New Algorithms for Regular Expression Matching

Philip Bille[*]

February 1, 2008

Abstract

In this paper we revisit the classical regular expression matching problem, namely, given a regular expression R and a string Q, decide if Q matches one of the strings specified by R. Let m and n be the length of R and Q, respectively. On a standard unit-cost RAM with word length $w \geq \log n$, we show that the problem can be solved in $O(m)$ space with the following running times:

$$\begin{cases} O(n\frac{m\log w}{w} + m\log w) & \text{if } m > w \\ O(n\log m + m\log m) & \text{if } \sqrt{w} < m \leq w \\ O(\min(n + m^2, n\log m + m\log m)) & \text{if } m \leq \sqrt{w}. \end{cases}$$

This improves the best known time bound among algorithms using $O(m)$ space. Whenever $w \geq \log^2 n$ it improves all known time bounds regardless of how much space is used.

1 Introduction

Regular expressions are a powerful and simple way to describe a set of strings. For this reason, they are often chosen as the input language for text processing applications. For instance, in the lexical analysis phase of compilers, regular expressions are often used to specify and distinguish tokens to be passed to the syntax analysis phase. Utilities such as Grep, the programming language Perl, and most modern text editors provide mechanisms for handling regular expressions. These applications all need to solve the classical Regular Expression Matching problem, namely, given a regular expression R and a string Q, decide if Q matches one of the strings specified by R.

The standard textbook solution, proposed by Thompson [11] in 1968, constructs a *non-deterministic finite automaton* (NFA) accepting all strings matching R. Subsequently, a state-set simulation checks if the NFA accepts Q. This leads to a simple $O(nm)$ time and $O(m)$ space algorithm, where m and n are the number of symbols in R and Q, respectively. The full details are reviewed later in Sec. 2 and can found in most textbooks on compilers (e.g. Aho et. al. [1]). Despite the importance of the problem, it took 24 years before the $O(nm)$ time bound was improved by Myers [8] in 1992, who achieved $O(\frac{nm}{\log n} + (n+m)\log n)$ time and $O(\frac{nm}{\log n})$ space. For most values of m and n this improves the $O(nm)$ algorithm by a $O(\log n)$ factor. Currently, this is the fastest known algorithm. Recently, Bille and Farach-Colton [4] showed how to reduce the space of Myers' solution to $O(n)$. Alternatively, they showed how to achieve a speedup of $O(\log m)$ over Thompson's algorithm while using $O(m)$ space. These results are all valid on a unit-cost RAM with w-bit words and a standard instruction set including addition, bitwise boolean operations, shifts, and multiplication. Each word is capable of holding a character of Q and hence $w \geq \log n$. The space complexities refer to the number of words used by the algorithm, not counting the input which is assumed to be read-only. All results presented here assume the same model. In this paper we present new algorithms achieving the following complexities:

[*]The IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark. Email: `beetle@itu.dk`. An extended abstract of this paper appeared in Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, 2006.

Theorem 1 *Given a regular expression R and a string Q of lengths m and n, respectively,* Regular Expression Matching *can be solved using $O(m)$ space with the following running times:*

$$\begin{cases} O(n\frac{m\log w}{w} + m\log w) & \text{if } m > w \\ O(n\log m + m\log m) & \text{if } \sqrt{w} < m \leq w \\ O(\min(n + m^2, n\log m + m\log m)) & \text{if } m \leq \sqrt{w}. \end{cases}$$

This represents the best known time bound among algorithms using $O(m)$ space. To compare these with previous results, consider a conservative word length of $w = \log n$. When the regular expression is "large", e.g., $m > \log n$, we achieve an $O(\frac{\log n}{\log\log n})$ factor speedup over Thompson's algorithm using $O(m)$ space. Hence, we simultaneously match the best known time and space bounds for the problem, with the exception of an $O(\log\log n)$ factor in time. More interestingly, consider the case when the regular expression is "small", e.g., $m = O(\log n)$. This is usually the case in most applications. To beat the $O(n\log n)$ time of Thompson's algorithm, the fast algorithms [4, 8] essentially convert the NFA mentioned above into a *deterministic finite automaton* (DFA) and then simulate this instead. Constructing and storing the DFA incurs an additional exponential time and space cost in m, i.e., $O(2^m) = O(n)$. However, the DFA can now be simulated in $O(n)$ time, leading to an $O(n)$ time and space algorithm. Surprisingly, our result shows that this exponential blow-up in m can be avoided with very little loss of efficiency. More precisely, we get an algorithm using $O(n\log\log n)$ time and $O(\log n)$ space. Hence, the space is improved exponentially at the cost of an $O(\log\log n)$ factor in time. In the case of an even smaller regular expression, e.g., $m = O(\sqrt{\log n})$, the slowdown can be eliminated and we achieve optimal $O(n)$ time. For larger word lengths our time bounds improve. In particular, when $w > \log n\log\log n$ the bound is better in all cases, except for $\sqrt{w} \leq m \leq w$, and when $w > \log^2 n$ it improves all known time bounds regardless of how much space is used.

The key to obtain our results is to avoid explicitly converting small NFAs into DFAs. Instead we show how to effectively simulate them directly using the parallelism available at the word-level of the machine model. The kind of idea is not new and has been applied to many other string matching problems, most famously, the Shift-Or algorithm [3], and the approximate string matching algorithm by Myers [9]. However, none of these algorithms can be easily extended to Regular Expression Matching. The main problem is the complicated dependencies between states in an NFA. Intuitively, a state may have long paths of ϵ-transitions to a large number of other states, all of which have to be traversed in parallel in the state-set simulation. To overcome this problem we develop several new techniques ultimately leading to Theorem 1. For instance, we introduce a new hierarchical decomposition of NFAs suitable for a parallel state-set simulation. We also show how state-set simulations of large NFAs efficiently reduces to simulating small NFAs.

The results presented in this paper are primarily of theoretical interest. However, we believe that most of the ideas are useful in practice. The previous algorithms require large tables for storing DFAs, and perform a long series of lookups in these tables. As the tables become large we can expect a high number of cache-misses during the lookups, thus limiting the speedup in practice. Since we avoid these tables, our algorithms do not suffer from this defect.

The paper is organized as follows. In Sec. 2 we review Thompson's NFA construction, and in Sec. 3 we present the above mentioned reduction. In Sec. 4 we present our first simple algorithm for the problem which is then improved in Sec. 5. Combining these algorithms with our reduction leads to Theorem 1. We conclude with a couple of remarks and open problems in Sec. 6.

2 Regular Expressions and Finite Automata

In this section we briefly review Thompson's construction and the standard state-set simulation. The set of *regular expressions* over an alphabet Σ are defined recursively as follows:

- A character $\alpha \in \Sigma$ is a regular expression.
- If S and T are regular expressions then so is the *concatenation*, $(S)\cdot(T)$, the *union*, $(S)|(T)$, and the *star*, $(S)^*$.

