Nordic Journal of Computing

STRING-MATCHING WITH AUTOMATA

Mehryar Mohri
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, U.S.A.
E-mail: mohri@research.att.com

Abstract. We present an algorithm to search in a text for the patterns of a regular set. Unlike many classical algorithms, we assume that the input of the algorithm is a deterministic automaton and not a regular expression. Our algorithm is based on the notion of failure function and mainly consists of efficiently constructing a new deterministic automaton. This construction is shown to be efficient. In particular, its space complexity is linear in the size of the obtained automaton.

Key words: Finite automata, pattern-matching, strings.

CR Classification: F.1.1, F.2.0, F.2.2, F.4.3

1. Introduction

Pattern-matching consists of finding the occurrences of a set of strings in a text. Two general approaches have been used to perform this task given a regular expression r describing the patterns. Both require a preprocessing stage, which consists of constructing an automaton representing the set described by the regular expression A^*r, where A is the alphabet of the text. This automaton is then used to recognize occurrences of the patterns in a text t (see [Aho *et al.* 1986], [Aho 1990], and [Crochemore and Rytter 1994] for a general survey of string-matching).

The first approach aims to construct a non-deterministic automaton, while the second yields a deterministic one. The choice between these methods depends on time-space tradeoffs. In general, the construction of a non-deterministic automaton corresponding to A^*r is linear in time and space ($O(|r|)$), but its use in recognition is quadratic ($O(|r| \cdot |t|)$). The preprocessing in the deterministic case, namely the construction of the automaton, is exponential in time and space ($O(2^{|r|})$), but the recognition of the patterns in a text t is then linear ($O(|t|)$).

In the particular case where the set of patterns is finite, more efficient algorithms have been designed. Aho and Corasick [1975] gave an algorithm (AC for short) allowing one to find occurrences of n patterns P_i ($0 \leq i \leq n$) in a text in linear time ($O(|t|)$). Their algorithm is based on an efficient construction of an automaton recognizing ($A^*(P_1 + \cdots + P_n)$) represented

Received December 1995. Revised February 1997.

with a failure function. It can be considered as a generalization of the well-known algorithm of Knuth *et al.* [1977] used in string-matching to multi-pattern matching. The complexity of the construction of the automaton required in AC is linear in time and space in the sum of the lengths of all patterns, more precisely in $O(\log |A| \cdot \sum_{i=1}^{n} |P_i|)$, where A is the alphabet of the patterns. Commentz-Walter [1979] gave an algorithm which is the extension of the Boyer-Moore algorithm [Boyer and Moore 1977] to the case of a finite set of strings. The complexity of her algorithm is quadratic though more efficient in practice than AC for shorts strings [Aho 1990]. Crochemore *et al.* [1993] gave a linear time version of this algorithm combining the use of the AC automaton with that of a suffix automaton.

The input to all these algorithms is a regular expression describing the set of patterns to search for. Here, we are concerned with the problem of searching for a set of patterns in the case where this set is directly given by a deterministic automaton. Although in many practical uses, such as those supported by tools in the Unix operating system, regular expressions are a very convenient way of representing the patterns, there are many other applications, especially in natural language processing [Mohri 1997], in which such a representation does not seem appropriate. Indeed, in those applications the deterministic automaton representing the patterns may have been obtained as a result of complex preprocessing. Besides, in most applications to that field the size of automata exceeds a hundred states and can reach millions. Thus, users cannot conveniently provide the corresponding regular expressions as the input of a pattern-matching algorithm.

In the following, we describe algorithms that help search a text for the patterns given by a deterministic automaton G. More precisely, we indicate how to construct a deterministic automaton representing the language $A^*L(G)$, where $L(G)$ represents the language recognized by G. This automaton can be used to determine whether t contains a pattern of $L(G)$. Our representation of this automaton is such that it also helps to identify easily the occurrences of the patterns found in t. We do not impose any preprocessing of the text. There exists a sublinear average time algorithm for searching for any regular expression in a preprocessed text represented by a Patricia tree [Baeza-Yates and Gonnet 1989].

Constructing a deterministic automaton representing $A^*L(G)$ from G can be done by adding loops labeled with elements of the alphabet A to the initial state and then use the classical powerset construction to obtain a deterministic automaton [Aho *et al.* 1986]. However, the size of the alphabet A can be very large in some applications and this may lead to deterministic automata with a huge number of transitions. In some natural language processing problems, for instance, A is the size of the whole dictionary of inflected forms of a natural language. It may reach several hundreds of thousands for languages such as English or French. Moreover, in other applications, such as error correction, the alphabet itself may not be known because it could depend on the text under consideration. Thus, one would like to provide a representation of the deterministic automaton that is independent of the alphabet and such that its number of transitions be as small as possible.

