1996

Searching for Ephemeral Subsequences in Strings

Alberto Apostolico

Mikhail J. Attalah

Report Number:
96-076

SEARCHING FOR EPHEMERAL SUBSEQUENCES IN STRINGS

Alberto Apostolico
Mikhail J. Atallah

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD-TR 96-076
December 1996

Searching for Ephemeral Subsequences in Strings

Alberto Apostolico*and Mikhail J. Atallah †
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
U.S.A.
{axa,mja}@cs.purdue.edu

Abstract

Let $T = a_1 \ldots a_n$ be a text where every symbol a_i has a *time stamp* t_i and a *duration* $d(a_i)$ associated with it. The time stamps of the a_i's are increasing, so that $j > i$ implies $t_j > t_i$. A text symbol a_i is *alive* at time t iff $t_i \leq t \leq t_i + d(a_i)$. A subsequence $a_{i_1} \ldots a_{i_m}$ of T is alive iff every a_{i_k} is alive at time t_{i_m}, that is, $t_{i_k} + d(a_{i_k}) \geq t_{i_m}$ for all $k \in \{1, \ldots, m-1\}$. We consider the problem of determining whether a given pattern $P = b_1 \ldots b_m$ occurs as an alive subsequence of T. We give an off-line (i.e., the pattern is known in advance) algorithm, running in $O(n+m)$ time. We also introduce and discuss data structures for fast on-line implementation.

Index Terms — Algorithms, pattern matching, ephemeral subsequence, DAWG, forward failure function, intrusion and misuse detection

*and Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy. axa@art.dei.unipd.it; partially supported by NSF grant CCR-92-01078, by NATO grant CRG 900293, by the National Research Council of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

†This author gratefully acknowledges support from the COAST Project at Purdue and its sponsors, in particular Hewlett Packard, DARPA, the National Security Agency, and the Office of Research and Development

1 Introduction

We consider the problem of detecting occurrences of a pattern string as a subsequence of a larger text string, under the special assumption that the textstring consists of *ephemeral* symbols, in the sense that each symbol can be used in a subsequence only within an interval or time duration that is fixed and typical of that symbol. Once this interval or time limit has elapsed, the symbol is no longer usable.

Variants of this problem arise in numerous applications, ranging from information retrieval to molecular sequence analysis. To give one example, we describe here the problem of detecting intrusion and misuse detection in a computer system [4], which originally motivated our study: in this application, the text string is the audit trail data generated by the system, where each event has a time stamp associated with it, whereas a pattern represents a sequence of events that indicate suspicious activity that might be indicative of an intrusion by an outsider, or misuse of the system by an insider. That time stamps and durations arise in this context is not surprising, since the significance of a sequence of events depends on the elapsed time between the events; for example, the significance of five failed login attempts in a system depends on the time that elapses between them, with more reason for alarm if they are in rapid succession than if days occur between them. Not all of the patterns described in [4] are of the kind we consider here, so that the algorithm we describe here will speed up detection of some but not all of the attack patterns described in [4]. However, we believe the ideas of the present paper can be useful as heuristics for improving detection of the more general patterns described in [4]. In what follows we no longer refer to the above motivation for the problem, instead we focus on the abstract algorithmic formulation given next.

First we review the notion of occurrence of a pattern as a subsequence in a text when there is no notion of time (no timestamps or durations). A pattern $P = b_1 \ldots b_m$ occurs as a subsequence of a text $T = a_1 \ldots a_n$ iff there exist indices $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ such that $a_{i_1} = b_1$, $a_{i_2} = b_2, \cdots, a_{i_m} = b_m$; in this case we say that P occurs at position i_m of T (note how this differs from the usual convention for occurrences as a substring rather than as a subsequence, which are usually said to occur at the position that matches the first position of the pattern rather than the last one). It is trivial to compute, in linear time, whether P occurs as a subsequence of T. What makes our problem different is that the symbols occurring in T have positive time (or position) stamps and durations, with the following significance. A symbol a_i that occurs at time t_i and has duration $d(a_i)$ is no longer alive at times that exceed $t_i + d(a_i)$. A subsequence of T is alive only

