Faster Regular Expression Matching

Philip Bille[1]* and Mikkel Thorup[2]

[1] Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark.
phbi@imm.dtu.dk
[2] AT&T Labs—Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932, USA. mthorup@research.att.com

Abstract. Regular expression matching is a key task (and often the computational bottleneck) in a variety of widely used software tools and applications, for instance, the `unix` `grep` and `sed` commands, scripting languages such as `awk` and `perl`, programs for analyzing massive data streams, etc. We show how to solve this ubiquitous task in linear space and $O(nm(\log\log n)/(\log n)^{3/2}+n+m)$ time where m is the length of the expression and n the length of the string. This is the first improvement for the dominant $O(nm/\log n)$ term in Myers' $O(nm/\log n+(n+m)\log n)$ bound [JACM 1992]. We also get improved bounds for external memory.

1 Introduction

Problem Regular expression matching is performed commonly as a primitive by many of today's computer systems, and has been so for almost half a century. With `unix/linux`, or on a Mac, we match regular expressions in large file systems using the command line utility `grep`. With the stream editor `sed` we further specify a replacement of the occurrences of the regular expression. A more integrated use of regular expression matching is found in the general text processing language `perl` [25] which is commonly used to convert between input/output formats. One of the basic features in `perl` is to match regular expressions and substitute some of the subexpressions. The regular expression matching is often the hard part whereas the subsequent substitution is easy, so if we could improve regular expression matching, we would improve a lot of data processing.

Historically, regular expression matching goes back to Kleene in the 1950s [14]. It became popular for practical use in text editors in the 1960s [24]. Compilers use regular expression matching for separating tokens in the lexical analysis phase [2]. New and interesting applications continue to appear in diverse areas such as XML querying [15, 18], protein searching [21], and Internet traffic analysis [12, 27].

Computational model We study the complexity of regular expression matching on the RAM model with standard word operations. This means that our

* Supported by the Danish Agency for Science, Technology, and Innovation. Work done while the author was at the IT University of Copenhagen.

algorithms can be implemented directly in standard imperative programming languages such as C [13] or C++ [23]. For the last thirty years, these programming languages have been commonly used to write efficient and portable code. Even if we code the main program in some higher level programming language, it is normally possible to invoke subroutines written in C for parts that have to run efficiently, and computational efficiency is what we study here. Most open source implementations of algorithms for regular expression, e.g., `grep`, `sed` and `perl`, are written in C. In practice, it would be impressive to gain a factor 2 in speed for this well-studied problem, but as theoreticians we will focus on getting the best asymptotic worst-case running time in terms of the problem size $n \to \infty$.

Current Bounds Let n and m be the lengths of the string Q and the regular expression R, respectively. Typically we assume $n > m$. The classical textbook solution to the problem by Thompson [24] from 1968 takes $O(nm)$ time. It uses a standard state-set simulation of a non-deterministic finite automaton (NFA) with $O(m)$ states produced from R. Using the NFA, we scan Q. Each of the n characters is processed in $O(m)$ time by progression of the NFA state set.

In 1985 Galil [9] asked if a faster algorithm could be derived. Myers [19] met this challenge in 1992 with an $O(nm/\log n + (n+m)\log n)$ time solution, thus improving the time complexity of Thompson algorithm by a $\log n$ factor for most values of n and m. Since then there has been several works mostly addressing issues of space and larger word length [4,5,22]. However, with no special assumptions on the word length the $O(nm/\log n)$ term from Myers' algorithms has not been touched. The fundamental reason is that all the previous approaches aim to speed-up the $O(m)$ time the NFA needs to process one character from the string. To do that we need at least $\Omega(m/w)$ time just to read or write the state set of the NFA. In fact, Bille [4] obtained this bound within a $\log w$ factor. However, we may have $w = O(\log n)$, so in terms of m and n, there is no hope of bypassing Myers' logarithmic improvement when working on one character at the time.

New Bounds In this paper we present a linear space regular expression matching algorithm with a running time of

$$O\left(\frac{nm\log\log n}{\log^{3/2} n} + n + m\right).$$

If the alphabet is bounded or if $m \leq n^{1-\varepsilon}$ for some positive constant ε, we can avoid the $\log\log n$ factor, getting a cleaner time bound of $O(nm/(\log n)^{3/2} + n + m)$. In any case, assuming $m, n \geq \log^{3/2} n$, this increases the improvement over Thompson's $O(nm)$ algorithm from the $\log n$ factor of Myers to almost a factor $(\log n)^{3/2}$.

