Solving Complex Problems Efficiently with Adaptive Automata

João José Neto

Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
Av. Prof. Luciano Gualberto, trav. 3, n. 158
CEP 05508-900 - Cidade Universitária - São Paulo - SP - Brasil
e-mail: jjneto@pcs.usp.br

Abstract - Adaptive technologies are based on the self-modifying property of some systems, which give their users a very powerful and convenient facility for expressing and handling complex problems. In order to accomplish self-modification, systems must be instructed on what exactly to modify, the correct place such an alteration is expected to occur and when adaptive actions have to take place. Therefore, one may turn a rule-based classic formalism into a corresponding adaptive one by allowing adaptive actions to be attached to its rules. In this paper we focus our attention on adaptive automata, an adaptive formalism based on structured pushdown automata whose transitions are allowed to hold adaptive actions responsible for self-modification activities. Then, we show adaptive automata-based solutions to several significant problems, and sketch performance comparisons to traditional solutions.
Keywords: self-modification, learning devices, adaptive automata, adaptive technology.

Introduction

Adaptive technology refers to the use of techniques and devices which are expected to react to their input by autonomously modifying their own behavior. Although implicitly used far ago, many adaptive devices, in particular automata [Shu95, Rub95, Jos98a] and grammars [Bur90, Chr90, Shu93] have been reported in the litterature.

Adaptive techniques have recently been given much closer attention and specific development, as testified by current events in the field of adaptive and reconfigurable computing.

The main idea around this concept lies on the ability adaptive devices must have of executing *adaptive actions*, which are self-modification procedures intended to be activated as reactions of adaptive formal devices to the occurrence of special events or situations.

In particular, *adaptive automata* [Jos94] constitute a particular class of adaptive devices whose subjacent traditional non-adaptive formalism is the pushdown automaton. When an adaptive action is attached to some transition of the pushdown automaton, such a transition is called an *adaptive transition*. When an adaptive transition is executed, that means some special condition has been detected. In response to the event, the attached adaptive actions are activated.

Adaptive actions may be designed to handle the occurrence of expected but not yet considered situations detected in the automaton's input. In response, the set of transitions of the automaton is modified by the adaptive action, so, after the current transition takes place, the automaton's shape (and consequently its behavior) will be correspondingly changed.

Consequently, a sequence of incoming transitions will be executed in the just updated environment, until it evolves again as a consequence of executing another adaptive transition, which will start over another sequence of transitions, and so forth.

It has been shown that adaptive automata are Turing-powerful devices [Jos93]. The resulting generality of the model, its learning capability due to its self-modification feature as well as the strength of its expressiveness make adaptive automata a very attractive device suitable for expressing complex facts and to handle difficult situations arisen when searching for computational solutions to complex problems.

In this paper, we address some rather difficult problems and show adaptive automata-based solutions to them: (1) efficient recognition of general palindromes, (2) acceptors for context-sensitive languages, (3) acceptors for anagrams on finite sets of symbols, (4) acceptors for asynchronously-merged languages, (5) acceptors for ambiguous languages, (6) simultaneous acceptor for a set of languages, (7) simultaneous detection of sub-strings which are sentences of any language in a set of languages.

Adaptive Automata

Structured pushdown automata are state machines composed by a set of finite-state-like mutually recursive sub-machines. Adaptive automata are structured pushdown automata whose state-transition rules may be attached adaptive actions. The expression below,

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

represents the general form of a rule in an adaptive automaton. The left-hand side of the expression refers to the current configuration of the adaptive automaton before the execution of the state transition, whereas its right-hand side encodes its configuration after the state transition.

The components of the 3-tuples encode the situation of the pushdown store, the state and the input data, respectively. After the 3-tuples, adaptive actions are optionally specified: the left-hand one represents modifications to be applied before the state transition, while the right-hand one specifies the changes to be imposed to the automaton after the transition.

Adaptive actions are calls to parametric *adaptive functions*, which may be roughly regarded as collections of *elementary adaptive actions* to be applied to the transition set of the automaton.

Parameters are special variables, used in adaptive functions, to which actual values (*arguments*) are assigned whenever the adaptive function is called.

