Matching a Set of Strings with Variable Length Don't Cares

Gregory Kucherov and Michaël Rusinowitch [*]

1 Introduction

Given an alphabet A, a *pattern* p is a sequence $(v_1, \dots, v_m)$ of words from A^* called *keywords.* We represent p as a single word $v_1@\dots@v_m$, where $@ \notin A$ is a distinguished symbol called *variable length don't care symbol.* Pattern p is said to *match* a text $t \in A^*$ if $t = u_0v_1u_1\dots u_{m-1}v_mu_m$ for some $u_0, \dots, u_m \in A^*$. In this paper we address the following problem: given a set P of patterns and a text t, test whether one of the patterns of P matches t.

Quoting Fisher and Paterson in the concluding section of [10], "a good algorithm for this (problem) would have obvious practical applications". For instance, as it was reported by Manber and Baeza-Yates [13], the DNA pattern TATA often appears after the pattern CAATCT within a variable length space. It may therefore be interesting to look for the general pattern CAATCT@TATA. If we are given a set of such general patterns, it is desirable to have an algorithm that searches for all of them simultaneously instead of searching consecutively for each one.

In this paper we propose an algorithm that solves the problem in time $O((|t| + |P|)\log|P|)$, where $|t|$ is the length of the text and $|P|$ is the total length of all keywords of P.

Several variants of the problem have been considered in the literature. Matching set of strings with "unit length don't care symbols" that match any individual letter, was studied in [10, 15]. Bertossi and Logi [5] have proposed an efficient parallel algorithm for finding in a text the occurrences of a single pattern with variable length don't-care symbols. Their algorithm has an $O(\log|t|)$ running time on $O(|t||P|/\log|t|)$ processors.

Our problem can also be viewed as testing membership of a word in a regular language of type $\cup_{i=1}^{n} A^*u_1^iA^*u_2^iA^*\dots A^*u_{m_i}^iA^*$. Note that any regular expression where the star operation only applies to the subexpression A (i.e. the union of all letters) can be reduced to the above form by distributing concatenation over union. An $O(|t||E|/\log|t|)$ solution for the case of a general regular expression E has been given by Myers [14].

The algorithm we propose here reads the text and the patterns from different tapes in the left-to-right fashion. The text is searched on-line, which means that the match is reported immediately after reading the shortest matched portion of the

*INRIA-Lorraine and CRIN/CNRS, Campus Scientifique, 615, rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy, France, email: {kucherov,rusi}@loria.fr

text. Moreover, every pattern is read in the on-line fashion too, in the sense that the algorithm starts reading a keyword in a pattern only when all previous keywords of this pattern have been found in the text. This allows keywords to be specified dynamically, possibly depending on the search situation, for example on the keywords of other patterns that have been found by that moment. This feature of the algorithm makes it, we believe, particularly useful for some applications.

In contrast to most of the existing string matching algorithms(see [1]) our algorithm is not composed of two successive stages – preprocessing the pattern (resp. the text) and reading through the text (resp. the pattern) – but has these two stages essentially interleaved. The basic data structure used in the algorithm is the DAWG (Directed Acyclic Word Graph) [6, 7]. The DAWG is a flexible and powerful data structure related to suffix trees and similar structures (see [1, section 6.2] for references and [17] for one of the most recent works). In particular, the DAWG was used in [6, 7] as an intermediate structure for constructing in linear time the minimal *factor automaton* for a (set of) word(s). An elegant linear time on-line algorithm for constructing the DAWG was proposed in these papers. Independently, the DAWG for a single word was studied by Crochemore [8, 9] under the name of *suffix automaton.* In particular, in [9] he has extended the DAWG to a matching automaton, similar to the well-known Aho-Corasick automaton, to derive a new string matching algorithm. The algorithm we propose in this paper uses on the one hand, Crochemore's idea of using the DAWG for string matching and on the other hand, the efficient DAWG construction given in [6, 7].

The paper is organized as follows. In Section 2 we present the DAWG and define on top of it our basic data structure. Section 3 explains how to modify the DAWG, namely how to append a letter to a keyword and how to unload a keyword from the DAWG. In Section 4 the DAWG is further extended to be used as a matching automaton for solving the variable length don't care problem. The pattern matching algorithm is then detailed, its correctness is proved, and its complexity is evaluated. Finally, concluding remarks are made in the last section.

2 The DAWG

2.1 Definitions and main properties

Our terminology and definitions of this section basically follow [7].

$|v|$ denotes the length of $v \in A^*$. If $v = v_1 w v_2$, then w is said to occur in v at *position* $|v_1|$ and at *end position* $|v_1 w|$. For $D = \{v_1, \ldots, v_n\}$, a position (resp. end position) of w in D refers to a pair $< i, j >$, where j is a position (resp. end position) of w in v_i. $\textit{end-pos}_D(w)$ is the set of all possible end positions of w in D. $\textit{pref}(v)$ (resp. $\textit{pref}(D)$) stands for the set of prefixes of v (resp. prefixes of the words from D). Similarly, $\textit{suff}(v)$ ($\textit{suff}(D)$) and $\textit{sub}(v)$ ($\textit{sub}(D)$) denote the set of suffixes and subwords respectively. ε denotes the empty word.

Our basic data structure is the *Directed Acyclic Word Graph (DAWG)* [6, 7].

Definition 1 *Let* $D = \{v_1, \ldots, v_n\} \subseteq A^*$. *For* $u, v \in \textit{sub}(D)$, *define* $u \equiv_D v$ *iff* $\textit{end-pos}_D(u) = \textit{end-pos}_D(v)$. $[u]_D$ *denotes the equivalence class of* u *w.r.t.* $\equiv_D$. *The DAWG* $\mathcal{A}_D$ *for* D *is a directed acyclic graph with set of nodes* $\{[u]_D | u \in \textit{sub}(D)\}$ *and* *set of edges* $\{([u]_D,[ua]_D) | u, ua \in sub(D), a \in A\}$. *The edges are labeled by letters in* A *so that the edge* $([u]_D,[ua]_D)$ *is labeled by* a. *The node* $[\varepsilon]_D$ *is called the* source *of* $\mathcal{A}_D$.

An example of a DAWG is given in Appendix A0. Viewed as a finite automaton with every state being accepting, the DAWG is a deterministic automaton recognizing the subwords of D. Moreover, except for accepting states, the DAWG is isomorphic to the *minimal* deterministic automaton recognizing the suffixes of D, where syntactically equal suffixes of different keywords are considered to be different. Formally, this automaton can be obtained by appending at the end of each $v_i \in D$ a distinct fresh symbol $\$_i$, then constructing the minimal deterministic automaton for the suffixes of the modified set, and then forgetting the accepting sink state together with all incoming $\$_i$-transitions. This construction ensures the uniqueness of the DAWG, the property that will be tacitly used throughout the paper. If D consists of a single keyword, this automaton called *suffix automaton* is just the minimal deterministic automaton recognizing the suffixes of D [8, 9].

