University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2005-2694

An Effective Algorithm for The Membership Problem for Extended Regular Expressions

Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA.
grosu@cs.uiuc.edu

Abstract. By adding the complement operator ($\neg$), extended regular expressions (*ERE*) can encode regular languages non-elementarily more succinctly than regular expressions. The *ERE* membership problem asks whether a word w of size n belongs to the language of an *ERE* R of size m. Unfortunately, the best known membership algorithms are either non-elementary in m or otherwise require space $\Omega(n^2)$ and time $\Omega(n^3)$; since in many practical applications n can be very large (in the order of billions, e.g., in testing where w represents the execution trace of some system), these space and time requirements could be prohibitive. In this paper we present a simple to implement *ERE* membership algorithm that runs in space $O(n \cdot (m + \log n) \cdot 2^m \cdot k)$ and in time $O(n^2 \cdot (m + \log n)^2 \cdot 2^m \cdot k)$, where k is the number of complement operators in R. The presented algorithm outperforms the best known algorithms when n is large.

Topics. Automata and formal languages, algorithms, data-structures.

1 Introduction

Regular expressions represent a compact and useful technique to specify patterns in strings. There are programming and/or scripting languages, such as Perl, which are mostly based on efficient implementations of pattern matching via regular expressions. Extended regular expressions (*ERE*s), which add complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition ($R^\star$) operators, make the description of regular languages more convenient and more succinct. The membership problem for an *ERE* R and a word w is to decide whether w is in the regular language generated by R. The size of w is typically much larger than that of R. Due to their simplicity and popularity, regular expressions, and implicitly the membership problem, have many applications and not only in computer science. For example, [7] suggests interesting applications in molecular biology. Many of today's programming languages have either builtin efficient regular expression membership algorithms or provide libraries for them. Testing is another interesting application area; the execution of physical processes or computer programs can be logged and then

searched for property violations. Since many safety properties are more naturally expressed as what should *not* happen or as *intersection* of several policies, *ERE* are particularly desirable. Moreover, since testing/logging sessions can be quite long, sometimes days or weeks, *ERE* membership algorithms that are efficient in the length of the word are highly preferred.

The simplest-minded solution would be generate a DFA or an NFA from R, and then to check the membership of w in linear time with n by simply traversing w letter-by-letter once. Unfortunately, this may not always be practical. This is because the size of the NFA or DFA can be non-elementarily larger than R [11]. Even if one succeeded to store such an immense automaton, checking the word against it would still be non-elementary, because one needs non-elementarily long labels for each state. There could admittedly be practical situations in which one can quickly generate a DFA or an NFA from R; if this is the case, then one should definitely use this simple algorithm. From a practical perspective, the work in this paper can be seen as an alternative to the simple-minded algorithm, when generating a standard automaton from R is not plausible.

There are several other *ERE* membership algorithms in precisely the same category. The first such algorithm was introduced in [4] in 1979, and ran in space $O(n^2 \cdot m)$ and time $O(n^3 \cdot m)$. A technique for speeding up membership algorithms by a factor of $\log n$ is presented in [9]. An interesting *ERE*-membership algorithm was then proposed in [3] in 1989, and was claimed to run in space and time $O(n^2 \cdot m)$. However, as stated in [8, 16], the algorithm in [3] actually has the same complexity as the one in [4], namely space $O(n^2 \cdot m)$ and time $O(n^3 \cdot m)$. Recently, a series of algorithms claiming better complexities have been published. First, [13] presents an algorithm based on synchronized automata for a special kind of *ERE*s, namely ones having intersection but not negation, claimed to run in space $O(n^2 \cdot k + n \cdot m)$ and time $O(n^2 \cdot m)$, where k is the number of complement operations in R. As acknowledged by the author of [13] in [14, 16, 15], his algorithm actually runs in time $O(n^3 \cdot m)$.

Two other algorithms claiming the same complexity as the one claimed in [13], but for general *ERE*s, have been independently published in 2002 and 2003 [8, 6]. As the author of [6] acknowledges [5], their algorithm was also mis-analyzed and actually runs in time $O(n^3 \cdot m)$ as well. Moreover, we agree with the authors of both [13] and [6] that the algorithm in [8] suffers from a similar mis-analysis [15, 5]. Indeed, even though the procedure *Update* is called $O(n^2 \cdot m)$ times as stated in the fifth line from the end of the paper [8], each call to *Update* may invoke a "for" loop (the line before the last one in the procedure *Update*), which in the worst case may loop

$O(n)$ times; even if one considers that the body of the loop takes constant time, that would still make the algorithm in [8] run in $O(n^3 \cdot m)$.

Consequently, all the attempts to asymptotically improve the direct 25-year-old algorithm in [4] failed. Motivated by efforts in monitoring and testing [1], where execution traces are typically much much larger then the *ERE*s, we took a fresh look at the *ERE*-membership problem in [10] – one focusing on developing *ERE* membership algorithms that are *low in n but still not non-elementary in m*. The algorithm that we proposed in [10] was claimed to run in time $O(n \cdot 2^{m^2})$, but, unfortunately, it was our turn to escape a subtle error in the analysis of our algorithm[1]. We believe now that the algorithm in [10] is probably *non-elementary* in m.

In this paper we present an *ERE* membership algorithm that runs in space $O(n \cdot (m + \log n) \cdot 2^m \cdot k)$ and in time $O(n^2 \cdot (m + \log n)^2 \cdot 2^m \cdot k)$. When n is asymptotically larger than $(m + \log n)^2 \cdot 2^m$, which is easily the case in many practical applications, e.g., testing, our algorithm outperforms all the existing *ERE* membership algorithms. Even if the algorithms in [3, 13, 8, 6] could be somehow modified to attain the claimed complexities, but this seems to be highly non-trivial, the algorithm presented in this paper would still outperform them space-wise when n is large ($\Omega(n \cdot \log n)$ rather than $\Omega(n^2)$), at the expense of just a $\log^2 n$ factor time-wise (this is not much considering the already existing n^2 factor).

The basic idea of our algorithm is to repeatedly cut the EREs at complement operators to obtain a data-structure of nested NFAs. Formally, this is performed by introducing novel notions of *contextual regular expressions* and *automata*. To achieve the effect of complementation at each cut point, special novel data-structures, called *jumping machines* and implemented using priority queues, are introduced; these encode information needed to "jump" to the next subword which is *not* in the corresponding language. The advantage of jumping machines is that one does not need to store (via indexes) all the subwords which are not in the language, as previous (unsuccessful) attempts did, but only the next one; so we drop a factor of n in storage. The price is that we need to store additional information, in the order of 2^m, to be able to jump to the next subword.

