Compact Recognizers of Episode Sequences

Alberto Apostolico[*] — *Purdue University & Università di Padova*

Mikhail J. Atallah [†] — *Purdue University*

Abstract

Given two strings $T = a_1 \dots a_n$ and $P = b_1 \dots b_m$ over an alphabet Σ, the problem of testing whether P occurs as a subsequence of T is trivially solved in linear time. It is also known that a simple $O(n \log |\Sigma|)$ time preprocessing of T makes it easy to decide subsequently for any P and in at most $|P| \log |\Sigma|$ character comparisons, whether P is a subsequence of T. These problems become more complicated if one asks instead whether P occurs as a subsequence of some substring Y of T of bounded length. This paper presents an automaton built on the textstring T and capable of identifying all distinct minimal substrings Y of X having P as a subsequence. By a substring Y being minimal with respect to P, it is meant that P is not a subsequence of any proper substring of Y. For every minimal substring Y, the automaton recognizes the occurrence of P having lexicographically smallest sequence of symbol positions in Y. It is not difficult to realize such an automaton in time and space $O(n^2)$ for a text of n characters. One result of this paper consists of bringing those bounds down to linear or $O(n \log n)$, respectively, depending on whether the alphabet is bounded or of arbitrary size, thereby matching the respective complexities of off-line exact string searching. Having built the automaton, the search for all lexicographically earliest occurrences of P in X is carried out in time $O(n + \sum_{i=1}^{m} rocc_i \cdot i \cdot \log n \cdot \log |\Sigma|)$, where $rocc_i$ is the number of distinct minimal substrings of T having $b_1 \dots b_i$ as a subsequence. All log factors appearing in the above bounds can be further reduced to $\log \log$ by resort to known integer-handling data structures.

Index Terms — Algorithms, pattern matching, subsequence and episode searching, DAWG, suffix automaton, compact subsequence automaton, skip-edge DAWG, forward failure function, skip-link.

[*]Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA, and Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy. `axa@cs.purdue.edu`. Work partially supported by NSF Grant CCR-9700276, by NATO Grant CRG 900293, by the National Research Council of Italy, and by British Engineering and Physical Sciences Research Council Grant GR/L19362.

[†]COAST Laboratory and Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, IN 47907, USA. `mja@cs.purdue.edu`. This author gratefully acknowledges support from the COAST Project at Purdue and its sponsors.

1 Introduction

We consider the problem of detecting occurrences of a *pattern* string as a subsequence of a substring of bounded length of a larger *text* string. Variants of this problem arise in numerous applications, ranging from information retrieval and data mining (see, e.g., [8]) to molecular sequence analysis (see, e.g., [9]) and intrusion and misuse detection in a computer system (see, e.g., [7]).

Recall that given a pattern $P = b_1 \dots b_m$ and a text $T = a_1 \dots a_n$ over some alphabet Σ, we say that P occurs as a *subsequence* of T 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 also say that the substring $Y = a_{i_1} a_{i_1+1} \dots a_{i_m}$ of T is a *realization* of P beginning at position i_1 and ending at position i_m in T. We reserve the term *occurrence* for the sequence $i_1 i_2 \dots i_m$. It is trivial to compute, in time linear in $|T|$, whether P occurs as a subsequence of T. Alternatively, a simple $O(n|\Sigma|)$ time preprocessing of T makes it easy to decide subsequently for any P, and in at most $|P|$ character comparisons, whether P is a subsequence of T. For this, all is needed is a pointer leading, for every position of T and every alphabet symbol, to the closest position occupied by that symbol. Slightly more complicated arrangements, such as developed in [2], can accommodate within preprocessing time $O(n \log |\Sigma|)$ and space linear in T also the case of an arbitrary alphabet size, though introducing an extra $\log |\Sigma|$ cost factor in the search for P.

These problems become more complicated if one asks instead whether T contains a realization Y of P of bounded length, since the earliest occurrence of P as a subsequence of T is not guaranteed to be a solution. In this case, one would need to apply the above scheme to all suffixes of T or find some other way to detect the *minimal* realizations Y of P in T, where a realization is minimal if no substring of Y is a realization of P. Algorithms for the so-called *episode matching* problem, which consists of finding the *earliest* occurrences of P in all minimal realizations of P in T have been given previously in [5]. An occurrence $i_1 i_2 \dots i_m$ of P in a realization Y is an earliest occurrence if the string $i_1 i_2 \dots i_m$ is lexicographically smallest with respect to any other possible occurrence of P in Y. The algorithms in [5] perform within roughly $O(nm)$ time, without resorting to any auxiliary structure or index based on the structure of the text.