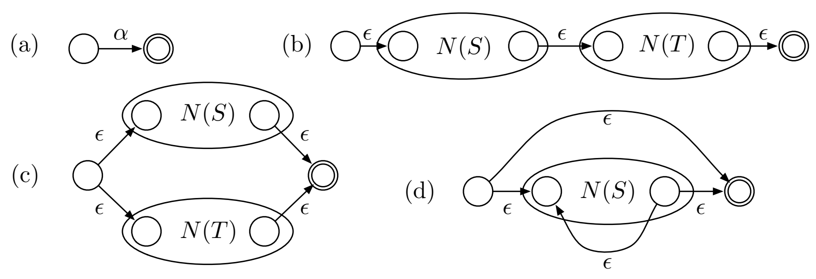

Figure 1: Thompson's NFA construction. The regular expression for a character $\alpha \in \Sigma$ corresponds to NFA (a). If S and T are regular expressions then $N(ST)$, $N(S|T)$, and $N(S^*)$ correspond to NFAs (b), (c), and (d), respectively. Accepting nodes are marked with a double circle.

Unnecessary parentheses can be removed by observing that $\cdot$ and $|$ is associative and by using the standard precedence of the operators, that is $*$ precedes $\cdot$, which in turn precedes $|$. We often remove the $\cdot$ when writing regular expressions.

The *language* $L(R)$ generated by R is the set of all strings matching R. The *parse tree* $T(R)$ of R is the binary rooted tree representing the hiearchical structure of R. Each leaf is labeled by a character in Σ and each internal node is labeled either $\cdot$, $|$, or $*$. A *finite automaton* is a tuple $A = (V, E, \delta, \theta, \phi)$, where

- V is a set of nodes called *states*,
- E is set of directed edges between states called *transitions*,
- $\delta : E \rightarrow \Sigma \cup \{\epsilon\}$ is a function assigning labels to transitions, and
- $\theta, \phi \in V$ are distinguished states called the *start state* and *accepting state*, respectively[1].

Intuitively, A is an edge-labeled directed graph with special start and accepting nodes. A is a *deterministic finite automaton* (DFA) if A does not contain any ϵ-transitions, and all outgoing transitions of any state have different labels. Otherwise, A is a *non-deterministic automaton* (NFA). We say that A *accepts* a string Q if there is a path from θ to ϕ such that the concatenation of labels on the path spells out Q. Thompson [11] showed how to recursively construct a NFA $N(R)$ accepting all strings in $L(R)$. The rules are presented below and illustrated in Fig. 1.

- $N(\alpha)$ is the automaton consisting of states θ_α, ϕ_α, and an α-transition from θ_α to ϕ_α.
- Let $N(S)$ and $N(T)$ be automata for regular expressions S and T with start and accepting states θ_S, θ_T, ϕ_S, and ϕ_T, respectively. Then, NFAs $N(S \cdot T)$, $N(S|T)$, and $N(S^*)$ are constructed as follows:

  $N(ST)$: Add start state θ_{ST} and accepting state ϕ_{ST}, and ϵ-transitions (θ_{ST}, θ_S), (ϕ_S, θ_T), and (ϕ_T, ϕ_{ST}).

  $N(S|T)$: Add start state $\theta_{S|T}$ and accepting state $\phi_{S|T}$, and add ϵ-transitions $(\theta_{S|T}, \theta_S)$, $(\theta_{S|T}, \theta_T)$, $(\phi_S, \phi_{S|T})$, and $(\phi_T, \phi_{S|T})$.

  $N(S^*)$: Add a new start state θ_{S^*} and accepting state ϕ_{S^*}, and ϵ-transitions (θ_{S^*}, θ_S), $(\theta_{S^*}, \phi_{S^*})$, (ϕ_S, ϕ_{S^*}), and (ϕ_S, θ_S).

Readers familiar with Thompson's construction will notice that $N(ST)$ is slightly different from the usual construction. This is done to simplify our later presentation and does not affect the worst case complexity of the problem. Any automaton produced by these rules we call a *Thompson-NFA* (TNFA). By construction, $N(R)$ has a single start and accepting state, denoted θ and ϕ, respectively. θ has no incoming transitions and

[1]Sometimes NFAs are allowed a *set* of accepting states, but this is not necessary for our purposes.

ϕ has no outgoing transitions. The total number of states is $2m$ and since each state has at most 2 outgoing transitions that the total number of transitions is at most $4m$. Furthermore, all incoming transitions have the same label, and we denote a state with incoming α-transitions an *α-state*. Note that the star construction in Fig. 1(d) introduces a transition from the accepting state of $N(S)$ to the start state of $N(S)$. All such transitions are called *back transitions* and all other transitions are *forward transitions*. We need the following property.

Lemma 1 (Myers [8]) *Any cycle-free path in a TNFA contains at most one back transition.*

For a string Q of length n the standard state-set simulation of $N(R)$ on Q produces a sequence of state-sets $S_0, \dots, S_n$. The ith set S_i, $0 \leq i \leq n$, consists of all states in $N(R)$ for which there is a path from θ that spells out the ith prefix of Q. The simulation can be implemented with the following simple operations. For a state-set S and a character $\alpha \in \Sigma$, define

$\mathsf{Move}(S, \alpha)$: Return the set of states reachable from S via a single α-transition.

$\mathsf{Close}(S)$: Return the set of states reachable from S via 0 or more ϵ-transitions.

Since the number of states and transitions in $N(R)$ is $O(m)$, both operations can be easily implemented in $O(m)$ time. The Close operation is often called an *ϵ-closure*. The simulation proceeds as follows: Initially, $S_0 := \mathsf{Close}(\{\theta\})$. If $Q[j] = \alpha$, $1 \leq j \leq n$, then $S_j := \mathsf{Close}(\mathsf{Move}(S_{j-1}, \alpha))$. Finally, $Q \in L(R)$ iff $\phi \in S_n$. Since each state-set S_j only depends on S_{j-1} this algorithm uses $O(mn)$ time and $O(m)$ space.

3 From Large to Small TNFAs

In this section we show how to simulate $N(R)$ by simulating a number of smaller TNFAs. We will use this to achieve our bounds when R is large.

