

Art Checklist

Journal Code:	INCO			
Article No:	3143			
	Disk Recd	Disk	Disk Usable	
Art #	Y/N	Format	Y/N	Remarks
Fig.1	Y	EPS	Y	
Fig.2a	Y	EPS	Y	
Fig.2b	Y	EPS	Y	
Fig.2c	Y	EPS	Y	
Fig.3	Y	EPS	Y	
Fig.4	Y	EPS	Y	
Fig.5	Y	EPS	Y	

Note:

1. .eps and .tif are preferred file formats.
2. Preferred resolution for line art is 600 dpi and that for Halftones is 300 dpi.
3. Art in the following formats is not supported: .jpeg, .gif, .ppt, .opj, .cdr, .bmp, .xls, .wmf and pdf.
4. Line weight should not be less than .27 pt.

Compact Recognizers of Episode Sequences

Alberto Apostolico¹

Dipartimento di Elettronica e Informatica, Università di Padova, Via Gradenigo 6/A, 35131 Padua, Italy, and Department of Computer Sciences, Purdue University, Computer Sciences Building, West Lafayette, Indiana 47907

E-mail: axa@cs.purdue.edu

and

Mikhail J. Atallah²

CERIAS Laboratory and Department of Computer Sciences, Purdue University, Recitation Building, West Lafayette, Indiana 47907

E-mail: mja@cs.purdue.edu

Received January 19, 1998; revised June 7, 1999

Given two strings $X = a_1 \dots a_n$ and $P = b_1 \dots b_m$ over an alphabet Σ , the problem of testing whether P occurs as a subsequence of X is trivially solved in linear time. It is also known that a simple $O(n \log |\Sigma|)$ time preprocessing of X makes it easy to decide subsequently, for any P and in at most $|P| \log |\Sigma|$ character comparisons, whether P is a subsequence of X . These problems become more complicated if one asks instead whether P occurs as a subsequence of some substring Y of X of bounded length. This paper presents an automaton built on the textstring X 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 the 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 corresponding complexities of automata constructions for offline exact string searching. Having built the automaton, the search for all lexicographically earliest occurrences of P in X is carried out in time $O(\sum_{i=1}^m \text{rocc}_i \cdot i)$ or $O(n + \sum_{i=1}^m \text{rocc}_i \cdot i \cdot \log n)$, depending on whether the alphabet is fixed or arbitrary, where rocc_i is the number of distinct minimal substrings of X having $b_1 \dots b_i$ as a subsequence (note that each such substring may occur many times in X but is counted only once in the bound). All log factors appearing in the above bounds can be further reduced to log log by resorting to known integer-handling data structures. © 2002 Elsevier Science (USA)

Key Words: algorithms; pattern matching; subsequence and episode searching; DAWG; suffix automaton; compact subsequence automaton; skip-edge DAWG; forward failure function; skip-link.

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., [10]) to molecular sequence analysis (see, e.g., [12]) and intrusion and misuse detection in a computer system (see, e.g., [9]).

Recall that given a pattern $P = b_1 \dots b_m$ and a text $X = a_1 \dots a_n$ over some alphabet Σ , we say that P occurs as a *subsequence* of X iff there exist indices $1 \leq i_1 < i_2 < \dots < i_m \leq n$ such that $a_{i_1} = b_1$, $a_{i_2} = b_2, \dots, a_{i_m} = b_m$; in this case we also say that the substring $Y = a_{i_1} a_{i_1+1} \dots a_{i_m}$ of X is a *realization* of P beginning at position i_1 and ending at position i_m in X . We reserve the term *occurrence* for the sequence $i_1 i_2 \dots i_m$. It is trivial to compute, in time linear in $|X|$, whether P occurs as a subsequence of

¹ Work partially supported by NSF Grant CCR-9700276, by NATO Grants CRG 900293 and PST.CLG.977017, by the National Research Council of Italy, by the Italian Ministry of University and Research under the National Project "Bioinformatics and Genomic Research," and by British Engineering and Physical Sciences Research Council Grant GR/L19362.

² Portions of this work were supported by Grant EIA-9903545 from the National Science Foundation and by sponsors of the Center for Education and Research in Information Assurance and Security.

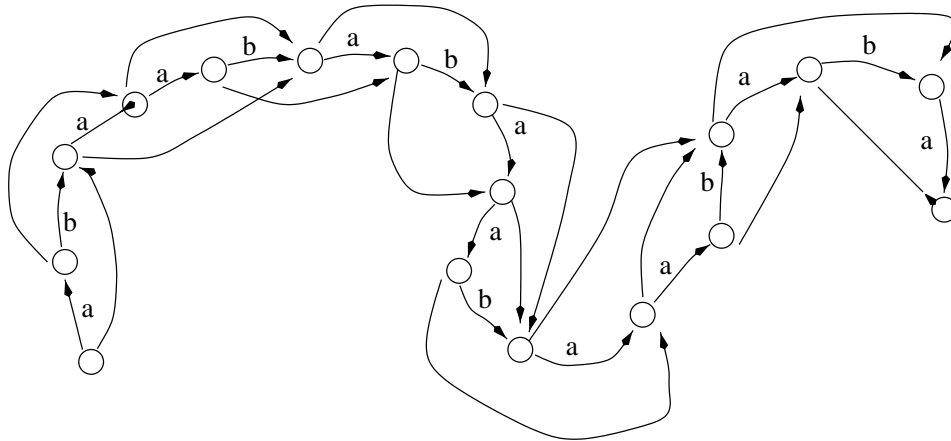


FIG. 1. Recognizer for the subsequences of *abaababaabaababa*, shown here without explicit labels on forward “skip” links.