if all of its symbols are alive at the timestamp of its last symbol: that is, if $a_{i_1} \ldots a_{i_m}$ is such that $t_{i_1} + d(a_{i_1}) \leq t_{i_m}$, $t_{i_2} + d(a_{i_2}) \leq t_{i_m}$, $\cdots$, $t_{i_{m-1}} + d(a_{i_{m-1}}) \leq t_{i_m}$. Our problem is to find whether there is an alive subsequence of T that equals P. The naive approach of using an automaton that advances from state 1 to state m as it reads T, occasionally retreating when a previously encountered symbol expires, runs into problems. In fact, we find it necessary to use an automaton whose state is described by an $(m-1)$-tuple of times, one for each proper prefix of the pattern. At first sight this will seem to imply an $O(mn)$ time algorithm, as there are events that cause $O(m)$ updates to occur in the $(m-1)$-tuple; one such event is the expiration time of a symbol on which $O(m)$ tuple entries depend, another is the occurrence of a symbol that appears $O(m)$ times in P and thus requires $O(m)$ updates to the tuple's time stamps. The contribution of this paper is a technique for achieving $O(m+n)$ time, even when the pattern contains repetitions.

Throughout the rest of the paper, by "occurrence" of a pattern we mean as a subsequence rather than as a substring. Note that, with appropriate setting of the time unit, choosing uniformly $d(a_i) = m$ reduces our problem to standard string searching, whereas choosing $d(a_i) = n$ reduces it to the problem of checking whether P is a subsequence of T.

2 The off-line algorithm

The algorithm consists of a scan of the text, while maintaining an $(m-1)$-tuple S of times satisfying the following invariant:

Invariant. When the scan of the text is done with a_i, $S(k)$ should equal the latest time after t_i at which the most recently encountered (that is, prior to a_{i+1}) prefix $b_1 \ldots b_k$ of the pattern will still be alive; if no subsequence $b_1 \ldots b_k$ of $a_1 \cdots a_i$ is alive after t_i then $S(k)$ is zero.

Observation 1 $S(1) \geq S(2) \geq \cdots \geq S(m-1)$.

Proof. If $b_1 \ldots b_k$ is still alive at time t then surely $b_1 \ldots b_{k-1}$ is still alive at that time. □

For every symbol a, we use $F(a)$ (mnemonic for "first occurrence of a") to denote the first occurrence of a in P; $F(a)$ is zero if a does not occur in P. That is, $F(a) = j$ iff $b_j = a$ and $b_i \neq a$ for all $i < j$. The function F can easily be computed in linear time, and the algorithm that follows assumes that this has already been done.

In addition to the function F, the preprocessing algorithm also computes, for each symbol a that occurs in P, a function L such that $L(a)$ is the largest index such that the portion of P between positions $F(a)$ and $L(a)$ consists of repetitions of symbol a, that is, $a = b_{F(a)} = b_{F(a)+1} = \cdots = b_{L(a)}$

and $b_{L(a)+1} \neq a$. Note that the last occurrence of a in P might be later than $L(a)$, if position $L(a)$ is followed by occurrences of symbols other than a and then by one or more later a's.

The algorithm uses an array S' to represent the S array, as follows. The array S' is identical to S with the following exceptions: If $F(a) < L(a)$ then $S(F(a) : L(a))$, which denotes the region of S between $S(F(a))$ and $S(L(a))$, is stored *circularly shifted* in the corresponding region of S', $S'(F(a) : L(a))$; that is, $S(F(a) : L(a))$ is represented by a circular list (which we refer to as $Circ(a)$) residing in $S'(F(a) : L(a))$, with $S'(F(a))$ considered a successor of $S'(L(a))$ in the circular list. Of course in that case we need to mark which place in $S'(F(a) : L(a))$ corresponds to $S(F(a))$, i.e. where to mark the beginning of the circular list $Circ(a)$: This information is stored in an array entry $s(a)$, that is, $S'(s(a)) = S(F(a))$, the successor of $S'(s(a))$ in the circular list $Circ(a)$ equals $S(F(a)+1)$, etc. The reason for this apparently strange representation is that it will enable us to update all the $L(a) - F(a) + 1$ entries of $S(F(a) : L(a))$ in constant time rather than in time proportional to $L(a) - F(a) + 1$ (as will become clear later). The circular list $Circ(a)$ is of course described by the triplet $F(a), L(a), s(a)$, so there is no need to store it separately (we use the $Circ(a)$ term as a notational convenience).