3.1 Clustering Parse Trees and Decomposing TNFAs

Let R be a regular expression of length m. We first show how to decompose $N(R)$ into smaller TNFAs. This decomposition is based on a simple clustering of the parse tree $T(R)$. A *cluster* C is a connected subgraph of $T(R)$ and a *cluster partition* CS is a partition of the nodes of $T(R)$ into node-disjoint clusters. Since $T(R)$ is a binary tree with $O(m)$ nodes, a simple top-down procedure provides the following result (see e.g. [8]):

Lemma 2 *Given a regular expression R of length m and a parameter x, a cluster partition CS of $T(R)$ can be constructed in $O(m)$ time such that $|CS| = O(\lceil m/x \rceil)$, and for any $C \in CS$, the number of nodes in C is at most x.*

For a cluster partition CS, edges adjacent to two clusters are *external edges* and all other edges are *internal edges*. Contracting all internal edges in CS induces a *macro tree*, where each cluster is represented by a single *macro node*. Let C_v and C_w be two clusters with corresponding macro nodes v and w. We say that C_v is the *parent cluster* (resp. *child cluster*) of C_w if v is the parent (resp. child) of w in the macro tree. The *root cluster and leaf clusters* are the clusters corresponding to the root and the leaves of the macro tree. An example clustering of a parse tree is shown in Fig. 2(b). Given a cluster partition CS of $T(R)$ we show how to divide $N(R)$ into a set of small nested TNFAs. Each cluster $C \in CS$ will correspond to a TNFA A, and we use the terms child, parent, root, and leaf for the TNFAs in the same way we do with clusters. For a cluster $C \in CS$ with children $C_1, \dots, C_l$, insert a special *pseudo-node* p_i, $1 \leq i \leq l$, in the middle of the external edge connecting C with C_i. We label each pseudo-node by a special character $\beta \notin \Sigma$. Let T_C be the tree induced by the set of nodes in C and $\{p_1, \dots, p_l\}$. Each leaf in T_C is labeled with a character from $\Sigma \cup \{\beta\}$, and hence T_C is a well-formed parse tree for some regular expression R_C over $\Sigma \cup \{\beta\}$. Now, the TNFA A corresponding to C is $N(R_C)$. In A, child TNFA A_i is represented by its start and accepting state θ_{A_i} and ϕ_{A_i} and a *pseudo-transition* labeled β connecting them. An example of these definitions is given in Fig. 2. We call any set of TNFAs obtained from a cluster partition as above a *nested decomposition* AS of $N(R)$.

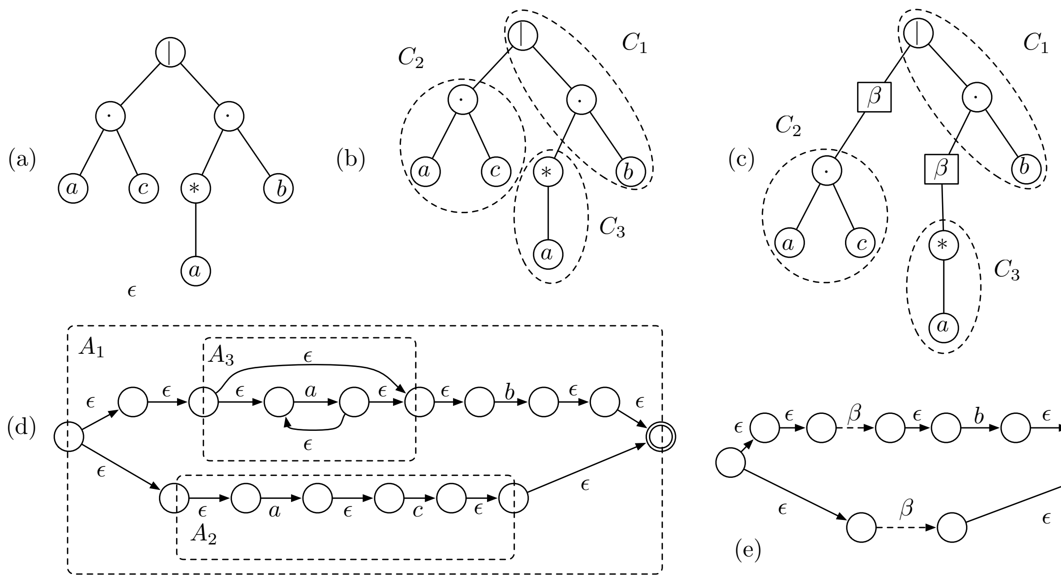

Figure 2: (a) The parse tree for the regular expression $ac|a^*b$. (b) A clustering of (a) into node-disjoint connected subtrees C_1, C_2, and C_3, each with at most 3 nodes. (c) The clustering from (b) extended with pseudo-nodes. (d) The nested decomposition of $N(ac|a^*b)$. (e) The TNFA corresponding to C_1.

Lemma 3 *Given a regular expression R of length m and a parameter x, a nested decomposition AS of $N(R)$ can be constructed in $O(m)$ time such that $|AS| = O(\lceil m/x \rceil)$, and for any $A \in AS$, the number of states in A is at most x.*

Proof. Construct the parse tree $T(R)$ for R and build a cluster partition CS according to Lemma 2 with parameter $y = \frac{x}{4} - \frac{1}{2}$. From CS build a nested decomposition AS as described above. Each $C \in CS$ corresponds to a TNFA $A \in AS$ and hence $|AS| = O(\lceil m/y \rceil) = O(\lceil m/x \rceil)$. Furthermore, if $|V(C)| \leq y$ we have $|V(T_C)| \leq 2y + 1$. Each node in T_C contributes two states to the corresponding TNFA A, and hence the total number of states in A is at most $4y + 2 = x$. Since the parse tree, the cluster partition, and the nested decomposition can be constructed in $O(m)$ time the result follows. $\square$

3.2 Simulating Large Automata

We now show how $N(R)$ can be simulated using the TNFAs in a nested decomposition. For this purpose we define a simple data structure to dynamically maintain the TNFAs. Let AS be a nested decomposition of $N(R)$ according to Lemma 3, for some parameter x. Let $A \in AS$ be a TNFA, let S_A be a state-set of A, let s be a state in A, and let $\alpha \in \Sigma$. A *simulation data structure* supports the 4 operations: $\mathsf{Move}_A(S_A, \alpha)$, $\mathsf{Close}_A(S_A)$, $\mathsf{Member}_A(S_A, s)$, and $\mathsf{Insert}_A(S_A, s)$. Here, the operations Move_A and Close_A are defined exactly as in Sec. 2, with the modification that they only work on A and not $N(R)$. The operation $\mathsf{Member}_A(S_A, s)$ returns yes if $s \in S_A$ and no otherwise and $\mathsf{Insert}_A(S_A, s)$ returns the set $S_A \cup \{s\}$.

In the following sections we consider various efficient implementations of simulation data structures. For now assume that we have a black-box data structure for each $A \in AS$. To simulate $N(R)$ we proceed as follows. First, fix an ordering of the TNFAs in the nested decomposition AS, e.g., by a preorder traversal of the tree represented given by the parent/child relationship of the TNFAs. The collection of state-sets for each TNFA in AS are represented in a *state-set array* X of length $|AS|$. The state-set array is indexed by the above numbering, that is, $X[i]$ is the state-set of the ith TNFA in AS. For notational convenience we write $X[A]$ to denote the entry in X corresponding to A. Note that a parent TNFA share two states with each child, and therefore a state may be represented more than once in X. To avoid complications we will always assure that X is *consistent*, meaning that if a state s is included in the state-set of some TNFA, then it is also included in the state-sets of all other TNFAs that share s. If $S = \bigcup_{A \in AS} X[A]$ we say that X *models* the state-set S and write $S \equiv X$.