Is exponential in the size of the *ERE* acceptable? We think that in most practical cases the answer is yes. In fact, many applications that need to test membership of a word to an NFA prefer to use an off-the-shelf NFA-to-DFA translator and then check membership to the DFA, an easier to implement task; the (single-)exponential blow-up in the size of the NFA tends not to be regarded as a practical limitation. Those who, by principle,

[1] We thank Prasanna Thati for detecting this error.

are firm supporters of "polynomial is better than exponential", can regard the result in this paper in the light of the previous failed attempts to improve the known polynomial bounds. At our knowledge, this is the first result which reduces the polynomial complexity w.r.t. n without incurring a non-elementary blow-up in m. Our algorithm does not only theoretically improves a long standing upper bound, but also can be easily implemented.

2 Preliminaries, Notations and Assumptions

Large numbers and memory. The need for novel *ERE* membership algorithms is motivated by their use in contexts where the length n of the word can be prohibitively large for the existing algorithms. Therefore, it would be *unreasonable* to assume that indexes $0 \leq i \leq n$ take constant space and time to be stored, incremented or compared. Consequently, in this paper, we regard numbers as arbitrarily long arrays of bits, so it takes logarithmic rather than constant space to represent a number. Also, we assume that it takes logarithmic time to increment or to compare numbers. Since *ERE* membership algorithms tend to require much space, it would also be unreasonable in this context to assume that a memory access takes constant time. To simplify our analyses, we assume that there is only one memory space and that any bit memory access takes time $O(\log M)$, where M is the total memory needed by the given algorithm. Thus, it takes time $O(\log n \cdot \log M)$ to store a number n in memory.

If one thinks that we are over-pessimistic here, or if one's implementation or use of our algorithm ensures that n is small enough in order for numbers to fit in one or very few computer words, then one can make this assumption explicit and the complexity of our algorithm will drop by a logarithmic factor. However, from now on in the paper we take the conservative side and assume that indexes, their basic operations, as well as memory accesses need logarithmic space and/or time. We make a simplifying assumption though: fixed-dimension matrix element access also takes time $O(\log M)$. More precisely, one can regard an access to an element $d[i, j, k]$ in a matrix $d[1..n, 1..n, 1..n]$ as a sequence of fixed length of indirect accesses $d[i][j][k]$, where d accesses a location where a vector of pointers $d[1..n]$ is found, then $d[i]$ is accessed which is a pointer to the vector of pointers $d[i][1..n]$, and then finally $d[i][j]$ is accessed which points to the vector of elements $d[i][j][1..n]$. Therefore, an access to an element of a matrix involves a fixed number of memory accesses, so we reasonably assume it to also take $O(\log M)$ time. Note that, since $O(\log n + \log m) = O(\log(n+m))$, if M is a polynomial in n and m then $O(\log M) = O(\log(n + m))$. With these assumptions, a careful analysis shows that the algorithm in [4] actu-

ally takes time $O(n^3 \cdot m \cdot \log(n+m))$. Regardless of how one analyzes *ERE* membership algorithms, the one presented in this paper is asymptotically better than the already existing ones when the word to test is very large.

Languages. In this paper, Σ is a set called *alphabet* whose elements are called *letters*, and X is a set of *variables*. The elements of $\Sigma^\star$, i.e., finite sequences of letters in Σ, are called *Σ-words* or simply *words*. We let ϵ denote the empty word. If $w \in \Sigma^\star$ then we let $|w|$ denote the *length* of w and w_i the ith letter of w. If w has n letters then we can also write w as $w_1 w_2 \cdots w_n$. If $1 \leq i \leq j \leq n$ then $w_i w_{i+1} \cdots w_j$ is the *subword* of w between i and j. If $i > j$ then $w_i w_{i+1} \cdots w_j$ is ϵ by convention. A language over Σ is a subset of $\Sigma^\star$. We let $\mathcal{L}_\Sigma$ denote the set of languages over Σ, i.e., the powerset $\mathcal{P}(\Sigma^\star)$. Let $\emptyset$ denote the empty language. If $L_1, L_2 \in \mathcal{L}_\Sigma$ then $L_1 \cdot L_2$ is the language $\{\alpha_1\alpha_2 \mid \alpha_1 \in L_1 \text{ and } \alpha_2 \in L_2\}$. If $L \in \mathcal{L}_\Sigma$ then $L^\star$ is $\{\alpha_1\alpha_2\cdots\alpha_n \mid n \geq 0 \text{ and } \alpha_1, \alpha_2, \ldots, \alpha_n \in L\}$ and $\neg L$ is $\Sigma^\star - L$.

Extended regular expressions. (*ERE*s) define languages by inductively applying *union* ($+$), *concatenation* ($\cdot$), *Kleene Closure* ($\star$), *intersection* ($\cap$), and *complementation* ($\neg$). The language of an *ERE* R, denoted by $L(R)$, is defined inductively as follows, where a is any letter in Σ: $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$, $L(R_1 + R_2) = L(R_1) \cup L(R_2)$, $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$, $L(R^\star) = (L(R))^\star$, $L(R_1 \cap R_2) = L(R_1) \cap L(R_2)$, $L(\neg R) = \neg L(R)$. One can define a procedure to check $\epsilon \in L(R)$ by just traversing R once. If R does not contain $\neg$ then it is a *regular expression* (*RE*). By applying De Morgan's law $R_1 \cap R_2 \equiv \neg(\neg R_1 + \neg R_2)$, *ERE*s can be linearly (in both time and size) translated into equivalent *ERE*s without intersection. Hence, in the sequel we consider expressions without intersection. If Σ is not understood from context, then we let ERE_Σ denote the set of *ERE*s over letters in Σ and let RE_Σ denote the set of *RE*s over Σ. We use R, R_1, R_2, R', etc., for *ERE*s, and r, r_1, r_2, r', etc., for *RE*s.

The *size* of an *ERE* is the total number of occurrences of letters and composition operators ($\cup$, $\cdot$, $\star$, and $\neg$) that it contains. We store *ERE*s as syntactic trees flattened as vectors in memory, each node keeping an encoding of its operation/letter; since each node in an *ERE* R of size m takes space $O(\log m)$, the space required to store R is $O(m \cdot \log m)$. In the context of an algorithm that needs memory $O(M)$, a traversal of R takes time $O(m \cdot \log m \cdot \log M)$. Since we frequently need to check if $\epsilon \in R$, we consider one additional bit in each node saying whether ϵ is in the language of the sub-*ERE* rooted in that node. It takes therefore $O(m \cdot \log m \cdot \log M)$ to calculate all these bits.

For any map $\varphi : X \to ERE_\Sigma$, we let $\varphi : ERE_{\Sigma \cup X} \to ERE_\Sigma$ also denote its unique extension to a *morphism*, that is, the map with $\varphi(\emptyset) = \emptyset$, $\varphi(\epsilon) = \epsilon$, $\varphi(a) = a$ for any $a \in \Sigma$, $\varphi(R_1 + R_2) = \varphi(R_1) + \varphi(R_2)$, $\varphi(R_1 \cdot R_2) = \varphi(R_1) \cdot \varphi(R_2)$, $\varphi(R^\star) = (\varphi(R))^\star$, and $\varphi(\neg R) = \neg\varphi(R)$; also, we let $\varphi_\neg : X \to ERE_\Sigma$ denote the map defined by $\varphi_\neg(x) = \neg\varphi(x)$.