We bypass the logarithmic limitation of previous approaches with a speed-up for the NFA processing of multiple characters at the time. This is, in itself, an obvious idea, but challenging to realize due to the complex interaction between the NFA and the input string (c.f. discussion at the end of Sec. 3)

Tabulation Our improved time bound is achieved via a better tabulation technique. The idea of using tables to improve calculations is old. As a nice early example, in 1792, Gaspard de Prony started preparing nineteen volumes of trigonometric and logarithm tables for the revolutionary French government. With the assistance of a small group of mathematicians, Prony divided the computations into a series of additions and subtractions. He then hired about eighty (human) computers to do the arithmetic.

Tabulation is also used on today's computers for the fastest regular expression matching. In particular, the implementation of Myers' algorithm [19], the `agrep` tool [26] and the `nrgrep` tool [20] are all based on tabulations of NFAs, and they outperform other tools. The basic point is to use table look-ups to replace a complicated NFA simulation that would look up transitions from multiple states.

Tabulation is a speed-up technique that only makes sense after we have settled on the basic combinatorial algorithm, in this case Thompson's algorithm which has stood unbeaten since 1968. In fact, this might very well be the asymptotically fastest worst-case efficient algorithm for regular expression matching if we ignore polylogarithmic factors. The goal in tabulation is now to compactly represent as complex subproblems as possible. Our contribution is to show that we with x bits can represent subproblems requiring $\Theta(x^{3/2}/\log x)$ steps in Thompson's algorithm. The limit of all previous tabulation approaches to this problem was to represent x steps with $\Theta(x)$ bits. We are breaking the linear bound by moving into a higher dimension of encoding, representing both part of the NFA and part of the input string in the x bits. Higher dimensional encodings are known for several other problems [3,6,16] but combining the NFA and string dimensions has been a challenge for regular expression matching.

Our encoding allows us to construct fixed universal tables that can later be used to solve arbitrary regular expression problems. The tables are constructed once and for all in $O(2^x)$ time and space. Using them we can match an expression of size m in a text of size n in $O(nm(\log x)/x^{3/2} + n + m)$ time, and this works well even in a streaming context. For the previous mentioned bounds, we could set $x = (\log_2 n)/2$ thus using $O(\sqrt{n})$ time and space on the tables. However, the same tables can be used to solve many regular expression matching problems. Therefore we may chose a larger x to create some large and powerful tables once and for all. These could be used when running `perl` where we often need to match many different regular expressions in a text.

External memory Our coding technique gives corresponding speed-ups for external memory with block size B and internal memory size $M \geq B$. Of course M and B could also represent smaller units in the memory hierarchy, e.g., cache of size M registers and cache line of length B. If $m = o(M)$, it is trivial to solve the regular expression matching problem with $O(n/B)$ I/O operations. Assuming $m = \Omega(M)$ we solve the problem with $O(nm/(\sqrt{M}B))$ I/O operations. This is a factor $\sqrt{M}$ better than what would be possible with previous algorithms regardless of the block size. Technically speaking, we use the internal memory to pack subproblems of complexity $M^{3/2}$ in Thompson's algorithm. Each subproblem is

read with M/B I/O operations, so the saving is a factor $M^{3/2}/(M/B) = \sqrt{M}B$. Previous solutions could only pack problems of complexity linear in M, so they could only save a factor $M/(M/B) = B$. In particular, our algorithm is the first to gain substantially in I/O operations from a large internal memory even if the blocks are small.

Summing up The theorem below formally states our results in terms of a resource parameter t capturing how much time and space we are willing to spend on the global tables.

Theorem 1. *On a unit-cost RAM with word length w and standard instruction set, for any parameter $t < 2^w$, we can do a general preprocessing for regular expression matching using $O(t)$ time and space. Subsequently, given any regular expression of length m and string of length n, we can perform the matching in*

$$O\left(\frac{nm \log\log t}{(\log t)^{3/2}} + n + m\right)$$

time using $O(t + m)$ space. Our matching only makes a single pass through the string, which may hence be presented as a stream.

Cast in terms of external memory with block size B and internal memory size M, $2B \leq M = O(m)$, we solve the regular expression matching problem with $O(nm/(\sqrt{M}B) + m/B \cdot \log_{M/B}(m/B))$ I/O operations using $O(m)$ space. Again we only perform a single pass over the string. After an $O(m/B \cdot \log_{M/B}(m/B))$ preprocessing of the pattern, we process string segments of length $\Theta(\sqrt{M})$ using $O(m/B)$ I/O operations.

As mentioned above, in many practical applications it makes sense once and for all to do the preprocessing with a fairly large t, fitting within the bounds of fast memory, and subsequently be able to solve lots of smaller regular expression matching problems quickly.

