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, these space and time requirements could be prohibitive. In this paper we present an *ERE* membership algorithm that runs in space $O(n \cdot (\log n + m) \cdot 2^m)$ and time $O(n^2 \cdot (\log n + m) \cdot 2^m)$. The presented algorithm outperforms the best known algorithms when n is exponentially larger than m.

1 Introduction

Regular expressions can compactly specify patterns in strings. 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.

Due to their simplicity and popularity, regular expressions, and implicitly the membership problem, have many applications. There are programming and/or scripting languages, such as Perl, which are mostly based on efficient implementations of pattern matching via regular expressions. Many languages either have builtin efficient regular expression membership algorithms or provide libraries for them. Testing is another application area; events produced by the execution of physical processes or computer programs can be logged and then searched for property violations. Also, [5] suggests applications in molecular biology. Since many properties are more naturally expressed as what should *not* happen or as *intersection* of several policies, *ERE*s are particularly desirable. Moreover, since the input words can be quite large (e.g., a chromosome can have hundreds of millions of nucleobases, or a log file can have billions of events), *ERE* membership algorithms that are efficient in the length of the word are highly preferred.

H. Seidl (Ed.): FOSSACS 2007, LNCS 4423, pp. 332–345, 2007.

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 [10]. Even if one succeeded to store such an immense automaton, running it would still be non-elementary on each letter in the input, 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 many other *ERE* membership algorithms in precisely the same category. The first such algorithm was introduced in [3] in 1979; it runs 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 [7]. Several *ERE* membership algorithms have been published since 1979, such as [2,12,13,14,6,4], improving slightly[1] the complexity of the now classic algorithm in [3]. More precisely, they reduced the space requirements to $O(n^2 \cdot k + n \cdot m)$ and the time to $O(n^3 \cdot k + n^2 \cdot m)$ or worse, where k is the number of complement operators in R. An *ERE* membership algorithm was presented in [9], which "rewrites" or "derives" the *ERE* by each letter in the input word; the lower-bound result in [10] tells that this algorithm is also worst-case non-elementary in the *ERE*, but, unlike in the simplistic NFA/DFA generation algorithm, the worst-case penalty is not paid upfront. At our knowledge, there are no *ERE* algorithms that are asymptotically better than the non-elementary-in-m one based on generation of NFA/DFA or than the dynamic-programming 27-year-old algorithm in [3].

In this paper we present an *ERE* membership algorithm that is not polynomial, but which avoids the non-elementary explosion in the size of the *ERE*. More precisely, it runs in space $O(n{\cdot}(\log n{+}m){\cdot}2^m)$ and in time $O(n^2{\cdot}(\log n{+}m){\cdot}2^m)$. When n is exponentially larger than m, in which case the "polynomial" algorithms would be exponential in the *ERE* anyway, our algorithm asymptotically outperforms all the known algorithms.

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, but only the next one; so we drop a factor of n in storage. The price to pay is that we need to store additional information to be able to jump to the next subword.

[1] Some of these algorithms originally claimed better improvements, but they turned out to be misanalysed – see [8] for a detailed discussion of this issue.

2 Preliminaries, Notations and Assumptions

Numbers and memory. To simplify the analysis and explanation of our algorithms, we adopt standard conventions about numbers and memory: numbers take constant time to store to and retrieve from memory, either independently or as elements of arrays or matrices. The companion report [8] gives an analysis of the algorithms presented in this paper considering that number storage/retrieval and operations on them (comparison, additions, etc.) and on memory require logarithmic time etc.; such pessimistic assumptions increase the time complexity of our algorithm by a logarithmic factor [8] (in the size of the input word and *ERE*). However, we still assume that logarithmic space is required to store a "large" number when it appears in more complex data-structures. For example, we build tables in our algorithm whose cells store pairs consisting of an index between 1 and n (n is the length of the input word) and a subset of states in an automaton having $O(m)$ states (m is the size of the input *ERE*); in this case, we will assume that it takes $O(\log n + m)$ space to store each cell in the table – this is what actually gives the "$\log n + m$" factor in the analysis of our algorithm. If one thinks that we are over-conservative here since n and m are small enough in practice that the number $n \cdot 2^m$ fits in a constant number of memory units (say, e.g., in two 64-bit words), then one can consider that our algorithm takes $O(n \cdot 2^m)$ space and $O(n^2 \cdot 2^m)$ time.