In some applications of exact 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., [1], [4]). 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 occurrences as well, then the time would become $O(|w| + occ)$, where *occ* denotes the total number of occurrences. These kinds of searches may be considered as *on line* with respect to the pattern, in the sense that preprocessing of the pattern is not allowed, but are *off-line* in terms of the ability to preprocess the text. In general, setting up efficient structures of this kind 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.

This paper addresses the construction of an automaton, based on the textstring T and suitable for identifying, for any given P, the set of all distinct minimal realizations of P in T. Specifically, the automaton recognizes, for each such realization Y, the earliest occurrence of P in Y. The preceding discussion suggests that it is not difficult to realize such an automaton in time and space $O(n^2)$ for a text of n characters. The main result of the paper consists of bringing those bounds down to linear space, thus matching that of the off-line exact string searching with subword graphs. Our construction can be used, in particular, in cases in which the symbols of P are affected by individual "expiration deadlines", expressed, e.g., in terms of positions of T that might elapse at most before next symbol (or, alternatively, the entire occurrence of pattern P) must be matched.

The paper is organized as follows. In next section, we review the basic structure of Directed Acyclic Word Graphs and outline an extension of it that constitutes a first, quadratic space realization of our automaton. A more compact implementation of the automaton is addressed in the following section. Such an implementation requires linear space but only in the case of a finite alphabet. The case of general alphabets is addressed in the last section, and it results in a tradeoff between seach time and space.

2 DAWGs and Skip-edge DAWGs

Our main concern in this section is to show how the text T can be preprocessed in a such a way, that a subsequent search for the earliest occurrences in T of all prefixes of any given P is carried out in time bounded by the size of the output rather than that of the input. 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)* [3] 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 P. Our modified graph can be built in time and space quadratic or linear in the length of the input, depending on whether the size of the input alphabet is arbitrary or bounded by a constant, respectively, and it can be searched for the earliest occurrences in all $rocc_i$ distinct realizations of P_i $(i = 1, 2, ..., m)$ in time

$$O(n + \sum_{i=1}^{m} rocc_i \cdot i \cdot \log n).$$

Note that a realization of P_i 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 $X = a_1 \ldots a_n$. First, we consider the following partial deterministic finite automaton recognizing 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 for $X = abbbaabaa$ is reported in Figure 1.

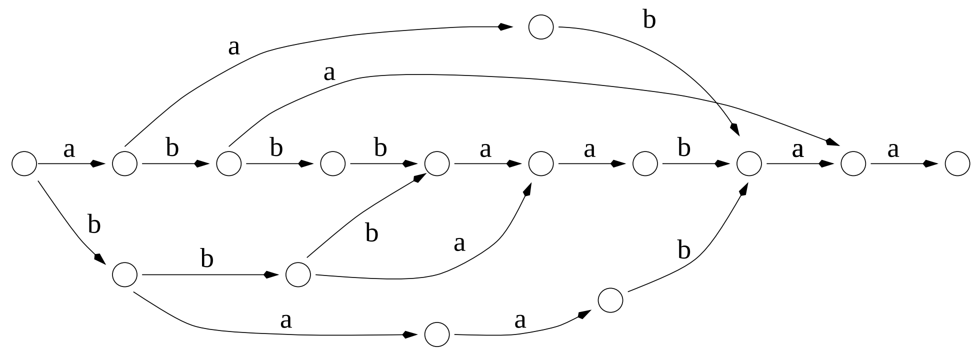

Figure 1: An example DAWG

We refer to, e.g., [3, 4], 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 [3, 4].