The problem of an efficient determinization of automata has been discussed by Perrin [1990] and Crochemore and Rytter [1994]. In case G represents a single string, one can provide a linear time linear space deterministic automaton representing $A^*L(G)$ using failure functions [Knuth *et al.* 1977]. In the following we extend the use of failure functions to the general case. In some cases the number of states of the minimal automaton representing $A^*L(G)$ is exponential in the size of G. Hence, in what follows we are mainly concerned with complexities depending on the size of the resulting automaton. Notice that typical examples of such blow-up cases are those corresponding to languages such as $(a+b)^*a(a+b)^n$, so they are of the shape $A^*L(G)$.

We first present an algorithm to compute the desired deterministic automaton from an acyclic automaton G, hence one representing a finite set of patterns. This algorithm can be considered as an extension of the AC algorithm to the case of automata. We then generalize the algorithm to deal with any deterministic automaton G.

2. Case of acyclic automata

We consider here a deterministic automaton $G = (V, i, F, A, \delta)$ with set of states V, initial state $i \in V$, set of final states $F \subseteq V$, A a finite alphabet and δ the state transition function mapping $V \times A$ to V. For any $p \in V$, we denote by $Trans[p]$ the set of transitions leaving p, and for any t in $Trans[p]$ we denote by $t.v$ the state reached by t, and by $t.l$ its label. E stands for the set of transitions of G. In this section, we assume that the automaton G is acyclic.

2.1 Algorithm

Our algorithm is close to the AC algorithm. Indeed, as in that algorithm we define a failure function s associating with each state q of G the longest proper suffix of the strings leading to q that is also prefix of the strings accepted by G. However, since we deal with automata, several distinct strings may reach the same state q. These strings might have longest proper suffixes, prefixes of strings of $L(G)$, that lead to different states when read from the initial state. To deal with such cases and make it possible to define s, we gradually transform the initial automaton G. Each state q is duplicated as many times as necessary such that we have a single possible default state associated with each state.

Fig. 1 and Fig. 2 illustrate this construction in a particular case. At each state of the automaton of Fig. 2, the first number refers to q and the second one following the slash to the default state $s(q)$. The state 2 of G has been duplicated so as to take into account the different possible default states (0 and 1). The use of the resulting automaton in recognition is the same as

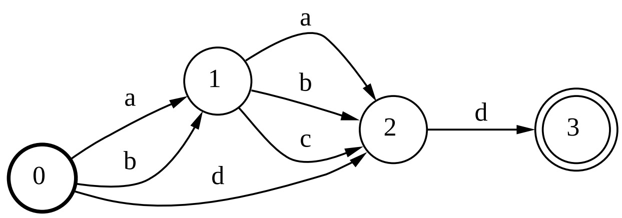

Fig. 1: Automaton G.

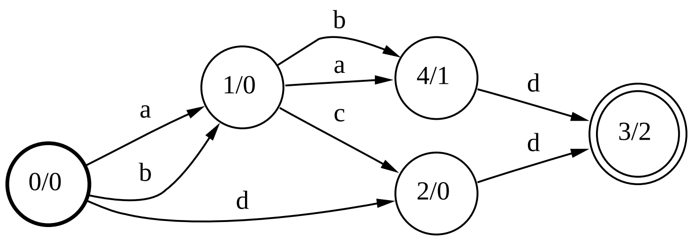

Fig. 2: Deterministic automaton representing $A^*L(G)$.

usual except that at a given state q ($q \neq i$), if the input symbol a corresponds to none of the transitions leaving q, a failure transition is made, that is the current state of the automaton becomes the default one $s(q)$.

In order to compute the failure function and duplicate states whenever necessary we use a breadth-first traversal of G so that states of level l be visited before those of level $l+1$. This is motivated by the fact that for $q \neq i$, the level of $s(q)$ is strictly less than that of q. More precisely, using the definition of s it is easy to prove the following:

$$\begin{array}{lll} \forall q, & d[q] \leq 1, & s[q] = i \\ \forall q, & d[q] \geq 2, & s[\delta(q,a)] = i \text{ if } (\forall k,\ \delta(s^k[q],a) \text{ undefined}) \\ \forall q, & d[q] \geq 2, & s[\delta(q,a)] = \delta(s^k[q],a) \text{ otherwise,} \\ & & k \text{ minimum such that } \delta(s^k[q],a) \text{ is defined} \end{array} \quad (1)$$

where $d[q]$ denotes the level of q. Thus, a breadth-first traversal guarantees that $s(q)$ can be computed following this process. Fig. 3 gives a pseudocode for the algorithm ACYCLIC-MATCHER computing a deterministic automaton recognizing $A^*L(G)$ from G.