In connection with very large data, note how the unit-RAM result and the external memory result complement each other. In both cases, the string may be a stream passed only once, which is perfect. In the normal case where $m = o(M)$, we pick a large t such that the $O(t+m)$ space fits in internal memory, and use the unit-RAM algorithm to process the stream in $O(n/B)$ I/O operations. However, in the extreme event that $m = \Omega(M)$, we apply our external memory algorithm.

Overview In this extended abstract, we only have room to present our algorithm for bounded size alphabets on the unit-cost word RAM. In Sec. 2 we define Thompson's standard automaton construction for regular expressions [24] and in Sec. 3 we describe how we decompose this automaton as done in the previous algorithms with logarithmic speedup. After that, we embark on our new attack on the problem. Our decomposition is different from that of Myers [19] because we want to tabulate the action of subautomatons, not just on a single input character, but on substrings of input characters. In Sec. 4 we describe the actual action of subautomatons on substrings and how to tabulate it.

2 Regular Expressions and Finite Automata

First we briefly review the classical concepts used in the paper. For more details see, e.g., Aho et al. [2]. The set of *regular expressions* over Σ are defined recursively as follows: A character $\alpha \in \Sigma$ is a regular expression, and if S and T are regular expressions then so is the *concatenation*, $(S) \cdot (T)$, the *union*, $(S)|(T)$, and the *star*, $(S)^*$. The *language* $L(R)$ generated by R is defined as follows: $L(\alpha) = \{\alpha\}$, $L(S \cdot T) = L(S) \cdot L(T)$, that is, any string formed by the concatenation of a string in $L(S)$ with a string in $L(T)$, $L(S)|L(T) = L(S) \cup L(T)$, and $L(S^*) = \bigcup_{i \geq 0} L(S)^i$, where $L(S)^0 = \{\epsilon\}$ and $L(S)^i = L(S)^{i-1} \cdot L(S)$, for $i > 0$. Here ϵ denotes the empty string. The *parse tree* $T(R)$ for R is the unique rooted binary tree representing the hierarchical structure of R. The leaves of $T(R)$ are labeled by a character from Σ and internal nodes are label by either $\cdot$, $|$, or $*$.

A *finite automaton* is a tuple $A = (V, E, \Sigma, \theta, \phi)$, where V is a set of nodes called *states*, E is a set of directed edges between states called *transitions* each labeled by a character from $\Sigma \cup \{\epsilon\}$, $\theta \in V$ is a *start state*, and $\phi \in V$ is an *accepting state*[3]. In short, A is an edge-labeled directed graph with a special start and accepting node. 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). If dealing with multiple automatons, we use a subscript $_A$ to indicate information associated with automaton A, e.g., θ_A is the start state of automaton A.

Given a string q and a path p in A we say that p and q *match* if the concatenation of the labels on the transitions in p is q. We also say that two paths p and p' match if they match the same string. The set of strings matching some path between states s and s' in A is denoted by $P_A(s, s')$. For state-sets S and S' we define $P_A(S, S') = \bigcup_{s \in S, s' \in S'} P_A(s, s')$. For a subset S of states in A and a string Q, define the *state-set transition*, $\delta_A(S, q)$, as the of states reachable from S through a path matching q. We say that A *accepts* the string q if $q \in P_A(\theta_A, \phi_A)$. Otherwise A *rejects* q. One may use a sequence of state-set transitions for a single character to test acceptance of a string Q of length n as follows. First set $S_0 := \{\theta_A\}$. For $i = 1, \ldots, n$ compute $S_i := \{\delta_A(S_{i-1}, Q[i])\}$. It follows inductively that q is accepted if and only if $\phi_A \in S_n$.

Given a regular expression R, an NFA A accepting precisely the strings in $L(R)$ can be obtained by several classic methods [10, 17, 24]. In particular, Thompson [24] gave the simple well-known construction in Fig. 1. We will call an automaton constructed with these rules a *Thompson NFA* (TNFA). Fig. 2 shows the TNFA for the regular expression $R = \texttt{a}\cdot\texttt{(a}^*\texttt{)}\cdot\texttt{(b|c)}$, along with lots of other information to be discussed in later sections.

A TNFA $N(R)$ for R has at most $2m$ states, at most $4m$ transitions, and can be computed in $O(m)$ time. With a breadth-first search of A we can compute a state-set transition for a single character in $O(m)$ time. Hence, we can

