An Adaptive Finite-State Automata Application to the problem of Reducing the Number of States in Approximate String Matching

Ricardo Luis de Azevedo da Rocha[1] **, João José Neto**[1]

[1]Laboratório de Linguagens e Técnicas Adaptativas – Escola Politécnica – USP
Av. Professor Luciano Gualberto, Trav. 3 N. 158
CEP 05508-900 São Paulo – SP – Brasil

luis.rocha@poli.usp.br,joao.jose@poli.usp.br

Abstract

This paper presents an alternative way to use finite-state automata in order to deal with approximate string matching. By exploring some adaptive features that enable any finite-state automaton model to change configuration during computational steps, dynamically deleting or creating new transitions, we can actually control the behavior and the topology of the automaton. We use these features for an application to approximate string matching trying to reduce the number of states required.

Keywords

adaptive devices, approximate string matching, finite-state automata, complexity, application

Workshop

VI WORKSHOP DE AGENTES Y SISTEMAS INTELIGENTES (WASI)

1. Introduction

Finite-state automata are a significant part of the theory of computation, and the models built from them are widely used in areas such as protocol specification, hardware design, and so forth. Despite the fact that such models are simple to create, they define the class of regular languages (type 3 languages) [8].

The use of simpler models to build complex structures is the main goal of some efforts, such as in [6, 11]. Those researches have in common the use of a basic model, and through some features, make them able to perform much more complex actions. Such as, to accept strings from type 0 languages [6]. The original adaptive automaton is based on a pushdown automaton [6], and the self-modifying automaton is based on the finite automaton.

We will use a simplified version of the adaptive automaton in our proposal, which is based on the finite automaton [7]. This paper addresses an application to the problem of approximate string matching using the adaptive finite automaton model.

The problem of approximate string matching may be directly addressed to finite-state automata. Since the early 1970's some algorithms have been developed to deal with it, such as the Knuth-Morris-Pratt [1], or Boyer-Moore [2], and Aho-Corasick [3]. All of them are based on finite-state automata principles.

The subsequent effort raised new ways of implementation, and some new kinds of models [10], but the problem remains unchanged:
Given a text string $T = t_1 t_2 \dots t_n$, a pattern $P = p_1 p_2 \dots p_m$, and an integer k, $k \leq m \leq n$, we are interested in finding all occurrences of a substring X in string T such that the distance $D(P, X)$ between the pattern P and X is less than or equal to k. $D(P, X)$ is some given function that quantifies how close P and X are, e.g. by counting unmatching symbols.

In the literature we find several ways for measuring such distance, e.g. the Hamming distance, where it is allowed to replace a character by another one, or the Levenshtein distance, where it is allowed to delete a character from the pattern or to insert a character into the pattern (the generalized Levenshtein distance allows two characters to be exchanged) [10].

The Hamming distance between two strings P and X, of equal length, may be defined as the number of positions with mismatching symbols in those strings. The Levenshtein distance between two strings P and X, not necessarily of equal length, may be defined as the minimal number of editing operations (insert, delete, and replace) needed to convert P into X [10, 4]. In this paper we will use the Hamming distance measure.

Some improvements have been achieved in the running time and the actual 'size' of the automaton along the process. Some researchers described new ways to implement automata models in order to get better performance on their algorithms. In the next section we will take into account a reduced model of finite automata, as developed in [4].

2. Finite-State Automata for String Matching

A nondeterministic finite-state automaton is a 5-tuple $M = (K, \Sigma, \delta, q_0, F)$, where K is a finite set of states, Σ is a finite set of input symbols, δ is a state transition function from $K \times (\Sigma \cup \{\epsilon\})$ to 2^K, $q_0 \in Q$ is the initial state, $F \subseteq K$ is the set of final states, [9].

Let us consider the example shown in figure 1, where there are five states in a nondeterministic finite-state automaton to exactly match any string ended with the sequence

$p_1p_2p_3p_4$. If we need an approximate string matching, the nondeterministic finite-state automaton should be like the one shown in figure 2.