We use a scalar β to store the index of the "boundary" between nonzero and zero entries S: The largest index i for which $S(i)$ is nonzero (if all of S is zero then $\beta = 0$).

Although the algorithm below refers to the logical structure S, the actual data structure used to represent it is the array S' (in conjunction with the arrays F, L, and s). We choose to present the algorithm in terms of S rather than S' in order not to clutter the exposition; when what happens physically in S' differs substantially from its logical counterpart in S, we shall explicitly point that out.

Initially all the entries of S are zero. The algorithm looks at $a_1, a_2, \ldots$ in that order and, for each symbol a_i of T that it looks at, it performs the following updates in S.

1. First we perform the following `expired-update` routine, whose purpose is to set to zero the entries of S whose expiration time is before t_i.
   `expired-update` routine:

   (a) Repeat the following (b) until $S(\beta) \geq t_i$ or $\beta = 0$:

   (b) If $S(\beta) < t_i$ then set $S(\beta) = 0$ and $\beta = \beta - 1$.

2. If $F(a_i) = 0$ (i.e., a_i does not occur in P) then we are done with the updates for a_i. Otherwise we continue with the next step.

3. We have $F(a_i) \neq 0$ (i.e., a_i occurs in P). We do the following.

   (a) If $a_i = b_{\beta+1}$ then we set $S(\beta+1) = \min\{S(\beta), t_i + d(a_i)\}$ and $\beta = \beta + 1$. If $\beta + 1 = m$ then we print "P occurs at position i".

   (b) If $S(F(a_i)) \neq 0$ then (i) we set $S(F(a_i)) = \min\{S(F(a_i) - 1), t_i + d(a_i)\}$, and (ii) if the update in (i) causes an increase to $S(F(a_i))$ compared to its old value, making it equal to $t_i + d(a_i)$, and if $F(a_i) < L(a_i)$, then for $k = F(a_i) + 1, \cdots, \min\{\beta, L(a_i)\}$ in turn we set $S(k) = S(k-1)$.

   Implementation note: This step is done in $O(1)$ time rather than in $O(k)$ time, because of the circular list $Circ(a)$ that is used in $S'(F(a) : L(a))$ to represent $S(F(a) : L(a))$. In effect, all we need to do in $Circ(a)$ is move $s(a_i)$ back by one position and write at that position the new value of $S(F(a_i))$.

3 Analysis

This section proves that the algorithm given in the previous section is correct and has time complexity $O(m+n)$.

3.1 Correctness

Step 1 of the algorithm is clearly needed to maintain the invariant, since we must set to zero all those $S(j)$'s for which even the most recent alive occurrence of $b_1 \ldots b_j$ in $a_1 \ldots a_{i-1}$ has an expiration time that is less than t_i.

Step 2 is justified because in that case symbol a_i does not appear in P.

Step 3 involves two different kinds of updates. We must justify these two kinds updates, and we must also prove that these are enough, i.e., that no other updates are needed.

We begin with Step 3(a), which is the easiest to justify: Since $a_i = b_{\beta+1}$ and since (by the definition of β) $b_1 \ldots b_\beta$ is alive until time $S(\beta)$, it follows that $b_1 \ldots b_{\beta+1}$ is alive until $S(\beta)$ or $t_i + d(a_i)$, whichever occurs first (i.e.. the smaller of the two).