The reader is referred to [6, 7] for a more detailed analysis of the DAWG, in particular for linear bounds on its size and the relationship between the DAWG and the suffix tree. The following property allows us to define an important tree structure on the nodes of $\mathcal{A}_D$.

Proposition 1 *For* $u, v \in suff(D)$, *if* $end\text{-}pos_D(u) \cap end\text{-}pos_D(v) \neq \emptyset$, *then either* $u \in suff(v)$ *or* $v \in suff(u)$. *This implies that*

(i) every $\equiv_D$*-equivalence class has a longest element called* the representative *of this class,*

(ii) if $end\text{-}pos_D(u) \cap end\text{-}pos_D(v) \neq \emptyset$, *then either* $end\text{-}pos_D(u) \subsetneq end\text{-}pos_D(v)$ *or* $end\text{-}pos_D(v) \subsetneq end\text{-}pos_D(u)$.

Property (ii) ensures that the subset relation on equivalence classes defines a tree structure on the nodes. If $end\text{-}pos_D(u) \subset end\text{-}pos_D(v)$, and for no $w \in sub(D)$, $end\text{-}pos_D(u) \subset end\text{-}pos_D(w) \subset end\text{-}pos_D(v)$, then we say that there exists a *suffix pointer* going from $[u]_D$ to $[v]_D$. $[u]_D$ is said to be a *child* of $[v]_D$ and $[v]_D$ the *parent* of $[u]_D$. The source of $\mathcal{A}_D$ is the root of this tree. The sequence of suffix pointers going from some node to the source is called the *suffix chain* of this node. The following lemma clarifies the relation between two nodes linked by a suffix pointer.

Lemma 1 ([7]) *Let* u *be the representative of* $[u]_D$. *Then any child of* $[u]_D$ *can be expressed as* $[au]_D$ *for some* $a \in A$.

If $au \in sub(D)$ (resp. $ua \in sub(D)$) for some $a \in A$, $u \in A^*$, then we say that a is a *left context* (resp. *right context*) of u in D. The lemma above shows that if u is the representative of $[u]_D$, then every child of $[u]_D$ corresponds to a distinct left context of u in D. This implies that each node has at most $|A|$ children. In contrast, edges of the DAWG refer to the *right context*: for every right context a of $u \in sub(D)$, the DAWG contains the edge $([u]_D,[ua]_D)$ labeled by a.

The following fact related to lemma 1 is also enlightening.

Lemma 2 *$u \in sub(D)$ is the representative of $[u]_D$ iff either u is a prefix of some keyword in D, or u has two or more distinct left contexts in D.*

The edges of the DAWG are divided into two categories: Assume that u is the representative of $[u]_D$. The edge $([u]_D, [ua]_D)$ is called *primary* if ua is the representative of $[ua]_D$, otherwise it is called *secondary*. The primary edges form a spanning tree of the DAWG rooted at the source. This tree can be also obtained by taking in the DAWG only the longest path from the source to each node. With each node $[u]_D$ we associate a number $depth([u]_D)$ which is defined as the depth of $[u]_D$ in the tree of primary edges. Equivalently, $depth([u]_D)$ is the length of the representative of $[u]_D$. Note that if the edge $([u]_D, [ua]_D)$ is primary, then $depth([ua]_D) = depth([u]_D) + 1$, otherwise $depth([ua]_D) > depth([u]_D) + 1$.

If $w \in pref(v_i)$ for some $v_i \in D$, then we call $[w]_D$ a *prefix node for v_i*. Note that by lemma 2, w is the representative of $[w]_D$. Besides, if $w = v_i$ for some $v_i \in D$, then the node $[w]_D$ is also called a *terminal node* for v_i.

2.2 Data structure

We assume that each node α of the DAWG is represented by a data structure providing the following attributes:

out(α, a): a reference to the target node of the edge issuing from α and labeled by a; out(α, a) = undefined when there is no such edge,

type(α, a): type(α, a) = primary if the edge issuing from α and labeled by a is primary, otherwise type(α, a) = secondary,

suf-pointer(α): a reference to the node pointed by the suffix pointer of α; suf-pointer(α) = undefined if α is the source,

depth(α): $depth(\alpha)$,

terminal(α): terminal(α) = null if α is not a terminal node, otherwise terminal(α) refers to a list of keywords for which α is terminal (we do not assume that all keywords are different and therefore a node can be terminal for several keywords). The list will be defined more precisely in section 4.2.

origin(α): a reference to the node that the primary edge to α comes from,

last-letter(α): the label of incoming edges to α (equivalently, the last letter of any word in α),

number-of-children(α): has three possible values $\{0, 1, \text{more-than-one}\}$. number-of-children$(\alpha)$ = 0 (respectively number-of-children(α) = 1, number-of-children(α) = more-than-one) if there are no (respectively one, more than one) suffix pointers that point to α,

child(α): refers to the only child of α when number-of-children$(\alpha) = 1$,

prefix-degree(α): the number of keywords in D for which α is a prefix node. prefix-degree$(\alpha) = 0$ if α is not a prefix node.

out and type implement the DAWG itself, the other attributes are needed for different purposes that will become clear in the following sections. We will use the same denotation $\mathcal{A}_D$ for the whole data structure described above. An example of the data structure is given in Appendix A0.

We always assume the alphabet to be of a fixed size. We assume the uniform RAM model of computation and then assume that retrieving, modifying and comparing any attribute values as well as creating and deleting a DAWG node is done in constant time.

3 Modifying a DAWG

3.1 Appending a letter to a keyword

A.Blumer, J.Blumer, Haussler, McConnell and Ehrenfeucht [7] (BBHME for short) proposed an algorithm to construct $\mathcal{A}_D$ for a given set D in time $O(|D|)$. The algorithm processes consecutively the patterns of D such that if $\{v_1, \ldots, v_i\}$ have been already processed, then the constructed data structure is $\mathcal{A}_{\{v_1,\ldots,v_i\}}$. Processing a pattern v_{i+1} (equivalently, extending $\mathcal{A}_{\{v_1,\ldots,v_i\}}$ to $\mathcal{A}_{\{v_1,\ldots,v_i,v_{i+1}\}}$) is called *loading* v_{i+1} (to $\mathcal{A}_{\{v_1,\ldots,v_i\}}$). Loading v_{i+1} to $\mathcal{A}_{\{v_1,\ldots,v_i\}}$ is done by scanning v_{i+1} from left to right such that if $w \in pref(v_{i+1})$ is an already processed prefix of v_{i+1}, then the constructed data structure is $\mathcal{A}_{\{v_1,\ldots,v_i,w\}}$. Therefore, processing patterns in the set as well as letters in the pattern is done in the *on-line* fashion, and a basic step of the algorithm amounts to extending $\mathcal{A}_{\{v_1,\ldots,v_i,w\}}$ to $\mathcal{A}_{\{v_1,\ldots,v_i,wa\}}$ for some $a \in A$.