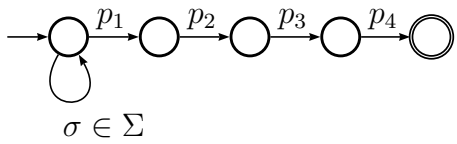

Figure 1: Nondeterministic Finite Automaton.

Figure 2 shows an automaton built for Hamming distance of $k = 3$, which allows exact matching at the first path, one mismatch at the second path, and so forth. At the end of each path there is a final state, which identifies the number of mismatches found in the string.

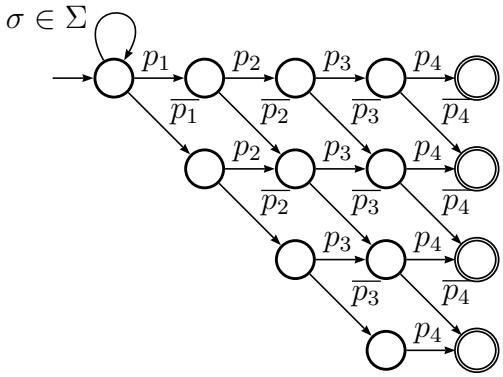

Figure 2: Nondeterministic Finite-State Automaton for Approximate String Matching of the String accepted by the automaton in Figure 1.

For the Levenshtein distance a similar automaton is slightly more complex, because it requires two further edit operations on the exact original string.

The research developped by Holub [4] has shown a way to reduce the number of states in such nondeterministic finite-state automaton, created to fit a given distance function (Hamming distance of $k = 3$). Holub's automaton is constructed without the need to identify the number of mismatches in the string, so there is only one final state, and each path may be significantly reduced, as shown in Figure 3.

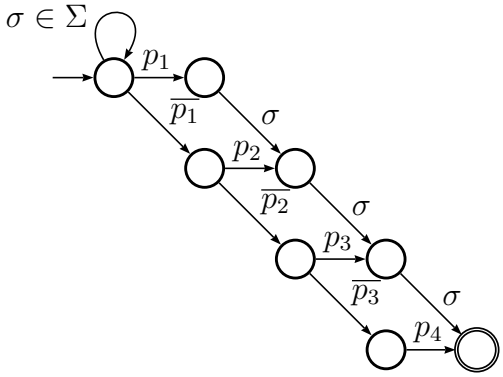

Figure 3: Holub's Nondeterministic Finite-State Automaton for Approximate String Matching.

In the next section we will use a different strategy, exploring dynamic features

added to the automaton in order to achieve a better result on the number of states, by dynamically keeping in the automaton only the strictly required states and transitions.

3. Adaptive Features

The main result shown by Holub in [4] is achieved by including in the automaton only the states strictly needed for handling the condition. By applying this idea and by creating the needed states at the execution time, we may significantly reduce the total number of states in the automaton.

By doing so we may use the guidelines in [5], where we find a formal introduction to adaptive devices, and explore the concept of general rule-driven devices which may be tailor-made just to fit our specific needs. From this point of view, we are able to create an automaton that is able to modify its own configuration according to the particular needs of the problem currently being solved.

Following [5] let us consider the nondeterministic finite automaton in figure 2 as our starting point for the creation of an adaptive finite-state automaton in the following way: $M_{sm} = (M, AM)$, where M_{sm} stands for adaptive finite-state automaton, M represents the original finite state automaton, and AM describes the mechanism that modifies the shape of the configuration of the automaton.