X . Alternatively, a simple $O(n|\Sigma|)$ time preprocessing of X makes it easy to decide subsequently for any P , and in at most $|P|$ character comparisons, whether P is a subsequence of X . For this, all that is needed is a pointer leading, for every position of X and every alphabet symbol, to the closest position occupied by that symbol, as exemplified in Fig. 1. Slightly more complicated arrangements, such as developed in [2], can accommodate within preprocessing time $O(n \log |\Sigma|)$ and space linear in X also the case of an arbitrary alphabet size, though introducing an extra $\log |\Sigma|$ cost factor in the search for P . We refer also to [3] for an additional discussion of subsequence searching.

These problems become more complicated if one asks instead whether X contains a realization Y of P of bounded length, since the earliest occurrence of P as a subsequence of X is not guaranteed to be a solution. In this case, one would need to apply the above scheme to all suffixes of X or find some other way to detect the *minimal* realizations Y of P in X , 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 X , have been given previously in [7]. 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 [7] 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, 6]). 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 w is the query pattern and occ denotes the total number of occurrences. These kinds of searches may be considered as *online* with respect to the pattern, in the sense that preprocessing of the pattern is not allowed, but are *offline* in terms of the ability to preprocess the text. In general, setting up efficient structures of this kind for nonexact matches seems quite hard: sometimes a small selection of options is faced that represent various compromises among a few space and time parameters. In [5, 11], the idea of limiting the search only to distinct substrings of the text is applied to perform approximate string matching with suffix trees.

This paper addresses the construction of an automaton, based on the textstring X and suitable for identifying, for any given P , the set of all distinct minimal realizations of P in X . Specifically, the automaton recognizes, for each such realization Y , the earliest occurrence of P in Y . As seen in the following section, it is not difficult to realize such an automaton in time and space $O(n^2)$ for a text of n characters over a fixed alphabet. The main result of the paper consists of bringing those bounds down to linear or $O(n \log n)$, depending on the assumption on alphabet size, matching the cost of preprocessing in offline 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 X that might elapse at most before the next symbol (or, alternatively, the entire occurrence of pattern P) must be matched.

The paper is organized as follows. In the 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 trade-off between search time and space.

2. DAWGS AND SKIP-EDGE DAWGS

Our main concern in this section is to show how the text X can be preprocessed in a such a way that a subsequent search for the earliest occurrences in X 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*) [4] associated with that word. Let \mathcal{W} be the set of all subwords of the text X , and let P_i ($i = 1, 2, \dots, m$) be the i th 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 ($i = 1, 2, \dots, m$) in time

$$O\left(\sum_{i=1}^m rocc_i \cdot i \cdot \log |\Sigma|\right).$$

Here a *realization* of P_i ($i = 1, 2, \dots, m$) is any minimal substring of X having $b_1 \dots b_i$ as a subsequence. 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. On the other hand, in the worst case where $P_m = P$ has $\Theta(n)$ distinct realizations in X , the above bound becomes $O(nm \log |\Sigma|)$, which is worse than the $O(nm)$ straightforward approach of searching independently for the earliest realization of P on every suffix of X .

We begin our discussion by recalling the structure of the DAWG for string $X = a_1 \dots a_n$. 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 $endpos_X(Y) = \{j : Y = a_i \dots a_j\}$ 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 occurrences 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 $[\epsilon]$, where ϵ is the empty word;
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 . As an example, the DAWG for $X = abbbaabaa$ is reported at the bottom of Fig. 2.

One way to obtain the DAWG is to append the endmarker to the string (for our example we choose the symbol c) and then perform the following steps. First, build the the digital tree of all suffixes of $ababaababaaababac$. This is done, e.g., by starting with an empty tree and introducing the suffixes in succession. The resulting tree is displayed at the top of Fig. 2. Next, apply to this tree the standard, linear time subtree isomorphism algorithm and merge all nodes that are roots of isomorphic subtrees. (The leaves are roots of the empty tree and are thus merged). This gives the structure in the middle of Fig. 2. Finally, remove all nodes and edges related to the endmarker symbol c .