Elementary adaptive actions are the only editing operations supplied by the formalism. Three different operations are allowed: inspection, deletion and insertion of transitions. The expression,

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

denotes any elementary adaptive actions by replacing the operator $\otimes$ by **?** for the inspection, **+** for the insertion and **-** for the deletion of a transition having the shape enclosed in brackets.

Note that both inspection and deletion elementary adaptive actions search the current set of transitions for any transition matching the given pattern. When such a transition is found, the *variables* used in place of any of the components of the elementary adaptive action are assigned the actual corresponding values in the matching transition. When used in an inspection or deletion elementary adaptive action, variables become either *undefined* (in the case no match was found) or *defined* (otherwise). Anyway, no further assignments to used variables are allowed.

An additional feature is also present in the model: the concept of *generator*. Generators are used to assign names to newly created states. They are also special variables, which are automatically assigned unique values at the start of the execution of an adaptive function, together with the assignment of argument values to the parameters. Again, once assigned, both generators and parameters are not allowed to change anymore.

More details on the formal aspects of adaptive automata may be found in [Jos94, Jos93].

A sampler of adaptive solutions for classical problems

The following text presents adaptive solutions for some rather difficult problems from the theory of computation, compiler construction, artificial intelligence and others. For each problem, classical solution approaches, based on usual automata are commented, alternative adaptive solutions are proposed and a comparison is done between them.

Although adaptive automata have been presented before in terms of an algebraic notation, the following examples will be denoted semi-formally through a graphical version of the automata. Adaptive actions will be informally presented through their functional description, explained in English.

Efficient Recognition of General Palindromes

The problem of accepting general palindromes is usually addressed through solutions based on non-deterministic algorithms [Lew81]. Palindromes are symmetrical strings over an alphabet. The main drawback in accepting such context-free languages is the difficulty in locating the center of the sentence in a single search.

Although pushdown automata are suitable to accept palindromes, non-determinism is needed when general palindromes are allowed. So, trial-and-error and backtracking mechanisms are essential for handling general palindromes in the general case.

An adaptive solution for this problem consists of reading the full input string and building two sequences of transitions, one for reconstructing the input string, and other for consuming the reconstructed reverse input string. The proposed adaptive automaton, depicted in figure 1,

- initializes the reconstructing machine and the consuming one.
- reads in the next symbol from the input string.
- adds to each sequence an adequate new transition corresponding to the symbol read.
- if this is not the last symbol in the input string, goes back to read the next symbol, otherwise executes the reconstructing sequence, then the consuming one, rejecting the input string if the final state of the consuming sequence is not reached, and accepting it otherwise.

Note that for any palindrome with length n the automaton takes n adaptive transitions to read the palindrome the first time. For each symbol read, one new transition is added to each auxiliary machine. After reading all symbols in the palindrome, both auxiliary sequences are ready for use, and their execution will take n trasnitions each. So, taking into account a fixed number (k) of additional control transitions, the total number of transitions taken by the automaton for accepting any n-length general palindrome is 3n+k, and the total transitions added to the original automaton is 2n. We conclude that space- and time-complexity for this automaton are both O(n). By construction, the proposed adaptive automaton operates deterministically, in contrast with the non-deterministic conventional automata usually required to solve the same problem.

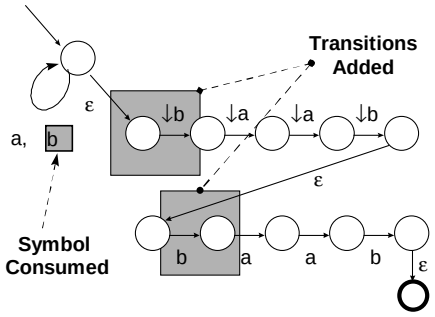

Figure 1 – An adaptive automaton for general palindromes

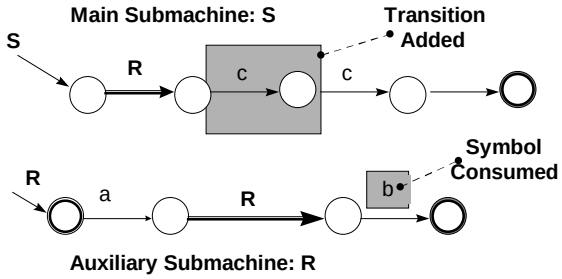

Figure 2 – An adaptive implementation for language $a^nb^nc^n$ with an underlying structured pushdown automaton with two sub-machines: **S** (main) and **R** (acceptor for the context-free prefix a^nb^n)

Acceptors for Context-Dependent Languages

The next example deals with the recognition of context-dependent languages. It is a well-known fact that no finite-state and no pushdown automata have power enough to accept context-dependent languages, and classical solutions for this problem are usually posed in terms of Turing machines or other equivalent devices.