We denote here by UNDEFINED a constant different from all states of G. A first-in first-out queue Q is used for managing the set of states to visit at each step according to a breadth-first search. The level of each state is computed and stored in the array d. The level computation is only useful in the case of cyclic automata examined in the next section.

```
ACYCLIC-MATCHER(G)
for each p ∈ V
  do  s[p] ← UNDEFINED
      f[p] ← p
s[i] ← i
d[i] ← 0
Q ← {i}
while Q ≠ ∅
  do  p ← head[Q]
      for each t ∈ Trans[p]
        do  q ← s[p]
            while (q ≠ i) and (δ(q, t.l) not defined)
                do      q ← s[q]
            if (p ≠ i) and (δ(q, t.l) defined)
                then    q ← δ(q, t.l)
            if (s[t.v] = UNDEFINED)
                then    s[t.v] ← q
                        d[t.v] ← d[p] + 1
                        if (q ∈ F)
                          then    F ← F ∪ {t.v}
                        LIST-INSERT(list[t.v], t.v)
                        ENQUEUE(Q, t.v)
            else if (there exists r ∈ list[f[t.v]] such that s[r] = q)
                then    t.v ← r
            else if (f[q] ≠ t.v)
                then    r ← COPY-STATE(t.v)    ▷ copy of t.v with same
                                                  transitions using f
                        s[r] ← q
                        f[r] ← f[t.v]
                        d[r] ← d[p] + 1
                        if (f[t.v] ∈ F)
                          then    F ← F ∪ {r}
                        LIST-INSERT(list[f[t.v]], r)
                        t.v ← r
                        ENQUEUE(Q, r)
            else    t.v ← q
      DEQUEUE(Q)
```

Fig. 3: Algorithm for the construction from G_2 of a deterministic automaton for $A^*L(G_2)$, the acyclic case.

The algorithm gradually modifies G by computing the default states at each step and by duplicating states whenever necessary. The duplication is performed by the function Copy-State which creates a new state q' copy of q with the same transitions as those q originally had. Since transitions leaving q may have been changed because of other previous duplications, for each state q we need to keep track of the original state of G which q is a copy of. This is done through the function f. Initially, $f[q] = q$ for all q since no copy has been done. So, when copying q, function Copy-State creates a new state q' with transitions $\delta(q', t.l) = f[t.v]$ for each $t \in Trans[q]$. In order to limit duplications to what is actually necessary, the list of duplicated states of each state q is stored in $list[q]$. Only if none of the elements of $list[q]$ has the desired default state is the state q duplicated. Notice also that in case the default state is a copy of q itself no duplication is necessary (condition of line 24). Besides copying transitions, Copy-State makes the new state a final state if the original state is final.

Theorem 1. *Let G be an acyclic deterministic automaton. This algorithm,* Acyclic-Matcher, *computes correctly a representation of a deterministic automaton recognizing $A^*L(G)$.*

Proof. The loop of lines 7-34 corresponds to a dynamic breadth-first traversal of G. Each state p is enqueued exactly once in Q and corresponds to one or more prefixes of strings of $L(G)$. Since the automaton is acyclic, the number of these prefixes is finite. Thus, the loop of lines 7-34 terminates. The loop of lines 9-33 is executed exactly $out-degree(p)$ times for each state p.

Lines 10-14 of the algorithm compute the default state q for each state $t.v$ reached by a transition from state p as previously described. The termination of the loop of lines 11-12 is ensured since for $p \neq i$ the level of $s(p)$ is strictly less than that of p.

The algorithm terminates since all loops do. Final states of the resulting automaton are either final states of the initial automaton, or those whose default states are final (lines 18-19 and 29-30). Hence, they correspond exactly to those paths from the initial state to these states which have a suffix in $L(G)$.