Let AA be a fixed set of actions (including a special *null adaptive action* a^0). AM contains a set of rules that may change the topology of the automaton. Let AR be the set of all possible sets of such rules for M_{sm}. In this case, any particular action $a^k \in AA$ maps the current set of rules $AR_t \subseteq AR$, defining the automaton into some other set of rules $AR_{t+1} \subseteq AR$, which will be the next set of rules defining the automaton until another action takes place.

$$a^k : AR \rightarrow AR$$

The mechanism AM associates each transition $t_i \in \delta$ of the original finite automaton to a corresponding pair of actions ba^p and $aa^p \in AA$:

$$AM \subseteq AA \times NR \times AA$$

This way we define an *adaptive rule* ar^p associated to a rule (transition) nr^p of the finite automaton as a 3-tuple in AM as:

$$ar^p = (ba^p, nr^p, aa^p)$$

For each of its moves the adaptive automaton M_{sm} applies the chosen rule ar^p in three steps:

- Activation of an adaptive action ba^p before applying the rule nr^p,
- Application of a rule (transition) nr^p, and,
- Execution of an adaptive action aa^p after the application of nr^p.

Transitions of the adaptive finite-automaton are defined this way: when the automaton is in state q^p and will perform a rule (transition) $nr^p = (ba^p, nr^p, aa^p)$ which leads to state s^{p+1} and modifies the set of states and transitions, there is an activation of the adaptive action ba^p, then the rule (transition) is performed and changes to state s^{p+1}, and then the adaptive action aa^p is performed, and we show those actions by $q^p \vdash s^{p+1}$.

The main feature of the adaptive finite-state automaton is its ability to apply modifications to the initial automaton, in order to change its original structure (states and transitions) in order to face the particular needs of the current problem.

Proposition 1 *The adaptive actions may be placed in order to change an automaton before or after the rule (transition).*

Proof: Following [7], and by induction on the adaptive steps. Suppose we want only to apply adaptive actions before the application of a rule (the proof is similar for the other case). We will create an additional step to each adaptive one which has an adaptive action after the application of a rule. We have that $q^p \vdash s^{p+1}$, and the transition is $ar^p = (ba^p, nr^p, aa^p)$, at state q^p

Base: $ar^p = (ba^p, nr^p, aa^p)$, making $p = 0$ we have two cases:
First supposing we have no adaptive steps, so there is no adaptive action to apply, which means $ar^0 = (ba^0, nr^0, aa^0) = nr^0$, and $q^0 \vdash s^0$.
For the second case suppose $p = 0$, and we have at least one adaptive action at step 0, so beginning from the initial step 0 we have $q^0 \vdash s^1$, and $ar^0 = (ba^0, nr^0, aa^0)$, so we need to create another adaptive step between the application of the rule nr^0 and aa^0), indeed we can apply:
$ar^0 = (ba^0, xnr^0, \epsilon) \overset{additional\ step}{\rightarrow} (aa^1, nr^1, \epsilon)$, where ϵ stands for an empty action. But then we need another state, a new one created by action xnr^0, say state x^1. So after changing the adaptive action ba^0 to xba^0 in order to delete the original transition $q^0 \vdash s^1$ with $ar^0 = (ba^0, nr^0, aa^0)$ and create two others in its place, but keeping the other actions of ba^0: $q^0 \vdash x^1$, and $x^1 \vdash s^2$, where the transition from q^0 to x^1 is described as $ar^0 = (xba^0, bnr^0, \epsilon)$, and from x^1 to s^2 as (aa^1, anr^1, ϵ), where rules bnr^0 and anr^1 act as the original action nr^0, but with an additional step.

Hyp.: $ar^q = (ba^q, nr^q, aa^q)$, $\forall q$ such as $0 \leq q \leq p$ can be split in two, where there are no adaptive actions after the transition of both.

Step: $ar^{p+1} = (ba^{p+1}, nr^{p+1}, aa^{p+1})$ As we have the application of the rules in steps, there must have been p steps of the form $ar^p = (ba^p, nr^p, aa^p)$, and each of these steps fall into the inductive hypothesis. The last step can be split in two just as before (in base case, where $p = 0$ with 1 adaptive step):
$ar^{p+1} = (xba^{p+1}, bnr^{p+1}, \epsilon) \overset{additional\ step}{\rightarrow} (aa^{p+2}, anr^{p+2}, \epsilon)$

$\diamondsuit$

Adaptive finite-state automata were designed from classic finite-state automata [8, 9], and may be viewed as a set of finite-state automata grouped together by the possibility of each one be replaced by some other one by changing the topology of the currently used one.

