Efficient Multi-Profile Filtering using Finite Automata

Lukas C. Faulstich*
<http://www.inf.fu-berlin.de/~faulstic>

Technical Report B 01-03

April 17, 2001

Abstract

The task of an alerting system is to observe events, produce a stream of event notifications, and deliver each notification to all clients whose profiles match this notification. The details of the delivery are specified by a notification policy that is part of a profile.

This report deals with the problem of matching an incoming document against a set of profiles. Focusing on this aspect, we can model profiles as unary predicates on the set of documents. If profiles are arbitrary unary predicates on documents, each profile must be checked separately. However, in many application domains such as digital libraries, much simpler predicates are common.

In this paper we focus on *substring matching*. We extend the well-known technique (cf. [1]) of constructing a deterministic finite state machine from a *single* search string and feeding the document as input to it. We show that any finite number of simple substring profiles can be checked *simultaneously* by a single deterministic finite state machine in linear time. We give a construction for this automaton and show that it has very modest storage requirements. We present methods for the incremental maintenance of this automaton in case of insertion or deletion of profiles.

Extensions of simple substring profiles to exact word matching, phrase matching, and profiles against different parts (fields) of a document are straight-forward. An extension to Boolean profiles based on post-processing by an additional layer of finite state machines is discussed. However, the advantage of linear running time is lost; the performance of this approach is similar to the methods presented in [8].

*Institut für Informatik, Freie Universität Berlin, Takustr. 9, D-14195 Berlin

1 Preliminaries

Let A be a finite alphabet and $w = a_1 \dots a_m \in A^*$ be a word over A. A substring $a_i \dots a_k$ denotes the empty word λ iff $k < i$. The set of *prefixes* of w is defined by *prefixes*$(w) = \{a_1 \dots a_j \mid j = 0 \dots m\}$. The set of *suffixes* of w is defined by *suffixes*$(w) = \{a_j \dots a_m \mid j = 1 \dots m+1\}$. Both sets include both w and λ. The set of *proper prefixes* of a word w is denoted by *pprefixes*(w) = *prefixes*$(w) - \{w\}$. The set *psuffixes*(w) is defined analogously.

These definitions can be extended to languages (sets of words): *prefixes*$(L) = \bigcup \{$*prefixes*$(w) \mid w \in L\}$ and *suffixes*$(L) = \bigcup \{$*suffixes*$(w) \mid w \in L\}$. In the same way we define the set of proper prefixes/suffixes of a language.

2 Construction

Let A be an alphabet. Let $W = \{w_1, \dots, w_n\} \subseteq A^*$ be a set of words over A. We call the elements of W *keywords*. We denote the set of indices of W by $N = \{1, \dots, n\}$ and use the function *index* $: W \longrightarrow N$ to map each keyword w_i to its index *index*$(w_i) = i$. We denote by $\bar{l} = \frac{1}{n}\Sigma_{i=1}^{n}|w_i|$ the average length of a keyword.

We construct a deterministic generalized sequential machine (GSM) [7] M_W that transforms a word $w \in A^*$ into a sequence $M(w) \in N^*$ comprised of the indices i of all words w_i contained in w, in the order of their occurrence.

A GSM is a finite automaton that outputs a word over an output alphabet at each transition. The output of the GSM for an input word is the concatenation of all these words, if the last transition leads to an accepting state.

One possible approach would be to construct a nondeterministic finite automaton (NFA) that accepts the language A^*WA^* and then turn it into a deterministic GSM. The construction of this NFA is shown in the appendix in Section 7. This construction is very intuitive. The transformation into a deterministic finite automaton (DFA) is a standard procedure, but turns out to be rather lengthy. Moreover, turning the DFA into a GSM would require a proof that the resulting GSM produces the correct output.

Hence we resort to a less intuitive but more direct approach: we construct a deterministic GSM from scratch and show that it solves our problem. To understand what the problems are we consider two examples.

Example 2.1
Let $W = \{w_1, w_2\} \subseteq \{a, b, c, d, e, f\}^*$ where $w_1 = \texttt{abc}$, $w_2 = \texttt{def}$. The GSM printing the indices of those w_i occurring in the input is shown in Fig. 1.

Note, that all states of this automaton are accepting states. We use the following notation: an edge from a state q to a state q' is labeled (i) with symbols $a_1, \dots, a_m \in A$ if there are productions $(qa_j \longrightarrow \lambda q')$ for $j = 1, \dots, m$, or (ii) with $a|o$ if there is a production $(qa \longrightarrow oq')$ with $o \neq \lambda$. The initial state is indicated by an incoming arrow without source node.