The resulting automaton accepts exactly $A^*L(G)$. Indeed, the definition of s ensures that the state p reached after reading an input string w corresponds to a path labeled with the longest suffix w' of w which is a prefix of $L(G)$. w is in $A^*L(G)$ iff w' has a suffix in $L(G)$, that is iff p is a final state. This ends the proof of the theorem. $\square$

Notice that the recognition is independent of the alphabet of the string to recognize. Although the construction of an automaton following Acyclic-Matcher only involves the alphabet of the strings represented by G, the resulting automaton recognizes $A^*L(G)$, for any A containing the alphabet of G.

2.2 Complexity and optimization

Thanks to the use of a failure function, the space complexity of the algorithm Acyclic-Matcher is linear in the size of the obtained automaton. Indeed, the size of the queue Q does not exceed the number of states V' of the resulting automaton since each state is exactly enqueued once. The total size of the lists $list[q]$ involved in the algorithm is also bounded by V' since two distinct lists have no element in common. The required size of the arrays s, f and d is equal to V'. Thus, the space complexity of Acyclic-Matcher is in $O(|V'| + |E'|)$, where E' is the set of transitions of the resulting automaton.

This is an interesting advantage of this algorithm especially in the case of automata containing more than several hundred thousands of states or transitions such as those involved in some natural language applications. The naive determinization algorithm applying the classical powerset construction to the automaton G provided with a loop labeled with all elements of A at the initial state is quadratic with respect to the number of states $V"$ it generates. Indeed, in that algorithm each of the states of the result is represented by a subset of $V"$. The size of each subset is less than $|V"|$, hence the algorithm is in $O(|V"|^2)$. This size is in fact bounded by $|V|$ since the elements of the subset all belong to the initial automaton. There exists k such that: $k \cdot \log |V"| \leq |V| \leq |V"|$. However, to be consistent with other expressions, we give complexities in terms of the sizes of the resulting automaton. The sum of the sizes of all subsets is indeed equivalent to $|V"|^2$ even in simple cases such as A^*a^n.

Another advantage of the algorithm Acyclic-Matcher is that it allows one to save the space required for the representation of many transitions which need to be explicitly indicated in the case of the naive algorithm, thanks to the use of the failure function.

In general, even if the initial automaton G is minimal, the result of the algorithm Acyclic-Matcher is not the minimal deterministic acceptor of $A^*L(G)$. For example, the application of this algorithm to the minimal automaton of the set $X = a(a + b + c) + bc(a + b) + cc$ does not provide the minimal acceptor of A^*X.

Notice that in any case the result of Acyclic-Matcher still recognizes the language $L(G)$ when used in the usual way without taking advantage of the failure function. But this is not necessarily the case for the minimal acceptor representing $A^*L(G)$. Since it also represents $L(G)$, the result of the algorithm constitutes a single device allowing one not only to know whether a given text contains a pattern of $L(G)$ but also to identify the occurrences of these patterns. Indeed, if a final state is reached while reading a prefix t' of a text t using the representation corresponding to $A^*L(G)$, then $|t'|$ corresponds to the ending position of some patterns of $L(G)$. Then reading the reverse string t'^R of t' from the reached state to the initial state in reverse using this automaton and the final states yields the list of all occurrences of the patterns ending at that position.

2.3 Complexity of the use of the resulting automaton

As in the case of the AC algorithm, this automaton offers linear time recognition. Indeed, each time a failure transition is made, the level of the control state of the automaton decreases. At most n forward transitions are made when processing a string of length n. Since the level of the state reached after processing this string is positive, the total number of failure or forward transitions is bounded by $2n$.

Using the same argument, it can be easily shown that the total number of failure transitions made in the algorithm ACYCLIC-MATCHER to compute s along each path from the initial state to a state with no leaving transition is bounded by the length of this path. So, if we assume that the test at line 22 is done in constant time $O(1)$, and that the insertions of lines 20 and 31 as well are in $O(1)$ on the average using perfect hashing methods (see [Aho and Lee 1986] and [Dietzfelbinger *et al.* 1988]), then on the average the time complexity of the algorithm is $O(\log|A| \cdot L)$ where L is the sum of the lengths of all the strings recognized by G. The $\log|A|$ factor corresponds to the cost of searching for a transition. In all these complexities $|A|$ can be replaced with $min\{|A|, e_{max}\}$, where e_{max} denotes the maximum out-degree of all states.

The algorithm has the same average complexity as the algorithm AC. However, one would like here to describe the complexity in terms of the size of the automaton since L might be exponentially larger than V'. The complexity of the algorithm AC is in $O(|V'|^2 \cdot \log|A|)$, if $|V'|$ denotes the number of nodes or edges of the trie representing the finite set of patterns. Indeed, at most $|V'|$ failure transitions are made for the computation of each default state.