Automata. *Non-deterministic finite automata (NFA) with ϵ-transitions* are used in this paper, i.e., tuples $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is an alphabet, $\delta : S \times (\Sigma \cup \{\epsilon\}) \to 2^S$ is a *transition function*, s_0 is an *initial state*, and F is a set of *final states*. We let NFA_Σ denote the set of such automata. It is well-known that one can associate an *NFA* A_r to any regular expression r. Moreover, the number of nodes and edges of A_r is linear with the size of r. Note, however, that the number of nodes and edges of an *NFA* may be significantly smaller than the total space needed to *store* the *NFA*. This is because one needs to store a *unique* label for each node in A_r, which needs logarithmic space in the number of nodes. The translation of an *RE* into an *NFA* that is most frequently used in practice, and the one that will also be considered in this paper, is perhaps the one due to Thompson [12], which is depicted in Figure 1.

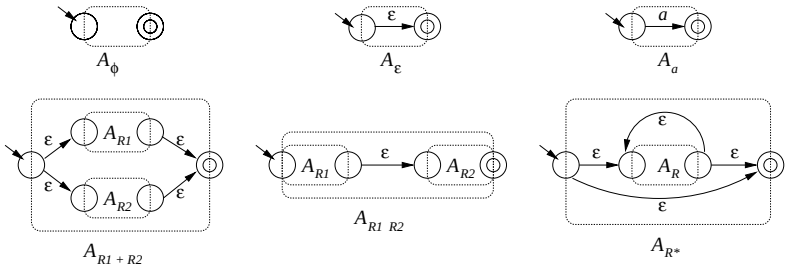

Fig. 1. Thompson's translation

$O(1)$ nodes and edges are added per constructor of the *RE*, so the total number of nodes/edges in the resulting *NFA* is linear with the size of the original *RE*. An important observation for this paper is that a letter a occurs exactly once in r iff a occurs on exactly one edge in A_r.

We assume a procedure Gen-NFA taking *RE*s to *NFA*s, using Thompson's construction. We are not interested here in how to store and handle *NFA*s efficiently, because this would not improve asymptotically our algorithm. We assume a simple encoding where each node together with its out-edges are grouped together starting with some given memory address,

and where each edge contains a label and a pointer to its successor state. Therefore, if m is the size of the *RE* r and M the size of memory, then GEN-NFA(r) takes space $O(m \cdot \log M)$ and time $O(m \cdot \log m \cdot \log M)$.

There are two other important *NFA* operations that can be performed in the same space/time as above, *ϵ-closure* and the *global step*. Given $Q \subseteq S$ and a letter a, the *ϵ-closure* of Q is the set $\overline{\delta}(Q, \epsilon)$ of states that can be reached starting with a state in Q and applying only ϵ-transitions, and the *global step* $\delta(Q, a)$ is the set of states $\cup_{s \in Q} \delta(s, a)$. We encode sets of m states in an *NFA* as vectors of m bits: 1 means that the corresponding state is in the set. The implementation of these operations is simple. The first, e.g., maintains a queue T of states, originally equal to Q, that still need to be processed; then it picks and removes a state from T and considers each of its ϵ-transitions. If a new state is found, add it to both T and the result set. Repeat until T becomes empty. Both these algorithms just traverse the automaton, so their complexity is $O(m \cdot \log M)$ for space and $O(m \cdot \log m \cdot \log M)$ for time. Note also that *intersection*, *union* and *emptiness* test on sets of states, represented as m-dimensional vectors, can be easily implemented in space $O(m)$ and time $O(m \cdot \log M)$.

Priority queues [2] are structures useful to maintain sets, supporting insertion and extraction of elements, as well as access to a "highest priority" element. They are routinely implemented in linear space using heaps flattened in vectors. The appealing aspect of priority queues is that insertion and extraction take log time, while accessing the highest priority element takes constant time. These numbers, however, assume that elements take constant space/time to store, access and compare. We here need to store *large* elements, so we cannot neglect their size.

Let $\mathcal{E}$ be the set of elements to be stored. Assume $\mathcal{E}$ has a potentially very large number η of elements, and that each element needs $O(\rho)$ space to be stored. In our application $\mathcal{E}$ is easily enumerable, so we assume that $\mathcal{E}$ is the set $\{1, 2, ..., \eta\}$ and ρ is $\log \eta$. Assume also that each element has a *key*, or a priority, which can be calculated and compared against other keys in $O(\log \eta)$. Moreover, since a priority queue can be implemented space-effectively using a heap as a complete binary tree flattened in a vector of η elements, one needs to handle indexes between 1 and η to access elements in the heap. Since η can be huge, assume that operations on indexes also take time $O(\log \eta)$. We need operations to initialize queues, to insert, extract and access top-priority elements. We next discuss each of these operations.

INITIALIZE($\mathcal{Q}, \mathcal{E}$)] initializes queue $\mathcal{Q}$ to maintain the elements in $\mathcal{E}$. It allocates a vector V of η cells, each cell of space $O(\log \eta)$. Since our priority queues do *not contain duplicates*, a vector B of η booleans is also

allocated, telling which elements are already in the queue. Thus, the total space required by INITIALIZE is $O(\eta \log \eta)$. Let us conservatively assume that each bit requires $O(\log M)$ time to be allocated/initialized, where M is the size of all the memory our algorithm requires. Then the time complexity of INITIALIZE is $O(\eta \cdot \log \eta \cdot \log M)$;

INSERT$(\mathcal{Q}, e)$ inserts e in $\mathcal{Q}$. When the size of elements and memory access time are neglected, this takes logarithmic time: one only needs to traverse and update bottom-up one path in the heap. However, under our large-number assumption, the complexity of INSERT is $O(\log^2 \eta \cdot \log M)$. Our sets $\mathcal{E}$ will contain a special "undefined" element $\perp$ that we do *not* want to be inserted in queues. To do that, and also to avoid duplicates, INSERT$(\mathcal{Q}, e)$ first checks whether $e = \perp$ or $B[e]$ is true, and exits if any of them holds. Otherwise, it inserts e in V and sets $B[e]$ to true;

EXTRACT-TOP$(\mathcal{Q})$ takes the same time as INSERT, $O(\log^2 \eta \cdot \log M)$. Once the top of $\mathcal{Q}$ is extracted, its corresponding entry in B is set to false.

TOP$(\mathcal{Q})$ returns the element at the top of the heap, without removing it, so it takes $O(\log \eta \cdot \log M)$.

In short, the complexity of priority queues grows by $\log \eta \cdot \log M$ when one does *not* neglect the size of elements and the memory access time.