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

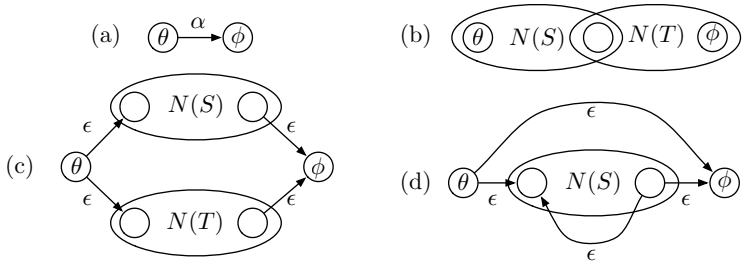


Fig. 1. Thompson's recursive NFA construction. The regular expression for a character $\alpha \in \Sigma$ corresponds to NFA (a). If S and T are regular expressions then $N(ST)$, $N(S|T)$, and $N(S^*)$ correspond to NFAs (b), (c), and (d), respectively. In each of these figures, the leftmost node θ and rightmost node ϕ are the start and the accept nodes, respectively. For the top recursive calls, these are the start and accept nodes of the overall automaton. In the recursions indicated, e.g., for $N(ST)$ in (b), we take the start node of the subautomaton $N(S)$ and identify with the state immediately to the left of $N(S)$ in (b). Similarly the accept node of $N(S)$ is identified with the state immediately to the right of $N(S)$ in (b).

test acceptance of a string Q of length n in $O(nm)$ time. This is Thompson's algorithm [24].

3 1-Dimensional Speed-Up

Myers' algorithm [19] and later variants [4, 5] all aim to speed-up Thompson's algorithm by improving the $O(m)$ bound for a state-set transition for a single character. We describe this approach in some detail below as we are going to reuse much of it our own algorithm. The version we present is for bounded size alphabets, i.e., each character can be coded with a constant number of bits.

For a desired speed-up of $x < w$, we will need some global tables of size $2^{O(x)}$. First we *decompose* $N(R)$ into a tree AS of $O(\lceil m/x \rceil)$ micro TNFAs, each of size at most x. In each $A \in AS$, each child TNFA C is represented by a start and accepting state and a *pseudo-transition* labeled $\beta \notin \Sigma$ connecting these. For example, Fig. 2, shows the TNFAs for the regular expression $R = \mathtt{a{\cdot}(a^*){\cdot}(b|c)}$ divided into 3 TNFAs $AS = \{A_1, A_2, A_3\}$, where $A_1 = N(\mathtt{a}^*)$, $A_2 = N(\mathtt{b|c})$, and $A_3 = N((\mathtt{a} \cdot \beta \cdot \beta)^*)$. Here the βs represent A_1 and A_2, respectively. We can always construct a decomposition of $N(R)$ as described above since we can partition the parse tree into subtrees using standard techniques and build the decomposition from the TNFAs induced by the subtrees, see e.g., Myers [19] for details.

The point in such a micro TNFA A is that we can code it uniquely via its parse tree using $O(x)$ bits. We shall use "A" to denote the bit coding of A. If $x = o(\log m)$, there will be many micro TNFAs with the same bit code. Technically A is an index to all information associated with A including both the code "A" and the children and parents of A. Since A has only x states, we

can also code its local state set S_A using x bits. Thus we can make a universal table T of size $2^{O(x)}$ that for every possible micro TNFA A of size $\leq x$, local state set S_A, and $\alpha \in \Sigma \cup \{\epsilon\}$, provides $T[\text{“}A\text{”}, S_A, \alpha] = \delta_A(S_A, \alpha)$ in constant time. Note that parents and children share some local states, and these states will have to be copies between their local state bitmaps.

Recall that our target is to compute a state-set transition $\delta_{N(R)}(S, \alpha)$ for a single character $\alpha \in \Sigma$. First we should traverse transitions labeled α from S and then traverse paths of ε-transitions. The challenge here is that paths of ε-transition may lead to distant TNFAs in the decomposition resulting in non-trivial dependencies between TNFAs. For example, suppose we start with state set $S = \{1\}$ in Fig. 2, and that we get the input character a. First we follow the only relevant a-transition in A_3 to state 2. Next we follow ε-transitions in A_1, leading us to states 3 and 5, and from there we follow ε-transitions in A_2 to states 6 and 8. We end up concluding that $\delta_{N(R)}(\{1\}, a) = \{2, 3, 5, 6, 8\}$.

Following the character transitions of α is in itself easy using our global table T. For every micro TNFA independently, we take the current local state set S_A and compute $S'_A = T[\text{“}A\text{”}, S_A, a] = \delta_A(S_A, \alpha)$. To handle the subsequent ε-transitions, Myers prove that any cycle-free path of ε-transition in a TNFA uses at most one of the back transitions we get from the star operators. This implies that we can compute the ε-closure in two depth-first traversals of the decomposition. Each of these traversals starts at the root, and are defined recursively for subtrees. When the traversal visits a micro TNFA A, it first sets $S'_A = T[A, S'_A, \varepsilon]$. Next, considering the children in order, one C at the time, it copies the accept state θ_C from S'_A to S'_C, perform an depth-first traversal down from C, and copy the accept state ϕ_C from S'_C to S_A. Finally, it sets $S'_A = T[A, S'_A, \varepsilon]$, and continue to the next child if any. If there were no back transitions, we would need only one such depth-first traversal. The total time we spend on a micro TNFA A is a constant plus a constant per child, adding up to $O(|AS|) = O(\lceil m/x \rceil)$ total time.