Property 1 For any 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 recognize all earliest occurrences of any given pattern P as a subsequence of X. Essentially, we need to endow every node α with a number of "downward failure links" or *skip-edges*. Each such link will be associated with a specific alphabet symbol, and the role of a link leaving α with label a will be to enable the transition to a descendant of α on a nontrivial (i.e., with at least two *original* edges) path labeled by a string in which symbol a occurs only as a suffix. Thus, a skip-edge labeled a is set from 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 occurs as 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 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 particular, 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 all longest paths in the graph that start at the source and are labeled by some prefix of P. (The depth of the search may be suitably bounded by taking into account the length of P, and lengths of the shunted 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 tight, an even tighter one being represented by the total number 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 construction itself is easily carried out in quadratic time, e.g., by adaptation of a depth-first visit of the DAWG, as follows. First, when the sink is first reached, it is given *nil* skip-edges for all alphabet symbols; next, every time we backtrack to a node α from some other node β, the label of arc (α, β) and the skip-edges defined at β are used to identify and issue (additional) skip-edges from α. We leave the details as an exercise. The main problem, however, is that storing this version of the augmented DAWG would take 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 improving on this space requirement.

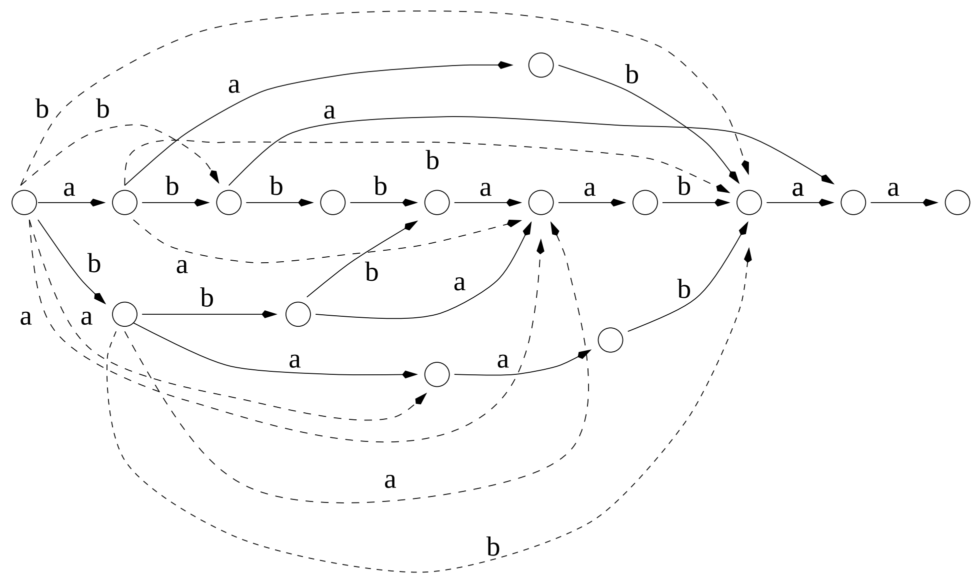

Figure 2: Adding skip-edges from the source and its two adjacent nodes

3 Compact skip-edge DAWGs

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 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 it is arbitrary otherwise. As for the extension, this 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 (any of these edges would be classified as either a "cross edge" or a "descendant edge" in the visit of the DAWG resulting in our 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, whence its size is linear in the length of X. We use the more convenient structure of $\mathcal{T}$ to describe how to set skip-edges and other auxiliary links. In actuality, edges are set on the DAWG.

Since the introduction of a skip-edge for every node and symbol would be too costly, we will endow with such edges only a fraction of the nodes. Specifically, our policy will result in a linear number of skip-edges being issued overall. From any node not endowed with a skip-edge on some desired symbol, the corresponding transition will be performed by first gaining access to a suitable node where such a skip-edge is available, and then by following that edge. In order to gain access from any node to its appropriate "surrogate" skip-edge, we need to resort to two additional families of auxiliary edges, respectively called *deferring edges* and *back edges*. The space overhead brought about by these auxiliary edges will be only $O(n \cdot |\Sigma|)$, hence linear when Σ is finite. The new edges will be labeled, just like skip-edges, but unlike skip-edges their traversal on a given input symbol does consume that symbol. Their full structure and management will be explained in due course.