3 Contextual Regular Expressions and Automata

Definition 1. *A **contextual regular expression over letters** Σ **and variables** X is a regular expression in $RE_{\Sigma \cup X}$ containing exactly one occurrence of each variable in X. We let $RE_\Sigma[X]$ denote the set of contextual regular expressions over Σ and X.*

The restriction to one variable does *not* apply to the language of a contextual *RE*. Indeed, if $r \in RE_\Sigma[X]$ then $\alpha \in L(r)$ can have zero, one or more occurrences of any $x \in X$. The motivation for contextual *RE*s comes from the fact that any *ERE* can be decomposed in a "root" contextual regular expression, together with an *ERE* with fewer complement operations associated to each variable. This well-founded decomposition of *ERE*s is a crucial step in our membership algorithm.

Proposition 1. *For any $R \in ERE_\Sigma$, there is a set of variables X, an $r \in RE_\Sigma[X]$, and a map $\varphi : X \to ERE_\Sigma$, such that $R = \varphi_\neg(r)$. Moreover, for any $x \in X$, the ERE $\varphi(x)$ contains strictly fewer complement operations than R. We call r the **root** of R.*

One can actually decompose an *ERE* R into (X, r, φ) in space $O(m_r \cdot \log M)$ and time $O(m_r \cdot \log m_r \cdot \log M)$. Figure 2 gives a possible implemen-

DECOMPOSE(R)
Input: *ERE* R
Output: triple (X, r, φ)
1. case R of
2. $\vdots$ $\{\emptyset, \epsilon\} \cup \Sigma$: return $(\emptyset, R, \emptyset)$
3. $\vdots$ $R_1 + R_2 : (X_1, r_1, \varphi_1) \leftarrow$ DECOMPOSE(R_1); $(X_2, r_2, \varphi_2) \leftarrow$ DECOMPOSE(R_2);
$\vdots$ return $(X_1 \cup X_2, r_1 + r_2, \varphi_1 \cup \varphi_2)$
4. $\vdots$ $R_1 \cdot R_2 : (X_1, r_1, \varphi_1) \leftarrow$ DECOMPOSE(R_1); $(X_2, r_2, \varphi_2) \leftarrow$ DECOMPOSE(R_2);
$\vdots$ return $(X_1 \cup X_2, r_1 \cdot r_2, \varphi_1 \cup \varphi_2)$
5. $\vdots$ $R'^{\star} : (X, r', \varphi) \leftarrow$ DECOMPOSE(R'); return $(X, r'^{\star}, \varphi)$
6. $\vdots$ $\neg R'$: return $(\{x\}, x, \varphi)$, where x is a fresh variable, and $\varphi(x) = R'$
7. endcase

Fig. 2. Procedure decomposing an *ERE* into a contextual *RE* and a substitution.

tation of the procedure DECOMPOSE performing the task of decomposing an *ERE* R into a triple (X, r, φ) as stated in Proposition 1. If one assumes that each call to DECOMPOSE is passed addresses where to write its resulting X, r, and φ, respectively, and that $\varphi(x)$ is a pointer to a sub-*ERE* of R, then one can obtain an implementation in which only $O(\log M)$ additional work is needed in each node of the tree representing r. Therefore, the total time of DECOMPOSE(R) remains the same as that of traversing r, say of size m_r, that is, $O(m_r \cdot \log m_r \cdot \log M)$. The space it requires to store its output is $O(m_r \cdot \log M)$, because $\varphi(x)$ is a pointer in memory, so it takes space $O(\log M)$.

Definition 2. *Automata in $NFA_{\Sigma \cup X}$ containing for each $x \in X$ exactly one edge labeled with x are called* ***contextual automata over letters Σ and variables X.*** *Let $NFA_{\Sigma}[X]$ denote the set of such automata. To emphasize their contextual nature, we write such automata as tuples $(S, \Sigma, X, \delta, s_0, F)$ rather then $(S, \Sigma \cup X, \delta, s_0, F)$. In any contextual automaton, let $in_x, out_x \in S$ denote respectively the source and the target states of the edge labeled x, for each $x \in X$.*

Note that Thompson's construction takes contextual *RE*s in $RE_{\Sigma}[X]$ to contextual *NFA*s in $NFA_{\Sigma}[X]$. One can associate any $R \in ERE_{\Sigma}$ a contextual automaton by first decomposing it into some (X, r, φ) and then taking GEN-NFA(r). Continuing this automata generation process for each $\varphi(x)$, one eventually gets a structure of "nested" *NFA*s, one for each complement operation in the original *ERE*. To ease the task of calculating ϵ-closures in such automata, we prefer to shortcut a nested *NFA* by an ϵ-transition whenever it contains ϵ in its language:

Definition 3. *Given $R \in ERE_\Sigma$ decomposing to (X, r, φ), the **root** NFA **of** R is the NFA returned by* GEN-NFA(r) *in which a new edge $\delta(in_x, \epsilon) = out_x$ is added for each $x \in X$ with $\epsilon \in L(\neg\varphi(x))$.*

With this, note that $\epsilon \in L(R)$ iff $\overline{\delta}(\{s_0\}, \epsilon) \cap F \neq \emptyset$. Let us next give an automata-based characterization for the membership of any w to $L(R)$.

Definition 4. *Let $w = w_1 w_2 \cdots w_n \in \Sigma^\star$, let $R \in ERE_\Sigma$ decompose to (X, r, φ), and let $(S, \Sigma, X, \delta, s_0, F)$ be the root NFA of R. Then we define Z_0, Z_1, Z_2, ..., Z_n as the smallest sets of states closed under the following:*

- *$s_0 \in Z_0$;*
- *$\overline{\delta}(Z_i, \epsilon) \subseteq Z_i$ for each $i \in \{0, 1, ..., n\}$;*
- *$\delta(Z_i, w_{i+1}) \subseteq Z_{i+1}$ for each $i \in \{0, 1, ..., n-1\}$;*
- *if $in_x \in Z_i$ for some $i \in \{0, 1, ..., n\}$ and $x \in X$ then $out_x \in Z_j$ for all $j \in \{i+1, ..., n\}$ with $w_{i+1} \cdots w_j \in L(\neg\varphi(x))$.*

Note that the "smallest sets" in the definition above makes sense, because sequences of sets closed under the operations above are also closed under componentwise intersection.

Proposition 2. *With the notation above, $w \in L(R)$ iff $Z_n \cap F \neq \emptyset$.*

The proposition above immediately implies that $w \notin L(R)$ iff $Z_n \cap F = \emptyset$. Since the definition of Z_0, Z_1, ..., Z_n is based on memberships of the subwords $w_{i+1} \cdots w_j$ to the languages $L(\varphi(x))$, which can be iteratively reduced to generating the root *NFA* of $\varphi(x)$ and then checking for emptiness the intersection of its final states with some corresponding Z set obtained like Z_n, one can now derive a membership algorithm based on root automata. In what follows we present an algorithm which, considering the information $w_{i+1} \cdots w_j \in L(\neg\varphi(x))$ encoded in some convenient way, calculates all the sets Z_0, Z_1, ..., Z_n and then checks for membership.