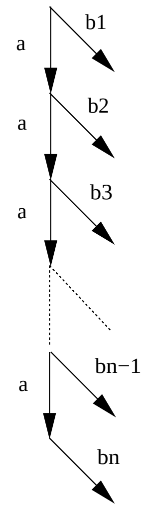

Fig. 4: Quadratic complexity of the AC algorithm.

We do not prove here that the best measure of the complexity of the AC algorithm is quadratic. There are cases where the quadratic complexity is reached, but one might find a simple way of improving the AC algorithm to avoid those cases, or consider that the complexity is quadratic with respect to the size of the alphabet. The case of the sets of patterns described by the regular expression $a^n(b_1 + \cdots + b_m)$ where the alphabet contains $\{a, b_1, \ldots, b_m\}$ clearly leads to a quadratic complexity since the corresponding trie would have $n + m$ transitions and the failure states computation would require $m(n-1)$ steps. The case of the sets $\{ab_1, aab_2, ..., a^n b_n\}$ with an alphabet containing $\{a, b_1, \ldots, b_n\}$ also leads to a quadratic complexity. The corresponding trie has $2n$ transitions and the failure states computation requires about $n(n-1)/2$ steps.

Indeed, the computation of the failure state when considering the transition b_i (Fig. 4), $1 \leq i \leq n$, requires examining the failure states of the state reached by a^i. There is no transition by b_i at those states. Hence, the search requires examining the transitions at all those i states.

The complexity of the algorithm Acyclic-Matcher is similar. It is in $O(|E'|\cdot|V'|\cdot\log|A|)$ where E' denotes the final number of transitions. Indeed, the loop of lines 9-33 is performed exactly $|E'|$ times, the number of failure transitions at each step is bounded by $|V'|$, and the size of the lists considered at line 22 is bounded by $|V'|$. In fact it is easy to prove that the number of failure transitions is even bounded by $|V|$. Recall that the complexity of the classical powerset construction is $O(|V"|^2 \cdot |V|^2 \cdot |A| \cdot \log|A|)$, where $V"$ is the set of states of the deterministic automaton it yields. Also, notice that if a better analysis of the AC algorithm gives a better complexity, it would also lead to a better complexity for our algorithm since it works in a very close way.

In practice, a simple modification of the algorithm allows one to speed up the construction. Rather than computing s for each adjacent state $t.v$ of p independently, one can compute them during a single series of failure transitions. Default states are successively examined from the state $s[p]$ until each state $t.v$ is assigned the value of the initial state or that of a state admitting the transition $t.l$, intermediate results being stored in an array of size $|A|$.

Also, once the automaton is constructed, the failure function s of the obtained automaton can be replaced with an optimized one r which avoids unnecessary failure transitions in a way similar to what is described for the AC algorithm [Aho 1990].

3. The general case

We consider here the general case of a deterministic automaton G not necessarily acyclic. The algorithm presented in the previous section does not apply in the general case. Indeed, if G contains cycles, there are states which can be reached by an infinite number of paths starting at the initial state.

In some cases, the paths reaching these states have distinct longest proper suffixes prefixes of $L(G)$. They correspond to the definition of distinct failure states. Using the algorithm of the previous section can then lead to the creation of infinitely many copies of such states. Fig. 5 and Fig. 6 illustrate this case.

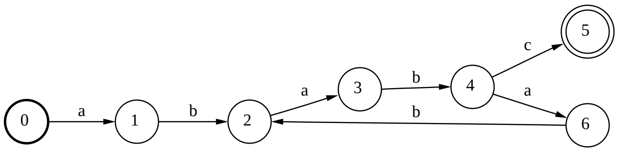

Fig. 5: Cyclic automaton G.

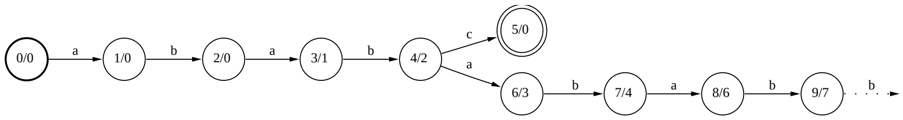

Fig. 6: Blow-up in the determinization.

The first states of the automaton obtained applying the algorithm of the previous section to the automaton of Fig. 5 are shown in Fig. 6. They indicate the endless creation of new states corresponding for instance to new suffixes of $(ab)^2((ab)^2)^*$ that are prefixes of $L(G)$. In the following, we explain how to modify this algorithm, using the same failure function, to avoid the endless creation of states and to obtain a deterministic automaton such as that of Fig. 7 which represents $A^*L(G)$.