The above algorithm assumes that we have built the global table T in $2^{O(x)}$ time and space. It also requires that we for a new regular expression R first construct the decomposition AS, and for each micro TNFA $A \in AS$ find the bit code “A”, but all this takes linear time. Thus, in total we can perform regular expression matching in time $O(nm/x + n + m)$.

To get a better time bound, we have to deal with an input segment q of super-constant length, yet we may only use constant time per micro TNFA. One basic problem is that we have to consider matching paths that leave a micro TNFA and returns as many times as there are characters in q, making a constant number of depth-first traversal sound elusive. Also, each state can correspond to multiple positions in q, but generally, we can only use a constant number of bits per state.

4 2-Dimensional Speed-Up

We will now show how to extend the 1-dimensional speed-up algorithm from the previous section to handle an input segment of length $y = \sqrt{x}$ within the same

$O(\lceil m/x \rceil)$ time bound that we used before on a single character. We are going to use some different global tables, but their total size will still be $2^{O(x)}$.

We will need to impose the restriction on the decomposition that each micro TNFA has at most two children. We get this if we decompose the parse tree using the tree partition technique that Frederickson [7] used for topology trees. As before, we get a decomposition tree AS of $O(\lceil m/x \rceil)$ micro TNFAs, each of size at most x, and now of with at most two children.

As useful new definitions, for any $A \in AS$, define $\overline{A}$ to be the TNFA induced by all states in A and descendants of A in the decomposition, and for any state-set S define $S|\overline{A}$ to be the restriction of S to states in $\overline{A}$.

Recall that one of our challenges we have is that a path p matching q may go in and out of the same micro TNFA many times. We are going to shortcut any downwards loop, that is, a segment s of p that leaves a micro TNFA A to go to a child C and later returns from C to A. When all downwards loops have been shortcut, we are left with a path p with a first part going up the decomposition tree, and a second part going down the decomposition tree. Either part may be empty. The basic point here is that we can follow all such shortcut paths if we do a bottom-up traversal followed by a top-down traversal.

The special thing about the downwards loop s going from A to a child C and later returning is that it matches an interval $q[i,j]$ of q that is accepted by $\overline{C}$. To shortcut the loop, we augment A with the information about all substrings accepted by $\overline{C}$, and do that for each of the at most 2 children of A. Even though $\overline{C}$ may be very large, there are only $\binom{|q|}{2} + |q| + 1 = \binom{|q|+1}{2} = O(y^2) = O(x)$ possible intervals of q.

In Sec. 4.1 we first show how to compute the augmented micro TNFAs and in Sec. 4.2 we show how to use this information to compute state-set transitions. We put the pieces together in Sec. 4.3 to get the full algorithm for regular expression matching.

4.1 Computing Accepted Substrings

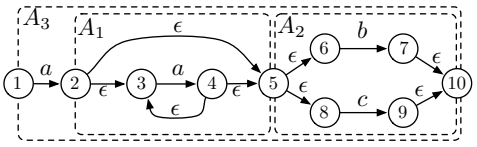
In a single bottom-up traversal of the decomposition we construct for each $A \in AS$ the set SS_A of substrings of q accepted by $\overline{A}$. These are represented as pairs of indices (i,j), that is, $(i,j) \in \mathrm{SS}_A$ iff $\overline{A}$ accepts $q[i,j]$. To compute SS_A, we first construct a "local" representation $A(q)$ of $\overline{A}$ such that for any states s, s' in A, we have

$$P_{A(q)}(s,s') = P_{\overline{A}}(s,s').$$

We construct $A(q)$ from A and the set SS_C from each child C of A if any. More precisely, we replace the pseudo-edge for child C with an NFA accepting the set of substrings of q indexed by SS_C. Having constructed $A(q)$, we compute the set SS_A from $A(q)$ as

$$\mathrm{SS}_A := \{(i,j) \mid \ q[i,j] \text{ is accepted by } A(q)\}.$$

We use the pair $(1,0)$ as a unique representation of the empty substring. In our example in Fig. 2, A_1 accepts $q[1,0] = \epsilon$, $q[1,1] = \mathtt{a}$, $q[1,2] = \mathtt{aa}$, and $q[2,2] = \mathtt{a}$



(a) $R = \texttt{a}\cdot(\texttt{a}^*)\cdot(\texttt{b}|\texttt{c})$ $\quad q = \texttt{aab}$