In these languages there are constructs in which the presence or the absence of specific language elements or constructs in some part of the input text may either allow or impose restrictions to the occurrence of corresponding syntactical constructs anywhere else in the sentence being handled.

A very frequent instance of such a situation occurs in programming languages [Pag81, Slo95], where many context-dependencies may be found, such as: (a) collecting names used in a program, (b) delimiting scopes for identifier names, (c) associating types to names according to the context, (d) matching the use of identifiers to their declaration, (e) checking the types associated to identifiers for coherence with their use within the program, (f) checking the number of arguments and their types in a subroutine, function or macro call against the corresponding parameters stated in their declaration, (g) declaring and expanding lexical macros, (h) declaring and expanding syntactical macros.

In this paper there is no space for details on each of the situations above, then we illustrate the topic with two adaptive automata for the classical context-sensitive language $a^nb^nc^n$, $n \geq 0$. Note the simplicity of both the structure, the operation and especially the performance of the adaptive solutions proposed.

One way to accept this language by means of an adaptive automaton is similar to what we showed in figure 1, above. The language may be accepted in three parts: one for a^n, other for b^n and other for c^n. An adaptive automaton may be constructed as follows: in an initial state the prefix a^n is consumed. For each symbol in this prefix, two corresponding additional transitions, consuming b and c respectively, are added to the (initially empty) sections accepting b^n and c^n. Therefore, after consuming a^n, the other sections will be able to accept b^n and c^n, respectively. Then, an empty transition from the initial state to the first state of the second section, and other from the final state of the second section to the first state to the third one will connect the automaton. The final state of the third section is the final state of the adaptive automaton.

For this adaptive automaton, the resulting complexity analysis is very similar to the one we showed before for general palindromes.

Another implementation for this language is illustrated in figure 2. Strings $a^nb^nc^n$ are recognized in two parts: First, the prefix a^nb^n is accepted by the recursive submachine R as a context-free language. Then the suffix c^n is consumed by the transitions on c in submachine S, added by an adaptive action driven by R each time a symbol b is found. Initially, S does not have any transition on c.

Space complexity analysis for this automaton is trivial: its total number of transitions after accepting the sentence will be O(n), considering the constant number of its initial transitions, plus the k=n/3 added transitions.

For time complexity, in order to accept any sentence of the form $a^kb^kc^k$ with n=3k, the automaton executes one initial call to submachine R, then k times the following: one transition on a, another recursive call to R, one transition on the corresponding b and one return transition to the calling submachine; finally, on the return to S, k more transitions on c are performed, and finally an empty transition to the final state is executed.

Therefore, the total number of transitions consuming symbols will be 3k=n; the total number of calls to R will be k+1=1+n/3; the number of the corresponding returns from R will be the same; the total number of empty transitions will be one. So, we have n+2(1+n/3)+1 transitions altogether, so the total cost is again O(n), despite the different costs associated to the various types of transitions.

It is easy to prove that rejection of input strings not in the language is also O(n) for this automaton.

Acceptors for Anagrams on a Finite Sets of Symbols

The adaptive automaton presented in this section intends to accept anagrams on a given set of symbols $A=\{a_1, a_2, ...a_n\}$. Obviously this is a simple problem that has a trivial classical solution through a tree-shaped finite-state automaton from whose depth-k non-final states emerge n-k transitions to depth-k+1 states, each consuming a symbol in A not consumed by any of the transitions in the path from the initial state of the automaton to the state under consideration. Such an automaton is both deterministic and minimal, and its performance is optimal. However, it is very space-consuming, for its total number of transitions is n(n-1)(n-2)...3.2.1=n!