We recognize two chains of states, one for matching `abc` $(q0, q1, q2)$ and one for matching `def` $(q0, q3, q4)$. What complicates this automaton is the need to treat the input symbols `a` and `b` consistently as starting symbols of w_1 and w_2, respectively, and to switch to the appropriate states $q1$ and $q3$. $\square$

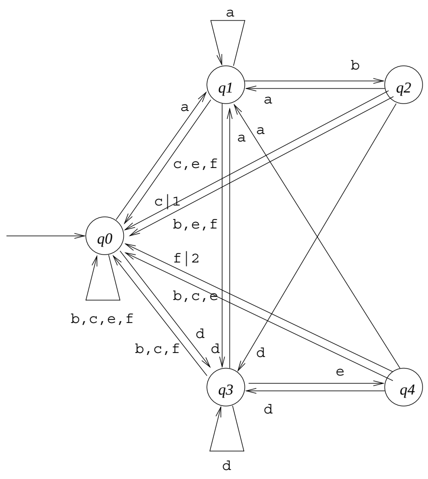

Figure 1: A GSM for $W = \{\texttt{abc}, \texttt{def}\}$.

Example 2.2
We now change W slightly by replacing `def` by `bcd`: Let $W = \{w_1, w_2\} \subseteq \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}^*$ where $w_1 = \texttt{abc}$ and $w_2 = \texttt{bcd}$. Note, that the prefix `bc` of w_2 occurs within w_1. This fact must be honored in $q2$ by switching upon input `c` to the same state $q4$ that is reached by input `bc` rather than to the initial state $q0$. This is depicted in Fig. 2

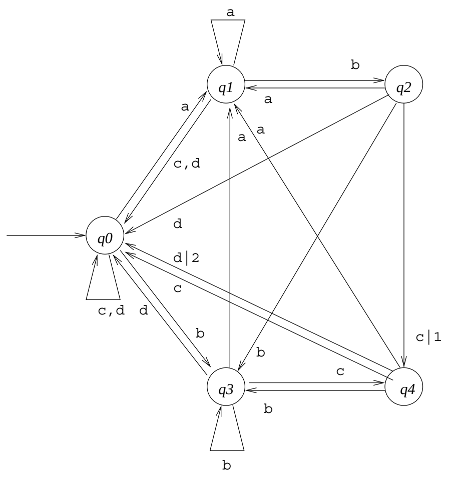

Figure 2: A GSM for $W = \{\texttt{abc}, \texttt{bcd}\}$.

$\square$

In general, we must keep book for every state to which words from W the input read so far can be completed. This means to recognize which prefixes of words from W have appeared as suffix of the input read so far. Hence we introduce the set $Q = \textit{pprefixes}(W)$ of all proper prefixes of words from W. We call the elements of Q W-prefixes. Note, that Q is finite since W is finite.

The function $\mathit{starts} : A^* \longrightarrow \mathcal{P}(Q)$ defined by $\mathit{starts}(w) = \mathit{suffixes}(w) \cap Q$ computes the set of W-prefixes occurring as suffixes of a word w. We define by $\mathit{maxstart}(w) = \max(\mathit{starts}(w))$ the longest suffix of w that is a W-prefix. Note, that the set $\mathit{starts}(w)$ is totally ordered by length since it is a finite subset of the totally ordered set $\mathit{suffixes}(w)$.

Lemma 2.1 Let $w \in A^*$. Then $\mathit{starts}(w) \subseteq \mathit{suffixes}(\mathit{maxstart}(w))$.

Proof: $v = \mathit{maxstart}(w)$ is by definition the maximal element of $\mathit{starts}(w)$. Furthermore $\mathit{starts}(w) = \mathit{suffixes}(w) \cap Q \subseteq \mathit{suffixes}(w)$. Let $u \in \mathit{starts}(w)$. Since both u and v are suffixes of w, but $|u| \leq |v|$, u must be a suffix of v. Hence $\mathit{starts}(w) \subseteq \mathit{suffixes}(\mathit{maxstart}(w))$. □

We denote by $\mathit{tailocc} : A^* \longrightarrow N^*$ the indices of all words in W that occur as suffixes of a word, ordered by size. It is defined by $\mathit{tailocc}(w) = i_1 \ldots i_k$ where $\{w_{i_1}, \ldots, w_{i_k}\} = \mathit{suffixes}(w) \cap W$, $|w_{i_j}| \leq |w_{i_{j+1}}|$ for $j = 1, \ldots, k-1$.

Construction 2.1

Now we construct the deterministic GSM $M_W = (A, N, Q, q_0, P)$ with input alphabet A, output alphabet N, state set Q, initial state $q_0 = \lambda$, and production set $P = P_{trans} \cup P_{acc}$ where

$$\begin{aligned} P_{trans} &= \{(\langle q\rangle a \to o\langle w\rangle) \mid q \in Q, a \in A, w = \mathit{maxstart}(qa), o = \mathit{tailocc}(qa)\} \\ P_{acc} &= \{(\langle q\rangle \to \lambda) \mid q \in Q\} \end{aligned}$$

(We use the syntax $\langle w\rangle$ to indicate that w is treated as a single symbol from Q rather than a word over A.) □

Example 2.3

To illustrate the construction described above, we return to Example 2.2 and build the automaton M_W for $W = \{\texttt{abc}, \texttt{bcd}\}$. This set of keywords yields the state set $Q = \{\langle\lambda\rangle, \langle a\rangle, \langle ab\rangle, \langle b\rangle, \langle bc\rangle\}$ consisting of all proper prefixes of `abc` and `bcd`, respectively.