The update to $S(F(a_i))$ done in Step 3(b) is justified for similar reasons as the above update of Step 3(a). But Step 3(b) also implicitly assumes that we never need to update only a portion of $S(F(a) : L(a))$, i.e., that whenever we improve $S(F(a))$ then we also improve $S(F(a)+1 : \min\{\beta, L(a)\})$. This too needs justification, as do the actual updates done in Step 3(b). The arrival of the symbol a_i marks the occurrence of a "more recent" alive subsequence of $a_1 \ldots a_i$ that equals $b_1, \cdots, b_{F(a_i)}$, an event that may or may not improve (i.e., increase) $S(F(a_i))$, depending on

whether the bottleneck for the expiration time of $b_1 \ldots b_{F(a_i)}$ $(= S(F(a_i)))$ is the expiration time of $b_1 \ldots b_{F(a_i)-1}$ (i.e., $S(F(a_i) - 1)$) or the expiration time of a_i (i.e., $t_i + d(a_i)$):

1. In the former case the expiration time of $b_1 \ldots b_{F(a_i)-1}$ is surely also the bottleneck for $S(F(a_i) + 1 : L(a_i))$, which justifies the lack of update by the algorithm for that case.

2. In the latter case its is the expiration time of the $(k - F(a_i) + 1)$ most recent occurrences of symbol a_i in T that determine $S(k)$, $F(a_i) < k \leq \min\{\beta, L(a_i)\}$. Therefore such an $S(k)$ improves with the arrival of a_i by increasing to the old value of $S(k-1)$, because the expiration time of the most recent $(k - F(a_i) + 1)$ occurrences of symbol a_i is the same as the expiration time of the most recent $(k - F(a_i) + 1 - 1)$ occurrences of that symbol before we looked at a_i, which is the old $S(k - 1)$. This justifies the update done by the algorithm in Step 3(b).

   We must still justify why we make no other updates than the ones done in Steps 3(a) and 3(b), that is, why we do not consider $S(j)$'s for which $b_j = a_i$ and $j > L(a_i) + 1$, unless $j = \beta + 1$ at the beginning of Step 3(a). To put it differently, we need to show that updating such an $S(j)$ by replacing it with $\min\{S(j - 1), t_i + d(a_i)\}$ is a "do nothing" operation that causes no change in $S(j)$. The proof is by contradiction: Suppose that there is at least one such an $S(j)$ that increases due to such an update, and let j be the earliest among those (that is, the smallest index j larger than $L(a_i) + 1$ such that $b_j = a_i$ and $S(j)$ increases because of the occurrence of a_i). Now, the improved (larger) expiration date of the new $S(j)$ must be determined by the new $S(j-1)$ rather than by $t_i + d(a_i)$, because otherwise $S(j)$ would equal $t_i + d(a_i)$, a contradiction since $b_{F(a_i)}$ is an earlier occurrence of symbol a_i in $b_1 \ldots b_{j-1}$ and thus must have an expiration time that is earlier than $t_i + d(a_i)$. Since $S(j)$ is determined by $S(j - 1)$, the only way the value of $S(j)$ increases is if the value of $S(j - 1)$ increases. So suppose $S(j - 1)$ increases. We distinguish two cases:

   (a) Case 1: $b_{j-1} = a_i$. This implies that $j - 1 > L(a_i) + 1$, contradicting the definition of j as the smallest index larger than $L(a_i) + 1$ such that $b_j = a_i$ and $S(j)$ increases ($j - 1$ satisfies all of these and is smaller than j).

   (b) Case 2: $b_{j-1} \neq a_i$. This contradicts the fact that $S(j - 1)$ increases, since the only way any $S(j - 1)$ can increase as a result of event a_i is if b_{j-1} equals symbol a_i.

4 Skip-edge DAWGs and on-line detection

In some applications of standard string searching, the text string is preprocessed in such a way that any subsequent query regarding pattern occurrence takes time proportional to the size of the pattern rather than that of the text. Notable among these constructions are those resulting in structures such as subword trees and graphs (refer to, e.g., [3]). Notice that the answer to the typical query is now only whether or not the pattern appears in the text. If one wanted to locate all the occurrence as well, then time would become $O(|w| + occ)$, where occ denotes the total number of occurrences. These kind of searches are sometimes classified as *on line*, in reference to the fact that preprocessing of the pattern is not allowed. In general, setting up efficient on line structures for non exact matches seems quite hard: sometimes a small selection of options is faced, that represent various compromises among a few space and time parameters. With a little simple preprocessing of a text string T, it becomes trivial to process any query as to whether a pattern P is a subsequence of T in $O(|P|)$ steps. All is needed is a pointer leading, for every position of T and every alphabet symbol, to the next position occupied by that symbol. Slightly more complicated arrangements readily accommodate the case of arbitrary alphabet size (see, e.g., [1]). However, this approach is no longer adequate in the presence of ephemeral symbols, since the earliest occurrence of P as a subsequence of T is not guaranteed to be a solution.