$A_1 = N(\texttt{a}^*)$ $\quad S = \{1, 3\}$

$A_2 = N(\texttt{b}|\texttt{c})$

$A_3 = N(\texttt{a} \cdot \beta \cdot \beta)$

(b) $\mathrm{SS}_{A_1} = \{(1,0),(1,1),(1,2),(2,2)\}$	$\mathrm{SS}_{A_2} = \{(3,3)\}$	$\mathrm{SS}_{A_3} = \{(1,3),(2,3)\}$
$A_1(q) = N(\epsilon\|\texttt{a}\|\texttt{aa})$	$A_2(q) = N(\texttt{b})$	$A_3(q) = N(\texttt{a} \cdot (\epsilon\|\texttt{a}\|\texttt{aa}) \cdot \texttt{b})$
(c) $\mathrm{PP}^{\phi}_{A_1} = \{1,2\}$	$\mathrm{PP}^{\phi}_{A_2} = \emptyset$	$\mathrm{PP}^{\phi}_{A_3} = \{3\}$
$\mathrm{PP}^{\theta}_{A_1} = \{1\}$	$\mathrm{PP}^{\theta}_{A_2} = \{1,2\}$	$\mathrm{PP}^{\theta}_{A_3} = \{0\}$
$S'_{A_1} = \emptyset$	$S'_{A_2} = \{7, 10\}$	$S'_{A_3} = \{10\}$

Fig. 2. (a) TNFA $N(R)$ for regular expression $R = \texttt{a}\cdot(\texttt{a}^*)\cdot(\texttt{b}|\texttt{c})$. $N(R)$ is decomposed into 3 TNFAs $AS = \{A_1, A_2, A_3\}$, where $A_1 = N(\texttt{a}^*)$, $A_2 = N(\texttt{b}|\texttt{c})$, and $A_3 = N((\texttt{a}\cdot\beta\cdot\beta)^*)$. Here the βs represent A_1 and A_2, respectively. The root of AS is A_3 and A_1 and A_2 are children of A_3. (b) The accepted substrings of $q = \texttt{aab}$ (the index $(1,0)$ indicates the empty string) and $A_1(q)$, $A_2(q)$, and $A_3(q)$. (c) The state-set transition computation for each phase on q on the state-set $S = \{1, 3\}$.

and A_2 accepts $q[3,3] = \texttt{b}$. Therefore $\mathrm{SS}_{A_1} = \{(1,0),(1,1),(1,2),(2,2)\}$ and $\mathrm{SS}_{A_2} = \{(3,3)\}$. Thus $A_1(q) = N(\epsilon|\texttt{a}|\texttt{aa})$ and $A_2(q) = N(\texttt{b})$. Inserting $A_1(q)$ and $A_2(q)$ in A_3 we get $A_3(q) = N(\texttt{a} \cdot (\epsilon|\texttt{a}|\texttt{aa}) \cdot \texttt{b})$, accepting $q[1,3] = \texttt{aab}$ and $q[2,3] = \texttt{ab}$, hence $\mathrm{SS}_{A_3} = \{(1,3),(2,3)\}$.

Encoding and Tabulation We encode q as a bit string "q" of length $O(y)$ and SS_A as a bit string "SS_A" with a bit for each possible interval of q, hence of length $\binom{|q|}{2} + |q| + 1 = \binom{|q|+1}{2} = O(y^2) = O(x)$. Our encoding "$A(q)$" of $A(q)$ consists of the bit-coding "A" of A from Sec. 3 followed by "q" and then the interval end-points in "SS_C" for each child C of A, that is,

$$\text{"}A(q)\text{"} = (\text{"}A\text{"}, \text{"}q\text{"}, \{\text{"}\mathrm{SS}_C\text{"} \mid C \text{ child of } A\}).$$

Thus "$A(q)$" is represented with $O(x + y + y^2) = O(x)$ bits. We precompute a global table SS providing

$$\text{"}\mathrm{SS}_A\text{"} = \mathrm{SS}[\text{"}A(q)\text{"}]$$

as a constant time look-up. The entries take $O(x)$ bits, so the table can be constructed in $2^{O(x)}$ time and space. Subsequently, for any substring q of length at most y, we can compute "SS_A" and "$A(q)$" for every $A \in AS$ in a bottom-up traversal in $O(\lceil m/x \rceil)$ total time.

4.2 Computing State-Set Transitions

We now show how to compute state-set transitions using the "SS_A" and "$A(q)$" computed above. We divide the algorithm into 3 phases. The first phase computes the set of prefixes of q that match a path from $S|\overline{A}$ to ϕ_A within $\overline{A}$, the second phase computes the set of prefixes of q that match a path from S to ϕ_A in $N(R)$, and finally, the third phase computes the local $S'_A = \delta_{N(R)}(S, q))|A$.