We exemplify the construction of the productions for the state $q = \langle ab\rangle$: For input symbol $a = \texttt{a}$ we consider $qa = \texttt{aba}$. Clearly no keyword from W appears in qa, hence the output $o = \mathit{tailocc}(qa)$ is the empty word. The longest state being a suffix of qa is $\langle a\rangle$, hence there is a production $\langle ab\rangle\texttt{a} \to \lambda\langle a\rangle$ in P_{trans}. Similarly, the input symbol $a = \texttt{b}$ yields as output the empty word and the successor state is $\langle b\rangle$. For input symbol $a = \texttt{d}$ there is no output as well and the successor state $q' = \mathit{maxstart}(\texttt{abd})$ is the state $\langle\lambda\rangle$ since no keyword starts with one of the suffixes `abd`, `bd`, or `d` of qa.

Input symbol $a = \texttt{c}$ yields $qa = \texttt{abc} = w_1 \in W$. Hence the output $o = 1$ is produced. Moreover, qa ends in the prefix `bc` of w_2 which requires the successor state $q' = \langle bc\rangle$.

The automaton M_W constructed this way is shown in Fig. 3. It turns out that this automaton is in fact isomorphic the automaton shown in shown in Fig. 2.

□

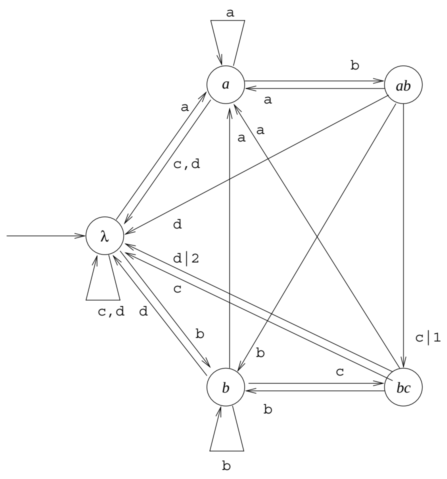

Figure 3: The GSM M_W for $W = \{\texttt{abc}, \texttt{bcd}\}$.

3 Correctness and Completeness

A GSM produces an output o for an input w iff there is a derivation $q_0 w \Rightarrow^* o$. The GSM M_W is deterministic as one can see easily from the definition of P_{trans} since for every $v \in Q$ and $a \in A$ there is exactly one production $(\langle v \rangle a \to o\langle w \rangle) \in P_{trans}$. This also shows that M_W can read arbitrary words from A^*. Since all states are accepting states (i.e., $\forall w \in Q \; \exists (\langle w \rangle \to \lambda) \in P_{acc}$), the language accepted by M_W is A^*.

Lemma 3.1 After reading an input sequence $w \in A^*$, the automaton M_W is in state $q = \mathit{maxstart}(w)$.

Proof: Lemma 3.1 obviously holds for $w = \lambda$ since the reached state is $q_0 = \lambda$ and $\mathit{maxstart}(\lambda) = \lambda$.

Let us assume that Lemma 3.1 holds for a certain $w \in A^*$. Let $a \in A$. Then by assumption M_W reaches the state $q = \mathit{maxstart}(w)$ by reading w. There exists a production $(\langle q \rangle a \to o\langle q' \rangle) \in P$ where $q' = \mathit{maxstart}(qa)$. Using this production, M_W reaches the state q' by reading wa.

It remains to show that $\mathit{maxstart}(qa) = \mathit{maxstart}(wa)$. Since $\mathit{maxstart}(wa)$ is a suffix of wa, it must have the form va where v is a suffix of w (*). By definition of *maxstart* the word $va = \mathit{maxstart}(wa)$ is in Q. By the definition of Q it follows that the prefix of a W-prefix is a W-prefix as well. Hence from $va \in Q$ it follows that also $v \in Q$ (**). From (*) and (**) it follows that v is in $\mathit{suffixes}(w) \cap Q = \mathit{starts}(w)$. By Lemma 2.1 it follows that $v \in \mathit{suffixes}(\mathit{maxstart}(w)) = \mathit{suffixes}(q)$. Hence $va \in \mathit{suffixes}(qa)$. As already noted, $va \in Q$. Together this implies $va \in \mathit{suffixes}(qa) \cap Q = \mathit{starts}(qa)$. Again we use Lemma 2.1 to conclude that

$$\mathit{maxstart}(wa) = va \in \mathit{suffixes}(\mathit{maxstart}(qa)) \tag{1}$$

.