The BBHME data structure has only attributes out, type and suf-pointer. However, the BBHME algorithm can be easily extended to maintain the additional attributes that we need for our purposes. Since the BBHME algorithm is fundamental for this paper, we give its pseudocode in appendix A1. We shortly comment the algorithm below.

Function APPEND-LETTER implements the main step. It takes the terminal node of w in $\mathcal{A}_{\{v_1,\ldots,v_i,w\}}$ and a letter a and outputs the terminal node for wa in $\mathcal{A}_{\{v_1,\ldots,v_i,wa\}}$. APPEND-LETTER creates, if necessary, a new node for $[wa]_{\{v_1,\ldots,v_i,wa\}}$, and then traverses the suffix chain of the node $[w]_{\{v_1,\ldots,v_i,w\}}$ (installing secondary edges to the new node) up to the first node with an outcoming a-edge. If this edge is primary, no further traversals have to be done. If it is secondary, the function SPLIT is called which creates another new node, installs its outcoming edges, updates suffix pointers, and then continues the traversal unless a node with a primary outcoming a-edge is found. Thus, at most two new nodes are created and the suffix chain of $[w]_{\{v_1,\ldots,v_i,w\}}$ is traversed up to the first primary outcoming a-edge.

In the paper we will be modifying the BBHME algorithm. In particular, some instructions will be added to the SPLIT function, which is indicated at line 4 of its code in appendix A1.

Functions APPEND-LETTER and SPLIT maintain additional attributes origin, last-letter, depth, number-of-children and child. As for origin, last-letter and depth, this is explicitely shown in the algorithm. number-of-children and child are updated every time the tree of suffix pointers is modified (lines 12,14,15 in APPEND-LETTER and lines 6,7 in SPLIT). Maintaining number-of-children is trivial. child can be implemented by organizing the set of children of each node in a double-linked list and keeping a pointer to the first child in the list. Deleting a child then takes time $O(1)$.

LOAD-KEYWORD(v) loads a keyword v by scanning it and iterating the APPEND-LETTER function. Also, LOAD-KEYWORD maintains the prefix-degree attribute. Maintaining terminal will be considered later.

The remarkable property of the algorithm, shown in [6, 7], is that it builds the DAWG for a set D in time $O(|D|)$ by iterating LOAD-KEYWORD(v) for every $v \in D$, starting with a one-node DAWG. Actually, loading an individual keyword v into $\mathcal{A}_D$ takes time linear on $|v|$ *regardless of the set* D.

Lemma 3 LOAD-KEYWORD(v) *runs in time* $O(|v|)$.

3.2 Unloading a keyword

In this section we give an algorithm that unloads a keyword v_{i+1} from $\mathcal{A}_{\{v_1,\dots,v_i,v_{i+1}\}}$. Starting from the terminal node $[v_{i+1}]_{\{v_1,\dots,v_i,v_{i+1}\}}$, the algorithm traces back the chain of primary edges and at each step undoes the modifications caused by appending a corresponding letter. Thus, the main step is inverse to APPEND-LETTER and amounts to transforming $\mathcal{A}_{\{v_1,\dots,v_i,wa\}}$ into $\mathcal{A}_{\{v_1,\dots,v_i,w\}}$ for $wa \in pref(v_{i+1})$. The modifications to be applied to the nodes of $\mathcal{A}_{\{v_1,\dots,v_i,wa\}}$ are described in the following lemma which is in a sense inverse to lemma 2.1 from [6].

Lemma 4 *(i) wa is not the representative of an equivalence class w.r.t. $\equiv_{\{v_1,\dots,v_i,w\}}$ iff either $wa \notin sub(\{v_1,\dots,v_i,w\})$ or wa has only one left context in $\{v_1,\dots,v_i,w\}$ (and hence has only one child) and $wa \notin pref(\{v_1,\dots,v_i,w\})$. In the first case the class $[wa]_{\{v_1,\dots,v_i,wa\}}$ is deleted. In the second case this class is merged with its child.*

(ii) Let $wa = u_1u_2a$ and u_2a is the representative of the class pointed to by the suffix pointer of $[wa]_{\{v_1,\dots,v_i,wa\}}$. Then u_2a is not the representative of an equivalence class w.r.t. $\equiv_{\{v_1,\dots,v_i,w\}}$ iff u_2a has only one left context in $\{v_1,\dots,v_i,w\}$ and $u_2a \notin pref(\{v_1,\dots,v_i,w\})$. In this case $[u_2a]_{\{v_1,\dots,v_i,wa\}}$ is merged with its single child.

(iii) There are no other modifications in the equivalence classes except those given in (i) and (ii).

The transformation of $\mathcal{A}_{\{v_1,\dots,v_i,wa\}}$ into $\mathcal{A}_{\{v_1,\dots,v_i,w\}}$ is done by modifying the DAWG according to lemma 4. The algorithm is given in appendix A2. Its short account follows.

Let *activenode* be $[wa]_{\{v_1,\dots,v_i,wa\}}$. A main function DELETE-LETTER takes *activenode*, finds the node $[w]_{\{v_1,\dots,v_i,wa\}}$ called *newactivenode* by retrieving origin(*activenode*), and then proceeds by case analysis. If *activenode* is a prefix node for a keyword other than v_{i+1}, or has two or more children, then no more work has to be done. If *activenode* is not a prefix node of any other keyword and has only one child, then it should be merged with this child which is done by an auxiliary function MERGE. Finally, if *activenode* has no children and is not a prefix node of any other keyword, this means that $wa \notin \{v_1,\dots,v_i,w\}$ and therefore *activenode* should be deleted. Before it is deleted, the suffix chain of *newactivenode* is traversed and outcoming secondary a-edges leading to *activenode* are deleted. Let *varnode* be the first node on the suffix chain with outcoming primary a-edge. It can be shown that *varnode* is actually $[u_2]_{\{v_1,\dots,v_i,wa\}}$ (lemma 4(ii)), and the node that this edge leads to, called *suffixnode*, is $[u_2a]_{\{v_1,\dots,v_i,wa\}}$. Once *suffixnode* is found, *activenode* is deleted together with its suffix pointer pointing to *suffixnode*. If *suffixnode* is a prefix node or has more than one child left, the algorithm terminates. Otherwise *suffixnode* has to be merged with its single child which is done by the MERGE function. MERGE acts inversely to the SPLIT function (see appendix A2 for details). Note that MERGE continues the traversal of *varnode* up to the first node with outcoming primary a-edge.