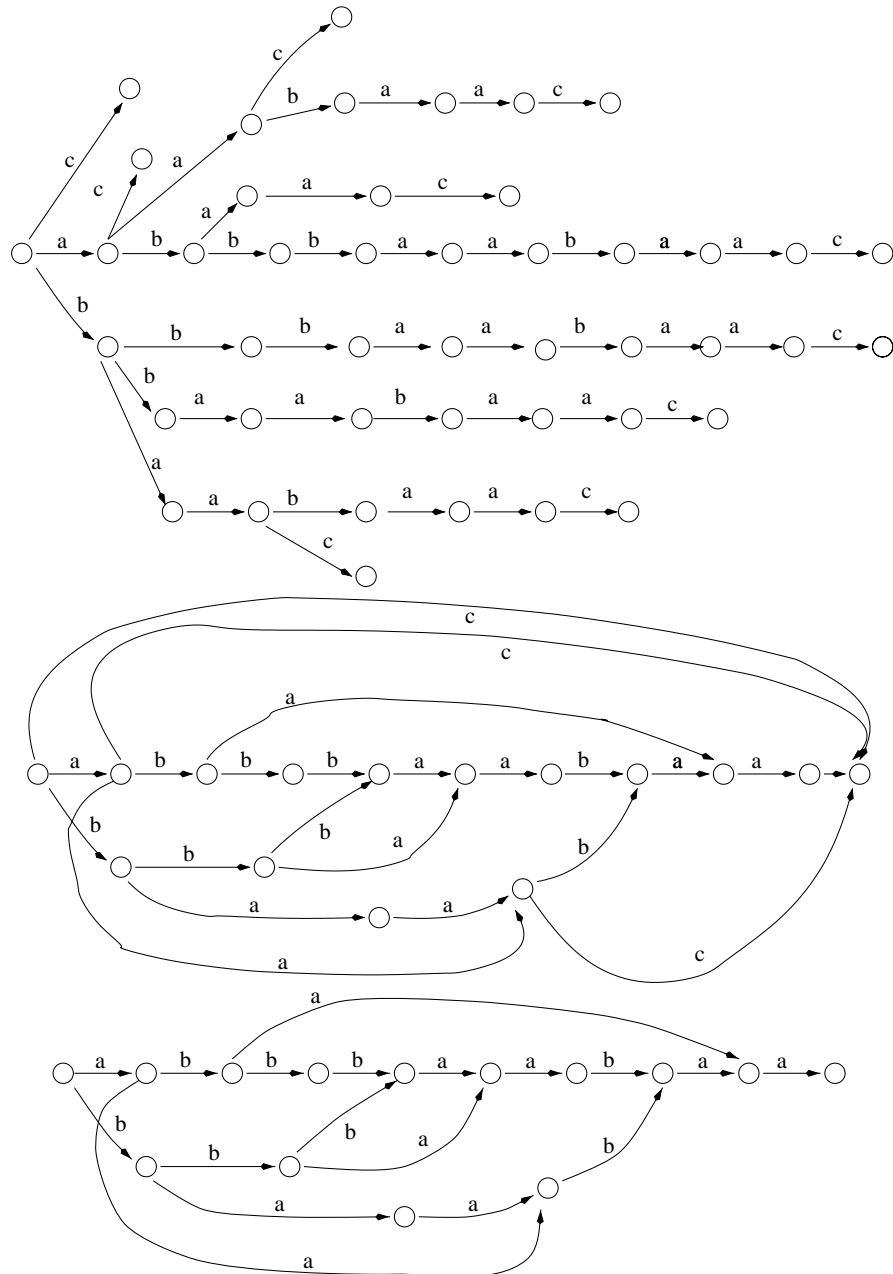


FIG. 2. The DAWG for $ababaababaaababa$ (shown at the bottom) may be derived from the digital tree of all suffixes of $ababaababaaababac$ (top: c is an arbitrary “endmarker”) by merging all isomorphic subtrees (middle) and then removing all structures related to the endmarker.

Whereas such an approach requires quadratic time and space, much more efficient and elegant constructions exist that take linear time and space. We refer to, e.g., [4, 6] for the details as well as for the correctness of the suboptimal construction given above. Here we recall some basic properties of the DAWG 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 [4, 6].

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 . Specifically, there is one such path for every suffix that is not a prefix of another suffix.