Our proposed adaptive automaton, as shown in figure 3, operates as follows: in an initial state, any symbol in A is consumed and the next state is the same initial state; any of these transitions are associated to an adaptive action **A** responsible for executing two operations: the elimination of the just executed transition from the adaptive automaton, avoiding its re-execution, and the elimination of a corresponding counting transitions in the path from the initial state to the final state of the automaton. Each of these counting transitions are attached an adaptive action **B**, which eliminates the special empty transition linking the sequence of counting transitions to the final state of the automaton.

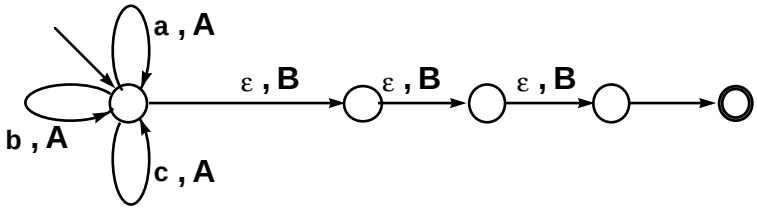

Figure 3 – Adaptive automaton accepting any anagrams to the set {a, b, c}. Adaptive action A removes the transition it is attached to and one of the empty transitions with B. Adaptive action B eliminates the rightmost empty transition, avoiding the automaton to reach its final state.

It is obvious that, for sets with n elements, the automaton requires only 2n+1 transitions instead of the former n! transitions calculated above. So, the space required for this adaptive automaton is O(n).

By definition, all sentences have length n. For any sentence, the automaton executes one transition for consuming each symbol. Each of these transition activates the adaptive action **A** , which in turn executes the following: one self removal, and one removal of an empty transition associated with **B** , one more for unlinking the transition following the removed one and a last one for re-linking it to the initial state of the automaton. Altogether, this adaptive action performs four editing operations on the automaton's set of transitions.

After consuming the full input string, the automaton will have eliminated all empty transitions associated to adaptive transition **B**, so it will execute only one more empty transition to reach the final state. Hence, the total cost for recognizing any sentence will be the cost of n transitions for consuming the symbols in the input string, plus the cost of executing 4.n elementary additions or removals of transitions, plus one final empty transition. The automaton's time response is therefore O(n) in this case, too.

Note that in case of shorter sequences, the input string will be exhausted before all transitions calling **B** are eliminated. Then, following its way towards the final state, the automaton will at least execute the adaptive action **B** once. **B** solely executes the elimination of the only transition that gives access to the final state, so avoiding the acceptance of the input string.

In case of longer sequences, some already used input symbol will be found for the second time in the input string, but at this time the corresponding consuming transition no more exists, so the sequence will also be rejected. Note that the time taken by any rejections is also O(n).

Acceptors for Asynchronously-Merged Sequences

The following case illustrates the adaptive implementation of languages defined as a composition of two or more other languages by asynchronously merging their sentences. For languages with non-disjoint input alphabets, non-determinism is unavoidable, so in this case it is not practical to use this method directly.

Classical solutions for this problem must consider all possible combinations of the sentences to be merged. As with many other combinatorial problems, even for a small number of short sequences, the expected number of sentences in the language is very large, requiring a finite-state automaton with a huge number of states and transitions for its acceptance.

With adaptive automata, it is easy to implement compact and efficient automata for this language. Figure 4 shows an adaptive automaton that accepts any merge of two short sequences, abcd and efg.

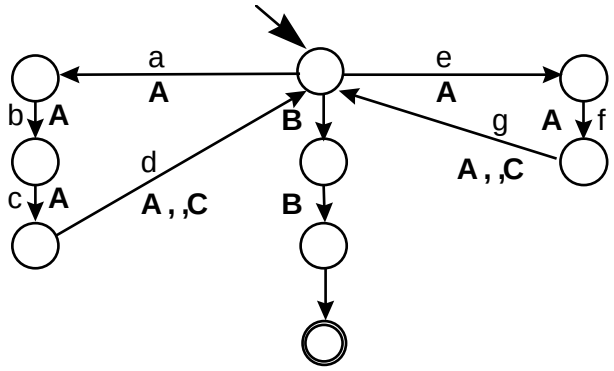

Figure 4 – Adaptive automaton accepting any merges of two sentences abcd and efg. Adaptive action A is self-removing. Adaptive action C removes one of the transitions calling B. Adaptive action B avoids accepting the input by eliminating the single way to the final state.

for each sequence to be merged, we build, departing from a common initial state, a chain of transitions consuming the symbols in the sequence. The transition consuming the last symbol of the sequence returns to the common initial state.