Similarly to the loading case, we will be extending the algorithm afterwards. In particular, instructions will be added to DELETE-LETTER and MERGE at line 9 and 2 respectively.

Maintenance of additional attributes origin, last-letter, depth, number-of-children and child is done similarly to the loading case.

UNLOAD-KEYWORD($[v]_D$) unloads pattern $v \in D$ from $\mathcal{A}_D$ by iterating DELETE-LETTER. Also, UNLOAD-KEYWORD maintains the prefix-degree attribute. Like in the case of insertion, UNLOAD-KEYWORD runs in time $O(|v|)$ *regardless of* D.

Lemma 5 UNLOAD-KEYWORD($[v]$) *runs in time* $O(|v|)$.

The proof goes along the same lines as for the case of insertion.

4 Matching a set of strings with variable length don't cares

4.1 Extending the DAWG for string matching

Crochemore noticed [9] that in the case of one keyword the DAWG can be used as a string matching automaton similar to that of Aho-Corasick, where suffix pointers play the role of failure transitions. The idea is to extend the current state *currentnode* with a counter *length* updated at each transition step. The procedure below describes a basic step of the algorithm. *currentletter* is assumed to be the current letter in the text.

```
UPDATE-CURRENT-NODE(currentnode, length, currentletter)
    while out(currentnode, currentletter) = undefined do
        if currentnode = source then
            return < currentnode, 0 >
        else currentnode := suf-pointer(currentnode)
            length := depth(currentnode)
    currentnode := out(currentnode, currentletter)
    length := length + 1
    return < currentnode, length >
```

The meaning of *currentnode* and *length* is given by the following proposition which is an extension of Proposition 2 of [9] for the multiple keyword case.

Proposition 2 *Assume that* D *is a set of keywords and* $t = t_1 \ldots t_n$ *is a text. Consider* $\mathcal{A}_D$ *and iterate* UPDATE-CURRENT-NODE(*currentnode*, *length*, *currentletter*) *on* t *with initial values currentnode = source, length* $= 0$, *and currentletter* $= t_1$. *At any step, if* $t_1 \ldots t_i$ *is the prefix of* t *scanned so far, and* w *is the longest word from* $suff(\{t_1 \ldots t_i\}) \cap sub(D)$, *then* w *belongs to currentnode (regarded as an equivalence class) and length* $= |w|$.

Crochemore used proposition 2 as a basis for a linear string matching algorithm in the case of a single keyword. An occurrence of the keyword is reported iff *currentstate* is terminal and, in addition, the current value of *length* is equal to depth(*currentnode*). The current position in the text is then the end position of the keyword occurrence. The linearity of the algorithm of proposition 2 can be shown using the same arguments as for the Aho-Corasick algorithm.

However, this idea does not extend to the multiple keyword case, since one or several keywords may occur at the current end position in the text even if *currentnode* is not terminal. This is the case for the keywords that are suffixes of $t_1 \ldots t_i$ shorter than the current value of *length*. To detect these occurrences, at every call to UPDATE-CURRENT-NODE the suffix chain of *currentnode* should be traversed and a match should be reported for every terminal node on the chain. A naive implementation of this traversal would lead to a prohibitive $O(|t||D|)$ search time.

One approach to the problem would be to attach to each node a pointer to the closest terminal node on the suffix chain. When the set of keywords is fixed once for all, this approach amounts to an additional preprocessing pass which can be done in time $O(|D|)$ and therefore does not affect the overall linear complexity bound. However, when the set of keywords is changing over time, which is our case, this approach becomes unsatisfactory, since modifying a single keyword may require $O(|D|)$ operations.

String matching for a changing set of keywords has been recently studied in the literature under the name of *dynamic dictionary matching*. Several solutions have been proposed [2, 3, 11, 4]. All of them, however, had to face a difficulty similar to the one described above. In terms of data structure, the problem amounts to finding for a node of a dynamically changing tree (in our case, tree of suffix pointers) the closest marked ancestor node (in our case, terminal node), where nodes are also marked and unmarked dynamically.

In this paper we borrow the solution proposed in [3] which consists in using the dynamic trees of Sleator and Tarjan [16]. The tree of suffix pointers is split into a forest by deleting all suffix pointers of terminal nodes. Thus, every terminal node in the tree becomes the root node of a tree in the forest. The forest is implemented using the dynamic trees technique of [16]. For shortness, we will call the DAWG augmented with this data structure the *extended DAWG*. Since finding the closest terminal node on the suffix chain of a node amounts to finding the root of its tree in the forest, this operation takes $O(\log |D|)$ time. We will denote by CLOSEST-TERMINAL(α) a function which implements this operation. It returns the closest terminal node on the suffix chain of α if such a node exists, and returns undefined otherwise.

On the other hand, creating, deleting and redirecting suffix pointers takes no longer a constant time, but time $O(\log |D|)$. Since both APPEND-LETTER and DELETE-LETTER requires a constant number of such operations, we restate lemmas 3 and 5 as follows.

Lemma 6 *On the extended DAWG,* LOAD-KEYWORD(v) *and* UNLOAD-KEYWORD($[v]$) *run in time* $O(|v| \log |D|)$.

4.2 Pattern matching algorithm

Assume that P is a finite set of strings $\{p_1, \ldots, p_n\}$ over $A \cup \{@\}$, where each $p_i \in P$ is written as $v_1^i @ v_2^i @ \ldots @ v_{m_i}^i$, for some $v_1^i, v_2^i, \ldots, v_{m_i}^i \in A^*$. According to our terminology, p_i's are called *patterns* and v_j's *keywords*. A pattern p_i matches a text $t \in A^*$, if $t = u_1 v_1^i u_2 \ldots u_{m_i} v_{m_i}^i u_{m_i+1}$ for some $u_1, u_2, \ldots, u_{m_i}, u_{m_i+1} \in A^*$. We address the following problem: given a set of patterns P and a text $t = t_1 \ldots t_k$, test whether one of the patterns of P matches t.

We assume that the text and every pattern is read from left to right from a separate input tape.

Let us first give an intuitive idea of the algorithm. At each moment of the text scan, the algorithm searches for a group of keywords, one from each pattern, represented by the DAWG. The search is done using the DAWG as an automaton similar to proposition 2. Every time a keyword is found, it is unloaded from the DAWG and the next keyword in the corresponding pattern is loaded instead. The crucial point is that the loading process is "spread over time" so that loading one letter of the keyword alternates with processing one letter of the text. In this way the correctness of the algorithm is ensured. Thus, unlike the usual automata string matching technique, the underlying automaton evolves over time adapting to the changing set of keywords.