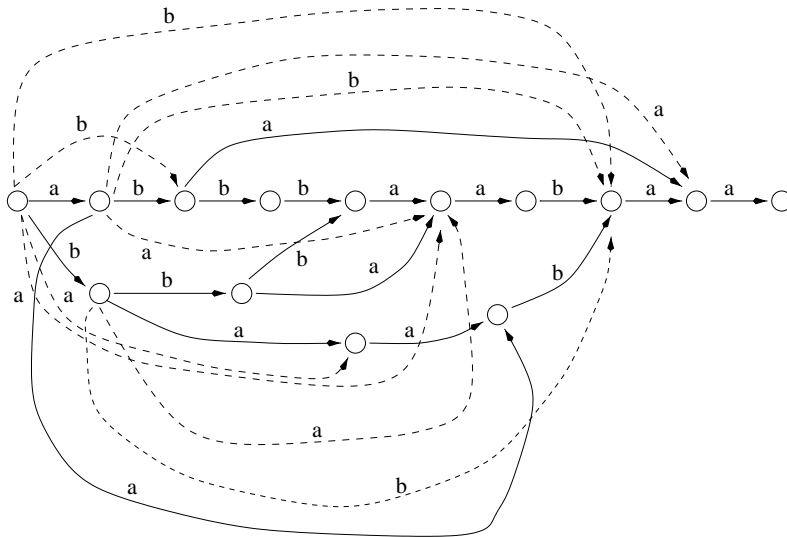


FIG. 3. Adding skip-edges from the source and its two adjacent nodes.

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 $|X| \leq E \leq 3|X| - 4$.

The structure of a basic subsequence detector such as the one of Fig. 1 suggests 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. Formally, a skip link labeled a is set from the node α associated with the equivalence class $[W]$ to any other node β , associated with some class $[WVa]$ such that $V \neq \epsilon$ and a does not appear in V . Thus, a skip-edge labeled a is issued from α to each one of its closest descendants other than children where an original incoming edge labeled a already exists. As an example, Fig. 3 displays a partially augmented version of the DAWG of Fig. 2, with skip-edges added only from the source and its two adjacent nodes. We will use the words *full skip-edge DAWG* to refer to the structure that would result from adding skip-edges to all nodes of the DAWG. 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 labeled with a prefix of P . The following theorem is used to summarize the discussion.

THEOREM 1. For any string X of n symbols, the full skip-edge DAWG of X can be built in $O(n^2|\Sigma|)$ time and space, and it can be searched for all $rocc_i$ earliest realizations of prefixes of any pattern P of m symbols in time

$$O\left(\sum_{i=1}^m rocc_i \cdot i \cdot \log |\Sigma|\right).$$

Proof. Having built the DAWG in time $O(n \log |\Sigma|)$ by one of the existing methods, the augmentation itself is easily carried out in $O(n^2|\Sigma|)$ time and space, e.g., by adaptation of a depth-first visit of the DAWG, as follows. First, when the sink is first reached, it gets assigned *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 . Essentially, α inherits all skip-edges of β except for those that have the same label as the arc (α, β) . Thus, it suffices to take the union of such a reduced set of skip-edges from β with the set of skip-edges possibly defined during previous visits of α . The bound follows from the fact that for every node and symbol, skip-edges might have to be directed to $\Theta(n)$ other nodes.

The time bound on searches is subtended by an immediate consequence of Property 1. Namely, we have 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 full skip-edge 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 . Therefore, the search for P is trivially performed, e.g., as a depth-first visit of all longest paths in the full skip-edge DAWG that start at the source and are labeled by some prefix of P . (Incidentally, 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 to each time we backtrack from a node corresponds a prefix of P which cannot be continued along the path being explored, whence the claimed time bound for searches. ■

The search-time bound of Theorem 1 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 also be added to the construction phase, thereby further reducing the number of skip-edges issued. The main problem, however, is that storing a full skip-edge DAWG would take an unrealistic $\Theta(n^2)$ worst-case space even when the alphabet size is a constant (cf. Fig. 3). The remainder of our discussion is devoted to improving on this space requirement.

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 some path (say, for example, the longest one) from that node to the sink is labeled precisely by the suffix $a_i a_{i+1} \dots a_n$ 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.

Let the *depth* of a node v in the DAWG be the length of the longest word W on a path from the source to v . We have then that, by the definition of a DAWG, every other path from the source to v is labeled by one of the consecutive suffixes of W down to a certain minimum length (these are the words in the equivalence class $[W]$ that occur in X only as suffixes of W). It also follows that, considering the set of immediate predecessors of v on these paths, their depths must be mutually different and each smaller than $|W|$. Finally, the depths of the backbone nodes must be given by the 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. 4). The extension consists simply of replicating 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 depth-first visit of the DAWG resulting in our tree). We stipulate that

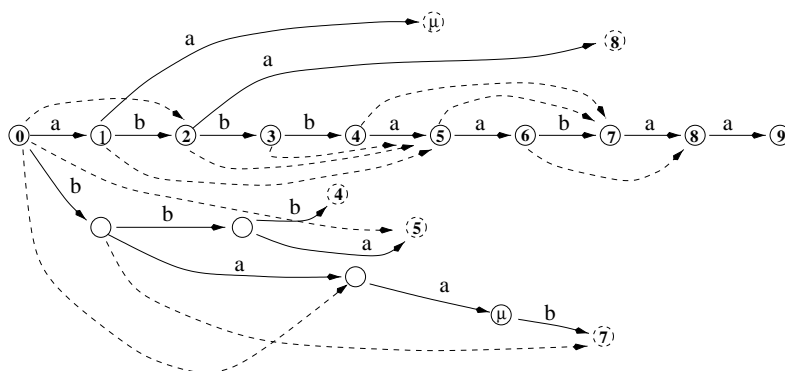


FIG. 4. An extended spanning tree T with sample skip-edges.

the duplicates of node ν are created in such a way that this will leave ν connected to the immediate predecessor μ of ν with the property that the depth of μ in the DAWG equals the depth of ν minus 1. Note that such a node μ must exist for each ν except the source. Note also that our spanning tree must contain a directed path that corresponds precisely to the backbone of the DAWG.