Next we show how to do a state-set simulation of $N(R)$ using the operations Move_{AS} and Close_{AS}, which we define below. These operations recursively update a state-set array using the simulation data structures. For any $A \in AS$, state-set array X, and $\alpha \in \Sigma$ define

```
$\mathsf{Move}_{AS}(A, X, \alpha)$:   1. $X[A] := \mathsf{Move}_A(X[A], \alpha)$
                      2. For each child $A_i$ of $A$ in topological order do
                         (a) $X := \mathsf{Move}_{AS}(A_i, X, \alpha)$
                         (b) If $\phi_{A_i} \in X[A_i]$ then $X[A] := \mathsf{Insert}_A(X[A], \phi_{A_i})$
                      3. Return $X$

$\mathsf{Close}_{AS}(A, X)$:      1. $X[A] := \mathsf{Close}_A(X[A])$
                      2. For each child $A_i$ of $A$ in topological order do
                         (a) If $\theta_{A_i} \in X[A]$ then $X[A_i] := \mathsf{Insert}_{A_i}(X[A_i], \theta_{A_i})$
                         (b) X := $\mathsf{Close}_{AS}(A_i, X)$
                         (c) If $\phi_{A_i} \in X[A_i]$ then $X[A] := \mathsf{Insert}_A(X[A], \phi_{A_i})$
                         (d) $X[A] := \mathsf{Close}_A(X[A])$
                      3. Return $X$
```

The Move_{AS} and Close_{AS} operations recursively traverses the nested decomposition top-down processing the children in topological order. At each child the shared start and accepting states are propagated in the state-set array. For simplicity, we have written Member_A using the symbol $\in$.

The state-set simulation of $N(R)$ on a string Q of length n produces the sequence of state-set arrays $X_0, \ldots, X_n$ as follows: Let A_r be the root automaton and let X be an empty state-set array (all entries in X are $\emptyset$). Initially, set $X[A_r] := \mathsf{Insert}_{A_r}(X[A_r], \theta_{A_r})$ and compute $X_0 := \mathsf{Close}_{AS}(A_r, \mathsf{Close}_{AS}(A_r, X))$. For $i > 0$ we compute X_i from X_{i-1} as follows:

$$X_i := \mathsf{Close}_{AS}(A_r, \mathsf{Close}_{AS}(A_r, \mathsf{Move}_{AS}(A_r, X_{i-1}, Q[i])))$$

Finally, we output $Q \in L(R)$ iff $\phi_{A_r} \in X_n[A_r]$. To see that this algorithm correctly solves Regular Expression Matching it suffices to show that for any i, $0 \leq i \leq n$, X_i correctly models the ith state-set S_i in the standard state-set simulation. We need the following lemma.

Lemma 4 *Let X be a state-set array and let A_r be the root TNFA in a nested decomposition AS. If S is the state-set modeled by X, then*

- $\mathsf{Move}(S, \alpha) \equiv \mathsf{Move}_{AS}(A_r, X, \alpha)$ *and*
- $\mathsf{Close}(S) \equiv \mathsf{Close}_{AS}(A_r, \mathsf{Close}_{AS}(A_r, X))$.

Proof. First consider the Move_{AS} operation. Let $\overline{A}$ be the TNFA induced by all states in A and descendants of A in the nested decomposition, i.e., $\overline{A}$ is obtained by recursively "unfolding" the pseudo-states and pseudo-transitions in A, replacing them by the TNFAs they represent. We show by induction that the state-array $X'_A := \mathsf{Move}_{AS}(A, X, \alpha)$ models $\mathsf{Move}(S, \alpha)$ on $\overline{A}$. In particular, plugging in $A = A_r$, we have that $\mathsf{Move}_{AS}(A_r, X, \alpha)$ models $\mathsf{Move}(S, \alpha)$ as required.

Initially, line 1 updates $X[A]$ to be the set of states reachable from a single α-transition in A. If A is a leaf, line 2 is completely bypassed and the result follows immediately. Otherwise, let $A_1, \ldots, A_l$ be the children of A in topological order. Any incoming transition to a state θ_{A_i} or outgoing transition from a state ϕ_{A_i} is an ϵ-transition by Thompson's construction. Hence, no endpoint of an α-transition in A can be shared with any of the children $A_1, \ldots, A_l$. It follows that after line 1 the updated $X[A]$ is the desired state-set, except for the shared states, which have not been handled yet. By induction, the recursive calls in line 2(a) handle the children. Among the shared states only the accepting ones, $\phi_{A_1}, \ldots, \phi_{A_l}$, may be the endpoint of an α-transition and therefore line 2(b) computes the correct state-set.

The Close_{AS} operation proceeds in a similar, though slightly more complicated fashion. Let $\widetilde{X}_A$ be the state-array modeling the set of states reachable via a path of *forward* ϵ-transitions in $\overline{A}$, and let $\widehat{X}_A$ be the state array modelling $\mathsf{Close}(S)$ in $\overline{A}$. We show by induction that if $X''_A := \mathsf{Close}_{AS}(A, X)$ then

$$\widetilde{X}_A \subseteq X''_A \subseteq \widehat{X}_A,$$

where the inclusion refers to the underlying state-sets modeled by the state-set arrays. Initially, line 1 updates $X[A] := \mathsf{Close}_A(X[A])$. If A is a leaf then clearly $X''_A = \widehat{X}_A$. Otherwise, let $A_1, \ldots, A_l$ be the children of A in topological order. Line 2 recursively update the children and propagate the start and accepting states in (a) and (c). Following each recursive call we again update $X[A] := \mathsf{Close}_A(X[A])$ in (d). No state is included in X''_A if there is no ϵ-path in A or through any child of A. Furthermore, since the children are processed in topological order it is straightforward to verify that the sequence of updates in line 2 ensure that X''_A contain all states reachable via a path of forward ϵ-transitions in A or through a child of A. Hence, by induction we have $\widetilde{X}_A \subseteq X''_A \subseteq \widehat{X}_A$ as desired.

