action, $\delta . n$ will be the time wasted in adaptive transitions.
- So, the maximum total time wasted to accept an n-length sentence will be

$$\alpha \cdot n + (\beta + \gamma)(n + 1) / 2 + \delta \cdot n \qquad (5)$$

From this reasoning, we can conclude that any deterministic adaptive automaton accepts an n-length input string in a time $\mathcal{O}(n)$, provided that the time taken by any adaptive action is limited to some finite upper bound δ.

In the general case, time-response of adaptive automaton will depend essentially on the behavior of their adaptive actions: for adaptive actions with some upper bound time response, the time-response function of the adaptive automaton will essentially follow that boundary function.

Space-complexity analysis

About space behavior, a similar reasoning may be done:

- assume that the underlying state machine of the adaptive automaton has m productions in its initial production set.
- the total number of production in the set is given at any instant by adding the initial m to the total number of inserted production, and subtracting the total number of deleted productions
- assume that adaptive actions may insert at most P productions, so that P is an upper-bound function for the number of added productions
- in the worst case, an adaptive action will insert p new productions to the current production set without deleting any production, so p is the maximum value that P can assume.
- assume that adaptive actions may delete at most Q productions each, so that Q is an upper-bound function for the number of deleted productions.
- in any case, Q is limited: if no deleting actions are performed, Q will be zero, and in the extreme case of the deleting action, Q will assume the cardinality of the current set of productions, which is always finite. Consequently, Q is irrelevant for worst-case investigation of space-behavior of adaptive automata.
- so, for bounded functions P, in the worst case, each adaptive action performed will insert p new productions, leading to a final production set with $m + p \cdot n$ productions after accepting any n-length sentence.
- therefore, in the hypothesis that P is bounded, the dimension of the production set will also grow as $\mathcal{O}(n)$ function.
- for arbitrary adaptive actions, P may be unbounded, and in this case the set is not guaranteed to remain finite

Illustrating Example

The behavior of adaptive automata suggests their use as a model for implementing knowledge acquisition devices.

As an application of adaptive automata to model learning devices, a little illustrating example in syntax learning is presented in this paper hereafter.

This example sketches the use adaptive automata in learning the syntax of a simple regular language from a set of representative samples of the language.

Positive samples allow constructing an initial acceptor for a superset of the desired language, and additional negative samples allow finding restrictions to be imposed to the automaton in order to obtain a correct acceptor for the desired language.

Syntax Learning

Natural languages may be viewed as practical complex cases of context-dependent languages, so context-free notations are insufficient to denote them, despite the common practice to state context-sensitive languages in terms of a base context-free language and a set of externally defined context-sensitive restrictions.

So, full grammatical formalization of natural languages may be approximated, to some extent, by underlying context-free grammars, whose productions may be attached context-sensitive restricting rules.

In other words, from automata viewpoint, an acceptor for such a context-dependent language would be built as an underlying pushdown acceptor, corresponding to the context-free base grammar, and a set of function calls, attached to its transitions, representing restrictions associated to context-dependencies.

This arrangement may be very naturally implemented by adaptive automata, which have been conceived to have an underlying structured pushdown automaton, representing the base context-free language, and a set of adaptive actions attached to its transitions, intended to impose progressive syntactical restrictions to the underlying automaton along the recognition process.

Hence, adaptive automata may be easily employed to intuitively model devices performing concepts related to formal definition and processing of natural language syntax.

Furthermore, because of their intrinsic dynamic behavior, adaptive automata seem to be very suitable to conceptually model the mechanics of many important learning features that are present in syntax-learning problems.

Proposed Technique

One major issue that must be faced when dealing with natural language inference is how to infer a base language for the desired language.

One way consists in adopting an initial hand-made base language definition. Sometimes, however, such a definition is not easily available, suggesting the existence of a previous inference of a regular or a context-free language in the syntax-learning process.

In order to learn a basic underlying syntax from a set of sample sentences, one can start from an initial finite-state automaton which accepts no string at all.

As new sentences are processed, this automaton should be modified to represent increasingly more receptive acceptors, towards some final version that correctly accept all the sentences in the sample.

By simple generalization procedures, the resulting automaton may be transformed into a device that accepts a wider syntax including the classes to which the sample sentences belong.

It is well known that in many cases this procedure does not generate the correct automaton, leading to a representation of some superset of the language instead.

Better approximations of the correct automaton may be obtained from that version by successively restricting the language it is able to accept.

This effect may be achieved by narrowing back the syntax accepted by the automaton, by using for that purpose information extracted from a second set of sample strings, whose members do not belong to the language.

As new samples from this second set are processed, the automaton should change into increasingly more restrictive acceptors, towards some final version that correctly describes the desired language.