Let \mathcal{T} be the resulting structure. Clearly, \mathcal{T} has the same number of edges as the DAWG. Moreover, each node of the DAWG is represented in \mathcal{T} at most once for every incoming edge. Therefore, the size of \mathcal{T} is linear in $|X|$. 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 $O(n \cdot |\Sigma|)$, and 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 not consume that symbol. Their full structure and management will be explained in due course.

We now describe the augmentation of the DAWG. With reference 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 original 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 γ 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. 5). 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 original 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 perform the following two actions. First, direct a skip-edge labeled a from γ to η . Next, consider the subtree of \mathcal{T} rooted at γ . Any path of \mathcal{T} in this tree that does not lead eventually to an arc labeled a (like the arc leading to node η) must end on a leaf. To every such leaf, direct from γ a deferring edge labeled a . This second action is always performed in the special case where γ is the root.

- **Case 3:** Node γ is a leaf. If γ is the sink, nothing is done. Otherwise, let μ be the original DAWG node of which γ is a replica. Back on \mathcal{T} , for each symbol a of the alphabet, follow every distinct path from μ until encountering a first occurrence of a or the sink. For every such path with no intervening branching nodes, direct a skip-edge from the leaf γ to the node at the end of the path. For every path traversing and proceeding past a branching node, direct a deferring edge from γ to the deepest one among such branching nodes. At the end, eliminate possible duplicates among the deferring edges that were introduced.

Figures 4 and 5 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- or deferring 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. 5). 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- or deferring edge labeled a is defined. We refer to γ as the a -*backup* of β , and we denote it by $back_a(\beta)$. Note that, by our construction of Case 2, such a backup node exists always. Clearly, our intent is that the effect of traversing a skip-edge as described in the previous section can now be achieved by traversing a small sequence of auxiliary edges. In the example of Fig. 5, the transition from node α to η is implemented by traversing in succession one

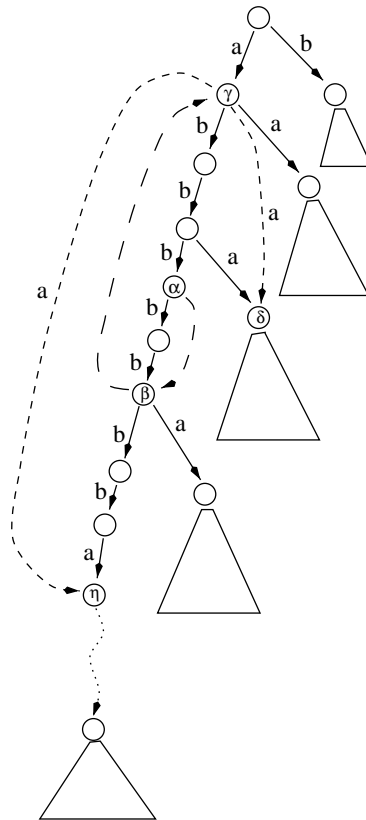


FIG. 5. A one-symbol transition from a node α of \mathcal{T} to its descendant η is implemented via three auxiliary-edge traversals: first, through a deferring link to the nearest branch node β ; next, from β to γ through a back-edge; finally, through the skip-edge from γ to η . To avoid unnecessarily cluttering the figure, not all edges are shown. 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 .

deferring edge, one back edge and one skip-edge. This complication is compensated by the following advantage.

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 auxiliary edge per alphabet symbol leaving nodes of outdegree 1, so that we can concentrate on branching nodes and leaves. As for the skip-edges directed from branching nodes, observe that for any symbol a and any node β , at most one skip-edge labeled a may reach β from some such node, due to the conventions made in Case 2. Indeed, if a skip-edge labeled a is set from some branching node α to another node β , then by construction no branching node on the path from α to β 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 . A similar argument holds for deferring edges directed towards every leaf of \mathcal{T} . Indeed, by the mechanics of Case 2, if a deferring edge labeled a is set from a branching node α to a leaf β , then there is no original edge labeled a on the path from α to β . Again, either α is the root or else there must be an original edge labeled a on the path from the closest branching ancestor of α to α itself, so that no deferring edge labeled a may be issued from a branching ancestor of α to a node in the subtree of \mathcal{T} rooted at α .