On the other hand we have $q = \mathit{maxstart}(w) \in \mathit{suffixes}(w)$ and hence $qa \in \mathit{suffixes}(wa)$. Since $\mathit{maxstart}(qa) \in \mathit{suffixes}(qa)$ we get by transitivity of the suffix relation

$$\mathit{maxstart}(qa) \in \mathit{suffixes}(\mathit{maxstart}(wa)) \tag{2}$$

Equations 1 and 2 show that $\mathit{maxstart}(qa)$ and $\mathit{maxstart}(wa)$ are suffixes of each other which implies $\mathit{maxstart}(qa) = \mathit{maxstart}(wa)$.

By structural induction, this proves the lemma for all $w \in A^*$. □

Theorem 3.1 The output o of automaton M_W for an input sequence $w \in A^*$ contains an index $i \in N$ iff $w_i \in W$ occurs in w.

Proof: Let w_i occur in w, i.e., $\exists u, u' \in A^* : w = uw_iu'$. Let $w_i = va$ for some $v \in A^*$ and $a \in A$. By Lemma 3.1 automaton M_W reaches the state $q = \mathit{maxstart}(uv)$ by reading input uv. v is a proper prefix of w_i and hence $v \in Q$. Therefore q must contain v as a suffix and accordingly qa must contain w_i as suffix. By reading the additional input symbol a the automaton uses a transition that outputs $\mathit{tailocc}(qa)$. Since w_i is a suffix of qa, $\mathit{tailocc}(qa)$ contains the index i. Hence the output of M_W for input w contains i.

On the other hand, if the output of M_W contains i, then after reading some prefix v of w, M_W must have made a transition $(\langle q\rangle a \longrightarrow o\langle q'\rangle)$ where $o = \mathit{tailocc}(qa)$ contains i. This means that w_i must be a suffix of qa. By Lemma 3.1 we know that $q = \mathit{maxstart}(v)$. In particular, q is a suffix of v and hence qa is a suffix of va. By transitivity of the suffix relation w_i is therefore a suffix of va and hence it occurs in the whole input w. □

Corollary 3.1 If two words from W occur in the input of automaton M_W, then their indices occur in the output of M_W in the same order.

Proof: Let w_i occur before w_j in input $w \in A^*$. This means there exist $u_1, v_1, u_2, v_2 \in A^*, u_2 \neq \lambda$ such that $w = u_1w_iv_1$ and $w_iv_1 = u_2w_jv_2$. By reading u_1w_i the automaton will reach state q_1 and output i after reading the last symbol of w_i, as shown in the proof of Theorem 3.1. Using the same argument we conclude that by reading the additional input u_2w_j the automaton will reach a state q_2 and output a word containing j after reading the last symbol of w_j. Hence j occurs after i in the output for w. □

4 Incremental Maintenance

Our aim is to support a large number of profiles submitted by independent users. Hence it is important to update the automaton M_W constructed in Section 2 *incrementally* when new profiles (i.e., elements of the set W) are entered or existing profiles are deleted.

4.1 Auxiliary Indices

Several indices are needed to support the efficient implementation of updates.

4.1.1 Overlap Count

When deleting an element w from W only those of its prefixes can be deleted from Q that are not at the same time prefixes of some other word from W. To avoid checking all elements of W we keep book on how many elements of W a state $q \in Q$ is a prefix of. This means we treat Q as a multiset by recording the multiplicity of each element being defined by the function *overlap* : $A^* \longrightarrow \mathbb{N}$ as follows:

$\mathit{overlap}(q) = |\{w \in W \mid q \in \mathit{pprefixes}(w)\}|$

Each counter $\mathit{overlap}(q)$ can be stored together with the corresponding state $\langle q \rangle$ of automaton M_W. To look up $\mathit{overlap}(w)$ for $w = a_1 \dots a_l$ we simulate automaton M_W with input w, but only as long as the reached state equals the input consumed so far. For this it is sufficient to check the length of the reached state. To accelerate this test we can attach to each state its length.

If the whole input sequence w is consumed this way, we have reached state $\langle q \rangle$ where $q = w$ and return the counter $\mathit{overlap}(q)$ attached to state $\langle q \rangle$. Otherwise $w \notin Q$ and hence we return $\mathit{overlap}(w) = 0$.

4.1.2 Tail Occurrences

The function *tailocc* lists all words $\mathit{tailocc}(w)$ from W that occur as suffix of a word w. To compute this function requires all elements of W to be checked. This computation can be sped up by organizing the reverse w^R of each word $w \in W$ in a trie T_W. For every $i = 1, \dots, n$ the trie stores at ${w_i}^R$ the entry i. The value of $\mathit{tailocc}(w)$ can be computed by traversing T_W along w^R as far as possible and returning all entries encountered on the way in reverse order.