Definition 5. *Given $w = w_1 w_2 \cdots w_n \in \Sigma^\star$ and $L \subseteq \Sigma^\star$, a map $t : \{0, 1, ..., n-1\} \times \{1, 2, ..., n\} \to \{0, 1\}$ **is a table for** w **and** L if and only if for any $0 \leq i < j \leq n$, $t[i][j] = 1$ iff $w_{i+1} \cdots w_j \in L$.*

The simplest way to represent a table is as (half) an $n \times n$ matrix of boolean values. As far as the calculation of Z_0, Z_1, ..., Z_n and the membership of w to R are concerned, a set of tables $\{t_x$ table for w and $\neg\varphi(x) \mid x \in X\}$ would *contain all the necessary information* regarding the map $\varphi : X \to ERE_\Sigma$. Figure 3 shows an *ERE* membership algorithm that generates the table of each *ERE*-subexpression occurring under a complement from the tables of its subexpressions.

```
MEMB-WITH-TABLES(w, R)
Input: w = w_1 w_2 ··· w_n ∈ Σ*,  R ∈ ERE
Output: true / false
Globals: Z_0, Z_1, ..., Z_n
1.  %GEN-TABLE-STRUCTURES
2.  Z_0 ← {s_0}
3.  for i ← 1, 2, ..., n do Z_i ← ∅ endfor
4.  for i ← 0, 1, ..., n do
5.  ⋮ %STEP-WITH-TABLES
6.  endfor
7.  return Z_n ∩ F ≠ ∅

GEN-TABLE(w, R)
Input: w = w_1 w_2 ··· w_n ∈ Σ*,  R ∈ ERE
Output: table t
 1.  %GEN-TABLE-STRUCTURES
 2.  for l = 0, 1, ..., n − 1 do
 3.  ⋮ Z_l ← {s_0}
 4.  ⋮ for i ← l + 1, ..., n do
 5.  ⋮ ⋮ Z_i ← ∅
 6.  ⋮ ⋮ t[l][i] ← 0
 7.  ⋮ endfor
 8.  ⋮ for i ← l, ..., n do
 9.  ⋮ ⋮ %STEP-WITH-TABLES
10.  ⋮ ⋮ if Z_i ∩ F = ∅ and (i > l) then
11.  ⋮ ⋮ ⋮ t[l][i] ← 1
12.  ⋮ ⋮ endif
13.  ⋮ endfor
14.  endfor
15.  return t

macro %GEN-TABLE-STRUCTURES
 1.  (X, r, φ) ← DECOMPOSE(R)
 2.  for all x ∈ X do
 3.  ⋮ t_x ← GEN-TABLE(w, φ(x))
 4.  endfor
 5.  (S, Σ, X, δ, s_0, F) ← GEN-NFA(r)
 6.  for all x ∈ X do
 7.  ⋮ if ε ∉ L(φ(x)) then
 8.  ⋮ ⋮ δ(in_x, ε) ← out_x
 9.  ⋮ endif
10.  endfor

macro %STEP-WITH-TABLES
 1.  Z_i ← δ̄(Z_i, ε)
 2.  if i < n then
 3.  ⋮ for all x ∈ X do
 4.  ⋮ ⋮ if in_x ∈ Z_i then
 5.  ⋮ ⋮ ⋮ for j ← i + 1, ..., n do
 6.  ⋮ ⋮ ⋮ ⋮ if t_x[i][j] then
 7.  ⋮ ⋮ ⋮ ⋮ ⋮ Z_j ← Z_j ∪ {out_x}
 8.  ⋮ ⋮ ⋮ ⋮ endif
 9.  ⋮ ⋮ ⋮ endfor
10.  ⋮ ⋮ endif
11.  ⋮ endfor
12.  ⋮ Z_{i+1} ← Z_{i+1} ∪ δ(Z_i, w_{i+1})
13.  endif
```

Fig. 3. Membership algorithm using tables.

Proposition 3. *The algorithm* MEMB-WITH-TABLES(w, R) *in Figure 3 returns* true *if and only if* $w \in L(R)$. *If* $|w| = n$, $|R| = m$, *and* R *contains* k *complement operations, then this algorithm runs in space* $O(n^2 \cdot k + n \cdot m + m \cdot \log(n + m))$ *and time* $O((n^3 \cdot k + n^2 \cdot m \cdot \log m) \cdot \log(n + m))$.

Proof. To simplify its presentation and analysis, the algorithm in Figure 3 is split into two procedures and 2 macros. The macros should be regarded "ad literam", that is, one should simply replace their "invocation" by their pseudocode, character-by-character. %GEN-TABLE-STRUCTURES assumes some *ERE* R and some word w, and first decomposes R into (X, r, φ), then generates the corresponding tables for each $\neg\varphi(x)$ (in fact, for (non-asymptotic) efficiency, the procedure GEN-TABLE is passed $\varphi(x)$, but note that at its Steps 10-11 it actually sets the table bits to 1 when the subword

is *not* in the language), and finally generates the root automaton of R. The macro %Step-With-Tables performs a "global step" in a root automaton. It assumes some step number i, corresponding to the latest processed letter in w, for which all sets Z_0, Z_1, ..., Z_{i-1} are already completely calculated and for which the sets Z_i, Z_{i+1}, ..., Z_n are only partially calculated, and finishes the calculation of Z_i, which only needs an ϵ-closure, and then updates the remaining Z_{i+1}, ..., Z_n as follows: if Z_i contains any special state in_x then the table t_x is consulted on its level $t_x[i]$ and all the sets Z_j with $w_{i+1} \cdots w_j \in L(\neg\varphi(x))$ are updated with the special state out_x; finally, the set Z_{i+1} is also updated by processing the next letter, w_{i+1}, in the current global state, Z_i. The procedure Gen-Table(w, R) will always be called on a sub-*ERE* R occurring under a complement in the original *ERE*, for which a table therefore needs to be generated. For each $0 \leq l \leq n-1$, it needs to set to 1 all the entries $t[l][i]$ for which $w_{l+1} \cdots w_i \in L(\neg R)$ (note that R is always some $\varphi(x)$ in its "parent" *ERE*). This can be done by first setting Z_l to $\{s_0\}$ and then simply traversing all the i's, completing Z_i and updating Z_{i+1}, ..., Z_n, and also checking whether Z_i contains any final state. The main procedure, Memb-With-Tables, is now self-explanatory. This algorithm follows more or less blindly Definition 4, so its correctness follows by Proposition 2.

Let us next calculate the complexity of this algorithm. One should first notice that the required memory M is polynomial in n and m, so bit memory accesses take $O(\log M) = O(\log(n+m))$. Note that the sets Z_0, Z_1, ..., Z_n can be reused at each invocation of Memb-With-Tables and/or Gen-Table, so we define them as global; these sets of states are represented as vectors of bits of size m, so they take total space $O(n \cdot m)$. Note also that not all the m bits of Z_0, Z_1, ..., Z_n are always needed: only m_r are necessary, where m_r is the size of the *RE* root r of the current *ERE* R.