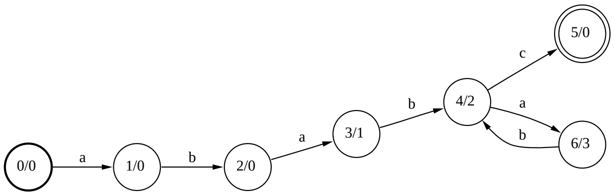

Fig. 7: Deterministic automaton representing $A^*L(G)$.

3.1 Algorithm

The general algorithm is an extension of the algorithm described in the previous section. It is based on the use of the same failure function s, and therefore uses the same breadth-first search order to define the resulting deterministic automaton. However, the duplication of states is here avoided in some cases. In fact, in those cases, which can only occur when the transition considered is a *back edge* or a *cross edge* in the sense of a breadth-first traversal, the original transition is modified in such a way that the destination state is no more a copy of the original destination state but some other state. This allows us to avoid the blow-up illustrated by the example above. For the sake of clarity, we give the complete pseudocode of the general algorithm here (Fig. 8), though it only slightly differs from the acyclic case.

The determinization we describe here has specific properties. It can be thought of as a powerset construction, though the sets are not explicitly constructed. The failure function s helps us to *abbreviate* the definition of each set.

For any string $u \in A^*$, we define $sp_{L(G)}(u)$ as the set of suffixes of u which are also prefixes of $L(G)$, and for any state p the set of its failure states $S(p) = \{p, s(p), \ldots, s^{n_p}(p) = i\}$. We show by recurrence on $|u|$ and by examining all alternatives in the algorithm that the strings of $sp_{L(G)}(u)$ reach one of the states of $S(p)$, $(\delta(i, u) = p)$:

$$\forall v \in A^*, v \in sp_{L(G)}(u) \Leftrightarrow \delta(i, v) \in S(p) \tag{2}$$

The condition clearly holds for $u = \epsilon$, since $S(i) = \{i\}$. Consider a transition (p, a, p') at line 9 of the algorithm, and let u be a string reaching p. Assume that the condition holds for all $v \in A^*$, $|v| \leq |u|$. The state q defined in 11-14 is the one reached by the longest proper suffix w of ua prefix of $L(G)$. Since $w \leq |u|$, we have:

$$\forall v \in A^*, v \in sp_{L(G)}(w) \Leftrightarrow \delta(i, v) \in S(q) \tag{3}$$

Now we need to show that in the resulting automaton, the destination state of the transition by a from p is a state p'' such that:

$$\forall v \in A^*, v \in sp_{L(G)}(ua) \Leftrightarrow \delta(i, v) \in (S(q) \cup \{p''\}) \tag{4}$$

This is clearly true if the destination state p'' is p' or if it is defined at lines 23 or 32, since then by construction $s(p'') = q$ (see lines 16, 22, and 26). The only other case is that of line 34 where p'' is defined to be q. The condition 4 is then realized too. This proves the assertion 2.

LEMMA 1. *Assume that the algorithm* MATCHER *yields a finite-state automaton G'. Then G' recognizes exactly $A^*L(G)$.*

```
MATCHER(G)
  for each p ∈ V
    do  s[p] ← UNDEFINED
        f[p] ← p
  s[i] ← i
  d[i] ← 0
  Q ← {i}
  while Q ≠ ∅
    do  p ← head[Q]
        for each t ∈ Trans[p]
          do  q ← s[p]
              while (q ≠ i) and (δ(q, t.l) not defined)
                    do    q ← s[q]
              if (p ≠ i) and (δ(q, t.l) defined)
                    then  q ← δ(q, t.l)
              if (s[t.v] = UNDEFINED)
                    then  s[t.v] ← q
                          d[t.v] ← d[p] + 1
                          if (q ∈ F)
                            then  F ← F ∪ {t.v}
                          LIST-INSERT(list[t.v], t.v)
                          ENQUEUE(Q, t.v)
              else if (there exists r ∈ list[f[t.v]] such that s[r] = q)
                    then  t.v ← r
              else if ((f[q] ≠ t.v) and ((d[t.v] ≥ d[p]) or
                          (t.v ∉ {f[s(q)], f[s^2(q)], ..., f[s^k(q)] = i}))
                    then  r ← COPY-STATE(t.v)    ▹ copy of t.v with same
                                                   transitions using f
                          s[r] ← q
                          f[r] ← f[t.v]
                          d[r] ← d[p] + 1
                          if (f[t.v] ∈ F)
                            then  F ← F ∪ {r}
                          LIST-INSERT(list[f[t.v]], r)
                          t.v ← r
                          ENQUEUE(Q, r)
              else  t.v ← q
        DEQUEUE(Q)
```

Fig. 8: Algorithm for the construction from G_2 of a deterministic automaton for $A^*L(G_2)$, the general case.