A similar induction shows that the state-set array $\mathsf{Close}_{AS}(A_r, X'')$ models the set of states reachable from X'' using a path consisting of forward ϵ-transitions and at most 1 back transition. However, by Lemma 1 this is exactly the set of states reachable by a path of ϵ-transitions. Hence, $\mathsf{Close}_{AS}(A_r, X'')$ models $\mathsf{Close}(S)$ and the result follows. $\square$

By Lemma 4 the state-set simulation can be done using the Close_{AS} and Move_{AS} operations and the complexity now directly depends on the complexities of the simulation data structure. Putting it all together the following reduction easily follows:

Lemma 5 *Let R be a regular expression of length m over alphabet Σ and let Q a string of length n. Given a simulation data structure for TNFAs with $x < m$ states over alphabet $\Sigma \cup \{\beta\}$, where $\beta \notin \Sigma$, that supports all operations in $O(t(x))$ time, using $O(s(x))$ space, and $O(p(x))$ preprocessing time,* Regular Expression Matching *for R and Q can be solved in $O(\frac{nm \cdot t(x)}{x} + \frac{m \cdot p(x)}{x})$ time using $O(\frac{m \cdot s(x)}{x})$ space.*

Proof. Given R first compute a nested decomposition AS of $N(R)$ using Lemma 3 for parameter x. For each TNFA $A \in AS$ sort A's children to topologically and keep pointers to start and accepting states. By Lemma 3 and since topological sort can be done in $O(m)$ time this step uses $O(m)$ time. The total space to represent the decomposition is $O(m)$. Each $A \in AS$ is a TNFA over the alphabet $\Sigma \cup \{\beta\}$ with at most x states and $|AS| = O(\frac{m}{x})$. Hence, constructing simulation data structures for all $A \in AS$ uses $O(\frac{mp(x)}{x})$ time and $O(\frac{ms(x)}{x})$ space. With the above algorithm the state-set simulation of $N(R)$ can now be done in $O(\frac{nm \cdot t(x)}{x})$ time, yielding the desired complexity. $\square$

The idea of decomposing TNFAs is also present in Myers' paper [8], though he does not give a "black-box" reduction as in Lemma 5. We believe that the framework provided by Lemma 5 helps to simplify the presentation of the algorithms significantly. We can restate Myers' result in our setting as the existence of a simulation data structure with $O(1)$ query time that uses $O(x \cdot 2^x)$ space and preprocessing time. For $x \leq \log(n/\log n)$ this achieves the result mentioned in the introduction. The key idea is to encode and tabulate the results of all queries (such an approach is frequently referred to as the "Four Russian Technique" [2]). Bille and Farach [4] give a more space-efficient encoding that does not use Lemma 5 as above. Instead they show how to encode *all possible* simulation data structures in total $O(2^x + m)$ time and space while maintaining $O(1)$ query time.

In the following sections we show how to efficiently avoid the large tables needed in the previous approaches. Instead we implement the operations of simulation data structures using the word-level parallelism of the machine model.

4 A Simple Algorithm

In this section we present a simple simulation data structure for TNFAs, and develop some of the ideas for the improved result of the next section. Let A be a TNFA with $m = O(\sqrt{w})$ states. We will show how to support all operations in $O(1)$ time using $O(m)$ space and $O(m^2)$ preprocessing time.

To build our simulation data structure for A, first sort all states in A in topological order ignoring the back transitions. We require that the endpoints of an α-transition are consecutive in this order. This is automatically guaranteed using a standard $O(m)$ time algorithm for topological sorting (see e.g. [5]). We will refer to states in A by their rank in this order. A state-set of A is represented using a bitstring $S = s_1 s_2 \dots s_m$ defined such that $s_i = 1$ iff node i is in the state-set. The simulation data structure consists of the following bitstrings:

- For each $\alpha \in \Sigma$, a string $D_\alpha = d_1 \dots d_m$ such that $d_i = 1$ iff i is an α-state.
- A string $E = 0e_{1,1}e_{1,2} \dots e_{1,m}0e_{2,1}e_{2,2} \dots e_{2,m}0 \dots 0e_{m,1}e_{m,2} \dots e_{m,m}$, where $e_{i,j} = 1$ iff i is ϵ-reachable from j. The zeros are *test bits* needed for the algorithm.
- Three constants $I = (10^m)^m$, $X = 1(0^m1)^{m-1}$, and $C = 1(0^{m-1}1)^{m-1}$. Note that I has a 1 in each test bit position[2].

The strings E, I, X, and C are easily computed in $O(m^2)$ time and use $O(m^2)$ bits. Since $m = O(\sqrt{w})$ only $O(1)$ space is needed to store these strings. We store D_α in a hashtable indexed by α. Since the total number of different characters in A can be at most m, the hashtable contains at most m entries. Using perfect hashing D_α can be represented in $O(m)$ space with $O(1)$ worst-case lookup time. The preprocessing time is expected $O(m)$ w.h.p.. To get a worst-case bound we use the deterministic dictionary of Hagerup et. al. [6] with $O(m \log m)$ worst-case preprocessing time. In total the data structure requires $O(m)$ space and $O(m^2)$ preprocessing time.

Next we show how to support each of the operations on A. Suppose $S = s_1 \dots s_m$ is a bitstring representing a state-set of A and $\alpha \in \Sigma$. The result of $\mathsf{Move}_A(S, \alpha)$ is given by

$$S' := (S >> 1) \,\&\, D_\alpha.$$

This should be understood as C notation, where the right-shift is unsigned. Readers familiar with the Shift-Or algorithm [3] will notice the similarity. To see the correctness, observe that state i is put in S' iff state $(i-1)$ is in S and the ith state is an α-state. Since the endpoints of α-transitions are consecutive in the topological order it follows that S' is correct. Here, state $(i-1)$ can only influence state i, and this makes the operation easy to implement in parallel. However, this is not the case for Close_A. Here, any state can potentially affect a large number of states reachable through long ϵ-paths. To deal with this we use the following steps.

$$\begin{aligned} Y &:= (S \times X) \,\&\, E \\ Z &:= ((Y \mid I) - (I >> m)) \,\&\, I \\ S' &:= ((Z \times C) << w - m(m+1)) >> w - m \end{aligned}$$

We describe in detail why this, at first glance somewhat cryptic sequence, correctly computes S' as the result of $\mathsf{Close}_A(S)$. The variables Y and Z are simply temporary variables inserted to increase the readability of

[2]We use exponentiation to denote repetition, i.e., $1^3 0 = 1110$.