In this section, we assume that the textstring is fixed and consider the problem of detecting, as fast as possible, whether a given pattern occurs as an ephemeral subsequence of the text. Our solution rests on an adaptation of the partial minimal automaton recognizing all subwords of a word, also known as the *DAWG (Directed Acyclic Word Graph)* [2] associated with that word. Let $\mathcal{V}$ be the set of all subwords of the text T, and P_i $(i = 1, 2, ..., m)$ be the ith prefix of a pattern P.. We say that a word $Y \in \mathcal{V}$ is a *realization* of P_i if (1) P_i is an ephemeral subsequence of Y but not of any subword of Y, and (2) no word $W \in \mathcal{V}$ has Y as a subword and P_{i+1} as an ephemeral subsequence. Note in particular that the first and last symbols of Y and P_i coincide. We show how to build the modified graph in linear time and then how to access it in time

$$O(min(n, \sum_{i=1}^{m} rocc_i \cdot |P_i|)),$$

where $rocc_i$ is the number of distinct realizations of P_i appearing in X. Note that a realization is a substring that may occur many times in X but is counted only once in our bound.

We begin our discussion by recalling the structure of the DAWG for string X. First, we consider the minimal partial deterministic finite automaton that recognizes all subwords of X. Given two

words X and Y, the *end-set* of Y in X is the set $endpos_X(Y) = \{j : Y = a_i...a_j\}$ for some i and j, $1 \leq i \leq j \leq n$. Two strings W and Y are equivalent on X if $endpos_X(W) = endpos_X(Y)$. The equivalence relation instituted in this way is denoted by $\equiv_X$ and partitions the set of all strings over Σ into equivalence classes. It is convenient to assume henceforth that our text string X is fixed, so that the equivalence class with respect to $\equiv_X$ of any word W can be denoted simply by $[W]$. Thus $[W]$ is the set of all strings that have occurences in X terminating at the same set of positions as W. Correspondingly, the finite automaton $\mathcal{A}$ recognizing all substrings of X will have one state for each of the equivalence classes of subwords of X under $\equiv_X$. Specifically:

1. The start state of $\mathcal{A}$ is $[\lambda]$
2. For any state $[W]$ and any symbol $a \in \Sigma$, there is a transition edge leading to state $[Wa]$;
3. The state corresponding to all strings that are not substrings of W, is the only nonaccepting state, all other states are accepting states.

Deleting from $\mathcal{A}$ above the nonaccepting state and all of its incoming arcs yields the DAWG associated with X. An example DAWG is reported in Figure 1.

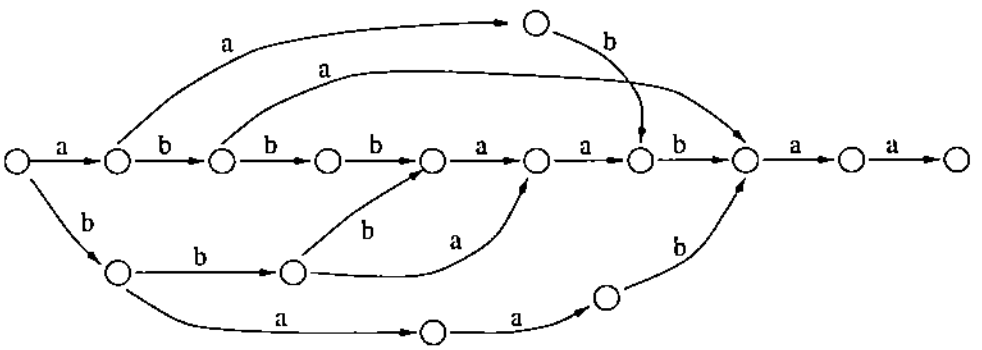

Figure 1: An example DAWG

We refer to, e.g., [2, 3], for the construction of a DAWG. Here we recall some basic properties of this structure. This is clearly a directed acyclic graph with one sink and one source, where every state lies on a path from the source to the sink. Moreover, the following two properties hold [2, 3].