Obviously the success of the procedure described above depends heavily on the adequacy of the sample used.

Implementation with Adaptive Automata

By means of an adequately designed adaptive automaton, it is possible to perform such a task of constructing an acceptor for the desired language by means of the following steps:

- start by processing samples of the language, that induces the adaptive automaton to configure itself as a finite-state automaton that accepts all given samples.
- after a sufficient sampling is made, the particular tree-shaped finite-state automaton, just obtained as the underlying state-machine of the adaptive automaton, may be transformed into an equivalent more efficient automaton by the use of well-known optimization transforms.
- usual generalization rules may then be applied to the automaton obtained in the previous step, in order to get an acceptor to a more general superset of the desired language, by using less states and transitions
- further manipulation of the automaton may detect self-embedded constructs, and use them to perform another transformation, leading to a structured pushdown automaton instead of a finite-state automaton
- negative samples may be also used in order to impose restrictions to the automaton, by specializing the context-free language, until a final formal description of the desired language is obtained

Examples

In this section, two applications are presented of adaptive automata in syntax learning of regular languages.

For the sake of simplicity, we introduce a simple adaptive automaton, to illustrate the utilization of adaptive actions, since the application shown in this section involve a lot of steps that transform an initial automaton into a final acceptor representing as close as possible the expected language.

This automaton is better described in [Jos93].

- in [Jos93] it is shown how to implement context dependencies in adaptive automata
- A similar method allow to perform the equivalent operation from adaptive grammatical formal devices

- Asimple mapping procedure enables automatic conversion from adaptive automata to adaptive grammars
- there is an algorithm that allows automatically generating structured pushdown automata directly from context-free grammars [Jos93]
- a similar algorithm allows automatic generation of adaptive grammars from adaptive automata

Applications

In this section, two applications of adaptive automata in syntax learning of regular languages are presented.

For the sake of simplicity, we introduce a simple adaptive automaton to illustrate the functionality of adaptive actions.

Example of adaptive automata (name collector) - This example is about a name collector. It recognizes identifiers in the input string, classifies them as previously found or not, and modify the automaton that implements the language in such a way that it can recognize any identifier lately collected [Jos93].

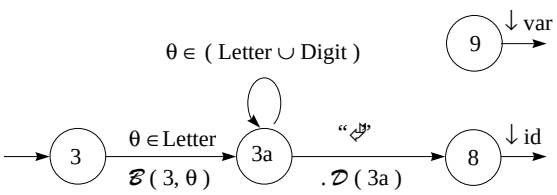

fig.1- name collector

Initial productions:

{ (3 , θ) : ℬ (3 , θ), → 3a ∀ θ ∈ Letter }
{ (3 , θ) : → 3a ∀ θ ∈ (Letter ∪ Digit) }
(3a , "Ã") : → 8 , 𝒟 (3a)
{ (γ z , 8, α) : → (γ , z , id α) ∀ z ∈ Γ }
{ (γ z , 9, α) : → (γ , z , var α) ∀ z ∈ Γ }

Adaptive functions:

ℬ (i, σ) : { j* :
+ [(i , σ) : → j]
{ + [(j , θ) : ℬ (j , θ), → 3a]
∀ θ ∈ (Letter ∪ Digit) }
+ [(j , "Ã") : → 8 ; 𝒟 (j)]
- [(i , σ) : ℬ (i, σ), → 3a]
}

𝒟 (i) : { :
- [(i , "Ã") : → 8 ; 𝒟 (i)]
+ [(i , "Ã") : → 9]
}

Algorithm to build a composed automaton from prefix- and suffix-automaton

A prefix-tree automaton is a tree-shaped automaton that deterministically accepts the finite language representing some positive sample of sentences of the language.

An auto-building prefix-tree automaton may be designed as such an adaptive automaton that its initial underlying machine shows only one state and one transition with an attached adaptive action (both the origin and the destination of this transition is the same single state, which represents the initial as well as the final state of this automaton).

This adaptive action allow the automaton to grow, driven by the sequence of input symbols, in such a way that at any end of accepted sentence it is a representation of the desired acceptor.

The resulting prefix-tree automaton is equivalent to a deterministic finite-state machine that recognizes the finite language corresponding to the set of all input sentences accepted so far.

Suffix-tree automata are also tree-shaped automata, and may be built similarly. However, suffix-tree automata are constructed in reverse order, from backward reading of the sentences in the positive sample of the language.

In order to enforce a single initial state, an additional auxiliary state is added as initial state, as well as empty transitions linking it to the starting state of all branches of the tree.

The resulting suffix-tree automaton will contain only one initial state and also only one final state.

In the drawings, adaptive actions have been omitted for clarity.

The composed automaton is built from the composition of the prefix-tree automaton with the corresponding suffix-tree automaton.