In this scenario, for deterministic languages, each state machine operates as a finite-state machine, except when handling complex (non-regular) constructs, when further accesses to the adaptive rules are needed in order to handle context-free or context-sensitive constructs, which are not handled by finite-state automata alone [5].

Adaptive finite-state automata may then be viewed as finite-state automata with the added ability to modify their current structure, e.g., to inspect the original model, and then to create new transitions and/or states, or to delete some of its existent transitions and/or states. With such feature, adaptive finite-state automata achieve the same computational power of Turing machines [6].

Therefore, adaptive finite state automata represent state machines (here we use the same concept of [8], in which any oriented graph that can symbolically represent, by its transitions or arrows, the behavior of a device, program, and so forth), which starts its operation as a simple finite-state machine, but as long as adaptive actions take place, some structural changes are applied to the initial model, modifying the set of states and

transitions of the original machine. Those changes are generated while performing the transitions during the operation of the machine, when an adaptive transition is found [5, 6].

As shown in Proposition 1 adaptive actions may be adequately placed in order to change an automaton only once, before or after the transition. By doing so the number of steps is decreased, as well as the number of actions needed. Table 1 shows the list of simple adaptive actions that can be used to build an adaptive function.

+: Action to add a new transition
-: Action to delete a transition
?: Action to search the set of transitions defining the current instance of the automaton for some given transition

Table 1: List of adaptive Actions

4. Application to approximate string matching using an adaptive finite-state automaton

When an adaptive finite-state automaton is in operation, states and transitions can be deleted from or added to the current state machine, but this can only happen if the transition that will occur in a step has attached adaptive actions designed to face the needs devised from the input string analysis. Such changes in the configuration of the device build a new state-machine, which replaces the current one. Then an additional step takes place, and so on. Such operation may be interpreted as path traversal in a state-machine space. As stated in [5], adaptive finite-state automata can be viewed as an extended version of the formalism for finite-state automaton.

So if we want to keep a maximum distance of k, we will need k adaptive actions, in order to count the number of mismatches between the string being searched for and the string being scanned. At each step, whenever a difference is found, an adaptive action takes place which changes the automaton creating a new state and by changing the set of adaptive (counting) actions. In figure 4 there is an example for $k = 3$, where the adaptive transitions are represented in gray, and they will only take place when they are actually needed. The adaptive transitions have one adaptive function to be taken after the transition to another state, symbolized by $p_i : \mathcal{A}$, where p_i stands for the symbol scanned, and $\mathcal{A}$ stands for an adaptive function named $\mathcal{A}$.

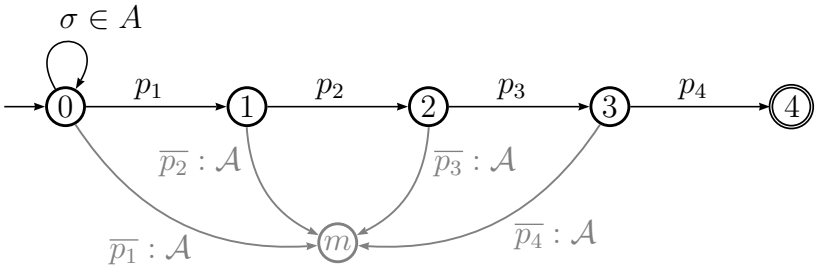

Figure 4: Adaptive Finite-state Automaton.

In figure 4 the adaptive transitions were drawn in gray, and they have more information than the others. For instance the arrow connecting state 1 to state m illustrates the adaptive transition of the automaton, the first symbol of the transition shows the actual transition symbol, and, after the comma we may find the adaptive actions, one before the colon (the action taken before the transition), and the other after the colon (the action

taken after the transition). We may perceive that the transitions in gray have actions attached to them that are expected to be performed after the transition takes place, meaning that action $\mathcal{A}$ changes the automaton afterwards the transition is performed.