Property 1 For sany word X, the sequence of labels on each distinct path from the source to the sink of the DAWG of X represents one distinct suffix of X.

Property 2 For any word X, the DAWG of X has a number of states Q such that $|X| + 1 \leq Q \leq 2|X| - 2$ and a number of edges E such that $|X| \leq E \leq 3|X| - 4$.

It is immediate to see how the DAWG of X may be adapted to test whether any given pattern P is an ephemeral subsequence of X. Essentially, we need to add, to every node α and for every alphabet symbol a such that no edge labeled a leaves α, a number of "downward failure links" or *skip edges* connecting α to each one of its closest descendants where an original incoming edge labeled a already exists. As an example, Figure 2 displays a partially augmented version of the DAWG of Figure 1 with skip edges added only to the source and its two adjacent nodes. An immediate consequence of Property 1 is that P is a subsequence of X beginning at some specific position i of X if and only if the following two conditions hold: (1) there is a path π labeled P from the source to some node α of the augmented version of the DAWG of X, and (2) it is possible to replace each skip edge in π with a chain of original edges in such a way that the resulting path would the original path from the source to α is labeled by consecutive symbols of X beginning with position i.

Clearly, the role of skip edges is to serve as shortcuts in the search. However, these edges also introduce "nondeterminism" in our automaton, in that now more than one path from the source may be labebed with a prefix of P. Even so, the search for P is trivially performed, e.g., as a depth-first visit of the graph where the depth is suitably bounded by taking into account the length of P, duration of individual symbols and lengths of the skipped paths. Each edge is traversed precisely once, and each time we backtrack from a node, this corresponds to a prefix of P which cannot be continued along the path being explored, whence the claimed time bound for searches. Such a bound is actually not very tight, a tighter one being represented by the total numebr of distinct nodes traversed. In practice, this may be expected to be proportional to some small power of the length of P. Consideration of symbol durations may be also added to the construction phase, thereby further reducing the number of skip edges issued. The main problem, however, is that storing this version of the augmented DAWG would charge an unrealistic $\Theta(n^2)$ space even when the alphabet size is a constant (cf. Fig. 2). The remainder of our discussion is devoted to improve on this space requirement.

Towards this end, observe that by Property 1 each node of the DAWG of X can be mapped to a position i in X in such a way that the path from that node to the sink is labeled precisely by the suffix $a_i a_{i+1} ... a_n$ of X. As is easy to check, such a mapping assignment can be carried out during the construction of the DAWG at no extra cost. Observe also that there is always a path labeled X in the DAWG of X. This path will be called the *backbone* of the DAWG, and its nodes will be numbered by consecutive integers from 0 (for the source) to n (for the sink).

In order to describe how skip links are issued on the DAWG, we resort to a slightly extended