All these transitions activate the adaptive action A, which removes the corresponding transition from the automaton, then disconnects the next transition from the chain and connects it to the initial state of the automaton. This procedure avoids the same symbol to be accepted more than once.

The transition consuming the last symbol in the sequence will execute adaptive action C as well, which deletes one of the counter empty transitions executing adaptive transition B. This one operates exactly as mentioned in the preceding section, and the counting represents in this case the number of complete sequences already consumed from the input string.

The operation of the automaton is simple: only the next input symbol allowed shows an alternative of transition departing from the initial state. Each symbol consumed by the automaton deletes the corresponding transition, and connects to the initial state the next transition, that will expect for the next allowed symbol in the corresponding sequence.

The state of the automaton will be set to its initial state after the execution of those self-deleting transitions, so that the following input will be expected to be one of those that are expected at the initial state at any moment in the recognition of the sentence.

Upon the input text is exhausted, the automaton will leave its initial state, following the way to the final state. If any transition with B is found in this way, the adaptive action B will remove the only transition allowing the automaton to accept the sentence. Otherwise, that empty transition will be followed, leading to the final acceptance state.

Naturally, the size of this automaton is very small, and corresponds to the sum of the sizes of the automata accepting the individual sequences to be merged, plus one counting transition per sequence (anything in the range [1, n]), plus one final empty transition. Therefore, its space demand is O(n), in contrast to the expected combinatorial size of the corresponding classical automaton.

Following the operation of this automaton, we observe that n transitions are taken to consume all symbols in an n-length input string, and n corresponding adaptive actions A are performed, each eliminating two transitions and adding one. Additionally, one adaptive action C is executed for each sequence to be merged. In the best case, there is only one sequence, and the adaptive action C will be executed only once. In the worst case, all sequence are of length one, and we will have n executions of C in this extreme case. These executions will also remove two transitions and add one each. The total expected number of transitions to be added or removed will be, in the best case, 3n+3, and in the worst case, 3n+3n=6n.

The total cost of the recognition of a sentence by this automaton is made up by the cost of the n+1 transitions executed by the automaton and that associated to the 3n+3 up to 6n modifications to the set of transitions, imposed by adaptive actions. Anyway, this cost is again O(n) for this case. Therefore, this linear space and time behavior renders the proposed model very practical and effective, in contrast to corresponding traditional combinatorial solutions.

Acceptors for Ambiguous Languages

Accepting ambiguous languages may be considered in several ways, according to how the acceptor is expected to be used. So, if we are interested in any possible interpretation for a sentence, then usual automata may be constructed that arbitrarily choose some particular interpretation and ignores the remaining ones. This effect is achieved by selecting one of the possibilities and discarding the others when more than one correct choice exists at some decision point in the accepting process.

If we are interested in building the set of all possible interpretations, then the acceptor must consider all options whenever some non-deterministic choice is done, in order to collect all recognition paths, which corresponds to collect all possible interpretations for the sentence.

Traditional sequential solutions for this problem often use classical backtracking techniques in the implementation of non-deterministic choices.

If any arbitrary solution is enough, then non-deterministic choices are reduced to selecting one of the available options and discarding the remaining ones. The resulting solution will depend on the selection policy, which may be chosen according to the user's particular needs.

If all possible solutions are required, non-deterministic choices will correspond to nodes in a multi-branch choice tree. Sequential implementations require traversing such a decision tree, and solutions are found whenever a leaf is reached in a depth-first traversal. Parallel implementation change non-determinism into parallelism. Breadth-first traversal is usual in this case, and each path may be processed in a separate processor. Hybrid implementations are also often chosen, in which more than one path are handled in each available processor in a multiprogramming environment, simulating a large number of virtual processors working in parallel.