That action does not create a new state, but changes all transitions with $\mathcal{A}$ as their attached action, replacing it by a call to another action, say $\mathcal{B}$, in order to update the distance being measured. But since we have $k = 3$, two other similar actions are needed.

Figure 4 also shows that the effect of creating or deleting transitions are to be taken into account when needed, so the overhead is not linear on the size of the text. It is necessary to perform a more accurate study on its effect.

As an example taken from figure 4, consider the string $p_1 p_k p_3 p_4$, where $p_k \neq p_i, 1 \leq i \leq 4$. This string is processed by the automaton leading to the configuration of figure 5, after processing substring $p_1 p_k$.

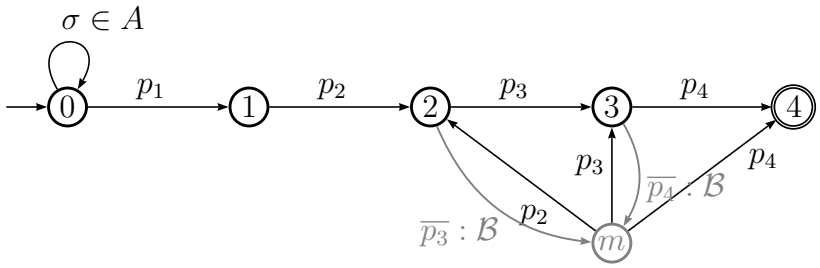

Figure 5: Adaptive Finite-state Automaton processing $p_1 p_k p_3 p_4$ after $p_1 p_k$.

As there is no need to create or delete states, only to create or delete transitions between the initial set of states, the set of states becomes fixed, and a fixed length state-transition table may be constructed. The transitions without adaptive actions can be performed in a simple fashion. To distinguish an adaptive transition from the others, it is only needed to mark them inside the transition table.

Any marked transition means an adaptive one, so after it had been performed the list of marked transitions is changed to another shape according to the current adaptive action, and then the next adaptive action, if present, is also changed.

In the example of Figure 4 there are three adaptive actions, namely $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$, which are defined as:

$$
\begin{aligned}
\mathcal{A} = [& \\
& + (m, p_2)\epsilon : \epsilon, \epsilon \rightarrow (2) \\
& + (m, p_3)\epsilon : \epsilon, \epsilon \rightarrow (3) \\
& + (m, p_4)\epsilon : \epsilon, \epsilon \rightarrow (4) \\
& - \{\forall x, y : (x, y)\mathcal{A} : \epsilon \rightarrow (m)\} \\
& + \{\forall x, y : (x, y)\mathcal{B} : \epsilon \rightarrow (m)\}] \\
\mathcal{B} = [& \\
& + (m, p_2)\epsilon : \epsilon, \epsilon \rightarrow (2) \\
& + (m, p_3)\epsilon : \epsilon, \epsilon \rightarrow (3) \\
& + (m, p_4)\epsilon : \epsilon, \epsilon \rightarrow (4) \\
& - \{\forall x, y : (x, y)\mathcal{B} : \epsilon \rightarrow (m)\} \\
& + \{\forall x, y : (x, y)\mathcal{C} : \epsilon \rightarrow (m)\}] \\
\mathcal{C} = [& \\
& + (m, p_2)\epsilon : \epsilon, \epsilon \rightarrow (2) \\
& + (m, p_3)\epsilon : \epsilon, \epsilon \rightarrow (3) \\
& + (m, p_4)\epsilon : \epsilon, \epsilon \rightarrow (4) \\
& - \{\forall x, y : (x, y)\mathcal{C} : \epsilon \rightarrow (m)\}]
\end{aligned}
$$

The action $\mathcal{A}$ just defined creates a set of non-adaptive transitions, (because we have ϵ before and after the colon), in order to return to the original (and correct) path.