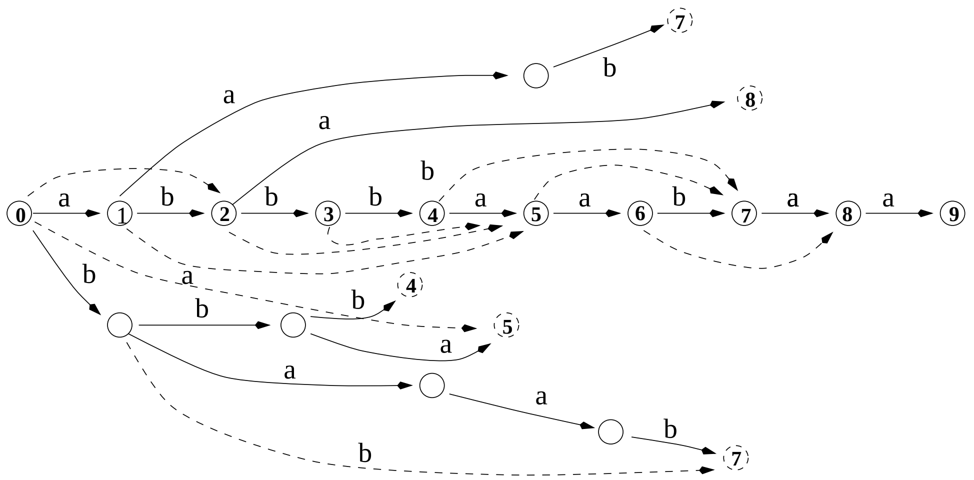

Figure 3: An extended spanning tree $\mathcal{T}$ with sample skip-edges

We now describe the augmentation of the DAWG.

First, specially tagged, *backbone* skip-edges 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 γ of $\mathcal{T}$, we distinguish the following cases.

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

- **Case 2:** Node γ is a branching node. The auxiliary edges to be possibly issued from γ are determined as follows (see also Fig. 4). Let η be a descendant other than a child of γ in $\mathcal{T}$, with an incoming edge labeled a, and let π be the longest ascending path from η such that no other edge of π is labeled a. If γ is the highest (i.e., closest to the root) branching node on π, then direct a skip-edge labeled a from γ to η.
  Consider now the subtree of $\mathcal{T}$ rooted at γ. Any path of $\mathcal{T}$ in this tree that does not lead eventually to to an arc labeled a (like the arc leading to node η) must end on a leaf. To every such leaf, direct a deferring edge labeled a from γ.

- **Case 3:** Node γ is a leaf. Let i be the position of X assigned to node γ under the mapping discussed earlier. For every symbol of the alphabet, the skip-edge from γ labeled by that symbol is copied from the homologous backbone skip-edge at node i of the backbone.

Figures 3 and 4 exemplify skip links for the backbone, the root and one of its children in the tree $\mathcal{T}$ of our example.

At this point and as a result of our construction policy, there may be branching nodes that do not get assigned any skip-edge. This may cause a search to stall in the middle of a downward path, for lack of appropriate direction. In order to prevent this problem, back edges are added to every such branching node, as follows (see Fig. 4). For every branching node β of $\mathcal{T}$ and every alphabet symbol a such that an a-labeled skip-edge from β is not defined, an edge labeled a is directed from β to the closest ancestor γ of β from which a skip-edge labeled a is defined. We refer to γ as the *a-backup* of β, and we denote it by $back_a(\beta)$. Clearly, our intent is that the effect of traversing a skip-edge as described in the previous section can now be achieved by traversing at most three auxiliary edges, namely, one deferring edge, one back edge and one skip-edge. This complication is compensated by the following advantage.