For the case in which we choose to accept any solution, in the most favorable situation, we have the deterministic case, in which only one choice will exist at any node of the decision tree. The number of tests required to determine the correct path will be the summation of the number of tests performed at each non-leaf node, that lies in the range [1, m] where m is the highest number of tests needed to decide among the available choices at any node all over the tree. In the worst case, the number of choices is the product of all individual number of choices in the nodes found in a path. For a decision tree of depth k, the total number of tests will be in the range k.[1, m]=[k, m.k], where k and m are constants. Therefore, for a given k, there is a constant p=m.k that represents the (linear) upper limit for the number of tests in this case.

For the other case, all valid paths must be identified, and the number of possibilities will be the product of the number of possible choices at each node in the path. If the number of choices at each non-leaf node is in the range [1, m], then for a decision tree with depth k, the total number of choices will be, in the worst case, m^k. For any path the cost of traversing one step in the tree is the cost of testing enough options to find the correct one. The number of required tests lie in the range [1, m]. So, for a tree with depth k, we will need, for any path, a total number of tests in the interval [k, m.k], so he total cost of finding all the m^k paths will lie in the interval m^k.[k, m.k]. So, in the worst case, $k.m^{k+1}$ tests are needed. In the best case, $k.m^k$ tests are enough. Therefore, as expected, the complexity of this solution is exponential. In other words, the solution goes worse with the growth of the number of chained non-deterministic choices.

Simultaneous Acceptor for a Set of Languages

In this example, we address the problem of simultaneously accepting multiple languages. A single input string is to be accepted by a single automaton, which checks it against all languages in a given set. For simplicity, we assume that deterministic automata are given for all languages.

The simplest idea consists of building an automaton that accepts the union of the desired languages, allowing very efficient acceptors to be constructed this way. However, this approach has a serious drawback: once a sentence is accepted by such a device, no information is available on the language or languages that sentence belongs to.

Another approach consists of performing separately the recognition of the sentence by each individual automaton associated to the various languages involved. This approach has also its own problems: for very long strings, for which a mismatch occurs near their tails, too much computational effort is wasted before starting a new trial against the next language to be tested.

Obviously a rather straightforward and effective approach is to use a parallel solution, so that each language is handled by a different processor working on its own copy of the input string. Although this seems to be the ideal solution for this problem, it may seldom be implemented in practice, for most usual computer installations are still based on low-cost sequential processors.

Simulation is an attractive alternative which uses multiprogramming to approximate the power of parallelism on sequential processors. As it would be expected, the portability of this solution is poor, for it is highly dependent on the features of the underlying operating system environment.

Our adaptive proposal for the solution of this problem will use this approach within the automaton itself, taking advantage of all good features of the latter approach without making it machine- or operating system-dependent. That is achieved by simulating the multiprogramming effect inside the automaton.

In our proposal, sketched in figure 5, an initial state holds a set of transitions with both source and destination states at this initial state. Each of these transitions corresponds to one of the symbols in the language's alphabet, and besides consuming it, it also forces the execution of a special adaptive action A whose role is to simulate the handling, by each of the automata accepting the elements in the set of desired languages, of the symbol that has just been consumed.