It recognizes a superset of the language accepted by those automata, including all sentences in the originating positive sample, and eventual further strings also.

The algorithm described below uses as input that pair of tree-shaped automata, and uses the following notation:

Ap - prefix-tree automaton	i - number denoted to the states of the Ap
As - suffix-tree automaton	j - number denoted to the states of the As and the Ac
Ac - composed automaton	p_o - a origin state of a transition in the prefix-tree automaton
σ, ρ - input symbol to be recognized	p_d - a destination state of a transition in prefix-tree automaton

Algorithm: Building of the composed automaton

Input: prefix- and suffix-tree automata for a single sample

Output: an acceptor for the language

Step 1: /* Initialization */
i = 1; j = 0; root of Ap = i; root of As = j;
in Ac: create an initial state; attach i to it;

Step 2: Let p_o be the current state in Ap. Traverse a transition, in Ap, from p_o.
Let p_d be the destination state and let σ be the symbol labeling this transition.
If p_d is null
then go to Step 17 /* to stop */
else if p_d has a number
then {$p_o = p_d$; go to Step 2;}
else {i = i+1; pd = i; $p_o = p_d$; p_d = NULL;}

Step 3: Traverse As since the root;
Search As for transitions labeled σ, departing from state i, with undefined destination state;
if (there exist such a transition)
then {select these transitions; j=j+1; go to Step 4;}
else go to Step 2;

Step 4: For all transitions found in step 3, attach j to the destination state;

Step 5: in Ac: /* activate the adaptive action */
Create a new state, attach j to it, and create a new transition labeled σ linking state j-1 to state j.

Step 6: /* find cycles */
Search As for transitions with label ρ, having destination state y number lower than origin state x.
if (there exist such a transition)
then {select these transitions; go to Step 7; }
else go to Step 16;

Step 7: Select a transition obtained in Step 6;
if (x = y) /* loop */
then go to Step 8;
else go to Step 11;

Step 8: in Ac: /* activate the adaptive action */
Search, in Ac, the state denoted by x;
Create a transition labeled ρ with both origin and destination states in x.

Step 9: Search for transitions, in Ap, with origin state x, label ρ, and undefined destination state;
if (there exist such a transition)
then {select these transitions; go to Step 10;}
else go to Step 16;

Step 10: For all transitions selected in Step 9,
attach y to the destination state;

Step11: Repeat steps 9 and 10 for all transitions in As, with destination state y, label ρ**,** and undefined origin state. In this case, attach x to the origin state. After performing this loop, go to Step 16;

Step 12: if (x > y)
then go to Step 13
else go to Step 16;

Step13: in Ac: /* activate the adaptive action */
Search, in Ac, the states numbered x and y;
Create a transition with origin state x, destination state y, and label ρ;

Step 14: Search Ap for transitions with origin state x and label ρ;
if (there exist such a transition)
then { select these transitions; go to Step 15; }
else go to Step 16;

Step 15: for all transitions selected in step 14, attach y to the destination state

Step 16: Repeat Step 7 until all transitions, obtained in Step 6, have been visited**,** then go back to Step 2;

Step 17: Stop /* all states are numbered */

First example: The first example infers the syntax of language L_1 = a b*c*d*.
The prefix- and suffix-tree automata will be built from positive sample

S+ = { a, ab, ac, ad, abc, abd, abcd, acd, abb, abbc, abbd }.

Figures 2 and 3 illustrate the prefix-tree and the suffix-tree automata built from the positive sample.

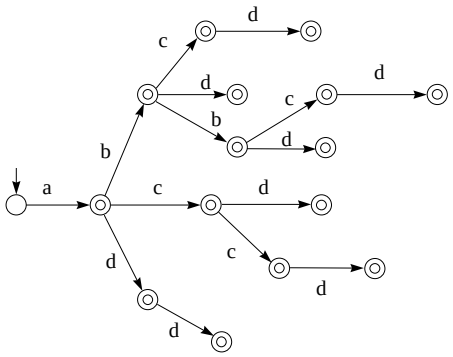

fig 2 - prefix-tree automaton

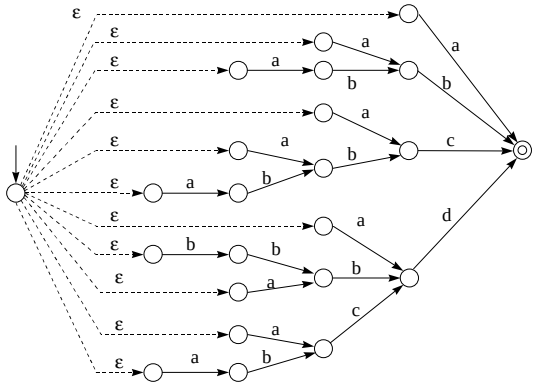

fig. 3 - suffix-tree automaton

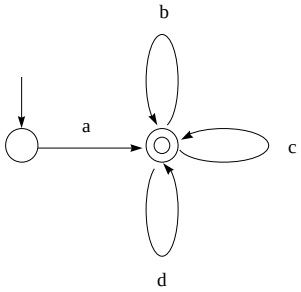

fig. 4 - composed automaton

Fig. 4 illustrates the composed automaton obtained by the application of the algorithm. This automaton is able to recognize the superset of strings of the language L_1.