Considering now the leaves, observe first that at most one skip-edge per symbol may be directed from a leaf to a node. To get a global bound on all deferring edges set from leaves, consider the compact trie of all suffixes of X . This trie has $O(n)$ leaves and arcs, and every branching node of \mathcal{T} can be mapped in a distinct branching node in the trie (indeed, \mathcal{T} can be obtained figuratively by pruning the

noncompact version of the trie). We will map each one of the deferring edges set from a leaf γ into an arc of the trie and in such a way that each arc of the trie is charged at most one such deferring edge per alphabet symbol. To see this mapping, let W be the word from the root of \mathcal{T} to γ , and let $W' = WVa$ be the extension of W that completes one of the paths ending in a first occurrence of a after crossing some branching nodes in the DAWG. The deferring edge relative to W' is charged to the edge of the trie where W' ends. Observe that, for any other extension $W'' = WV'a$ of W appearing in X and such that a has no occurrence in V' , V and V' must diverge. In other words, W'' must have a path in the trie that diverges from that of W' and thus will charge an arc of the trie different from the one charged by W' . Moreover, since no prefix of W ends on a leaf of \mathcal{T} , then no ancestor of γ can produce the same charges. In conclusion, for each leaf of \mathcal{T} and symbol of Σ a distinct arc of the trie is charged, whence the total number of auxiliary edges of this kind per symbol is bounded by the length of X . ■

LEMMA 2. *P has a realization Y in X beginning with a_i and ending with 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 a sequence containing at most two deferring edges and at most one back edge; (iii) there is a path labeled $Y = a_i a_{i+1} \dots a_j$ in the DAWG from the source to the node $v = [Y]$ (a replica of) which is reached by the last arc of σ .*

Proof. The proof is by induction on the length m of the pattern. Let $m = 1$ and 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). For $m > 1$, assume that the claim holds for all patterns of lengths $1, 2, \dots, m - 1$ and that we have matched the prefix $P_{m-1} = b_1 b_2 \dots b_{m-1}$ of P down to some node α of \mathcal{T} while maintaining (i)–(iii). Hence, for some index $f < j$, there is a path from the root of \mathcal{T} to α labeled $Y_f = a_i a_{i+1} \dots a_f$ and such that Y_f is a realization of P_{m-1} .

Given that P has a realization Y , then there must be a path in the DAWG labeled $a_{f+1} \dots a_j$ and originating at the node represented in \mathcal{T} by α . The path itself has an image in \mathcal{T} , perhaps fragmented in a sequence of segments. Considering this image, the claim is then easily checked if the last symbol b_m of P is consumed through either an original arc or a defined skip-edge leaving α and shunting the path labeled $a_{f+1} \dots a_j$. Assuming that neither type of edge is defined at α , then by construction there must be a deferring edge labeled $a = b_m$ leading through a path labeled by a prefix $a_{f+1} \dots a_k$ of $a_{f+1} \dots a_j$ ($k < j$) either to a branching node, call it β , or to some leaf λ .

In this second case, we have either a defined skip-edge leading to the final node, which would conclude the argument, or one more deferring edge that will take from λ to a branching node. Considering such a branching node, it must be the image in \mathcal{T} of some node of the DAWG on the path from α labeled $a_{k+1} \dots a_j$. Hence from this moment on we can reset our reasoning and assume to be in the same case as if we were on a branching node β , except for the fact that we would now start with the handicap of having already consumed up to two deferring edges.

Assume then that we are on a branching node β , possibly having already traversed one or two deferring edge, and with some suffix of $a_{k+1} \dots a_j$, ($f < k < j$) still to be matched. Clearly, if β has a defined a -labeled skip- or original edge, this concludes the argument. Thus, the only remaining cases of interest occur when no a -labeled skip- or original edge is defined from β . In such an event, the link to $\gamma = \text{back}_a(\beta)$ is traversed instead, as depicted in Fig. 5.

Let η be any of the descendants of β such that η is connected to β through a nontrivial original path π of \mathcal{T} in which symbol a appears precisely as a suffix 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 η . 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 $\text{back}_a(\beta) = \gamma$. Therefore, if we choose η to be the node at the end of the path originating from β and labeled by the suffix of $a_{k+1} \dots a_j$ that remains at this point to be consumed, we have that a skip-edge labeled a must exist from γ to η . Traversal of this edge achieves propagation of points (i)–(iii) from $P_{m-1} = b_1 b_2 \dots b_{m-1}$ to P within the transitions of at most two deferring edges, one back edge and one skip-edge.

The proof of the converse is straightforward and thus is omitted. ■

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- or deferring edge labeled a leaving a node like the node γ of Fig. 5, with some such edges leading to descendants of γ that are not simultaneously descendants of α . In Fig. 5, an instance of such a situation is represented by node δ .

We assume that all auxiliary (i.e., skip- or deferring) 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 auxiliary edge from γ are found as consecutive entries in the auxiliary edge list of γ associated with symbol a . This list or part of it will be traversed from left to right in our search, as follows naturally from the structure of a depth-first visit of a graph. The convention is also made that the back edge from β to γ points directly to the auxiliary edge associated with the leftmost descendant of β . During a search, the beginning of the sublist at γ relative to descendants of β is immediately accessed from β in this way, and the skip- and deferring edges in that sublist are scanned sequentially while the subtree of \mathcal{T} rooted at β is being explored. The following theorem summarizes the discussion.