Let us first analyze %Gen-Table-Structures, both with respect to space and time. Note that this macro is invoked by both Memb-With-Tables and Gen-Table, and both of these have a current *ERE* R; let m_r be the size of the *RE* root r of R. Step 1 takes space $O(m_r \cdot \log M)$ and time $O(m_r \cdot \log m_r \cdot \log M)$, including the time to update the bits stating the membership of ϵ to the language of each subterm of R. Steps 2-4 take space $O(\sum_{x \in X} space_{\mathrm{GT}(x)})$ and time $O(\sum_{x \in X} time_{\mathrm{GT}(x)})$, where $space_{\mathrm{GT}(x)}$ and $time_{\mathrm{GT}(x)}$ are the space and the time of Gen-Table$(w, \varphi(x))$. The $O(n^2)$ space needed to store the table t_x will be counted as part of $space_{\mathrm{GT}(x)}$; what is assigned to t_x is a pointer to the table already generated by Gen-Table$(w, \varphi(x))$. Step 5 takes space $O(m_r \cdot \log M)$ and time $O(m_r \cdot \log m_r \cdot \log M)$; assume the worst case space here, so adding new edges

(at most one per node) to the automaton later will not require additional space. Since $\varphi(x)$ already contains the information $\epsilon \in \varphi(s)$ and since no new space is needed to add a new edge to a node in the automaton, Steps 6-10 take constant space and $O(m_r \cdot \log M)$ time. Summing all these up, we obtain that %GEN-TABLE-STRUCTURES takes space $O(m_r \cdot \log M + \sum_{x \in X} space_{\mathrm{GT}(x)})$ and time $O(m_r \cdot \log m_r \cdot \log M + \sum_{x \in X} time_{\mathrm{GT}(x)})$.

Let us now analyze %STEP-WITH-TABLES. The space for the global sets Z_0, Z_1, ..., Z_n has been already counted, and the space for the other operations can be reused, so this macro should take constant space in a good implementation. Anyhow, we can afford to assume, conservatively, that the space needed by the various operations is not reused, so the total space of %STEP-WITH-TABLES is $O(m_r \cdot \log M)$. Steps 1 and 11 take time $O(m_r \cdot \log m_r \cdot \log M)$ and Step 7 takes $O(\log M)$, so the total time taken by %STEP-WITH-TABLES is $O((|X| \cdot n + m_r \cdot \log m_r) \cdot \log M)$.

Let us next analyze GEN-TABLE. Since it needs to create the table t of size $O(n^2)$, one can readily see that it takes space $O(n^2 + m_r \cdot \log M + \sum_{x \in X} space_{\mathrm{GT}(x)})$. Step 1 takes time $O(m_r \cdot \log m_r \cdot \log M + \sum_{x \in X} time_{\mathrm{GT}(x)})$. Steps 8-12, taking the major time in the outmost loop, take time $O(n \cdot (|X| \cdot n + m_r \cdot \log m_r) \cdot \log M)$, so the total time taken by GEN-TABLE is $O(n^2 \cdot (n \cdot |X| + m_r \cdot \log m_r) \cdot \log M + \sum_{x \in X} time_{\mathrm{GT}(x)})$.

We can now analyze the main procedure, MEMB-WITH-TABLES. Without making explicit the space and time of invoked GEN-TABLE, one can readily see that MEMB-WITH-TABLES takes space $O(m_r \cdot \log M + \sum_{x \in X} space_{\mathrm{GT}(x)})$ and time $O(n^2 \cdot |X| \cdot \log M + n \cdot m_r \cdot \log m_r \cdot \log M + \sum_{x \in X} time_{\mathrm{GT}(x)})$. To complete the analysis, note that GEN-TABLE is eventually invoked exactly once on every *ERE* R' with $\neg R'$ a subterm of the original *ERE* R. Since the sum of all the sizes $m_{r'}$ of the *RE* roots of these *ERE*s R' is $O(m)$, one can relatively easily see that the total space of MEMB-WITH-TABLES is $O(n^2 \cdot k + m \cdot M)$ plus the total space $O(n \cdot m)$ to store Z_0, Z_1, ..., Z_n; since $O(\log M) = O(log(n+m))$, the total space required by MEMB-WITH-TABLES is $O(n^2 \cdot k + n \cdot m + m \cdot \log(n+m))$. One can similarly calculate the total time of MEMB-WITH-TABLES to $O((n^3 \cdot k + n^2 \cdot m \cdot \log m) \cdot \log(n+m))$.

The space above can be non-asymptotically improved, by noting that once a table is calculated for an *ERE*, the tables of its subexpressions are not necessary anymore, so their space can be reused. Since in practical situations n is larger than m, the algorithm in Figure 3 provides a slight improvement over the one in [4]. Unfortunately, it still requires space $\Omega(n^2)$. Since n is expected to be a very large number, typically much much larger than m, the space required by this algorithm can be prohibitively large in many applications of interest. Clearly, the problem here comes from storing

the tables t_x for $x \in X$, each requiring $\Theta(n^2)$ space. We will next see that one can significantly reduce the required space as a function of n, namely from n^2 to $n \cdot \log n$. The idea is to encode the languages of $\varphi(x)$ for $x \in X$ in a more space effective fashion.

4 An Effective ERE Membership Algorithm

Definition 6. *A **jumping machine** $\mathcal{P} = (P, p_0, \pi)$ consists of set P of **states**, an **initial state** p_0, and a **jumping map** $\pi : \{0, 1, ..., n-1\} \times P \to (\{1, 2, ..., n\} \times P) \cup \{\perp\}$ with the property that for any $0 \leq i < n$ and any $p \in P$, if $\pi(i, p) = (j, p')$ then $i < j$. Given $0 \leq i < n$, we let $\pi(i)$ denote the set $\{j_1, j_2, ..., j_{n_i}\}$ with $\pi(i, p_0) = (j_1, p_1)$, $\pi(j_1, p_1) = (j_2, p_2)$, ..., $\pi(j_{n_i-1}, p_{n_i-1}) = (j_{n_i}, p_{n_i})$, $\pi(j_{n_i}, p_{n_i}) = \perp$. Given word $w = w_1 w_2 \cdots w_n$ and language L, we say that (P, p_0, π) **is a jumping machine for** w **and** L if and only if $\pi(i) = \{j \mid j > i,\ w_{i+1} \cdots w_j \in L\}$.*

Therefore, a jumping machine provides a mechanism to generate the sets $\pi(i)$ in a stepwise manner. A jumping machine for w and L can therefore eventually produce the same information as a table for w and L. However, the advantage of jumping machines in contrast to tables is that they may require *much less space* to be stored. Indeed, a machine (P, p_0, π) can be encoded in space $\Theta(n \cdot |P| \cdot (\log n + \log |P|))$, namely when encoded as a $n \times |P|$ matrix storing in each cell an element in $(\{1, 2, ..., n\} \times P) \cup \{\perp\}$. This space can be roughly approximated with $\Theta(n \cdot \log n)$ when n is significantly larger than $|P|$, as opposed to $\Theta(n^2)$ as required by tables. Figure 4 shows an *ERE* membership algorithm based on jumping machines, that modifies the one in Figure 3 appropriately.