Languages. In this paper, Σ is a finite 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 $\neg$ and $\cap$ 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

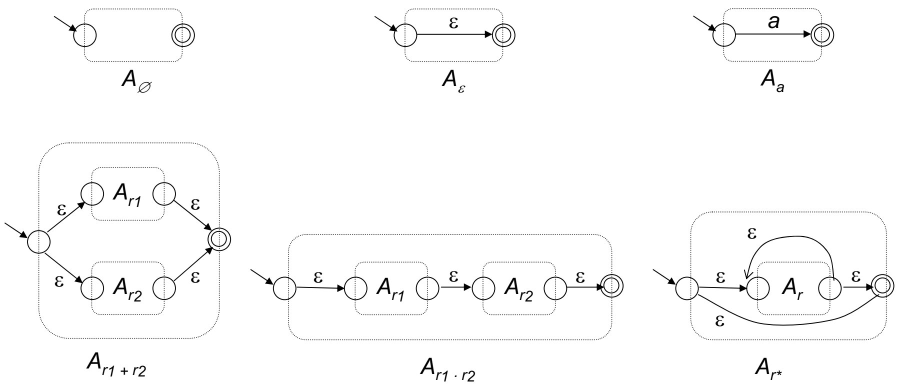

Fig. 1. Thompson's translation

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 ($+$, $\cdot$, $\star$, and $\neg$) that it contains. We will frequently need to check if $\epsilon \in L(R)$ for various subexpressions R of the original *ERE*; we can calculate all these ϵ-memberships in linear time in the original *ERE*, by a simple DFS traversal, updating the ϵ-membership of each subexpression from the corresponding ϵ-memberships of its subexpressions.

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. A common *RE*-to-*NFA* translation, due to Thompson [11], is shown in Figure 1. The resulting *NFA* is linear with the size of the original *RE*. An important observation for this paper is that a letter x occurs exactly once in r iff x occurs on exactly one edge in A_r.

We assume a linear-time linear-space procedure Gen-NFA taking *RE*s to *NFA*s, using Thompson's construction. There are two important *NFA* operations that can be performed in linear space/time, namely *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 can encode sets of states in an *NFA* of m states 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 (the latter is initially empty). Repeat until T is empty. Also, *intersection*, *union* and *emptiness* test on sets of states represented as vectors of size m take $O(m)$.

Priority queues. [1] 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, which can be initialized in linear time; we assume an $O(|\mathcal{E}|)$ procedure Initialize($\mathcal{Q}, \mathcal{E}$) that initializes queue $\mathcal{Q}$ to hold the elements $\mathcal{E}$ (also called "heapify"). 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 will accordingly tune these numbers (conservatively) when the elements in $\mathcal{E}$ require more than constant space to be stored.

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 operators 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 operators than R. We call r the **root** of R.*

In what follows we assume a procedure Decompose that takes *ERE*s R to triples (X, r, φ) as above. If one uses pointers to refer to regular (sub)expressions, then one can decompose an *ERE* R into (X, r, φ) in $O(m_r)$ space and time, where m_r is the size of r.

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 operator 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 component-wise 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* $w_{i+1} \cdots w_j \in L$*.*

```
MEMB-WITH-TABLES($w, R$)
Input: $w = w_1 w_2 \cdots w_n \in \Sigma^\star$, $R \in ERE$
Output: true / false
Globals: $Z_0, Z_1, ..., Z_n$
%GEN-TABLE-STRUCTURES
$Z_0 \leftarrow \{s_0\}$
for $i \leftarrow 1, 2, ..., n$ do $Z_i \leftarrow \emptyset$ endfor
for $i \leftarrow 0, 1, ..., n$ do
    %STEP-WITH-TABLES
endfor
return $Z_n \cap F \neq \emptyset$

GEN-TABLE($w, R$)
Input: $w = w_1 w_2 \cdots w_n \in \Sigma^\star$, $R \in ERE$
Output: table $t$
%GEN-TABLE-STRUCTURES
for $l = 0, 1, ..., n-1$ do
    $Z_l \leftarrow \{s_0\}$
    for $i \leftarrow l+1, ..., n$ do
        $Z_i \leftarrow \emptyset$
        $t[l][i] \leftarrow 0$
    endfor
    for $i \leftarrow l, ..., n$ do
        %STEP-WITH-TABLES
        if $Z_i \cap F = \emptyset$ and $(i > l)$ then
            $t[l][i] \leftarrow 1$
        endif
    endfor
endfor
return $t$

macro %GEN-TABLE-STRUCTURES
$(X, r, \varphi) \leftarrow$ DECOMPOSE($R$)
for all $x \in X$ do
    $t_x \leftarrow$ GEN-TABLE($w, \varphi(x)$)
endfor
$(S, \Sigma, X, \delta, s_0, F) \leftarrow$ GEN-NFA($r$)
for all $x \in X$ do
    if $\epsilon \notin L(\varphi(x))$ then
        $\delta(in_x, \epsilon) \leftarrow out_x$
    endif
endfor

macro %STEP-WITH-TABLES
$Z_i \leftarrow \overline{\delta}(Z_i, \epsilon)$
if $i < n$ then
    for all $x \in X$ do
        if $in_x \in Z_i$ then
            for $j \leftarrow i+1, ..., n$ do
                if $t_x[i][j]$ then
                    $Z_j \leftarrow Z_j \cup \{out_x\}$
                endif
            endfor
        endif
    endfor
    $Z_{i+1} \leftarrow Z_{i+1} \cup \delta(Z_i, w_{i+1})$
endif
```