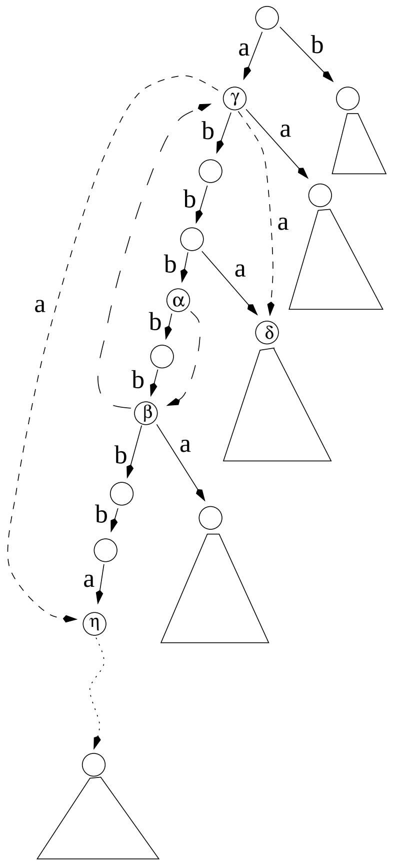

Figure 4: A one-symbol transition from a node α of T to its descendant η requires at most three edge-traversals: first, through a deferring link to the nearest branch node β; next from β to γ through a back-edge; finally, through the skip-edge from γ to η. Note that the presence of another a-labeled skip-edge from γ to δ introduces ambiguity as to which direction to take once the search has reached γ

Lemma 1 *The total number of auxiliary edges in $\mathcal{T}$, whence also in the augmented DAWG of X, is $O(|X| \cdot |\Sigma|)$.*

Proof. There is at most one deferring or back edge per alphabet symbol for each node where such an edge is defined. Also, the claim is certainly true for backbone nodes, leaves and tree nodes of outdegree 1. As for the skip links directed from branching nodes, observe that for any symbol a and any node β, at most one skip-edge labeled a may reach β. The same is true for deferring edges issued on every leaf of $\mathcal{T}$. Indeed, if a deferring edge labeled a is set from a branching node α to such a leaf, then by construction no branching node on the path from α to that leaf can be issued an a-labeled skip-edge. Also by construction, either α is the root or else there must be an edge labeled a on the path from the closest branching ancestor of α to α itself. Hence, no skip-edge labeled a could possibly be set from a branching ancestor of α to a node in the subtree of $\mathcal{T}$ rooted at α. In conclusion, for each symbol of Σ the total number of these edges is bounded by the length of X, by Property 2. $\square$

Lemma 2 *P has a realization Y in X beginning with a_i and ending at a_j if and only if there is a sequence σ of arcs in $\mathcal{T}$ with the following properties: (i) the concatenation of consecutive labels on the original and skip-edges of σ spells out P; (ii) any original or skip-edge of σ is followed by at most one deferring and at most one back edge, in this order; (iii) there is a path labeled $Y = a_i a_{i+1} ... a_j$ from the source to the node $\nu = [Y]$ which is reached by the last arc of σ.*

Proof. Assume that P has a realization Y in X as stated. By the definition of $\mathcal{T}$, there must be an original arc corresponding to $b_1 = P_1 = Y_1 = a_i$. The node reached by this arc satisfies trivially points $(i - iii)$. Assuming now that we have matched a prefix $P_i = b_1 b_2 ... b_i$ of P up to some node α in our structure while maintaining $(i - iii)$. The assertion is easily checked if next symbol b_{i+1} of P is consumed through either an original arc or a skip-edge, the latter being possibly preceded by a deferring edge. Then, the only case of concern occurs when, possibly after having traversed a deferring edge labeled, say, a, one is led from some node α to a branching node β from which no a-labeled skip- or original edge is defined. In such an event, the link to $\gamma = back_a(\beta)$ is traversed instead, as shown in Fig. 4.

Let η be a descendant of β such that either η is reached through a nontrivial path π of $\mathcal{T}$ in which symbol a appears precisely as a suffix, or else η is a leaf of $\mathcal{T}$ and there is no a-labeled original edge from β to η. We claim that there is a skip-edge labeled a from γ to β. In fact, by our selection of π, there is no other edge labeled a on the path from β to the parent node of η. Assuming one such edge existed on the path from γ to β, then β itself or a branching ancestor of β other than γ would have a-labeled skip-edges defined, thereby contradicting that $back_a(\beta) = \gamma$.

In summary, having matched some prefix of P up to some node α in our structure, we always know how to reach in at most two additional transitions a node γ, possibly different from α, from which to perform a consistently labeled skip-edge transition to a descendant η of α, in response to the next symbol of P. The node reached through this edge satisfies points $(i - iii)$ relative to the new matched prefix.