Let us turn to a formal description. Let $t[1:l] = t_1 \ldots t_l$ be a prefix of t scanned so far. For every $p_i = v_1^i @ v_2^i @ \ldots @ v_{m_i}^i \in P$, consider a decomposition

$$t[1:l] = u_1^i v_1^i u_2^i \ldots v_{j_i-1}^i u_{j_i}^i \tag{1}$$

for $j_i \in [1, m_i], u_1^i, u_2^i, \ldots, u_{j_i}^i \in A^*$, such that

- for every $r \in [1, j_i - 1]$, $v_r^i \notin sub(u_r^i v_r^i) \setminus suff(u_r^i v_r^i)$,
- $v_{j_i}^i \notin sub(u_{j_i}^i)$.

Clearly, under the conditions above, decomposition (1) is unique. The intuition is that the *leftmost* occurrence of each pattern is looked for, that is the leftmost occurrence of every keyword that follows the occurrence of the preceding keyword.

Consider decompositions (1) for every $i \in [1, n]$. We now define the state of the matching process after l letters of the text have been processed. We first introduce some terminology. For every $i \in [1, n]$, $v_{j_i}^i$ is called an *active* keyword. If $|u_{j_i}^i| < |v_{j_i}^i|$ for $i \in [1, n]$, then both the pattern p_i and its active keyword $v_{j_i}^i$ are said to be *under loading*. For each $i \in [1, n]$, define $\bar{v}_{j_i}^i = v_{j_i}^i[1:q]$ where $q = min\{|u_{j_i}^i|, |v_{j_i}^i|\}$. Thus, if p_i is under loading then $\bar{v}_{j_i}^i$ is the prefix of $v_{j_i}^i$ of length $|u_{j_i}^i|$, otherwise $\bar{v}_{j_i}^i = v_{j_i}^i$. The current situation of the matching process is represented by a data structure consisting of three components given below together with their invariant conditions:

1. The extended DAWG for the set $V = \{\bar{v}_{j_1}^1, \ldots, \bar{v}_{j_n}^n\}$, defined as in Sections 2, 4.1, except that if $v_{j_i}^i$ is a keyword under loading, then the node $[\bar{v}_{j_i}^i]_V$ is not considered terminal. We call these nodes *port nodes*. Intuitively, a port node refers to a node in the DAWG to which the next letter of the corresponding keyword under loading should be appended. If $v_{j_i}^i$ has been completely loaded, that is $\bar{v}_{j_i}^i = v_{j_i}^i$, then $[\bar{v}_{j_i}^i]_V$ is terminal for $v_{j_i}^i$ and the list terminal($[\bar{v}_{j_i}^i]$) contains a reference to p_i. In other words, if α is a terminal node, terminal(α) is the list of patterns p_i such that α is terminal for the active keyword of p_i.

2. A distinguished node in the DAWG called *currentnode*, together with a counter *length*. *currentnode* is the node $[w]_V$, where w is the longest word in $\mathit{suff}(t[1:l]) \cap \mathit{sub}(\{\bar{v}^1_{j_1}, \ldots, \bar{v}^n_{j_n}\})$, and $\mathit{length} = |w|$.

3. A double linked list of patterns under loading each element of which has a reference to the corresponding pattern p_i and a reference to the corresponding port node in the DAWG.

A basic step of the algorithm consists of three stages. First, for each keyword under loading, the next letter is inserted into the DAWG using the APPEND-LETTER procedure and the port node is updated. If the keyword has been loaded completely, then the corresponding port node becomes terminal and the corresponding pattern is deleted from the list of patterns under loading. Secondly, *currentnode* and *length* are updated using the UPDATE-CURRENT-NODE procedure. Finally, the suffix chain of *currentnode* is traversed looking for terminal nodes. Each such node corresponds to one or several active keywords that occur at the current end position in the text. Each detected matching keyword is unloaded from the DAWG using the UNLOAD-KEYWORD algorithm, and the following keyword in the pattern becomes under loading with the source being the port node.

To define the algorithm, functions SPLIT, UNLOAD-KEYWORD and MERGE from Section 3 should be slightly modified. The reason for modifying SPLIT is that the node which has to be split (*targetnode* in the SPLIT algorithm) may happen to be the actual value of *currentnode*. *currentnode* should then be updated so that condition 2 above be preserved. The following instruction has to be inserted into the SPLIT algorithm at line 4 (Appendix A1).

```
4.1   if $currentnode = targetnode$ then
4.2       if $length \leq \mathrm{depth}(newtargetnode)$ then $currentnode := newtargetnode$
```

Similarly, each of the functions DELETE-LETTER and MERGE may be led to delete a node which is actually *currentnode*, in which case *currentnode* must be updated. Again, the new value is computed in order to preserve condition 2. The following instructions have to be inserted into the DELETE-LETTER algorithm at line 9 (Appendix A2).

```
9.1       if $currentnode = activenode$ then
9.2           $currentnode := \mathrm{suf\text{-}pointer}(activenode)$
9.3           $length := \mathrm{depth}(currentnode)$
```

The instruction below has to be inserted into the MERGE algorithm at line 2 (Appendix A2).

```
2   if $currentnode = targetnode$ then $currentnode := newtargetnode$
```

Note that modified functions SPLIT, DELETE-LETTER and MERGE may now change *currentnode* and *length* as a side effect. The modifications will be further discussed in Section 4.3.

We are now ready to give the complete algorithm. t denotes a subject text and READ-LETTER(t) returns the scanned letter. For a pattern p under loading, READ-LETTER(p) returns the next letter of p, and PORT-NODE(p) refers to the corresponding port node.

```
MATCH($t, P = \{p_1, \ldots, p_n\}$)
1    create a node currentnode
2    length := 0
3    set the list of patterns under loading to be $\{p_1, \ldots, p_n\}$
4    for each $p_i$ do PORT-NODE($p_i$) := currentnode
5    while the end of $t$ has not been reached do
         %STAGE 1
6        for each pattern under loading $p_i$ do
7            portnode := PORT-NODE($p_i$)
8            patternletter := READ-LETTER($p_i$)
9            newportnode := APPEND-LETTER(portnode, patternletter)
10           prefix-degree(newportnode) := prefix-degree(newportnode) + 1
11           if all letters of the active keyword of $p_i$ have been read then
12               delete $p_i$ from the list of patterns under loading
13               mark newportnode as a terminal node unless it was already the case
14                   and add $p_i$ to the list terminal(newportnode)
15           else PORT-NODE($p_i$) := newportnode
         %STAGE 2
16       currentletter := READ-LETTER($t$)
17       < currentnode, length >:=
                 UPDATE-CURRENT-NODE(currentnode, length, currentletter)
         %STAGE 3
18       if currentnode is terminal and depth(currentnode) = length then
19           closestterminal := currentnode
20       else closestterminal := CLOSEST-TERMINAL(currentnode)
21       while closestterminal $\neq$ undefined do
22           unmark closestterminal as a terminal node
23           currentterminal := closestterminal
24           closestterminal := CLOSEST-TERMINAL(closestterminal)
25           for each $p_i$ from the list terminal(currentterminal) do
26               if all keywords of $p_i$ have been read then
27                   output "$p_i$ occurs in $t$" and stop
28               else UNLOAD-KEYWORD(currentterminal)
29                   add $p_i$ to the list of patterns under loading
30                   PORT-NODE($p_i$) := source
31   output "$t$ does not have occurrences of $P$"
```