Fig. 2. Membership algorithm using tables

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 2 shows an *ERE* membership algorithm that generates the table of each *ERE*-subexpression occurring under a complement from the tables of its subexpressions.

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

Proof. To simplify its presentation and analysis, the algorithm in Figure 2 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. 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 $O(n \cdot m)$.

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 and time $O(m_r)$, including the time to update the bits stating the membership of ϵ to the language of each subexpression of r (and thus 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 and time $O(m_r)$; 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 $\epsilon \in L(\varphi(x))$ 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 $O(m_r)$ time. Summing all these up, we obtain that %Gen-Table-Structures takes space $O(m_r + \sum_{x \in X} space_{\mathrm{GT}(x)})$ and time $O(m_r + \sum_{x \in X} time_{\mathrm{GT}(x)})$.

Let us now analyze %Step-With-Tables. The space for the global sets Z_0, Z_1, ..., Z_n has already been counted, and the space for the other operators 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 operators is not reused, so the total space of %STEP-WITH-TABLES is $O(m_r)$. Steps 1 and 12 take time $O(m_r)$ and Step 7 takes $O(1)$, so the total time taken by %STEP-WITH-TABLES is $O(|X| \cdot n + m_r)$.

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 $O(n^2 + m_r + \sum_{x \in X} \mathit{space}_{\mathrm{GT}(x)})$. Step 1 takes time $O(m_r + \sum_{x \in X} \mathit{time}_{\mathrm{GT}(x)})$. Steps 8-12, taking the major time in the outmost loop, take time $O(n \cdot (|X| \cdot n + m_r))$, so the total time taken by GEN-TABLE is $O(n^3 \cdot |X| + n^2 \cdot m_r + \sum_{x \in X} \mathit{time}_{\mathrm{GT}(x)})$.

We can now analyze the main procedure, MEMB-WITH-TABLES. Without making explicit the space and time consumed by GEN-TABLE, one can readily see that MEMB-WITH-TABLES takes space $O(m_r + \sum_{x \in X} \mathit{space}_{\mathrm{GT}(x)})$ and time $O(n^2 \cdot |X| + n \cdot m_r \cdot + \sum_{x \in X} \mathit{time}_{\mathrm{GT}(x)})$. 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 $O(n^2 \cdot k + m)$ plus the total space $O(n \cdot m)$ to store Z_0, Z_1, ..., Z_n, that is, $O(n^2 \cdot k + n \cdot m)$. One can similarly calculate the total time of MEMB-WITH-TABLES to $O(n^3 \cdot k + n^2 \cdot m)$.