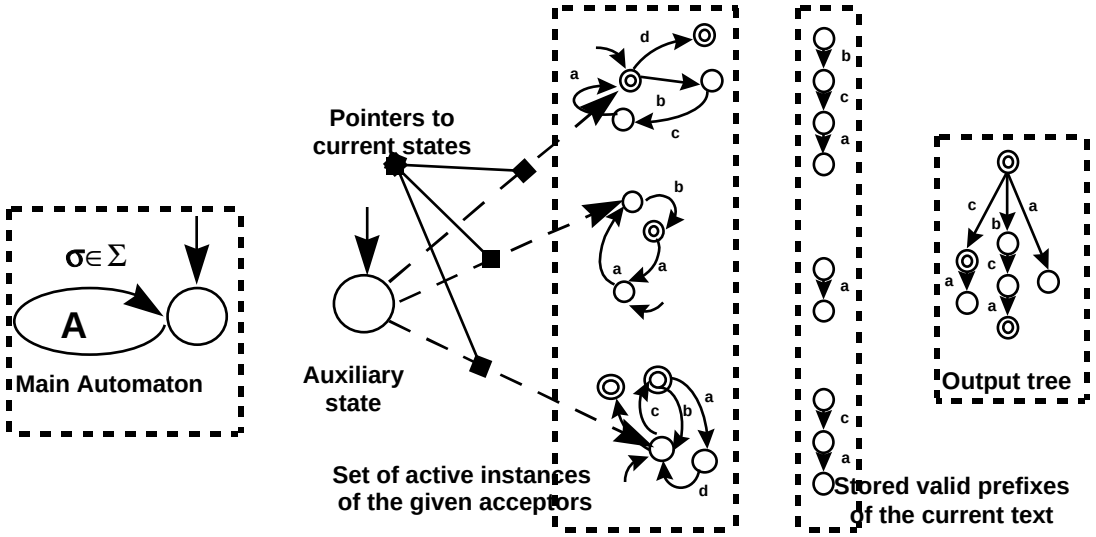

Figure 5 - Simplified scheme of an adaptive automaton implementing an acceptor for multiple languages

The operation of adaptive action A is as follows: when a new symbol is read in at the initial state of the automaton, this action is executed. A special auxiliary state is used for holding a set of empty transitions pointing the simulated current state of all automata (represented as sub-machines) that accept the languages in the set of desired languages.

For each of these pointers, the corresponding transition is initially removed from the automaton, and other transition is put in its place by the adaptive action A, pointing the state the corresponding automaton would reach if it were fed with the symbol just read in. In case of rejection of the input sequence, the pointer to the current state of the corresponding automaton is discarded. After repeating this procedure for all languages, the execution of A is over, and a new symbol is read in.

This process is repeated until the input stream is exhausted. Then, an adaptive action B is executed which tests all current states pointed by all remaining special empty transitions, then discards any transition not pointing a final state, and executes a transition to the final state of the adaptive automaton, leaving pointers to the final states of all sub-machines that would have accepted the input string.

For an input alphabet with k symbols, the proposed automaton has k transitions at its initial state, plus one to reach the final state. Additionally, the auxiliary state will hold at most p pointer transitions to the simulated current states. Finally, we must add the total number of transitions in the given p automata. That gives $k+1+p+m_1+m_2+...+m_p$, where m_i is the number of transitions of the automaton associated to the i-th language. This is an upper limit. As a lower limit we have $k+1+m_1+m_2+...+m_p$ transitions in case of rejection of the input string by all p automata. Note that both limits are constant and do not depend on the length of the input string.

In order to perform its operation on a sentence of length n, this automaton will perform n transitions with attached adaptive action A at its initial state, then one empty transition with adaptive action B . The total number of transitions executed is therefore n+1. In addition to the transitions, there are n executions of adaptive action A , plus one execution of B.

In order to calculate the contribution of A , in the case of p languages, we must consider that each of the p times adaptive action A is executed, it performs one removal and at most one insertion of the transition pointing the current state of an automaton corresponding to some different language (no insertions will occur in case of rejection). Hence, the total number of elementary actions performed by each execution of A is in the range [1, 2]. The total contribution of this adaptive action will be in [p, 2.p]. Adaptive action B will execute at most p tests and p removals. Therefore, the total number of elementary adaptive actions performed by B is in the range [0, 2.p].

We conclude that the total number of elementary adaptive actions performed by A and B will be in the interval [p, 4.p], so the cost of this simultaneous language recognition for an n-length sentence will be the cost of executing n+1 transitions plus the cost of executing at least n.p and at most 4.n.p elementary adaptive actions. Since p is constant in this problem, this cost is O(n).

Simultaneous Sub-string Detection in a Set of Languages.

An interesting and useful variant of the preceding problem refers to the detection, within a given string, of all occurrences of substrings corresponding to sentences in any language of a set of languages. This is a rather complex problem in the sense that it repeats the solution of the preceding one once for each new symbol read in, for each symbol potentially starts a new sentence. In addition, the same symbol may start not only one, but several sentences of different lengths, for any sentence may be a prefix of others. So, we may expect that any solution for this problem will be quite expensive.