The converse of the proof is straightforward and thus is omitted. $\square$

Based on Lemma 2, a search for the realizations of a pattern in the augmented DAWG of X may be carried out along the lines of a bounded-depth visit of a directed graph. The elementary downward step in a search consists of following an original edge or a skip-edge, depending on which one is found. The details are then obvious whenever such an edge actually exists. The problem that we need to examine in detail is that for any symbol a there may be more than one skip-edge labeled a leaving γ, and some such edges may lead to descendants of γ that are not simultaneously descendants of α. One instance of this is represented by node δ in Fig. 4.

We can assume that all skip-edges leaving a node γ under the same symbol label a are arranged in a list dedicated to a, sorted, say, according to the left-to-right order of the descendants of γ. Thus, in particular, any descendants of the node α of our example that are reachable by an a-labeled skip-edge from γ would be found as consecutive entries in the skip-edge list of γ associated with symbol a. This list or part of it will be traversed left to right in our search, as follows naturally from the structure of a depth-first visit of a graph. In order to understand at which point the sublist relative to descendants of α begins and ends, we can use some standard auxiliary information such as pre- and postorder ordinal number of the nodes of $\mathcal{T}$. Then, in the list of skip-edges from γ, the beginning of the possible sublist relative to descendants of α is found as the smallest preorder index larger than the preorder index of α. Such an index may be located at this point in one of two possible ways, i.e., by performing a binary search on the list of skip-edges at γ, or by stepwise advancement of a pointer through successive positions of the list. The first approach will result in a $\Theta(\log n)$ factor multiplying the cost of each step and thus the overall cost. The second, adds to the global cost a term linear in the length n of X, since each unit advance on the list at γ can be charged uniquely to a distinct edge of $\mathcal{T}$. Running the two above list implementations concurrently leads to the following summary for our discussion.

Theorem 1 *The compact skip-edge DAWG associated with X supports the search for all earliest occurrences of a pattern $P = b_1 b_2 ... b_m$ in X in time*

$$O(\mathrm{Min}(n \;+\; \sum_{i=1}^{m} rocc_i \cdot i, \;\; \sum_{i=1}^{m} rocc_i \cdot i \cdot \log n)),$$

where $rocc_i$ is the number of distinct realizations in X of the prefix P_i of P.

As already noted, a realization is a substring that may occur many times in X but is counted only once in our bound. It is not difficult to modify the DAWG augmentation highlighted in the previous section so as to build the compact variant described here. Again, the core paradigm is a bottom-up computation on $\mathcal{T}$, except that this time skip-edge lists may be assigned to branching nodes only on a temporary basis: whenever, climbing back towards the root from some node β, an ancestor branching node α before any intervening edges labeled a, then the a-labeled skip-edge list of β is surrendered to α. The process takes linear time and space for fixed alphabets. Symbol durations may be taken into account both during construction as well as in the searches, possibly resulting in additional savings. The details are tedious but straightforward and are left to the reader.

4 Generalizing to unbounded alphabets

When the alphabet size $|\Sigma|$ is not a constant independent of the length n of T, we face the choice of implementing also the (original) adjacency list of a node of the DAWG as either a linear list or a balanced tree. The first option leaves space unaffected but introduces slowdown by a linear multiplicative factor in worst-case searches. The second introduces some linear number of extra nodes but now the overhead of a search is only a multiplicative factor $O(\log|\Sigma|)$. Below, we assume this second choice is made. Rather straightforward adaptations to the structure discussed in the previous section would lead to a statement similar to Theorem 1, except for an $O(\log|\Sigma|)$ factor in the time bound. Here, however, we are more interested in the fact that when the alphabet size is no longer a constant Lemma 1 collapses, as the number of auxiliary edges needed in the DAWG may become quadratic. In this Section, we show that a transducer supporting search time

$$O(\text{Min}(n \ + \sum_{i=1}^{m} rocc_i \cdot i, \ \sum_{i=1}^{m} rocc_i \cdot i \cdot \log n) \cdot \log|\Sigma|),$$

can in fact be built within $O(n \log|\Sigma|)$ time and linear space.