PROOF. Consider a state p of G', and let w be a string reaching p. In the algorithm p is made final iff there exists $q \in S(p)$ such that q be a copy of a final state of the original automaton G. Using the assertion 2 and the fact that by construction the states of G' are all accessible, this is equivalent to the existence of a string v in $sp_{L(G)}(w) \cap L(G)$. Thus w is recognized by G' iff it admits a suffix $v \in L(G)$. This proves the lemma. □

Denote by V' the set of states of the result, for each state $p \in V'$, by $f(p)$ the state p is a copy of, and consider the mapping Ψ defined by:

$$\begin{array}{rrl}\Psi : & V' \longrightarrow & W \\ & q \longmapsto & (f(q), f(s(q)), \ldots, f(s^{n_q}(i)) = i)\end{array}$$

where W is the set of tuples made of elements of V with repetitions. Notice that the size of such tuple can be infinite. The following lemma will help us to prove the correctness of the algorithm.

LEMMA 2. *The number of states of the result of* MATCHER *is finite.*

PROOF. To prove this assertion, we show that Ψ is injective and that its image is finite.
Ψ is injective. Indeed, let (q, q') be in V'^2, $\Psi(q) = \Psi(q')$ implies $n_q = n_{q'}$:

$$\forall i, 0 \leq i \leq n_q,\ f(s^i(q)) = f(s^i(q')) \tag{5}$$

Recall that $s^{n_q}(q) = s^{n_{q'}}(q') = i$, and consider the states $s^{n_q-1}(q)$ and $s^{n_{q'}-1}(q')$. They are copies of the same state: $f(s^{n_q-1}(q)) = f(s^{n_{q'}-1}(q'))$. Since the algorithm is such that no copy is made when states have the same failure states,

$$[s(s^{n_q-1}(q)) = s(s^{n_{q'}-1}(q')) = i] \Rightarrow [s^{n_q-1}(q) = s^{n_{q'}-1}(q')] \tag{6}$$

Using the same arguments, a direct recurrence leads to $s(q) = s(q')$ and $q = q'$.
We now show that for all $p \in V'$, the number of elements of $\Psi(p)$ is less than $|V|$. Indeed, assume that there exists a state p such that $\Psi(p)$ has strictly more than $|V|$ elements. Then, there exist two indices k_1 and k_2, $1 \leq k_1 < k_2 \leq n_p$, such that $s^{k_1}(p)$ and $s^{k_2}(p)$ are copies of the same states: $f(s^{k_1}(p)) = f(s^{k_2}(p))$. We have $s^{k_2-k_1}(s^{k_1}(p)) = s^{k_2}(p)$, and by definition of s, the level of $s^{k_2}(p)$ is less than that of $s^{k_1}(p)$: $d[s^{k_2}(p)] < d[s^{k_1}(p)]$. Hence, the state $s^{k_1}(p)$ has been created as a copy of $f(s^{k_2}(p))$ when considering a back edge or a cross edge ($d[t.v] < d[p]$ using the notation of Fig. 8). But $t.v = f(s^{k_2}(p)) = f(s^{k_2-k_1}(s^{k_1}(p)))$, thus the conditions of line 24 of the algorithm do not hold with $q = s^{k_1}(p)$. This contradicts the fact that $s^{k_1}(p)$ has been created as a copy of $f(s^{k_2}(p))$. Hence, for any state p, $\Psi(p)$ has $|V|$ or less than $|V|$ elements. The image of Ψ is included in $2^{|V|}$ and is finite. Since Ψ is injective, this implies that the number of states of the result is finite. □

Using the two previous lemmas we can now prove the following theorem.