Theorem 1. MEMB-WITH-MACHINES(w, R) *in Figure 4 returns* true *iff* $w \in L(R)$. *If* $|w| = n$, $|R| = m$, *and* R *contains* k *complement operations, then* MEMB-WITH-MACHINES(w, R) *runs in space* $O(n \cdot (m + \log n) \cdot 2^m \cdot k)$ *and in time* $O(n^2 \cdot (m + \log n)^2 \cdot 2^m \cdot k)$.

Proof. One may show the correctness of this algorithm by analogy with the table-based algorithm in Figure 3, which is the reason for which we actually presented the table-based algorithm. In the table-based algorithm, given an *ERE* R that decomposed to (X, r, φ), we maintained a table t_x listing *explicitly* the entire "future" of each $\varphi(x)$ w.r.t. the remaining suffix of w (i.e., the set of future indexes $1 \leq j \leq n$ for which the special state out_x needs to be added to the current set of states Z_j at that moment). We now maintain a jumping machine $\mathcal{P}_x = (P_x, p_0^x, \pi)$ instead, which, at any "moment", i.e., index $0 \leq i \leq n-1$, "knows" explicitly only the

first future moment when out_x needs to be considered, namely the one given by the first component of $\pi[i][\{p_0^x\}]$. However, the jumping machine also "freezes" its corresponding state at that future moment (the second component of $\pi[i][\{p_0^x\}]$), so that it *implicitly* "knows" how to generate the entire information in the corresponding table in the table-based algorithm; but this will be done on a *by-need* basis.

Like in the table-based algorithm, the ultimate purpose of the data-structures, jumping machines in this case, is to detect the future indexes at which the special states out_x need to be included in the set of (future) current states. In the table-based algorithm, the sets Z_0, Z_1, ..., Z_n accumulated this information progressively, by simply transferring it from the tables. Since the tables are not available anymore, when the special state in_x is encountered during the global step of the root automaton, we need to store somewhere the first future moment, say i, that out_x needs to be considered. That informal "somewhere" can be effectively replaced by a *priority queue* data-structure, $\mathcal{Q}_x$. Since the state in_x can be encountered several times before that moment i, each time starting a new "jumping session" in $\mathcal{P}_x$, we need to store *all* the first future moments to consider out_x of all the "sessions" that the jumping machine $\mathcal{P}_x$ can be in. Then at any global step of the algorithm, one needs to check whether any of the jumping machine sessions "predicted" the current moment as one to include out_x. If that is the case then, besides including out_x in the current global state, one also needs to advance the corresponding session in jumping machine to its next "predicted" moment to include the state out_x. This is what Steps 1-9 in %Step-With-Machines do. To accomplish this task properly, we store not only the first future moments of each session in the priority queue, but also the corresponding jumping machine session. Since several different sessions in $\mathcal{P}_x$ could have predicted the same current moment, all these sessions need to be advanced to their next predicted future moments to consider out_x (Steps 4-7 in %Step-With-Machines). Making the intuitions above rigorous, the algorithm Memb-With-Machines in Figure 4 flows in a one-to-one analogy to the table-based algorithm in Figure 3. As a "synchronization" point in this analogy, note that Z at Step 10 in %Step-With-Machines corresponds to Z_i at Step 1 in %Step-With-Tables.

Let us next analyze the space and time complexity of this algorithm. Note first that the total memory M required by this algorithm is polynomial in n and exponential in m, so $O(\log M) = O(m + \log n)$. Following a similar analysis to that of %Gen-Table-Structures, one immediately gets that %Gen-Machine-Structures requires space $O(m_r \cdot \log M + \sum_{x \in X} space_{\mathrm{GM}(x)})$ and time $O(m_r \cdot \log m_r \cdot \log M + \sum_{x \in X} time_{\mathrm{GM}(x)})$. Gen-

MACHINE tells us that P_x will have size 2^{m_x}, where m_x is the size of the root of $\varphi(x)$. Considering $\mathcal{E}_x := \{1, 2, ..., n\} \times P_x$ of size $\eta_x := n \cdot 2^{m_x}$ in the analysis of INITIALIZE of a queue, one obtains that %INITIALIZE-QUEUES takes space $O(n \cdot \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}))$ and time $O(n \cdot \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) \cdot \log M)$. %STEP-WITH-MACHINES is invoked at places where all the memory it needs is allocated, so it takes constant space. The crucial observation in the time analysis of %STEP-WITH-MACHINES is that the loop at Steps 4-7 executes at most 2^{m_x} times, because there can be at most that many pairs (i, p) in total and because we do not allow duplicates in queues. Therefore, Steps 1-9 take time $O(\sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) \cdot \log M)$. Steps 10-18 only add time $O(m_r \cdot \log m_r \cdot \log M)$, where m_r is the size of r, so the total time of %STEP-WITH-MACHINES is $O((\sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) + m_r \cdot \log m_r) \cdot \log M)$.

Let us now analyze the remaining two procedures. Step 1 in each of them takes space $O(m_r \cdot \log M + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$. GEN-MACHINE needs to allocate a jumping machine, whose space is dominated by the matrix π of size $n \times 2^{m_r}$ keeping elements in $\{1, 2, ..., n\} \times 2^S$, so each element of size $\log(n \cdot 2^{m_r})$. Therefore, the total space required by π is $O(n \cdot 2^{m_r} \cdot \log(n \cdot 2^{m_r}))$. Since %INITIALIZE-QUEUES at Step 5 can reuse the same space for each iteration of the loop at Steps 4-13, we conclude that the total space required by GEN-MACHINE is $O(n \cdot (2^{m_r} \cdot \log(n \cdot 2^{m_r}) + \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x})) + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$. Time-wise, note that the loops at Steps 3 and 4, respectively, add a factor of $n \cdot 2^{m_r}$ to the time of Steps 5-12. After calculations, we get that the total time of GEN-MACHINE is $O(n^2 \cdot 2^{m_r} \cdot (m_r \cdot \log m_r + \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x})) \cdot \log M + \sum_{x \in X} \mathit{time}_{\mathrm{GM}(x)})$. Without making explicit the space and time of the invoked GEN-MACHINE, one can quickly see that MEMB-WITH-MACHINES takes space $O(m_r \cdot \log M + n \cdot \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$ and time $O(n \cdot (m_r \cdot \log m_r + \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x})) \cdot \log M + \sum_{x \in X} \mathit{time}_{\mathrm{GM}(x)})$.