4.3 Correctness of the algorithm

To prove the correctness and completeness of MATCH we verify by induction that conditions 1-3 of Section 4.2 are invariant under the main **while**-loop. More precisely, we prove that the set $\{\bar{v}^1_{j_1}, \ldots, \bar{v}^n_{j_n}\}$, *currentnode*, *length*, and the list of patterns under loading satisfy conditions 1-3 at every step of the algorithm. The correctness and completeness would then follow from decomposition (1). Below we give an outline of the proof.

We first consider conditions 1 and 3. Consider decomposition (1) for some $i \in [1, n]$ and assume that $u^i_{j_i} = \bar{v}^i_{j_i} = \varepsilon$. (In other words, consider the first step when $v^i_{j_i}$ is active.) Since one letter is read from the text at every iteration (line 16), $u^i_{j_i}$ in decomposition (1) is extended by one letter (unless an occurrence of $v^i_{j_i}$ is found, see below). While $v^i_{j_i}$ is under loading, $\bar{v}^i_{j_i}$ is extended by one letter at every iteration too (line 8). Therefore, the algorithm keeps $\bar{v}^i_{j_i}$ to be a proper prefix of $v^i_{j_i}$ of length $|u^i_{j_i}|$. When all letters of $v^i_{j_i}$ are loaded, then $\bar{v}^i_{j_i} = v^i_{j_i}$, and since $|u^i_{j_i}| = |\bar{v}^i_{j_i}|$, then $v^i_{j_i}$ should cease to be under loading (condition 1). This corresponds to instruction 12 in the algorithm that deletes p_i from the list of patterns under loading as soon as $v^i_{j_i}$ has been read completely. At subsequent iterations, $\bar{v}^i_{j_i}$ does not change, unless $v^i_{j_i}$ is unloaded, which happens iff $[v^i_{j_i}]$ is on the suffix chain of *currentnode*. Since by condition 2 the member of *currentnode* (regarded as an equivalence class) of length *length* is a suffix of the text read so far, then so is $v^i_{j_i}$. With respect to decomposition (1), this means that j_i is incremented by 1 with $u^i_{j_i+1} = \varepsilon$. This completes the induction. The above arguments together with the uniqueness of the DAWG (Section 2.1) shows that at every moment the values of $\bar{v}^i_{j_i}$'s and the list of patterns under loading are correct.

We now prove that *currentnode* and *length* verify condition 2 at every step of the algorithm. Assume that condition 2 is verified at the beginning of the iteration of the **while**-loop. Let $V = \{\bar{v}^1_{j_1}, \ldots, \bar{v}^n_{j_n}\}$ be the current underlying set of words and w be the longest word from $suff(t[1:l]) \cap pref(V)$, where $t[1:l]$ is the prefix of the text read so far. By condition 2, *currentnode* is $[w]_V$ and *length* $= |w|$.

The first stage extends the DAWG to a set $\tilde{V} = \{\tilde{v}_1, \ldots, \tilde{v}_n\}$, where each $\tilde{v}_i$ is either $\bar{v}^i_{j_i}$ or $\bar{v}^i_{j_i}a$ for some $a \in A$. During this stage, *currentnode* is kept to be the equivalence class of w w.r.t. the changing set V. The only point when *currentnode* may need to be updated is when this node is split into two by the SPLIT function. In this case we should decide which of the two resulting classes contains w and then becomes a new value of *currentnode*. This is decided according to the value of *length*. The update of *currentnode* is done by instruction 4 added to SPLIT in the previous section. The correctness follows from a more detailed analysis of SPLIT that can be found in [6].

At the second stage, the next letter is read from the text, which means that index l in condition 2 is incremented, and then *currentnode* and *length* are updated by UPDATE-CURRENT-NODE. We have to show that after that, *currentnode* and *length* verify condition 2, which means that *currentnode* is $[\tilde{w}]_{\tilde{V}}$, where $\tilde{w}$ is the longest word from $suff(t[1:l+1]) \cap pref(\tilde{V})$, and *length* is $|\tilde{w}|$. The proof of this part is similar to that of proposition 2.

At the third stage the keywords are detected which occur at the current end position in the text. Clearly, all these keywords are suffixes of w. w itself is a keyword iff *currentnode* is a terminal node and w is its representative, that is depth(*currentnode*) $=$ *length*. The keywords which are proper suffixes of w have their terminal nodes on the suffix chain of w. Thus, all matching keywords are detected by the algorithm. Each matching keyword, when detected, is unloaded from $\tilde{V}$. We have to show again that condition 2 is preserved under this transformation. Consider an elementary deletion step (DELETE-LETTER) which consists in transforming the DAWG from some set $\hat{V} = \{\hat{v}_1, \ldots, \hat{v}_{n-1}, \hat{v}_n a\}$ to $\hat{V}' = \{\hat{v}_1, \ldots, \hat{v}_{n-1}, \hat{v}_n\}$. If $w \in suff(\hat{v}_n a)$ and $w \notin sub(\hat{V}')$, then w should be reset to its longest suffix which belongs to $sub(\hat{V}')$. The corresponding modification of *currentnode* and *length* is done by the instructions added to DELETE-LETTER in the previous section. The instruction added to MERGE updates *currentnode* whenever this class is merged with another one. This modification is inverse to the one done by the SPLIT function.

We summarize the discussion above in the following theorem.

Theorem 1 *The algorithm* MATCH(t, P) *is correct and complete, i.e. it detects an occurrence of patterns of P in t iff there is one.*

It is important to note that the correctness of the algorithm is essentially due to the fact that the process of keyword loading is synchronized with the text scan. If a whole keyword had been loaded immediately after the previous one has been found, a correct maintenance of *currentnode* would become impossible.

4.4 Complexity of the algorithm

In this section we evaluate the time complexity of MATCH(t, P). Define $|P| = \sum_{i=1}^{n} \sum_{j=1}^{n_i} |v_j^i|$ and $d = \sum_{i=1}^{n} \max\{|v_j^i| \mid j \in [1 : n_i]\}$. Distinguishing d and $|P|$ in the complexity analysis is useful for applications in which patterns of P are long sequences of short keywords.