In Fig. 4 the resulting trie T_W is demonstrated for a small set W of keywords. In this example, $\mathit{tailocc}(\mathit{metadata})$ could be computed by starting from the root of the trie and traversing its topmost branch. Two entries will be encountered, namely 1 (for *data*) and 4 (for *metadata*). Hence $\mathit{tailocc}(\mathit{metadata}) = 41$.

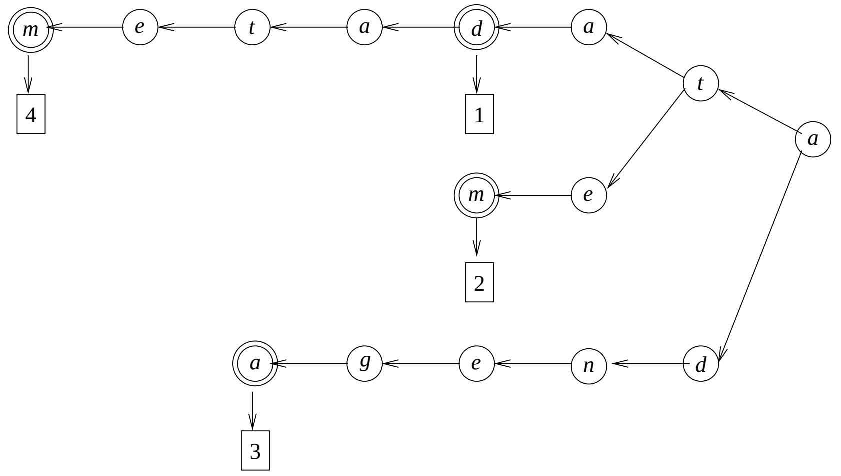

Figure 4: Trie T_W for $W = \{\mathit{data}, \mathit{meta}, \mathit{agenda}, \mathit{metadata}\}$.

The lookup in T_W can be executed in time linear in the length of w. On the other hand the number of steps is limited by the depth of the trie which equals the maximum keyword length $l_{\max}$. The average depth of a leaf of T_W is somewhere between the average keyword length $\bar{l}$ and $l_{\max}$.

Insert or delete require linear time in the length of the word to be inserted or deleted. In the case of a deletion this takes in average $\bar{l}$ steps. If we assume that the distribution of new keywords equals the distribution of existing keywords in W, this holds for insertions as well.

The trie contains a number of nodes that equals in the worst case the total number of symbols stored in the trie. This number can be expressed as $n \cdot \bar{l}$ where $n = |W|$.

Since every node stores at most $|A|$ references to subnodes the maximum storage requirements of the trie are proportional to $|A| \cdot n \cdot \bar{l}$. Both $|A|$ and $\bar{l}$ can be assumed to be fixed, and hence the storage requirements are linear in n.

4.1.3 Starts

To compute the function $\mathit{starts}(w)$ all W-prefixes from Q must be checked. This can be avoided by organizing the set Q^R of all reversed W-prefixes in a trie T_Q that for every $q \in Q$ stores at q^R the entry q. To compute $\mathit{starts}(w)$ the trie is traversed along w^R and all entries $q \in Q$ encountered on the way are collected as result. The last entry encountered points to the maximum W-prefix q that is a suffix of w, i.e., $q = \mathit{maxstart}(w)$.

As for the trie T_W above, the computation of $\mathit{starts}(w)$ or $\mathit{maxstart}(w)$ requires a number of steps linear in $|w|$, but no more than the depth of the trie along the path indicated by w. This depth is limited by $l_{\max}$.

Insert and delete of a keyword w with $|w| = l$ require insertion or deletion of at most l proper prefixes of w, each of which takes a number of steps linear in its length. This results in a total of $O(\bar{l}^2)$ steps.

The storage requirements of T_Q are $O(|A| \cdot \bar{l}^2 \cdot n) = O(n)$.

4.2 Insert

We discuss the insertion of a new keyword $w_{n+1} = a_1 \dots a_l \in A^*$. We denote the updated set of keywords by $W' = W \uplus \{w_{n+1}\}$. The trie T_W is updated by inserting ${w_{n+1}}^R$ which results in the modified trie $T_{W'}$.

We abbreviate the prefixes of w_{n+1} by $q_j := a_1 \dots a_j$ for $j = 0, \dots l$.

Let $k = \max\{q_j \in Q \mid j = 0, \dots, l-1\}$ be the length of the maximum prefix of w_{n+1} that is already a W-prefix. This number k can be computed by simulating automaton M_W with input w_{n+1} as long as the reached state equals the consumed input (cf. Section 4.1.1). The number of symbols consumed this way equals k.

We now construct the updated automaton M'. Its set Q' of states is the old set of states plus the proper prefixes of w_{n+1}. More formally, $Q' = Q \uplus \{q_j \mid k < j < l\}$. The overlap count $\mathit{overlap}(q')$ for all proper prefixes q_j, $j = 0, \dots, l-1$ is incremented. For all new states q_j, $j = k+1, \dots, l-1$, the reverse ${q_j}^R$ is inserted into trie T_Q, yielding the modified trie $T_{Q'}$.