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. Like the algorithms in [2,12,13,14,6,4], the algorithm in Figure 2 provides only a slight improvement over the classic one in [3]. Unfortunately, all known membership algorithms, including the one above, still require space $\Omega(n^2)$, which can be prohibitively large in many applications of interest. 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*** $\pi : \{0, 1, ..., n-1\} \times P \to (\{1, 2, ..., n\} \times P) \cup \{\bot\}$ *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* $\pi(i, p_0) = (j_1, p_1)$*,* $\pi(j_1, p_1) = (j_2, p_2)$*, ...,* $\pi(j_{n_i - 1}, p_{n_i - 1}) = (j_{n_i}, p_{n_i})$*,* $\pi(j_{n_i}, p_{n_i}) = \bot$*. 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

```
Memb-With-Machines($w, R$)
Input: $w = w_1 w_2 \cdots w_n \in \Sigma^\star$
       $R \in ERE$
Output: true / false
Globals: $Z', Z$
%Gen-Machine-Structures
%Initialize-Queues
$Z' \leftarrow \{s_0\}$
for $i \leftarrow 0, 1, ..., n$ do
    %Step-With-Machines
endfor
return $Z \cap F \neq \emptyset$
```

```
Gen-Machine($w, R$)
Input: $w = w_1 w_2 \cdots w_n \in \Sigma^\star$
       $R \in ERE$
Output: machine $(P, p_0, \pi)$
%Gen-Machine-Structures
$P \leftarrow 2^S$;  $p_0 \leftarrow \{s_0\}$
for $l = 0, 1, ..., n-1$ do
    for all $p \in P$ do
        %Initialize-Queues
        $Z' \leftarrow p$
        for $i \leftarrow l, ..., n$ do
            %Step-With-Machines
            if $Z \cap F = \emptyset$ and $(i > l)$ then
                $\pi_x[l][p] \leftarrow i$;  break-loop
            endif
        endfor
    endfor
endfor
return $(P, p_0, \pi)$
```

```
macro %Gen-Machine-Structures
$(X, r, \varphi) \leftarrow$ Decompose($R$)
for all $x \in X$ do
    $(P_x, p_0^x, \pi_x) \leftarrow$ Gen-Machine($w, \varphi(x)$)
endfor
$(S, \Sigma, X, \delta, s_0, F) \leftarrow$ Gen-NFA($r$)
for all $x \in X$ do
    if $\epsilon \notin L(\varphi(x))$ then
        $\delta(in_x, \epsilon) \leftarrow out_x$
    endif
endfor
```

```
macro %Initialize-Queues
for all $x \in X$ do
    Initialize($\mathcal{Q}_x, \{1, 2, ..., n\} \times P_x$)
endfor
```

```
macro %Step-With-Machines
for all $x \in X$ do
    if $key$(Top($\mathcal{Q}_x$)) equals $i$ then
        $Z' \leftarrow Z' \cup \{out_x\}$
        while $key$(Top($\mathcal{Q}_x$)) equals $i$ do
            $(i, p_x) \leftarrow$ Extract-Top($\mathcal{Q}_x$)
            Insert($\mathcal{Q}_x, \pi_x[i][p_x]$)
        endwhile
    endif
endfor
$Z \leftarrow \overline{\delta}(Z', \epsilon)$
if $i < n$ then
    for all $x \in X$ do
        if $in_x \in Z$ then
            Insert($\mathcal{Q}_x, \pi_x[i][p_0^x]$)
        endif
    endfor
    $Z' \leftarrow \delta(Z, w_{i+1})$
endif
```

Fig. 3. Membership algorithm using jumping machines

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 $(\{1, 2, ..., n\} \times P) \cup \{\bot\}$. 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 3 shows an *ERE* membership algorithm based on jumping machines, that modifies the one in Figure 2 correspondingly.

Theorem 1. Memb-With-Machines(w, R) *in Figure 3 returns* true *iff* $w \in L(R)$. *If* $|w| = n$ *and* $|R| = m$ *then* Memb-With-Machines(w, R) *runs in space* $O(n \cdot (\log n + m) \cdot 2^m)$ *and in time* $O(n^2 \cdot (\log n + m) \cdot 2^m)$.

Proof. One may show the correctness of this algorithm by analogy with the table-based algorithm in Figure 2, 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 the 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 3 flows in a one-to-one analogy to the table-based algorithm in Figure 2. 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. Following a similar analysis to that of %Gen-Table-Structures, one immediately gets that %Gen-Machine-Structures requires space $O(m_r + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$ and time $O(m_r + \sum_{x \in X} \mathit{time}_{\mathrm{GM}(x)})$. Gen-Machine tells us that P_x will have size 2^{m_x}, where m_x is the size of the root of $\varphi(x)$. Since we need to insert $|\{1, 2, ..., n\} \times P_x|$, which is $n \cdot 2^{m_x}$, in the priority queue $\mathcal{Q}_x$, and since each element requires space $\log(n \cdot 2^{m_x})$ to be stored as part of the Initialize step of the queue, one obtains that %Initialize-Queues takes space and time $O(n \cdot \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}))$. %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 $O(\sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}))$. Steps 10-18 only add time $O(m_r)$, where m_r is the size of r, so the total time of %Step-With-Machines is $O(\sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) + m_r)$.