Action $\mathcal{A}$ deletes all self-reference adaptive transitions, and creates a new set of transitions that change the deleted ones into corresponding self-reference adaptive transitions calling $\mathcal{B}$.

Therefore any other mismatch will perform a similar task with action $\mathcal{B}$, changing it into $\mathcal{C}$, and so forth, until the last calling level, when the string is finally rejected.

This sort of sequence of actions creating transitions which call other actions, and replace the existing transitions, allows the automaton to control and 'count' existing mismatches.

If a distance k is needed, we will have to consider k different adaptive actions, so that every action will delete itself when executed, and create another action, the next in sequence, in order to replace it. This way, we may keep track of the distance, as mentioned before, by 'counting' the number of actions already performed.

This sort of behavior has some advantages, we can significantly reduce the number of control states needed. Actually there must be one control state, as showed by the example of figure 4, but it can become clearer and more manageable if we create more than one state. Splitting the mistakes from the states, in order to keep a different state to each k. Doing so allows us to just 'connect' or 'disconnect' existing transitions, becoming easier to manipulate the state-transition table.

5. Conclusions

This paper presented an alternate way to reduce the number of states of a finite-state automaton, when dealing with approximate string matching. The actual 'size' of the automaton model can be reduced to a level very close to the one corresponding to exact string matching.

In fact, the number of initial states is the same of the finite-state automaton for exact match, plus one state. At each mismatch found, the transitions of the automaton are replaced by others, in order to keep control of the number of mismatches, until the number of mismatches reaches k, then no other mismatch will be accepted (because the adaptive transitions are replaced by simple transitions).

However, such a reduction has a price: we will need to replace the transitions at execution time (deleting the existing, and creating others to take their place), despite the fact that it will only be executed when needed. But if we create one state to each level of mistake (k states at all), we will be able manipulate the state-transition table much more easily. In this paper we have not addressed directly the time-complexity of the solution, but we intend to do it.

There are other research studies taking into account this adaptive feature, and we noticed that there are results and practical simulations that uses it, such as [11]. However, they share a common focus, they are connected to the processing of context-free or context-sensitive languages [6].

References

[1] Knuth, D. E., Morris, J. H., Pratt, V. R., Fast pattern matching in strings. *SIAM J. Comput.* **6, 2** (1977) 323-350.

[2] Boyer, R. S., Moore, J. S., A fast string searching algorithm. *Comm. ACM* **20, 10** (1977) 62-72.

[3] Aho, A. V., Corasick, M. J., Efficient string matching: an aid to bibliographic search. *Comm. ACM* **18, 6** (1975) 333-340.

[4] Holub, J., Reduced Nondeterministic Finite Automata for Approximate String Matching. *Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University* (August 1996) 19-27.

[5] J. J. Neto, Adaptive Rule-driven Devices - General Formulation and Case Study. *CIAA 2001: Proceedings of the 6th International Conference on Implementation and Application of Automata* (2001).

[6] J. J. Neto, Adaptive Automata for Context-Depend-ent Languages. *ACM SIGPLAN Notices* **29, 9** (Sep-tember, 1994) 115-124.

[7] J. J. Neto; C.A. B. Pariente, Adaptive Automata - a revisited proposal. *CIAA 2002: Proceedings of the 7th International Conference on Implementation and Application of Automata - Lecture Notes in Computer Science* **2608** (July, 2002) 158-168.

[8] J. Hopcroft, and J. Ullman, Formal Languages and their Relation to Automata. Addison-Wesley. 1969.

[9] H. Lewis, and C. Papadimitriou, Elements of the Theory of Computation. Prentice-Hall. 1998.

[10] Navarro, G., A guided tour to approximate string matching. *ACM Computing Surveys* **33, 1** (March, 2001) 31-88.

[11] R. Rubinstein, and J. N. Shutt, Self-modifying finite automata. *Proceedings of the 13th IFIP World Computer Congress, 1994* (1994) 493-498.