the computation. Let $S = s_1 \dots s_m$. Initially, $S \times X$ concatenates m copies of S with a zero bit between each copy, that is,

$$S \times X = s_1 \dots s_m \times 1(0^m1)^{m-1} = (0s_1 \dots s_m)^m.$$

The bitwise & with E gives

$$Y = 0y_{1,1}y_{1,2} \dots y_{1,m}0y_{2,1}y_{2,2} \dots y_{2,m}0 \dots 0y_{m,1}y_{m,2} \dots y_{m,m},$$

where $y_{i,j} = 1$ iff state j is in S and state i is ϵ-reachable from j. In other words, the substring $Y_i = y_{i,1} \dots y_{i,m}$ indicates the set of states in S that have a path of ϵ-transitions to i. Hence, state i should be included in $\mathsf{Close}_A(S)$ precisely if at least one of the bits in Y_i is 1. This is determined next. First $(Y \,|\, I) - (I >> m)$ sets all test bits to 1 and subtracts the test bits shifted right by m positions. This ensures that if all positions in Y_i are 0, the ith test bit in the result is 0 and otherwise 1. The test bits are then extracted with a bitwise & with I, producing the string $Z = z_1 0^m z_2 0^m \dots z_m 0^m$. This is almost what we want since $z_i = 1$ iff state i is in $\mathsf{Close}_A(S)$. The final computation *compresses* the Z into the desired format. The multiplication produces the following length $2m^2$ string:

$$\begin{aligned} Z \times C &= z_1 0^m z_2 0^m \dots z_m 0^m \times 1(0^{m-1}1)^{m-1} \\ &= z_1 0^{m-1} z_1 z_2 0^{m-2} \cdots z_1 \dots z_k 0^{m-k} \cdots z_1 \dots z_{m-1} 0 z_1 \dots z_m 0 z_2 \dots z_m \cdots 0^k z_{k+1} \dots z_m \cdots 0^{m-1} z_m 0^m \end{aligned}$$

In particular, positions $m(m-1)+1$ through m^2 (from the left) contain the test bits compressed into a string of length m. The two shifts zeroes all other bits and moves this substring to the rightmost position in the word, producing the final result. Since $m = O(\sqrt{w})$ all of the above operations can be done in constant time.

Finally, observe that Insert_A and Member_A are trivially implemented in constant time. Thus,

Lemma 6 *For any TNFA with $m = O(\sqrt{w})$ states there is a simulation data structure using $O(m)$ space and $O(m^2)$ preprocessing time which supports all operations in $O(1)$ time.*

The main bottleneck in the above data structure is the string E that represents all ϵ-paths. On a TNFA with m states E requires at least m^2 bits and hence this approach only works for $m = O(\sqrt{w})$. In the next section we show how to use the structure of TNFAs to do better.

5 Overcoming the ϵ-closure Bottleneck

In this section we show how to compute an ϵ-closure on a TNFA with $m = O(w)$ states in $O(\log m)$ time. Compared with the result of the previous section we quadratically increase the size of the TNFA at the expense of using logarithmic time. The algorithm is easily extended to an efficient simulation data structure. The key idea is a new hierarchical decomposition of TNFAs described below.

5.1 Partial-TNFAs and Separator Trees

First we need some definitions. Let A be a TNFA with parse tree T. Each node v in T uniquely corresponds to two states in A, namely, the start and accepting states $\theta_{A'}$ and $\phi_{A'}$ of the TNFA A' with the parse tree consisting of v and all descendants of v. We say v *associates* the states $S(v) = \{\theta_{A'}, \phi_{A'}\}$. In general, if C is a cluster of T, i.e., any connected subgraph of T, we say C associates the *set* of states $S(C) = \cup_{v \in C} S(v)$. We define the *partial-TNFA* (pTNFA) for C, as the directed, labeled subgraph of A induced by the set of states $S(C)$. In particular, A is a pTNFA since it is induced by $S(T)$. The two states associated by the root node of C are defined to be the start and accepting state of the corresponding pTNFA. We need the following result.

Lemma 7 *For any pTNFA P with $m > 2$ states there exists a partitioning of P into two subgraphs P_O and P_I such that*

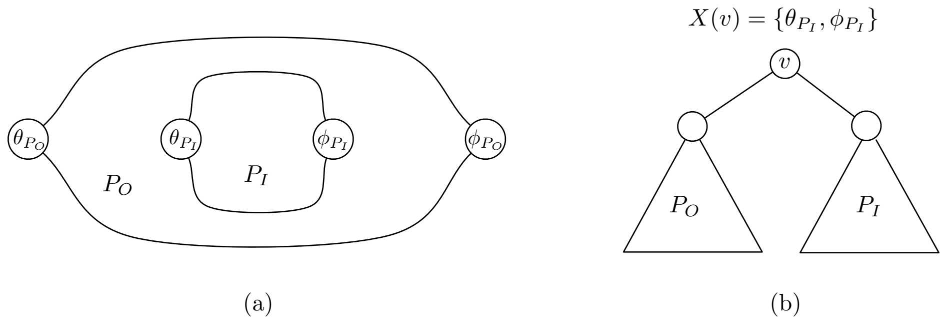

Figure 3: (a) Inner and outer pTNFAs. (b) The corresponding separator tree construction.

(i) P_O *and* P_I *are pTNFAs with at most* $2/3m + 2$ *states each,*

(ii) any transition from P_O *to* P_I *ends in* θ_{P_I} *and any transition from* P_I *to* P_O *starts in* ϕ_{P_I}*, and*

(iii) the partitioning can be computed in $O(m)$ *time.*

Proof. Let P be pTNFA with $m > 2$ states and let C be the corresponding cluster with t nodes. Since C is a binary tree with more than 1 node, Jordan's classical result [7] establishes that we can find in $O(t)$ time an edge e in C whose removal splits C into two clusters each with at most $2/3t + 1$ nodes. These two clusters correspond to two pTNFAs, P_O and P_I, and since $m = 2t$ each of these have at most $2/3m + 2$ states. Hence, (i) and (iii) follows. For (ii) assume w.l.o.g. that P_O is the pTNFA containing the start and accepting state of P, i.e., $\theta_{P_O} = \theta_P$ and $\phi_{P_O} = \phi_P$. Then, P_O is the pTNFA obtained from P by removing all states of P_I. From Thompson's construction it is easy to check that any transition from P_O to P_I ends in θ_{P_I} and any transition from P_I to P_O must start in ϕ_{P_I}. $\square$

Intuitively, if we draw P, P_I is "surrounded" by P_O, and therefore we will often refer to P_I and P_O as the *inner pTNFA* and the *outer pTNFA*, respectively (see Fig. 3(a)). Applying Lemma 7 recursively gives the following essential data structure. Let P be a pTNFA with m states. The *separator tree* for P is a binary, rooted tree B defined as follows: If $m = 2$, i.e., P is a trivial pTNFA consisting of two states θ_P and ϕ_P, then B is a single leaf node v that stores the set $X(v) = \{\theta_P, \phi_P\}$. Otherwise ($m > 2$), compute P_O and P_I according to Lemma 7. The root v of B stores the set $X(v) = \{\theta_{P_I}, \phi_{P_I}\}$, and the children of v are roots of separator trees for P_O and P_I, respectively (see Fig. 3(b)).

With the above construction each node in the separator tree naturally correspond to a pTNFA, e.g., the root corresponds to P, the children to P_I and P_O, and so on. We denote the pTNFA corresponding to node v in B by $P(v)$. A simple induction combined with Lemma 7(i) shows that if v is a node of depth k then $P(v)$ contains at most $(\frac{2}{3})^k m + 6$ states. Hence, the depth of B is at most $d = \log_{3/2} m + O(1)$. By Lemma 7(iii) each level of B can be computed in $O(m)$ time and thus B can be computed in $O(m \log m)$ total time.