THEOREM 2. *Let G be a deterministic automaton. The algorithm of Fig. 8,* MATCHER*, computes correctly a representation of a deterministic automaton recognizing $A^*L(G)$.*

PROOF. Lemma 2 shows that the algorithm terminates. By lemma 1, we know that the resulting automaton represents exactly $A^*L(G)$ using the failure function s. This proves the theorem. □

3.2 Complexity

Using the same arguments as in the acyclic case, one can easily prove that the space complexity of the algorithm is linear in the size of the resulting automaton $O(|V'| + |E'|)$, thanks to the use of a failure function. The time complexity of the algorithm is $O(|E'| \cdot |V'| \cdot \log |A|)$ since the number of default transitions made at line 12 or for the condition of line 24 is bounded by the total number of final states V, and since the size of the lists of line 22 is bounded by $|V'|$. While in the powerset construction, at a given state $\{q_1, \ldots, q_m\}$ and for a given letter, one needs to consult each state q_i to find possible transitions leaving this state labeled with that letter, this occurs only in the worst case using MATCHER. Indeed, only if the default state of q_m admits no transition by the considered letter is the following state considered, etc. Also, thanks to the use of a failure function, not all the transitions created in the naive algorithm need to be constructed here.

4. Conclusion

The construction of an automaton representing $A^*L(G)$ is needed in many problems of lexical and syntactic analysis, and in speech processing. In particular, they are very useful when dealing with local grammars represented by automata or transducers [Mohri 1994].

We have fully implemented the algorithms described in the previous sections. Results in practical cases in natural language processing show them to be very efficient. We used an implementation of our algorithm to improve a version of a program close to the Unix command *sed*. On large texts, it turned out to be 4 times faster than those obtained with the Unix command without having recourse to any optimization. Also, this algorithm allowed us to successfully determinize automata arising in speech processing for which the naive powerset construction was computationally intractable. The resulting sizes of the determinized automata exceed 1.5 million states in those cases.

Although we did not show it here, these algorithms can be easily adapted so as to create according to the *lazy transition evaluation technique* described by Aho [1990] just that deterministic automaton needed to match a given text.

Acknowledgements

A preliminary version of this paper appeared in the Proceedings of CPM'95, Springer-Verlag, LNCS 937, 1995. Submission of this paper in final form to the Nordic Journal of Computing was invited in view of the strong endorsement contained in the conference reviews.

I thank Maxime Crochemore for several discussions on this work, and Fernando Pereira for helpful comments on earlier draft of this paper.

References

AHO, A. V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Leuwen, J. Van, Editor. Elsevier, Amsterdam, 255–300.

AHO, A. V. AND CORASICK, M. J. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Communication of the Association for Computing Machinery 18 (6)*, 333–340.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers, Principles, Techniques and Tools*. Addison Wesley.

AHO, A.V. AND LEE, D. 1986. Storing a dynamic sparse table. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, 55–60.

BAEZA-YATES, RICARDO A. AND GONNET, GASTON H. 1989. Efficient text searching of regular expressions. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, ICALP '89*, Volume Lecture Notes in Computer Science, Springer-Verlag, Berlin, 46–62.

BOYER, R.S. AND MOORE, J.S. 1977. A Fast String-Searching Algorithm. *Communication of ACM 20*, 762–772.

COMMENTZ-WALTER, B. 1979. A String Matching Algorithm Fast on the Average. *Automata, Languages and Programming Lecture Notes in Computer Science, Springer-Verlag, Berlin*, 118–132.

CROCHEMORE, M., CZUMAJ, A., GASIENIEC, L., JAROMINEK, S., LECROQ, T., PLANDOWSKI, W., AND RYTTER, W. 1993. Fast Multi-Pattern Matching. Technical Report IGM 93-3, Institut Gaspard Monge.

CROCHEMORE, M. AND RYTTER, W. 1994. *Text Algorithms*. Oxford University Press.

DIETZFELBINGER, M., KARLIN, A., MEHLHORN, K., HEIDE, F. MEYER AUF DER, ROHNERT, H., AND TARJAN, R. E. 1988. Dynamic perfect hashing: upper and lower bounds. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 524–531.

KNUTH, D.E., JR, J.H. MORRIS, AND PRATT, V.R. 1977. Fast Pattern Matching in Strings. *SIAM Journal of Comput. Syst. Sci. 6*, 323–350.

MOHRI, MEHRYAR. 1994. Syntactic Analysis by Local Grammars Automata: an Efficient Algorithm. In *Proceedings of the International Conference on Computational Lexicography (COMPLEX 94)*. Linguistic Institute, Hungarian Academy of Science: Budapest, Hungary.

MOHRI, MEHRYAR. 1997. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics 23*, 1–42.

PERRIN, D. 1990. Finite Automata. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, Leuwen, J. Van, Editor. Elsevier, Amsterdam, 1–57.