Let us first focus on the time taken by stage 2, that is on the total time of executing UPDATE-CURRENT-NODE (instruction 17). One can show that this time is $O(|t|)$ using a standard argument of amortizing the number of iterations of the **while**-loop in UPDATE-CURRENT-NODE over all letters of t (cf [1]).

Let us analyse stage 3 now. CLOSEST-TERMINAL(α) runs in time $O(\log d)$, and within one execution of stage 3 there is one more call to CLOSEST-TERMINAL than terminal nodes on the suffix chain of *currentnode*. Therefore, each call to CLOSEST-TERMINAL but one is followed by unloading at least one keyword. Each iteration of the **for**-loop (line 25) either unloads one keyword or stops the whole run of the algorithm. Clearly, every keyword of P can be loaded and unloaded at most once during the run of MATCH. Unloading a keyword v_j^i using UNLOAD-KEYWORD takes $O(|v_j^i| \log d)$ time by lemma 6. Since the list of patterns under loading is implemented as a double linked list, instruction 29 as well as instruction 12 of stage 1 is done in time $O(1)$. To sum up, the time spent on stage 3 during the whole run of MATCH can be evaluated as $O(|t| \log d + \sum_{i=1}^{n} \sum_{j=1}^{n_j} |v_j^i| \log d) = O((|t| + |P|) \log d)$.

Now let us turn to stage 1. Each iteration of the **for**-loop (line 6) calls to APPEND-LETTER (line 9) which is the only individual step taking non-constant time. Thus, it remains to evaluate the complexity of the loading process. Here, however, we face a difficulty. To describe it, we forget for a moment about the auxiliary dynamic tree structure defined in Section 4.1 which introduces a $\log d$ factor in appending a letter (lemma 6). The problem is that although by lemma 3, loading a keyword takes time linear on its length, this result does not generally hold for our mode of loading. The reason is that in our case, loading letters of a keyword alternates with loading letters of other keywords and unloading some keywords, while the proof of lemma 3 in [6] assumes tacitly that the DAWG does not change between loadings of two consecutive letters of a keyword. In the rest of this section we outline a solution to this problem which takes linear time *with respect to the set of all loaded keywords.* The complete description of the solution is far too long to be given here, and is left to the full version of the paper.

Recall from section 3.1 that calling APPEND-LETTER(α, a) provokes a traversal of the suffix chain of α up to the first node with an outcoming primary a-edge. During stage 1, such a traversal is made for the port node α and the current letter a of every keyword under loading. The solution consists in synchronizing the traversals and imposing an order of traversing parts of the suffix chains of the port nodes.

Assume that the same letter a is appended to two port nodes α_1 and α_2, and assume that the suffix chains of α_1 and α_2 have a common part. A careful analysis of possible situations shows that after both loadings the branching node will have an outcoming primary a-edge and all other nodes of the common part will have outcoming secondary a-edges going to the same node as the primary one does. This suggests the principle that *common parts of the suffix chains of nodes extended by the same letter can be treated once.*

It is not too difficult to see how this principle can be implemented. The simplest way is to perform the traversal in two passes. In the first pass the suffix chain of every port node is traversed and the visited nodes are marked with the letter to be appended to the port node. The traversal stops if the node is already marked with the same letter. This node is additionally marked as a *branching node.* In the second pass the loading process is performed using the marking so that the common parts (delimited by branching nodes) are traversed once. A more complicated task is to prove that the above principle preserves linearity. The idea of the proof is to amortize all suffix chain traversals over the *total length of all keywords under loading.* In this way we show that the total time taken by loading keywords during the matching algorithm is $O(|P|)$.

If the auxiliary dynamic tree structure of suffix pointers has to be maintained (Section 4.1), appending each letter requires additional $O(\log d)$ time, and the whole loading time is then $O(|P| \log d)$.

Summarizing the complexity of all stages, we state the following result :

Theorem 2 MATCH(t, P) *runs in time* $O((|t| + |P|) \log d)$.

5 Concluding Remarks

In this paper we have designed an efficient algorithm for matching a text against a set of patterns with variable length don't cares. Note that this problem can be considered as a generalization of the dynamic dictionary matching (DDM) problem [2, 3, 11, 4] in that the dictionary (underlying set of words) changes *during* the text search. In particular, the technique of using the DAWG as a matching automaton together with the algorithms of modifying the DAWG used in this paper, constitute yet another solution of the DDM problem, that matches the same complexity bounds as in [2, 3, 11]. Also, our method meets a similar obstacle as the DDM algorithms (see section 4.1), which gives rise to the $\log d$ factor in the complexity bound (theorem 2). The obstacle amounts to the detection of keywords that are prefixes (in case of [3, 4]) or suffixes (in our case) of a given subword of another keyword. In [4] a new solution to this problem was proposed, based on the reduction to the *parenthesis maintenance problem*, which improved the complexity bounds by the $\log \log d$ factor. This solution can be plugged into our algorithm, allowing a similar improvement.

Note that our algorithm detects the leftmost occurrence of the patterns in the text. This is an obvious drawback for those applications that require to find all occurrences. However, it can be easily seen that even for a single pattern the number of occurrences may be exponential, which makes impossible an efficient algorithm that outputs all of them. Note however that the number of occurrences may be computed in polynomial time using the dynamic programming technique.

References

[1] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. Elsevier Science Publishers B. V. (North-Holland), 1990.

[2] Amihood Amir and Martin Farach. Adaptive dictionary matching. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, San Juan (Puerto Rico)*, pages 760–766. IEEE computer society press, October 1991.

[3] Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Park Kunsoo. Dynamic dictionary matching. To appear in Journal of Computer and System Sciences, June 1993.

[4] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, Austin (TX)*, pages 392–401, January 1993. to appear in Information and Computation.

[5] Alan A. Bertossi and Filippo Logi. Parallel string matching with variable length don't cares. *Journal of parallel and distributed computing*, 22:229–234, 1994.

[6] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[7] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

[8] Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[9] Maxime Crochemore. String matching with constraints. In *Proceedings International Symposium on Mathematical Foundations of Computer Science*, volume 324 of *Lecture Notes in Computer Science*, pages 44–58. Springer-Verlag, 1988.

[10] Michael J. Fisher and Michael S. Paterson. String-matching and other products. In R. M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125. American Mathematical Society, Providence, RI, 1974.

[11] Ramana M. Idury and Alejandro A. Schäffer. Dynamic dictionary matching with failure functions. *Theoretical Computer Science*, 131:295–310, 1994.

[12] G. Kucherov and M. Rusinowitch. Complexity of testing ground reducibility for linear word rewriting systems with variables. In *Proceedings 4th International Workshop on Conditional and Typed Term Rewriting Systems, Jerusalem (Israel)*, July 1994. to appear in the LNCS series.