THEOREM 2. *The compact skip-edge DAWG associated with X supports the search for all earliest occurrences of a pattern $P = b_1 b_2 \dots b_m$ in X in time*

$$O\left(\sum_{i=1}^m \text{rocc}_i \cdot i \cdot \log |\Sigma|\right),$$

where rocc_i is the number of distinct realizations in X of the prefix $P_i = b_1 b_2 \dots b_i$ 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 to build the compact variant described here. Again, the core paradigm is a bottom-up computation on \mathcal{T} , except that this time lists of skip- and deferring edges may be assigned to branching nodes only on a temporary basis: whenever, climbing back toward the root from some node β , an ancestor branching node α is encountered before any intervening edges labeled a , then the a -labeled skip- and deferring edge lists of β are surrendered to α , and β is simultaneously appended to a list *Back* of branching nodes awaiting back edges. As soon as (because of an intervening original edge labeled a or having reached the root) the a -labeled lists are permanently assigned to some node α , appropriate back edges are also directed from every node in the list *Back*, and *Back* itself is disposed of. A similar process governs the introduction of skip- and deferring edges from leaves. Recall that a leaf of \mathcal{T} is in fact a replica of a same *confluence node* of the DAWG. This not only shows that it is possible to compute this class of auxiliary edges bottom-up, but also suggests that for a group of leaves replicating a same node of the DAWG it suffices to issue the edges at the node of \mathcal{T} that is the image of that DAWG node and let the replicas simply point to it. Note that as long as we insist on reasoning in terms of \mathcal{T} , these deferring edges from leaves must be suitably marked, lest they be confused with those issued at branching nodes and play havoc with the search as described in Lemma 2.

The overall process takes time and space linear in the structure at the outset, which is linear in $|X|$ for fixed alphabets. Symbol durations may be taken into account both during construction and 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 X , we face the choice of implementing 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 2, 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

$$O\left(n + \sum_{i=1}^m i \cdot \text{rocc}_i \log n\right)$$

can in fact be built within $O(n \log n)$ 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].

DEFINITION. With $p = j \bmod |\Sigma|$ ($j = 1, 2, \dots, n$), `CLOSE`[j] contains the smallest position larger than j where there is an occurrence of s_p , the p th symbol of the alphabet.

Thus, `CLOSE` is regarded as subdivided into blocks of size $|\Sigma|$, the entries of which are cyclically assigned to the different symbols of the alphabet. It is trivial to compute `CLOSE` from X , in linear time. Let now $\text{closest}(i, p)$ be the closest instance of s_p to the right of position i (if there is no such occurrence set $\text{closest}(i, p) = n + 1$). The following property holds.

LEMMA 3. *Given the table `CLOSE` and the sorted list of occurrences of s_p , $\text{closest}(i, p)$ can be computed for any p in $O(\log |\Sigma|)$ time.*

We refer to [2] for a proof of Lemma 3. The main idea is that two accesses of the form `CLOSE`[$\lfloor i/|\Sigma| \rfloor \cdot |\Sigma| + p$] and `CLOSE`[$\lfloor i/|\Sigma| \rfloor \cdot |\Sigma| + |\Sigma| + p$] either must 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, and 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 *inside* each chain of unary nodes present in \mathcal{T} . A key element in making this latter fact possible is the circumstance, already noted, that we can map every path to the sink of the DAWG, and 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: we may either scan the chain sequentially or search through it logarithmically. The first option results in adding to the overall time complexity a term linear in n ; the second requires additional *ad hoc* auxiliary links at the rate of at most $2 \log n$ per node, of which $\log n$ point upward and at most as many point downward. The overhead introduced by the second option is $O(\log n)$ per transition, which absorbs the $O(\log |\Sigma|)$ time possibly charged by closest . The same scheme can be adapted to get rid of skip-edges directed from the leaves.

We still face a potential of $\Theta(|\Sigma|)$ deferring edges per chain node and as many backup edges per branching node. These edges 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 $\gamma = \text{back}_a(\beta)$ there can be no edge labeled a , but such an edge must exist on the path from the closest branching ancestor of γ to 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 deferring edges that lead to leaves, these edges can be entirely suppressed at the cost of visiting the subtrees of \mathcal{T} involved at search time: this introduces work linear overall, since, e.g., in a breadth-first search it suffices to visit each subtree once.

Finally, we consider the collection of all deferring edges that originate at an arbitrary leaf. Recall that when a deferring edge labeled a is set from a leaf γ to a branching node β , this is done with the intent of making accessible during searches a final target node η that is found along a unary chain connecting β to its closest branching descendant (or leaf) ν . Specifically, η is the node at the end of the first edge labeled a on the chain connecting β to ν . In analogy with what was discussed earlier, η can be reached in logarithmic time from ν through an application of *closest*. The problem is thus how to be prepared to reach nodes such as ν , during searches, without dedicating one separate deferring edge to every such node.