We construct the production set of M' in three steps. First we provide default productions for the new states that are consistent with the productions of the existing automaton M_W. More formally, for q_j, $j = k+1, \dots, l-1$ and input symbol a we copy the corresponding production of $q = \mathit{maxstart}(q_j)$.

Provided that state q_j is reached by consuming an input string ending in the suffix q_j, the resulting automaton reacts just like the old one.

In the second step we correct those productions $\langle q\rangle a \longrightarrow o\langle q'\rangle$ that now have to lead to one of the new states. This is the case if a state q ends in a prefix q_j of w_{n+1} for $j = k, \dots, l-2$, the input symbol a equals a_{j+1}, and $|q'| < j+1$. Then the new state is q_{j+1} instead of q'. To find those states that end in q_j, we look up the node corresponding to ${q_j}^R$ in $T_{Q'}$. All entries occurring in the subtree rooted in this node correspond to states that have q_j as suffix.

In the final step we add the index $n+1$ to the output of those productions $\langle q\rangle a \longrightarrow o\langle q'\rangle$ where qa contains w_{n+1} as a suffix. This means that q ends in q_{l-1} and $a = a_l$. $T_{Q'}$ can be used to find all these states q efficiently.

4.3 Delete

Deleting a keyword $w = a_1 \dots a_l$ from W can be achieved in two ways: either by just removing all occurrences of *indexw* from output strings in productions or by additionally removing all traces of w in M_W which yields the automaton M' corresponding to $W' = W - \{w\}$.

We abbreviate the prefixes of w by $q_j := a_1 \dots a_j$, for $j = 0, \dots l$.

Removing *indexw* from the output requires to look up all states q ending in q_{l-1} in T_Q and correcting the production $\langle q\rangle a_l \longrightarrow o\langle q'\rangle$ by deleting *indexw* from o.

If the automaton M_W will be left this way, w must be marked as deleted in T_W. The re-insertion of w then requires just removing this mark and then executing the third phase of the insertion procedure (cf. Section 4.2).

If every trace of w is to be deleted from M_W, we need two additional phases. In phase two we decrement $\mathit{overlap}(q_j)$ for $j = 0, \dots l-1$, yielding $\mathit{overlap}'$. Let $k = \max\{j = 0, \dots, l-1 \mid \mathit{overlap}'(q_j) > 0\}$. Then $\mathit{overlap}'(q_j) = 0$ for $j > k$. We now delete all states $q_{k+1}, \dots, q_{l-1}$ from Q and T_Q since they are not needed for other keywords from W. This yields Q' and $T_{Q'}$.

In the phase three we correct those productions leading to the deleted states $q_{k+1}, \dots, q_{l-1}$. To this purpose we look up in $T_{Q'}$ for each q_j, $j = k+1, \dots, l-1$ the states q ending in q_{j-1} and replace the production $\langle q\rangle a_j \longrightarrow o\langle q'\rangle$ by $\langle q\rangle a_j \longrightarrow o\langle \mathit{maxstart}(q_j)\rangle$. Since $\mathit{maxstart}(q_j)$ can be obtained by looking up q_j in $T_{Q'}$ and finding the entry for the maximum suffix of q_j (cf. Section 4.1.3). Hence we can start with $j = l-1$ and combine the computation of $\mathit{maxstart}(q_{j-1})$ and of the states ending in q_{j-1}.

5 Complexity

Automaton M_W executes $l = |w|$ transitions for an input w. Each transition is comprised of (i) reading an input symbol and (ii) possibly printing several output symbols. The total length of the output is the number m of occurrences of words from W in the input. Hence the runtime of M_W is $O(l+m)$. Since every input symbol must be consumed and every occurrence of a word from W has to announced, there can be no algorithm with a better runtime complexity.

The storage requirements of M_W are $|P_{trans}| = |Q| \cdot |A| \leq (\Sigma_{i=1}^n |w_i|) \cdot |A|$ if we store the automaton in form of a transition table. The size of Q is limited by $\Sigma_{i=1}^n |w_i| = n \cdot \bar{l}$ where $\bar{l}$ is the average length of a keyword.

If one would use n automata M_i for checking each profile $A^*\{w_i\}A^*$ separately, one would have runtime $O(n \cdot l + m)$ which means that the runtime would grow linearly in the number of profiles. The storage requirements for each M_i would be $|w_i| \cdot |A|$ which results in the same total storage requirements as for M_W.

6 Complex Profiles

In the last section, we considered profiles to be defined by languages of the form $A^*\{w\}A^*$. Now we introduce several extensions:

1. exact, prefix, suffix, and substring match; phrases
2. field specifications
3. Boolean operators
4. proximity operators

We introduce the set Π of all possible profiles and the current set $P \subset \Pi$ of profiles that are to be matched.