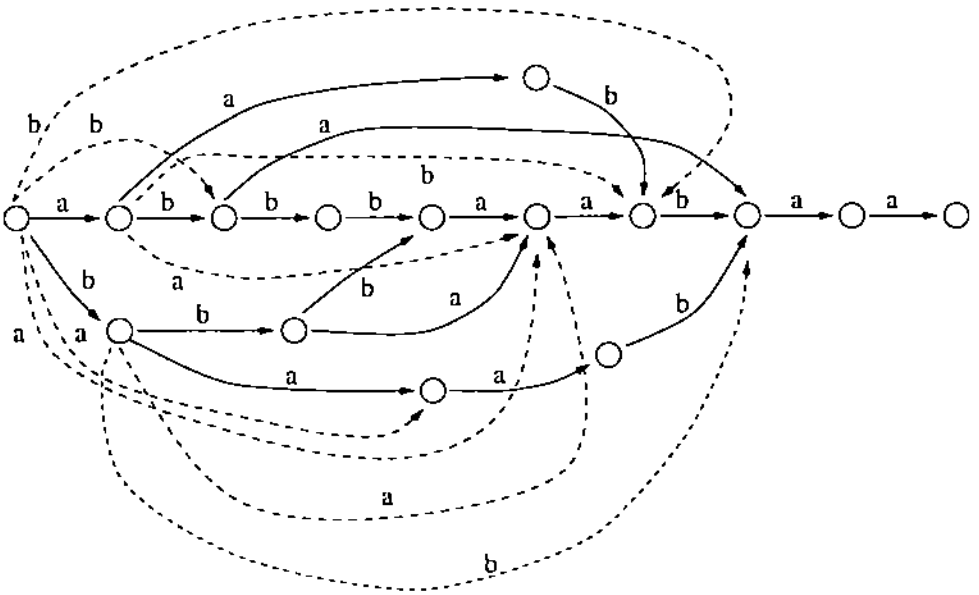

Figure 2: Partially augmented DAWG

version of a spanning tree of the DAWG (see Fig. 3). Our spanning tree must contain a directed path that corresponds precisely to the backbone of the DAWG, but is arbitrary otherwise. The extension consists simply of duplicating the nodes of the DAWG that are adjacent to the leaves of the spanning tree, so as to bring into the final structure also the edges connecting those nodes (each of these edges would be classified as either a cross edge or a descendant edge in the visit of the DAWG that produced the tree). Let $\mathcal{T}$ be the resulting structure. Clearly, $\mathcal{T}$ has the same number of edges and at most twice the number of nodes of the DAWG. We use the structure of $\mathcal{T}$ to describe how to set skip edges. In actuality, edges are set on the DAWG.

First, specially tagged, *backbone* skip links are directed from each backbone node and each alphabet symbol to the closest (sink-wards) backbone node reached by an edge labeled by that symbol, or to $(n+1)$ if no such edge is found. With reference now to a generic node α of $\mathcal{T}$, we distinguish the following cases.

- **Case 1** Node α has outdegree 1. Assume that the edge leaving α is labeled a, and consider the path π from α to a branching node or leaf of $\mathcal{T}$, whichever comes first. For every first occurrence on π of an edge (β, γ) labeled $\bar{a} \neq a$, direct a skip edge labeled $\bar{a}$ from α to γ. For every symbol of the alphabet not encountered on π set a skip pointer from α to the branching node or leaf found at the end of π.

- **Case 2** Node α is a branching node. The skip edges possibly to be issued from α are determined as follows. Let β be a descendant other than a child of α in $\mathcal{T}$, with an incoming

the root or else there must be an edge labeled a on the path from the branching node immediately above α and α. Hence, no branching node from the root to α could direct an a labeled skip edge to any node in the subtree of T rooted at α. In conclusion, also the total number of these edges is bounded by the length of X, by Property 2. □

It is not difficult to carry out the augmentation of the DAWG along the paradigm of a visit of the underlying directed graph, hence in linear time and space. The details are tedious but straightforward, and are left for an exercise. A search is similarly organized as the visit of a directed graph, except this time we need to carry along and constantly update a vector skip telling us for every symbol of the alphabet, which branching node holds the information regarding the pertinent skip edge. A pointer to the position in the descendant list of that node would tell us exactly which one of the possibly many a-labeled skip edges should be followed at any given time. Since any pattern search always originates at the root, there is no risk of ever bypassing information on skip edges.

Finally, we point out that symbol durations should be taken into account during the search. This is, however, straightforward, and results in additional time savings.

References

[1] A. Apostolico and C. Guerra, The Longest Common Subsequence Problem Revisited, *Algoritmica*, 2, 315-336 (1987).

[2] A. Blumer. J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas, The Smallest Automaton Recognizing the Subwords of a Text, *Theoretical Computer Science* , 40, 31-55 (1985).

[3] M. Crochemore and W. Rytter, *Text Algorithms*, Oxford University Press, New York (1994).

[4] S. Kumar and E.H. Spafford, "A Pattern-Matching Model for Instrusion Detection," *Proceedings of the National Computer Security Conference*, 1994, pp. 11–21.