5.2 A Recursive ϵ-Closure Algorithm

We now present a simple ϵ-closure algorithm for a pTNFA, which recursively traverses the separator tree B. We first give the high level idea and then show how it can be implemented in $O(1)$ time for each level of B. Since the depth of B is $O(\log m)$ this leads to the desired result. For a pTNFA P with m states, a separator tree B for P, and a node v in B define

```
Close_{P(v)}(S):  1. Compute the set $Z \subseteq X(v)$ of states in $X(v)$ that are $\epsilon$-reachable from $S$ in $P(v)$.
                  2. If $v$ is a leaf return $S' := Z$, else let $u$ and $w$ be the children of $v$, respectively:
                     (a) Compute the set $G \subseteq V(P(v))$ of states in $P(v)$ that are $\epsilon$-reachable from $Z$.
```

(b) Return $S' := \mathsf{Close}_{P(u)}((S \cup G) \cap V(P(u))) \cup \mathsf{Close}_{P(w)}((S \cup G) \cap V(P(w)))$.

Lemma 8 *For any node v in the separator tree of a pTNFA P, $\mathsf{Close}_{P(v)}(S)$ computes the set of states in $P(v)$ reachable via a path of ϵ-transitions.*

Proof. Let $\widehat{S}$ be the set of states in $P(v)$ reachable via a path of ϵ-transitions. We need to show that $\widehat{S} = S'$. It is easy to check that any state in S' is reachable via a path of ϵ-transitions and hence $S' \subseteq \widehat{S}$. We show the other direction by induction on the separator tree. If v is leaf then the set of states in $P(v)$ is exactly $X(v)$. Since $S' = Z$ the claim follows. Otherwise, let u and w be the children of v, and assume w.l.o.g. that $X(v) = \{\theta_{P(u)}, \phi_{P(u)}\}$. Consider a path p of ϵ-transitions from state s to state s'. There are two cases to consider:

Case 1: $s' \in V(P(u))$. If p consists entirely of states in $P(u)$ then by induction it follows that $s' \in \mathsf{Close}_{P(u)}(S \cap V(P(u)))$. Otherwise, p contain a state from $P(w)$. However, by Lemma 7(ii) $\theta_{P(u)}$ is on p and hence $\theta_{P(u)} \in Z$. It follows that $s' \in G$ and therefore $s' \in \mathsf{Close}_{P(u)}(G \cap V(P(u)))$.

Case 2: $s' \in V(P(w))$. As above, with the exception that $\phi_{P(u)}$ is now the state in Z.

In all cases $s' \in S'$ and the result follows. □

5.3 Implementing the Algorithm

Next we show how to efficiently implement the above algorithm in parallel. The key ingredient is a compact mapping of states into positions in bitstrings. Suppose B is the separator tree of depth d for a pTNFA P with m states. The *separator mapping* M maps the states of P into an interval of integers $[1, l]$, where $l = 3 \cdot 2^d$. The mapping is defined recursively according to the separator tree. Let v be the root of B. If v is a leaf node the interval is $[1, 3]$. The two states of P, θ_P and ϕ_P, are mapped to positions 2 and 3, respectively, while position 1 is left intentionally unmapped. Otherwise, let u and w be the children of v. Recursively, map $P(u)$ to the interval $[1, l/2]$ and $P(w)$ to the interval $[l/2 + 1, l]$. Since the separator tree contains at most 2^d leaves and each contribute 3 positions the mapping is well-defined. The size of the interval for P is $l = 3 \cdot 2^{\log_{3/2} m + O(1)} = O(m)$. We will use the unmapped positions as test bits in our algorithm.

The separator mapping compactly maps all pTNFAs represented in B into small intervals. Specifically, if v is a node at depth k in B, then $P(v)$ is mapped to an interval of size $l/2^k$ of the form $[(i-1) \cdot \frac{l}{2^k} + 1, i \cdot \frac{l}{2^k}]$, for some $1 \leq i \leq 2^k$. The intervals that correspond to a pTNFA $P(v)$ are *mapped* and all other intervals are *unmapped*. We will refer to a state s of P by its mapped position $M(s)$. A state-set of P is represented by a bitstring S such that, for all mapped positions i, $S[i] = 1$ iff the i is in the state-set. Since $m = O(w)$, state-sets are represented in a constant number of words.

To implement the algorithm we define a simple data structure consisting of four length l bitstrings X_k^θ, X_k^ϕ, E_k^θ, and E_k^ϕ for each level k of the separator tree. For notational convenience, we will consider the strings at level k as two-dimensional arrays consisting of 2^k intervals of length $l/2^k$, i.e., $X_k^\theta[i, j]$ is position j in the ith interval of X_k^θ. If the ith interval at level k is unmapped then all positions in this interval are 0 in all four strings. Otherwise, suppose that the interval corresponds to a pTNFA $P(v)$ and let $X(v) = \{\theta_v, \phi_v\}$. The strings are defined as follows:

$$X_k^\theta[i,j] = 1 \text{ iff } \theta_v \text{ is } \epsilon\text{-reachable in } P(v) \text{ from state } j,$$
$$E_k^\theta[i,j] = 1 \text{ iff state } j \text{ is } \epsilon\text{-reachable in } P(v) \text{ from } \theta_v,$$
$$X_k^\phi[i,j] = 1 \text{ iff } \phi_v \text{ is } \epsilon\text{-reachable in } P(v) \text{ from state } j,$$
$$E_k^\phi[i,j] = 1 \text{ iff state } j \text{ is } \epsilon\text{-reachable in } P(v) \text{ from } \phi_v.$$

In addtion to these, we also store a string I_k containing a test bit for each interval, that is, $I_k[i, j] = 1$ iff $j = 1$. Since the depth of B is $O(\log m)$ the strings use $O(\log m)$ words. With a simple depth-first search they can all be computed in $O(m \log m)$ time.