Phase 1: Computing path prefixes in $\overline{A}$ to ϕ_A The first phase is a bottom-up traversal. For each $A \in AS$, it computes the set PP^{ϕ}_{A} of prefixes of q matched by a path in $\overline{A}$ from $S|\overline{A}$ to the accept state ϕ_A of A. The prefixes are represented via their end indices, so $j \in \mathrm{PP}^{\phi}_{A}$ iff $q[1,j] \in P_{\overline{A}}(S|\overline{A}, \phi_A)$. The set PP^{ϕ}_{A} is constructed from $A(q)$ and the sets PP^{ϕ}_{C} from the children C of A:

$$\mathrm{PP}^{\phi}_{A} := \{j \mid q[1,j] \in P_{A(q)}(S_A, \phi_A)\} \cup \bigcup_{C \text{ child of } A} \{j \mid i \in \mathrm{PP}^{\phi}_{C} \text{ and } q[i+1,j] \in P_{A(q)}(\phi_C, \phi_A)\}$$

In Fig. 2 we can reach ϕ_{A_1} from state 3 in $A_1(q)$ using $q[1,1]$ or $q[1,2]$ and therefore $\mathrm{PP}^{\phi}_{A_1} = \{1,2\}$. There are no states from S in A_2 so $\mathrm{PP}^{\phi}_{A_2} = \emptyset$. Finally, we can reach ϕ_{A_3} both from state 1 and 3 parsing $q[1,3]$ so $\mathrm{PP}^{\phi}_{A_3} = \{3\}$.

Phase 2: Computing path prefixes in $N(R)$ to θ_A The second phase is a top-down traversal of the decomposition. For each $A \in AS$ we compute the set PP^{θ}_{A} of prefixes of q matched by a path in all of $N(R)$ from S to the start state θ_A of A. The prefixes are represented via their end indices, so $j \in \mathrm{PP}^{\phi}_{A}$ iff $q[1,j] \in P_N(S, \theta_A)$.

If A is the root automaton, we trivially have $\mathrm{PP}^{\theta}_{A} = \{0\}$ if $\theta_A \in S$ and otherwise $\mathrm{PP}^{\theta}_{A} = \emptyset$. Hence we may assume that A has a parent B, and we will use PP^{θ}_{B} to compute PP^{θ}_{A}. We also use $B(q)$ and PP^{ϕ}_{C} from phase 1 for each child C of B, including $C = A$. The computation is now done as

$$\begin{aligned}\mathrm{PP}^{\theta}_{A} :=& \{j \mid q[1,j] \in P_{B(q)}(S_B, \theta_A)\} \\ &\cup \{j \mid i \in \mathrm{PP}^{\theta}_{B} \text{ and } q[i+1,j] \in P_{B(q)}(\theta_B, \theta_A)\} \\ &\cup \bigcup_{C \text{ child of } B} \{j \mid i \in \mathrm{PP}^{\phi}_{C} \text{ and } q[i+1,j] \in P_{B(q)}(\phi_C, \theta_A)\}\end{aligned}$$

In Fig. 2 we have that $\mathrm{PP}^{\theta}_{A_3} = \{0\}$. We can reach θ_{A_1} with $q[1,1]$ from state 1 in $A_3(q)$ and hence $\mathrm{PP}^{\theta}_{A_1} = \{1\}$. We can reach θ_{A_2} with $q[1,1]$ and $q[1,2]$ in $A_3(q)$ from state 1 and hence $1, 2 \in \mathrm{PP}^{\theta}_{A_2}$. From $\mathrm{PP}^{\phi}_{A_1} = \{1,2\}$ we also get that $1, 2 \in \mathrm{PP}^{\theta}_{A_2}$. Hence, $\mathrm{PP}^{\theta}_{A_2} = \{1,2\}$.

Phase 3: Updating state-sets The third and final phase traverses the decomposition in any order. For each $A \in AS$, we compute the $S'_A = \delta_{N(R)}(S,q)|A$. Then the desired state set transition $S' = \delta_{N(R)}(S,q)$ is just the union of these sets. Recall that $y = |q|$.

The set S' is now computed based on $A(q)$, PP^{ϕ}_{C} from phase 1 for each child C of A, and PP^{θ}_{A} from phase 2:

$$S'_A := \{s' \mid q \in P_{A(q)}(S_A, s')\} \cup \{s' \mid i \in \mathrm{PP}^{\theta}_{A} \text{ and } q[i+1,y] \in P_{A(q)}(\theta_A, s')\} \cup \bigcup_{C \text{ child of } A} \{s' \mid i \in \mathrm{PP}^{\phi}_{C} \text{ and } q[i+1,y] \in P_{A(q)}(\phi_C, s')\}$$

In Fig. 2 we have that $S'_{A_1} = \emptyset$, $S'_{A_2} = \{7, 10\}$, and $S'_{A_3} = \{10\}$, so $S' = \delta_{N(R)}(S,q) = \{7, 10\}$.