[13] Udi Manber and Ricardo Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, 1991.

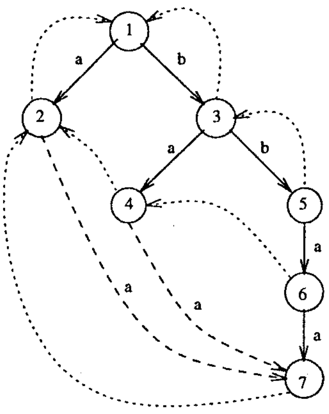
[14] Gene Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, April 1992.

[15] Ron Y. Pinter. Efficient string matching with don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Pattern Matching*, volume F12 of *ASI Series*, pages 11–29. Springer-Verlag, 1985.

[16] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26:362–391, 1983.

[17] Esko Ukkonen. On-line construction of suffix-trees. Report A-1993-1, University of Helsinki, Department of Computer Science, February 1993.

A0 Example of a DAWG



This is the DAWG for the set $\{ba, bbaa\}$. Nodes 1-7 correspond respectively to the equivalence classes $\{\varepsilon\}$, $\{a\}$, $\{b\}$, $\{ba\}$, $\{bb\}$, $\{bba\}$, $\{aa, baa, bbaa\}$. Primary edges are drawn with normal arrows, secondary edges with dashed arrows, and suffix pointers with dotted arrows. Depth of nodes 1-7 is 0, 1, 1, 2, 2, 3, 4 respectively. Nodes 3, 4, 5, 6, 7 are prefix nodes with prefix-degree(3) $=$ 2 and prefix-degree(4) $=$ prefix-degree(5) $=$ prefix-degree(6) $=$ prefix-degree(7) $=$ 1. Nodes 4 and 7 are terminal nodes.
number-of-children(1) $=$ number-of-children(2) $=$ more-than-one,
number-of-children(3) $=$ number-of-children(4) $=$ 1,
number-of-children(5) $=$ number-of-children(6) $=$ number-of-children(7) $=$ 0

A1 Extending the DAWG

```
APPEND-LETTER($activenode, a$)
1    if out($activenode, a$) $\neq$ undefined then
2        if type($activenode, a$) = primary then
3            return out($activenode, a$)
4        else return SPLIT($activenode$, out($activenode, a$))
5    else create a new node $newactivenode$ and set number-of-children($newactivenode$) := 0
6        create a new primary $a$-edge ($activenode, newactivenode$) and set
          origin($newactivenode$) := $activenode$, last-letter($newactivenode$) := $a$,
          depth($newactivenode$) := depth($activenode$) + 1
7        $varnode$ := suf-pointer($activenode$)
8        while $varnode \neq$ undefined and out($varnode, a$) = undefined do
9            create a new secondary $a$-edge ($varnode, newactivenode$)
10           $varnode$ := suf-pointer($varnode$)
11       if $varnode$ = undefined then
12           create a suffix pointer from $newactivenode$ to $source$
13       elseif type($varnode, a$) = primary then
14           create a suffix pointer from $newactivenode$ to out($varnode, a$)
15       else create a suffix pointer from $newactivenode$
                                    to SPLIT($varnode$, out($varnode, a$))
16       return $newactivenode$
```

```
SPLIT($originnode, targetnode$)
1    create a new node $newtargetnode$
2    $appendedletter$ := last-letter($targetnode$)
3    replace the secondary edge ($originnode, targetnode$) by a primary edge
      ($originnode, newtargetnode$) with the same label and set
      origin($newtargetnode$) := $originnode$, last-letter($newtargetnode$) := $appendedletter$,
      depth($newtargetnode$) := depth($originnode$) + 1
4    instructions are added at this line in section 4.2
5    for every outcoming edge of $targetnode$, create a secondary outcoming
          edge of $newtargetnode$ with the same label and going to the same node
6    create a suffix pointer of $newtargetnode$ pointing to suf-pointer($targetnode$)
7    redirect the suffix pointer of $targetnode$ to point to $newtargetnode$
8    $varnode$ := suf-pointer($originnode$)
9    while $varnode \neq$ undefined and type($varnode, appendedletter$) = secondary do
10       redirect the secondary edge of $varnode$ (labeled by $appendedletter$)
                                              to point to $newtargetnode$
11       $varnode$ := suf-pointer($varnode$)
12   return $newtargetnode$
```

```
LOAD-KEYWORD($v = v_1 \dots v_n$)
1    $activenode$ := $source$
2    for $i$ := 1 to $n$ do
3        $activenode$ := APPEND-LETTER($activenode, v_i$)
4        prefix-degree($activenode$) := prefix-degree($activenode$) + 1
```

A2 Reducing the DAWG

```
DELETE-LETTER(activenode)
1   newactivenode := origin(activenode)
2   deletedletter := last-letter(activenode)
3   if prefix-degree(activenode)>0 or number-of-children(activenode) = more-than-one then
4       return newactivenode
5   elseif number-of-children(activenode) = 1 then
6       MERGE(activenode, newactivenode, deletedletter)
7       return newactivenode
8   else delete the primary edge (newactivenode, activenode) labeled by deletedletter
9       instructions are added at this line in section 4.2
10      varnode := suf-pointer(newactivenode)
11      while varnode $\neq$ undefined and type(varnode, deletedletter) = secondary do
12          delete the secondary edge (varnode, activenode) labeled by deletedletter
13          varnode := suf-pointer(varnode)
14      delete activenode
15      if varnode $\neq$ undefined then
16          suffixnode := out(varnode, deletedletter)
17          if prefix-degree(suffixnode) = 0 and number-of-children(suffixnode) = 1 then
18              MERGE(suffixnode, varnode, deletedletter)
19      return newactivenode
```

```
MERGE(targetnode, originnode, deletedletter)
1   newtargetnode := child(targetnode)
2   instructions are added at this line in section 4.2
3   replace the primary edge (originnode, targetnode) labeled by deletedletter
            by a secondary edge (originnode, newtargetnode) with the same label
4   varnode := suf-pointer(originnode)
5   while varnode $\neq$ undefined and type(varnode, deletedletter) = secondary do
6       redirect the secondary edge (varnode, targetnode) labeled by deletedletter
                                                            to newtargetnode
7       varnode := suf-pointer(varnode)
8   redirect the suffix pointer of newtargetnode to point to suf-pointer(targetnode)
9   delete the suffix pointer of targetnode
10  delete targetnode
```

```
UNLOAD-KEYWORD(terminalnode)
1   activenode := terminalnode
2   while activenode $\neq$ source do
3       prefix-degree(activenode) := prefix-degree(activenode) - 1
4       activenode := DELETE-LETTER(activenode)
```