Let S be a bitstring representing a state-set of A. We implement the operation $\mathsf{Close}_A(S)$ by computing a sequence of intermediate strings $S_0, \ldots, S_d$ each corresponding to a level in the above recursive algorithm. Initially, $S_0 := S$ and the final string S_d is the result of $\mathsf{Close}_A(S)$. At level k, $0 \leq k < d$, we compute S_{k+1} from S_k as follows. Let $t = l/2^k - 1$.

$$\begin{aligned}
Y^\theta &:= S_k \;\&\; X_k^\theta \\
Z^\theta &:= ((Y^\theta \mid I_k) - (I_k >> t)) \;\&\; I_k \\
F^\theta &:= Z^\theta - (Z^\theta >> t) \\
G^\theta &:= F^\theta \;\&\; E_k^\theta \\
Y^\phi &:= S_k \;\&\; X_k^\phi \\
Z^\phi &:= ((Y^\phi \mid I_k) - (I_k >> t)) \;\&\; I_k \\
F^\phi &:= Z^\phi - (Z^\phi >> t) \\
G^\phi &:= F^\phi \;\&\; E_k^\phi \\
S_{k+1} &:= S_k \mid G^\theta \mid G^\phi
\end{aligned}$$

We argue that the computation correctly simulates (in parallel) a level of the recursive algorithm. Assume that at the beginning of level k the string S_k represents the state-set corresponding the recursive algorithm after k levels. We interpret S_k as divided into $r = l/2^k$ intervals of length $t = l/2^k - 1$, each prefixed with a test bit, i.e.,

$$S_k = 0s_{1,1}s_{1,2}\ldots s_{1,t}0s_{2,1}s_{2,2}\ldots s_{2,t}0\ldots 0s_{r,1}s_{r,2}\ldots s_{r,t}$$

Assume first that all these intervals are mapped intervals corresponding to pTNFAs $P(v_1), \ldots, P(v_r)$, and let $X(v_i) = \{\theta_{v_i}, \phi_{v_i}\}$, $1 \leq i \leq r$. Initially, $S_k \;\&\; X_k^\theta$ produces the string

$$Y^\theta = 0y_{1,1}y_{1,2}\ldots y_{1,t}0y_{2,1}y_{2,2}\ldots y_{2,t}0\ldots 0y_{r,1}y_{r,2}\ldots y_{r,t},$$

where $y_{i,j} = 1$ iff θ_{v_i} is ϵ-reachable in $P(v_i)$ from state j and j is in S_k. Then, similar to the second line in the simple algorithm, $(Y^\theta \mid I_k) - (I_k >> t) \;\&\; I_k$ produces a string of test bits $Z^\theta = z_1 0^t z_2 0^t \ldots z_r 0^t$, where $z_i = 1$ iff at least one of $y_{i,1} \ldots y_{i,t}$ is 1. In other words, $z_i = 1$ iff θ_{v_i} is ϵ-reachable in $P(v_i)$ from any state in $S_k \cap V(P(v_i))$. Intuitively, the Z^θ corresponds to the "θ-part" of the of Z-set in the recursive algorithm. Next we "copy" the test bits to get the string $F^\theta = Z^\theta - (Z^\theta >> t) = 0z_1^t 0z_2^t \ldots 0z_r^t$. The bitwise & with E_k^θ gives

$$G^\theta = 0g_{1,1}g_{1,2}\ldots g_{1,t}0g_{2,1}g_{2,2}\ldots g_{2,t}0\ldots 0g_{r,1}g_{r,2}\ldots g_{r,t}.$$

By definition, $g_{i,j} = 1$ iff state j is ϵ-reachable in $P(v_i)$ from θ_{v_i} and $z_i = 1$. In other words, G^θ represents, for $1 \leq i \leq r$, the states in $P(v_i)$ that are ϵ-reachable from $S_k \cap V(P(v_i))$ through θ_{v_i}. Again, notice the correspondance with the G-set in the recursive algorithm. The next 4 lines are identical to first 4 with the exception that θ is exchanged by ϕ. Hence, G^ϕ represents the states that ϵ-reachable through $\phi_{v_1}, \ldots, \phi_{v_r}$.

Finally, $S_k \mid G^\theta \mid G^\phi$ computes the union of the states in S_k, G^θ, and G^ϕ producing the desired state-set S_{k+1} for the next level of the recursion. In the above, we assumed that all intervals were mapped. If this is not the case it is easy to check that the algorithm is still correct since the string in our data structure contain 0s in all unmapped intervals. The algorithm uses constant time for each of the $d = O(\log m)$ levels and hence the total time is $O(\log m)$.

5.4 The Simulation Data Structure

Next we show how to get a full simulation data structure. First, note that in the separator mapping the endpoints of the α-transitions are consecutive (as in Sec. 4). It follows that we can use the same algorithm as in the previous section to compute Move_A in $O(1)$ time. This requires a dictionary of bitstrings, D_α, using additional $O(m)$ space and $O(m \log m)$ preprocessing time. The Insert_A, and Member_A operations are trivially implemented in $O(1)$. Putting it all together we have:

Lemma 9 *For a TNFA with $m = O(w)$ states there is a simulation data structure using $O(m)$ space and $O(m \log m)$ preprocessing time which supports all operations in $O(\log m)$ time.*

Combining the simulation data structures from Lemmas 6 and 9 with the reduction from Lemma 5 and taking the best result gives Theorem 1. Note that the simple simulation data structure is the fastest when $m = O(\sqrt{w})$ and n is sufficiently large compared to m.

6 Remarks and Open Problems

The presented algorithms assume a unit-cost multiplication operation. Since this operation is not in AC^0 (the class of circuits of polynomial size (in w), constant depth, and unbounded fan-in) it is interesting to reconsider what happens with our results if we remove multiplication from our machine model. The simulation data structure from Sec. 4 uses multiplication to compute Close_A and also for the constant time hashing to access D_α. On the other hand, the algorithm of Sec. 5 only uses multiplication for the hashing. However, Lemma 9 still holds since we can simply replace the hashing by binary search tree, which uses $O(\log m)$ time. It follows that Theorem 1 still holds except for the $O(n + m^2)$ bound in the last line.

Another interestring point is to compare our results with the classical Shift-Or algorithm by Baeza-Yates and Gonnet [3] for exact pattern matching. Like ours, their algorithm simulates a NFA with m states using word-level parallelism. The structure of this NFA permits a very efficient simulation with an $O(w)$ speedup of the simple $O(nm)$ time simulation. Our results generalize this to regular expressions with a slightly worse speedup of $O(w/\log w)$. We wonder if it is possible to remove the $O(\log w)$ factor separating these bounds.

From a practical viewpoint, the simple algorithm of Sec. 4 seems very promising since only about 15 instructions are needed to carry out a step in the state-set simulation. Combined with ideas from [10] we believe that this could lead to a practical improvement over previous algorithms.

7 Acknowledgments

The author wishes to thank Rasmus Pagh and Inge Li Gørtz for many comments and interesting discussions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph (in russian). english translation in soviet math. dokl. 11, 1209-1210, 1975. *Dokl. Acad. Nauk.*, 194:487–488, 1970.

[3] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[4] P. Bille and M. Farach-Colton. Fast and compact regular expression matching, 2005. Submitted to a journal. Preprint availiable at `arxiv.org/cs/0509069`.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, second edition.* MIT Press, 2001.

[6] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. of Algorithms*, 41(1):69–85, 2001.

[7] C. Jordan. Sur les assemblages des lignes. *J. Reine Angew. Math.*, 70:185–190, 1869.

[8] E. W. Myers. A four-russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.

[9] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

[10] G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, November 2004.

[11] K. Thompson. Regular expression search algorithm. *Comm. of the ACM*, 11:419–422, 1968.