6.1 Atomic Profiles

Definition 6.1 (Atomic Profile)
Let $*$ and $\_$ be dedicated symbols not in A. The symbol $\_$ represents white space in the input.
An *atomic profile* has one of the forms

1. w
2. $w*$
3. $*w$
4. $*w*$

where $w \in A^+(\_A^+)^*$. We call the set of all atomic profiles Π_{atomic}. □

We match arbitrary atomic profiles as follows:
The alphabet A is extended to $A' = A \cup \{\_\}$. We encode a phrase of k words $w_1, w_2, \ldots, w_m \in A^*$ as a single word $w_1\_w_2\_\ldots\_w_m \in A'^*$. Atomic profiles of the forms $w, w*, *w, *w*$ are stored in W as $\_w\_, \_w, w\_, w$, respectively. We say, a profile p *subscribes* to the element of W that is derived from p. The GSM M_W is then constructed as described in Section 2.
We preprocess the input sequence by prepending and appending a $\_$ symbol and replacing every sequence of whitespace characters by a single $\_$ symbol. This preprocessing can be carried out by a very simple GSM M_{pre}.
We postprocess the output of automaton M_W by marking for each output symbol $i \in N$ all profiles $p \in P$ that subscribe to w_i. After processing the whole document, notification messages are sent for all marked profiles.

6.2 Qualified Profiles

Definition 6.2 (Qualified Profile)

Let F be a finite set of *field names.*

We assume a function *field* : $F \times A^* \longrightarrow A^*$ that selects from an input document w the content *field*(f, w) of each field f. The required format of the input documents and how the fields are identified and extracted is domain-specific.

The set of *qualified* profiles Π_{quali} contains all profiles of the form $f : p$ where $f \in F$ and $p \in \Pi_{atomic}$. □

A set $P \subseteq \Pi_{quali}$ of qualified profiles is handled by building for each field $f \in F$ the set $P_f = \{p \mid p : f \in P\}$ and the set W_f of keywords derived from P_f. Then for each nonempty W_f the corresponding automaton M_{W_f} is constructed.

An incoming document is processed by extracting every field f and feeding it to the automaton M_{W_f} for f.

6.3 Boolean Profiles

Definition 6.3 (Boolean Profile)

A Boolean profile over a set Π of so-called *simple* profiles has one of the forms

1. p where $p \in \Pi$
2. $p_1 \vee p_2$ where p_1 and p_2 are Boolean profiles over Π
3. $p_1 \wedge p_2$ where p_1 and p_2 are Boolean profiles over Π
4. $\neg p$ where p is a Boolean profile over Π

We call Π_{bool} the set of Boolean profiles over Π_{quali}. □

A Boolean profile over Π matches a document if it is fulfilled when being interpreted as a propositional formula where each simple profile $p \in \Pi$ is fulfilled if p matches the current document.

The naive approach to evaluate a set of Boolean profiles from Π_{bool} is to run the automaton M_{W_f} for each field f and marking for each output symbol the corresponding profile as being fulfilled. After processing all fields, each Boolean profile is evaluated and if it is fulfilled, a notification message is sent to its owner.

If the average selectivity of a profile is small, the naive approach means a large overhead because all profiles have to be evaluated, but only few actually match.

6.3.1 A GSM-based approach

Can the GSM-based technique presented in this work extended to Boolean profiles? A natural approach would be to build a GSM M_W for all simple profiles occurring in the Boolean profiles and then feed the output to a number of automata working in parallel, each of which is responsible for matching a single Boolean profile. This approach is based on the idea that a Boolean profile can be interpreted as a regular expression over the output alphabet N of M_W. Hence it is straight-forward to construct a finite state automaton M_p for each Boolean profile p.

However, the approach to execute all these automata M_p for each output symbol $i \in N$ produced by M_W is not efficient since the symbol i is relevant only to a (typically small) fraction of the profiles. An enhanced approach would therefore manage for every output symbol i an inverted list containing all automata M_p for which i is relevant. Still there is an overhead because all automata for which i is relevant have to be executed. The vision would be to have a single automaton that handles all Boolean profiles in parallel, just like M_W handles all simple profiles in parallel. It is straight-forward to combine several finite state machines to a single finite state machine that emulates all the component machines in parallel[1] Unfortunately the state space of the combined machine is the Cartesian product of the component state spaces. This combinatorial explosion means that the transition tables may exceed the available memory and the current state cannot be stored in a single integer variable. It might make sense to combine smaller groups of automata with similar sets of relevant input symbols into larger automata as long as the size does not exceed a certain limit. This modest approach would also facilitate incremental maintenance. To identify groups of similar automata one could employ incremental clustering algorithms.