In the terms of the discussion of Lemma 1, the idea could be again to coalesce in a same deferring edge all of those deferring edges from γ that would be charged to a same arc of the suffix trie of X and let *closest* discern at search time among the individual symbols present on that arc. In the specific case we are considering, this trie arc would be one that maps, in the DAWG, to the path connecting β to ν . However, this time this is not enough, since not every symbol of Σ is guaranteed to appear in every DAWG chain or trie arc. We must go one step further and coalesce all of the at most $|\Sigma|$ edges reaching down along a given path of the trie into the deepest one among those edges. The intent is that, during a search, the table *closest* will be used to climb back to the appropriate depth and symbol. We need to show that this is done consistently, i.e., that a connected path supporting this climb is guaranteed to exist in \mathcal{T} .

Let W be the word associated with γ and $W' = WV$ a shortest extension of W such that V contains at least one instance of every symbol of Σ and W' ends at a branching node of the suffix trie. Let \hat{W} and \hat{W}' be, respectively, the longest words in the equivalence classes $[W]$ and $[W']$, and recall that γ is a replica of the node of \mathcal{T} corresponding to \hat{W} . Clearly, there must be a path in the DAWG connecting the node of $[W]$ to that of $[W']$ and labeled V . Moreover, the DAWG node corresponding to $[W']$ must be a branching node, because such is the corresponding node in the trie. By our construction of \mathcal{T} , such a branching node must exist also in this tree, and it must be connected to the root through a path labeled \hat{W}' . Since V is a suffix of \hat{W}' , a connected path labeled V exists in \mathcal{T} as claimed.

We conclude by pointing out that all log factors appearing 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 [8]. We conjecture that the $\log n$ factors could be made to disappear entirely by resorting to amortized finger searches such as, e.g., in [2].

5. CONCLUSION

We have described a data structure suitable for reporting the occurrence of a pattern string as a constrained subsequence of another string. Since the full-fledged data structure would be too bulky in practical allocations, a more compact, sparse version was built where space saving is traded in exchange for some overhead on search time. Both of these parameters are perhaps susceptible of further improvement. In particular, it is not clear that the bounds attained for fixed alphabet sizes cannot be extended without penalty to the case of an unrestricted alphabet. Nontrivial estimates or bounds on the terms $rocc_i$ that appear in our complexities may shed more light on the expected or worst case performance of a search. Finally, little is known about indices that would return, in time linear in the pattern size, whether or not any given pattern occurs as an episode subsequence of the textstring.

ACKNOWLEDGMENTS

We are indebted to the referees for their thorough scrutiny of the original version of this paper and for their many valuable comments.

REFERENCES

1. Apostolico, A., and Galil, Z. (Eds.) (1997), *Pattern Matching Algorithms*, Oxford University Press, New York.
2. Apostolico, A., and Guerra, C. (1987), The longest common subsequence problem revisited, *Algorithmica* **2**, 315–336.
3. Baeza-Yates, R. (1991), Searching subsequences, *Theoret. Comput. Sci.* **78**, 373–376.

4. Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M. T., and Seiferas, J. (1985), The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40**, 31–55.
5. Cobbs, A. (1995), Fast approximate matchings using suffix trees, in *CPM'95, Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching* (Z. Galil and E. Ukkonen, Eds.), Lecture Notes in Computer Science, Vol. 937, pp. 41–54, Springer-Verlag, Berlin/New York.
6. Crochemore, M., and Rytter, W. (1994), *Text Algorithms*, Oxford University Press, New York.
7. Das, G., Fleischer, R., Gąsieniek, L., Gunopulos, D., Kärkkäinen, J. (1997), Episode matching, in *CPM'97, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching* (A. Apostolico and J. Hein, Eds.), Lecture Notes in Computer Science, Vol. 1264, pp. 12–27, Springer-Verlag, Berlin/New York.
8. Johnson, D. B. (1982), A priority queue in which initialization and queue operations take $O(\log \log n)$ time, *Math. System Theory* **15**, 295–309.
9. Kumar, S., and Spafford, E. H. (1994), A pattern-matching model for intrusion detection, in *Proceedings of the National Computer Security Conference*, pp. 11–21.
10. Mannila, H., Toivonen, H., and Vercamo, A. I. (1997), Discovering frequent episodes in event sequences, *Data Mining Knowledge Discovery* **1**, 259–289.
11. Ukkonen, E. (1993), Approximate string matching with suffix trees, in *CPM'93, Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching* (A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, Eds.), Lecture Notes in Computer Science, Vol. 684, pp. 228–242, Springer-Verlag, Berlin/New York.
12. Waterman, M. (1995), *Introduction to Computational Biology*, Chapman & Hall, London.