Let us now put all these together by iteratively expanding all the $\mathit{space}_{\mathrm{GM}(x)}$ and $\mathit{time}_{\mathrm{GM}(x)}$. Let is first calculate the space. Note that if one iteratively expands the terms $\mathit{space}_{\mathrm{GM}(x)}$ that occur in the space complexity of MEMB-WITH-MACHINES, then each term of the form $n \cdot 2^{m_x} \cdot \log(n \cdot 2^{m_x})$ will occur exactly twice. The resulting space then will be $O(m_r \cdot \log M + n \cdot \sum_{r'} 2^{m_{r'}} \cdot \log(n \cdot 2^{m_{r'}}))$, where r' ranges over all the *RE* roots of all *ERE*s R' occurring under a $\neg$ operator in the original *ERE*, and $m_{r'}$ is the size of r'. Supposing that m' is the largest of the $m_{r'}$ sizes, the space becomes $O(m_r \cdot \log M + n \cdot k \cdot 2^{m'} \cdot \log(n \cdot 2^{m'}))$. Since $O(\log M) = O(m + \log n)$, since $m_r, m' \leq m$, and since we are more concerned about the size of n and the complexity of our algorithm with respect to n rather than m (as far as it does not become non-elementary in m), for writing pur-

poses we overestimate the space required by MEMB-WITH-MACHINES to $O(n \cdot (m + \log n) \cdot 2^m \cdot k)$. The total time of MEMB-WITH-MACHINES can be calculated in a similar manner to $O(n^2 \cdot (m + \log n)^2 \cdot 2^m \cdot k)$.

5 Conclusion

Previous known algorithms to test whether a word of size n is in the language of an *ERE* of size m are either space/time non-elementary in m or otherwise space $\Omega(n^2)$ and time $\Omega(n^3)$. Several attempts in the last 25 years to asymptotically improve these bounds failed. Existing applications in which n is huge and much much larger than m, motivate algorithms that are close to linear and n, but obviously not non-elementary in m. In this paper we presented an algorithm which is simply exponential in m but is in the order of $n \cdot \log n$ space-wise and $n^2 \cdot \log^2 n$ time-wise.

References

1. F. Chen, M. D'Amorim, and G. Rosu. A Formal Monitoring-based Framework for Software Development and Analysis. In *Proceedings of ICFEM'04*, volume 3308 of *LNCS*, pages 357–372. Springer, 2004.
2. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
3. S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
4. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
5. L. Ilie, 2004. Personal communication.
6. L. Ilie, B. Shan, and S. Yu. Fast algorithms for ERE matching and searching. In *Proc. of STACS'03*, volume 2607 of *LNCS*, pages 179–190, 2003.
7. J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.
8. O. Kupferman and S. Zuhovitzky. An improved algorithm for the memb. problem for ERE. In *Proc. of MFCS'02*, volume 2420 of *LNCS*, pages 446–458, 2002.
9. G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
10. G. Roşu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 499–514. Springer, 2003.
11. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). pages 1–9. ACM Press, 1973.
12. K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
13. H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proc. of MFCS'00*, volume 1893 of *LNCS*, pages 699–708, 2000.
14. H. Yamamoto. A new recognition algorithm for extended regular expressions. In *Proceedings of ISAAC'01*, volume 2223 of *LNCS*, pages 257–267, 2001.
15. H. Yamamoto, 2004. Personal communication.
16. H. Yamamoto and T. Miyazaki. A fast bit-parallel algorithm for matching ERE. In *Proc. of COCOON'03*, volume 2697 of *LNCS*, pages 222–231, 2003.

```
MEMB-WITH-MACHINES(w, R)
Input: w = w_1 w_2 ··· w_n ∈ Σ*
       R ∈ ERE
Output: true / false
Globals: Z', Z
1.  %GEN-MACHINE-STRUCTURES
2.  %INITIALIZE-QUEUES
3.  Z' ← {s_0}
4.  for i ← 0, 1, ..., n do
5.  ⋮ %STEP-WITH-MACHINES
6.  endfor
7.  return Z ∩ F ≠ ∅
```

```
GEN-MACHINE(w, R)
Input: w = w_1 w_2 ··· w_n ∈ Σ*
       R ∈ ERE
Output: machine (P, p_0, π)
 1.  %GEN-MACHINE-STRUCTURES
 2.  P ← 2^S;  p_0 ← {s_0}
 3.  for l = 0, 1, ..., n − 1 do
 4.  ⋮ for all p ∈ P do
 5.  ⋮ ⋮ %INITIALIZE-QUEUES
 6.  ⋮ ⋮ Z' ← p
 7.  ⋮ ⋮ for i ← l, ..., n do
 8.  ⋮ ⋮ ⋮ %STEP-WITH-MACHINES
 9.  ⋮ ⋮ ⋮ if Z ∩ F = ∅ and (i > l) then
10.  ⋮ ⋮ ⋮ ⋮ π_x[l][p] ← i;  break-loop
11.  ⋮ ⋮ ⋮ endif
12.  ⋮ ⋮ endfor
13.  ⋮ endfor
14.  endfor
15.  return (P, p_0, π)
```

```
macro %GEN-MACHINE-STRUCTURES
 1.  (X, r, φ) ← DECOMPOSE(R)
 2.  for all x ∈ X do
 3.  ⋮ (P_x, p_0^x, π_x) ← GEN-MACHINE(w, φ(x))
 4.  endfor
 5.  (S, Σ, X, δ, s_0, F) ← GEN-NFA(r)
 6.  for all x ∈ X do
 7.  ⋮ if ε ∉ L(φ(x)) then
 8.  ⋮ ⋮ δ(in_x, ε) ← out_x
 9.  ⋮ endif
10.  endfor
```

```
macro %INITIALIZE-QUEUES
1.  for all x ∈ X do
2.  ⋮ INITIALIZE(Q_x, {1, 2, ..., n} × P_x)
3.  endfor
```

```
macro %STEP-WITH-MACHINES
 1.  for all x ∈ X do
 2.  ⋮ if key(TOP(Q_x)) equals i then
 3.  ⋮ ⋮ Z' ← Z' ∪ {out_x}
 4.  ⋮ ⋮ while key(TOP(Q_x)) equals i do
 5.  ⋮ ⋮ ⋮ (i, p_x) ← EXTRACT-TOP(Q_x)
 6.  ⋮ ⋮ ⋮ INSERT(Q_x, π_x[i][p_x])
 7.  ⋮ ⋮ endwhile
 8.  ⋮ endif
 9.  endfor
10.  Z ← δ̄(Z', ε)
11.  if i < n then
12.  ⋮ for all x ∈ X do
13.  ⋮ ⋮ if in_x ∈ Z then
14.  ⋮ ⋮ ⋮ INSERT(Q_x, π_x[i][p_0^x])
15.  ⋮ ⋮ endif
16.  ⋮ endfor
17.  ⋮ Z' ← δ(Z, w_{i+1})
18.  endif
```

Fig. 4. Membership algorithm using jumping machines.