6.3.2 Index structures for selective dissemination of information

Yan and Garcia-Molina describe in [8] several index methods for efficiently matching a large set of conjunctive profiles. All index methods discussed there have in common that for all words inverse lists are managed that indicate in which profile a word occurs. Here we shortly outline the *counting method*: for each profile, a counter exists that is initialized to zero. For each element in the *set* of words occurring in the input document, the inverse list is looked up and the counters of all profiles mentioned in this list are incremented. If a counter reaches the length of its corresponding profile, a notification for this profile is sent. Because of words occurring more than once in the input document the counting method does not work on the sequence of words in a document but requires that this sequence is converted into a set.

A weakness of the index methods described by Yan and Garcia-Molina is the missing support for substring matches. Building the set of all substrings occurring in an input document would be a highly inefficient approach. By employing the GSM-based technique presented in this work, the set of all keywords occurring as substrings in the input document can be computed efficiently.

Another weakness of the approach of Yan and Garcia-Molina is its limitation to conjunctive profiles. Although it is straight-forward to support profiles in disjunctive form and hence arbitrary Boolean profiles, this may cause a combinatorial explosion of the number of profiles. This is for instance the case when a thesaurus is used to expand terms in a profile into disjunctions of related terms.

[1]This construction is for instance used to prove that the class of regular languages is closed with respect to intersection.

6.3.3 Composite events in active databases

A problem very similar to the matching a large number of Boolean profiles against a document occurs in active databases: there the task is to identify composite events in a stream of atomic events that occur as the database content changes. There exist various algebras for composite events, but most of them support the Boolean operators. If one interprets a document as a stream of atomic events describing occurrences of words or substrings, then a Boolean profile can be seen as a composite event triggering a rule that sends a notification to the profile owner.

There are various approaches to monitor active databases for composite events, using techniques such as colored petrinets [3] or operator graphs [4, 2, 5]. For an overview of active database technology in general, please see [6].

7 Conclusion

In this report we have presented a method for simultaneously matching in linear time a set of search strings against an input document. We have introduced incremental algorithms for maintaining the finite state machine constructed from the set of search strings. Several extensions of the profile language have been discussed. A straight-forward extension to Boolean profiles turned out not to be possible. We have sketched an approach for Boolean profiles based on multiple finite state machines that achieves an efficiency similar to the methods presented in [8].

References

[1] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 606–617, Los Altos, CA 94022, USA, 1995. Morgan Kaufmann Publishers.

[3] S. Gatziu and K. R. Dittrich. Detecting Composite Events in Active Databases Using Petri Nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering (RIDE)- Active Database Systems*, pages 2–9, 1994.

[4] Eric N. Hanson. Rule condition testing and action execution in Ariel. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, volume 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 49–58, New York, NY 10036, USA, 1992. ACM Press.

[5] U. Jaeger. SMILE - A framework for lossless situation detection. In *Proceedings of the 5th Annual Workshop on Information Technologies and Systems WITS*, pages 110–119, 1995.

[6] U. Jaeger and J. C. Freytag. An annotated bibliography on active databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(1):58–69, March 1995.

[7] Arto Salomaa. *Computation and Automata.* Cambridge University Press, 1985.

[8] Tak W. Yan and Héctor García-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, June 1994.

A. Constructing a NFA

It is quite easy to construct a nondeterministic finite automaton (NFA) that accepts the language $L = A^*WA^*$. The graph of this automaton M_{NFA} is shown in Fig. 5. There is a common initial state q_0, from which n chains depart each of which matches a word w_i. Each chain ends in an accepting state. Additionally, the initial state has a self-edge for each element of the input alphabet A.

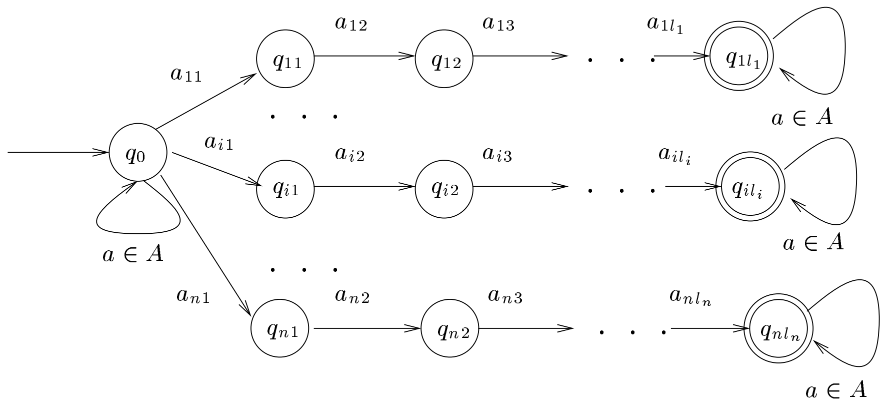

Figure 5: The nondeterministic finite automaton M_{NFA} accepting A^*WA^*.