Let us now analyze the remaining two procedures. Step 1 in each of them takes space $O(m_r + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$. 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 $O(n^2 \cdot 2^{m_r} \cdot (m_r + \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x})) + \sum_{x \in X} \mathit{time}_{\mathrm{GM}(x)})$. Without making explicit the space and time of the invoked Gen-Machine, one can quickly see that Memb-With-Machines takes space $O(m_r + n \cdot \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x}) + \sum_{x \in X} \mathit{space}_{\mathrm{GM}(x)})$ and time $O(n \cdot (m_r + \sum_{x \in X} 2^{m_x} \cdot \log(n \cdot 2^{m_x})) + \sum_{x \in X} \mathit{time}_{\mathrm{GM}(x)})$.

Let us now put all these together by iteratively expanding all the $\mathit{space}_{\mathrm{GM}(x)}$ and $\mathit{time}_{\mathrm{GM}(x)}$. Let us 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 $O(m_r + n \cdot \sum_{r'} 2^{m_{r'}} \cdot \log(n \cdot 2^{m_{r'}}))$, which is $O(m_r + n \cdot \log n \cdot \sum_{r'} 2^{m_{r'}} + n \cdot \sum_{r'} m_{r'} \cdot 2^{m_{r'}})$, 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'. Since $m_{r'} \leq m$ and since $\sum_{r'} 2^{m'_r} \leq 2^{\sum_{r'} m_{r'}} = 2^m$, by overestimation we get that the space required by Memb-With-Machines is $O(n \cdot (\log n + m) \cdot 2^m)$. The total time of Memb-With-Machines can be calculated in a similar manner to $O(n^2 \cdot (\log n + m) \cdot 2^m)$.

Corollary 1. *If $n > 2^m$, then our ERE-membership algorithm above runs in space $O(n \cdot \log n \cdot 2^m)$ and time $O(n^2 \cdot \log n \cdot 2^m)$.*

Hence, if $n > 2^m$, our algorithm above runs in space $O(n \cdot \log n \cdot 2^m)$, compared to $O(n^2 \cdot k + n \cdot m)$ (k is the number of complements in the *ERE*), the space required by the best known algorithms to solve the same problem; all algorithms, including ours, run in time a factor of n larger than their corresponding space. When does the algorithm proposed in this paper outperform the other algorithms? Roughly speaking, if $k = \Theta(m)$ then by the monotonicity of the function $x/\log x$ when $x > 2$, one gets $n/\log n > 2^m/m$, that is, that $n \cdot \log n \cdot 2^m$ is asymptotically better than $n^2 \cdot k$, so our algorithm wins. On the other hand, if $k = \Theta(1)$ then our algorithm again wins, but this time when $n > m \cdot 2^m$; however, if $k = \Theta(1)$ then one is likely better off using the standard NFA-to-DFA-then-complement algorithm.

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 $\Omega(n^3)$. In the 27 years that passed since the first non-elementary algorithm has been given in [3], several algorithms for the *ERE* membership problem have been proposed. Unfortunately, none of them improved significantly the original algorithm in [3]. In particular, all the current non-elementary-in-m algorithms require space $\Omega(n^2)$, which is prohibitive in the context of some applications of interest. For example, $\Omega(n^2)$ means more than 1TB of memory when n is 1 million, and more than what today's technology can offer when n is larger than 1 billion. 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 $n^2 \cdot \log n$ time-wise. The proposed algorithm outperforms the known polynomial algorithms when n is exponentially larger than m.

A novel data-structure, called jumping machine, was also introduced in this paper and played a crucial technical role in our algorithm. It would be interesting to investigate to what extent the jumping machines can be used for improving other automata-theoretic constructions (that use two dimensional tables).

References

1. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
2. S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
3. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
4. L. Ilie, B. Shan, and S. Yu. Fast algorithms for extended regular expression matching and searching. In *Proceedings of STACS'03*, volume 2607 of *LNCS*, pages 179–190, 2003.
5. J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2): 211–243, 1995.

6. O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *Proc. of MFCS'02*, volume 2420 of *LNCS*, pages 446–458, 2002.
7. G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
8. G. Roşu. An effective algorithm for the membership problem for extended regular expressions. Technical Report UIUCDCS-R-2005-2964, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.
9. 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.
10. L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). pages 1–9. ACM Press, 1973.
11. K. Thompson. Regular expression search algorithm. *CACM*, 11(6):419–422, 1968.
12. 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.
13. H. Yamamoto. A new recognition algorithm for extended regular expressions. In *Proceedings of ISAAC'01*, volume 2223 of *LNCS*, pages 257–267, 2001.
14. H. Yamamoto and T. Miyazaki. A fast bit-parallel algorithm for matching extended regular expressions. In *Proc. of COCOON'03*, volume 2697 of *LNCS*, pages 222–231, 2003.