The idea is of course to forfeit many skip-edges and other auxiliary edges and pay for this sparsification with a $\log|\Sigma|$ overhead on some elementary transitions. We explain first how this can work on the original array in which X is stored. We resort to a global table CLOSE, defined as follows [2]: CLOSE is regarded as subdivided into blocks of size $|\Sigma|$, With $p = j \bmod |\Sigma|$ ($j = 1, 2, ..., n$), CLOSE$[j]$ contains the smallest position larger than j where there is an occurrence of s_p, the pth symbol of the alphabet. It is trivial to compute CLOSE from X, in linear time. Let now $closest(i,p)$ be the closest instance of s_p to the right of position i (if there is no such occurrence set $closest(i,p) = n+1$). Then, $closest(i,p)$ can be computed from CLOSE and the sorted list of occurrences of s_p in X in $O(\log|\Sigma|)$ time. We refer to [2] for details. The main idea is that two accesses of the form CLOSE$[\lfloor i/|\Sigma|\rfloor \cdot |\Sigma| + \ p\ ]$ and CLOSE$[\lfloor i/|\Sigma|\rfloor \cdot |\Sigma| \ + \ |\Sigma| + \ p\ ]$ must either identify the desired occurrence or else will define an interval of at most $|\Sigma|$ entries in the occurrence list of s_p, within which the desired occurrence can be found by binary search, hence in $O(\log|\Sigma|)$ time. Note that the symbols of X can be partitioned into individual symbol-occurrence lists in $O(n \log|\Sigma|)$ overall time, and that those lists occupy linear space collectively.

The above construction enables us immediately to get rid of all skip-edges issued on the backbone and *inside* each chain of unary nodes present in $\mathcal{T}$. A key element in making this latter fact possible is the circumstance, already remarked, that we can map every path to the sink of the DAWG, hence also every such maximal chain, to a substring of a suffix (hence, to an interval of positions) of X. In fact, once such an interval is identified, an application of *closest* will tell how far down along the chain one should go. Along these lines, we only need to show how a downward transition on $\mathcal{T}$ is performed following the identification made by *closest* of the node that we want to reach: again, we may either scan the chain sequentially or search through it logarithmically, using additional *ad-hoc* auxiliary links (at most $2 \log n$ per node, of which $\log n$ point upward and at most as many point downwards). Except for the possible $O(\log |\Sigma|)$ time charged by *closest*, the rest of the work is going to be absorbed in the global time bound of Theorem 1.

We still face a potential of $\Theta(|\Sigma|)$ deferring edges per chain node and leaf, and as many backup edges per branching node. The chain nodes are easy to accommodate: all deferring edges from a node point to a same branching node and can thus coalesce into a single "downward failure link". As for the backup edges, recall that by definition, on a path π between β and $\gamma = back_a(\beta)$ there can be no edge labeled a, but such an edge must exist on the path from the closest branching ancestor of γ and γ. Let trivially each node of $\mathcal{T}$ be given as a label the starting position of the earliest suffix of X whose path passes through that node. Then, we can use the table *closest* on array X to find the distance of this arc from η, climb to it on $\mathcal{T}$ using at most $\log n$ auxiliary upward links, and finally reach γ through the downward failure link. Considering now the deferrring edges that lead to leaves, they can be entirely suppressed at the cost of visiting the subtrees of $\mathcal{T}$ involved at search time: this introduces linear overall work, since it suffices to visit each subtree once.

We conclude by pointing out that all log factors apperaring in our claims can be reduced to $\log log$ at the expense of some additional bookkeeping, by deploying data structures especially suited for storing integers in a known range [6]. It is also likely that the $\log n$ factors could be made to disappear entirely by resort to amortized finger searches such as, e.g., in [2].

References

[1] A. Apostolico and Z. Galil (Eds.), *Pattern Matching Algorithms*, Oxford University Press, New York (1997).

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

[3] 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).

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

[5] G. Das, R. Fleischer, L. Gąsieniek, D. Gunopulos, J. Kärkkäinen, Episode Matching, *CPM'97, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, (A. Apostolico and J. Hein, Eds.), Springer Verlag LNCS **1264**, 12-27 (1997).

[6] D.B. Johnson, A Priority Queue in which Initialization and Queue Operations Take $O(\log\log n)$ Time, *Math. Sys. Th.*, **15**, 295-309 (1982).

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

[8] H. Mannila, H. Toivonen and A.I. Vercamo, Discovering Frequent Episodes in Sequences, *KDD'95, Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining*, AAAI Press, 210-215 (1995).

[9] M. Waterman, *Introduction to Computational Biology*, Chapman and Hall (1995).