The first problem may be addressed by starting the recognition process on new instances of the acceptors for each new symbol read in. For context-free and regular languages, the acceptors do not change as they operate, so we may use multiple pointer transitions to the current state associated to each input substring, pointing states on the same automaton. For context-sensitive languages, adaptive actions may modify the shape of the automaton, so whenever a new symbol is read in, a new copy of each automaton must be created in order to allow the recognition of the new substring.

If we are expected to collect all sentences recognized from the input string, a tree-shaped automaton may be constructed such that any path from its initial state to any of its final states collects the symbols constituting the desired sentences.

If we must have the sentences classified by language, then one of such tree-shaped automata has to be constructed for each language, so that any new sentence detected in the input string is immediately added to the tree associated to the corresponding language.

So, for each symbol for which a consuming transition is simulated for some language, the corresponding path is created or updated in the corresponding tree-shaped automaton. Whenever a

final state is reached in the simulated transition, we consider that the end of a new sentence is reached, and a corresponding final state is set in the tree-shaped automaton as well.

Conclusion

Adaptive automata are formal devices that have the same theoretical power of Turing Machine, in the sense that they are able to represent complex context-sensitive languages. Furthermore, any computational phenomena that may be described by context-sensitive languages may be handled by that device as well.

In this work, we have shown through a few significant application examples how adaptive automata may be employed as very powerful tools that have a strong potential to represent practical, efficient, compact and elegant solutions for several very complex practical problems.

The behavior of adaptive automata as piecewise finite-state- or structured pushdown-automata render them easy to understand and very adequate as effective implementation models.

The informal complexity analysis made for the application examples above has shown that a good portion of the addressed subjects showed to be very effectively handled by adaptive automata, whose behavior may be considered, in a significant majority of cases, far better, in both space and time aspects, than usual equivalent solutions making use of classical techniques.

The adaptive automata-based solutions studied in this paper are rather good solutions for a set of problems with a significant degree of computational or implementation complexity, and showed to be excellent implementation models for their class of problems.

These and many other results encourage investment on the subject of this research. Parallel works are now in progress at our institution exploring adaptive automata as programming paradigms, as computation models, as language implementation models, etc. In a near future we expect to have designed and implemented a full high-level adaptive paradigm language system based exclusively on adaptive automata, starting from its grammatical conception, including context-dependent aspects, run-time environment and the semantics of its dynamic behavior.

References

[Bur90] Burshtein, B. Generation and Recognition of Formal Languages by Modifiable Grammars SIGPLAN Notices 25 n.12, Dec. 1990, pp. 45-53

[Chr90] Christiansen A Survey of Adaptable Grammars SIGPLAN Notices vol. 25 n. 11, Nov. 1990, pp.35-44

[Jos93] José Neto, J. Contribuições à Metodologia de Construção de Compiladores Livre-Docência post-doctoral thesis,, Escola Politécnica da USP, 1993. In Portuguese.

[Jos94] José Neto, J. Adaptive automata for context-dependent languages ACM SIGPLAN Notices, 1994.

[Jos98a] José Neto, J., Almeida Jr., J.R., Santos, J.M.N. Synchronized Statecharts for Reactive Systems Proceedings of the IASTED International Conference on Applied Modelling and Simulation, p. 246-251 Honolulu, Hawaii, 1998

[Lew81] Lewis, H.R., Papadimitriou, C.H. Elements of the Theory of Computation Prentice Hall, 1981

[Pag81] Pagan, F.G. Formal Specification of Programming Languages: A Panoramic Primer Prentice-Hall, 1981

[Rub95] Rubinstein, R.; Shutt, J.N. Self-modifying finite automata - Basic Definitions and results. Technical Report WPI-CS-TR-95-2. Worcester Polytechnic Institute, Worcester, Massachusetts, August, 1995, 14p.

[Shu93] Shutt, J.N. Recursive Adaptable Grammars Ph D. Thesis, Worcester Polytechnic Institute, 1993

[Shu95] Shutt, J.N. Self-modifying finite automata - Power and limitations. Technical Report WPI-CS-TR-95-4. Worcester Polytechnic Institute, Worcester, Massachusetts, December, 1995, 32p.

[Slo95] Slonneger, K. and Kurtz, B.L. Formal Syntax and Semantics of Programming Languages - a Laboratory-based Approach Addison Wesley, 1995