Second example: This example is intended to infer regular language L_2 = ab (c | de*f) g* from positive sample
S+ = { ab, abc,abdf, abdeeef, abg, abggg, abdefdefccg, abcdefdfgg }

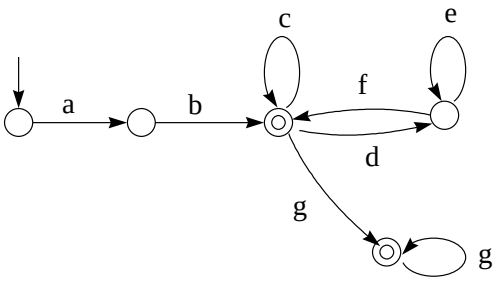

fig. 5 - positive automaton

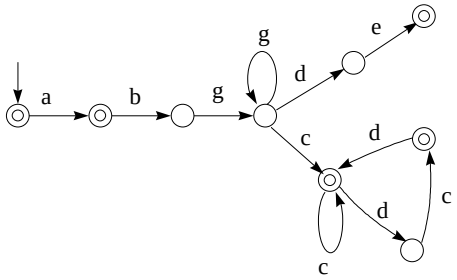

fig. 6 - negative automaton

In this case, the positive automaton recognizes a superset of language L_2. A negative sample of representative non-sentences helps changing the automaton to reject the corresponding syntax.

Note that token g precedes symbols c, d, e or f in all strings of the chosen negative sample.

An example of such a sample is

S- = { ε, a, abgc, abgcc, abgggc, abgde, abgcdcd }

Applying the same algorithm presented before, the resulting automaton is:

Submit the string simultaneously to both automata - positive and negative - and accept it if and only if it is accepted by the positive automaton and rejected by the negative automaton.

If speed matters, merge both automata into a single faster equivalent one.

This faster automaton may be easily constructed by applying classical methods.

Conclusions

It is possible and practical to use adaptive formalisms in order to make inferences, syntactical analysis and other tasks common in natural language processing.

The powerful features of adaptive automata - ease to express learning activities and potential high efficiency of the resulting automaton - may be used and explored with this practice, allowing the construction of efficient working devices for natural language processing.

References

[Dup94] DUPONT, P. Regular grammatical inference from positive and negative samples by genetic search: the GIG method. In: (ICGI'94) Lecture Notes in Artificial Intelligence, n.862, Springer Verlag, pp.236-245, 1994.

[Dup96] DUPONT, P. Incremental regular inference. In: (ICGI'96) Lecture Notes in Artificial Intelligence, n.1147, Springer Verlag, pages 222-237, 1996.

[Fu75a] FU, K.S.; BOOTH, T.L. Grammatical Inference: Introduction and Survey - Part I, IEEE Transactions on Systems, Man, and Cybernetics, v.5, n.1, pp.95-111, 1975; Part II, v.5, n.4, pp.409-423, 1975.

[Jos93] JOSÉ NETO, J. Contribuição à metodologia de construção de compiladores. São Paulo, 1993, 272p. Tese (Livre-Docência) Escola Politécnica, Universidade de São Paulo.

[Jos94] JOSÉ NETO, J. Adaptive automata for context-dependent languages. ACM SIGPLAN NOTICES. V.29, n.9, p.115-124, 1994.

[Par96] PAREKH, R.; HONAVAR V. An incremental interactive algorithm for regular grammar inference. In: (ICGI'96) Lectures Notes in Computer Science, Springer Verlag, vol.1147, pp.238-250, 1996.

[Par97] PAREKH, R.; NICHITIU, C.; HONAVAR V. A polynomial time incremental algorithm for regular grammar inference. Technical Report #97-03. Department of Computer Science, Iowa State University, Jan 17, 1997.

[Par97] PAREKH, R.; HONAVAR V. Learning DFA form simple examples. Proceedings of the Eighth International Workshop on Algorithmic Learning Theory (ALT'97), Sendai, Japan, Oct 6-7, 1997.

[Rub95b] RUBINSTEIN, R.S.; SHUTT. J.N. Self-modifying finite automata: An introduction, Information processing letters, v.56, n.4, 24, p.185-90, 1995.