Encoding and Tabulation We encode each of the sets PP_A^ϕ and PP_A^θ as bit strings "PP_A^ϕ" and "PP_A^θ" of length $y = \sqrt{x}$ where the jth bit is set iff the index j is in the set. The output set S'_A is represented as the input set S_A with a local bit string "S'_A" of length x.

For phase 1, we have a table PP^ϕ so that we can set

$$\text{“}\mathrm{PP}_A^\phi\text{”} := \mathrm{PP}^\phi[\text{“}A(q)\text{”}, \text{“}S_A\text{”}, \{\text{“}\mathrm{PP}_C^\phi\text{”} \mid C \text{ child of } A\}].$$

For phase 2, we have a table PP^θ so that for a child A of B, we can set

$$\text{“}\mathrm{PP}_A^\theta\text{”} := \mathrm{PP}^\theta[\text{“}B(q)\text{”}, \text{“}S_B\text{”}, \text{“}\mathrm{PP}_B^\theta\text{”}, \{\text{“}\mathrm{PP}_C^\phi\text{”} \mid C \text{ child of } B\}].$$

Finally, for phase 3, we have a table S', so that we can set

$$\text{“}S'_A\text{”} := S'[\text{“}A(q)\text{”}, \text{“}S_A\text{”}, \text{“}\mathrm{PP}_A^\theta\text{”}, \{\text{“}\mathrm{PP}_C^\phi\text{”} \mid C \text{ child of } B\}].$$

The entries in each table use $O(x)$ bits and hence we can construct the tables in $2^{O(x)}$ time and space. It follows that given any state-set S and input segment q of length at most y, we can compute $\delta_{N(R)}(S, q)$ in $O(\lceil m/x \rceil)$ total time.

Note that the construction only depends on the fixed alphabet Σ, and the parameters x and y, so the tables may be reused for any regular expression matching problem over the same alphabet.

4.3 The Algorithm

Let $t < 2^w$ be a bound on the space devoted to tables. Choosing $x = y^2 = \varepsilon \log t$ we construct and build all tables in $O(t)$ time and space. Given a regular expression R of length m and a string Q of length n, we solve the regular matching problem using $\lceil n/y \rceil$ state-set transitions computations as described above. To process a new input segment, we only need the global tables, the $O(m)$ space decomposition of R, and the state-set resulting from the preceeding input. Hence, the full algorithm uses space $O(t + m)$ and time $O(m + (m/x + y) \cdot n/y) = O(n + m + nm/\log^{3/2} t)$. Note that if Q is represented as a stream, we are simply processing substrings of length y, one at the time in a single pass, and hence our algorithm also works in this context.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.
2. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
3. V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. On economic construction of the transitive closure of a directed graph. *Dokl. Acad. Nauk.*, 194:487–488, 1970.
4. P. Bille. New algorithms for regular expression matching. In *Proc. 33rd ICALP, LNCS 4051*, pages 643–654, 2006.

5. P. Bille and M. Farach-Colton. Fast and compact regular expression matching. *Theoret. Comput. Sci.*, 409:486 – 496, 2008.
6. T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th STOC*, pages 590–598, 2007.
7. G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997. Announced at FOCS'91.
8. M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. System Sci.*, 48(3):533–551, 1994.
9. Z. Galil. Open problems in stringology. In A. Apostolico and Z. Galil, editors, *Combinatorial problems on words, NATO ASI Series, Vol. F12*, pages 1–8. 1985.
10. V. M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16(5):1–53, 1961.
11. T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.
12. T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proc. 23nd ICDE*, pages 1315–1319, 2007.
13. B. Kernighan and D. Ritchie. *The C Programming Language (2nd Ed.)*. Prentice-Hall, 1988. First edition from 1978.
14. S. C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies, Ann. Math. Stud. No. 34*, pages 3–41. 1956.
15. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. 27th VLDB*, pages 361–370, 2001.
16. W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.
17. R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Computers*, 9(1):39–47, 1960.
18. M. Murata. Extended path expressions of XML. In *Proc. 20th PODS*, pages 126–137, 2001.
19. E. W. Myers. A four-russian algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, 1992.
20. G. Navarro. NR-grep: a fast and flexible pattern-matching tool. *Softw. Pract. Exper.*, 31(13):1265–1312, 2001.
21. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comp. Biology*, 10(6):903–923, 2003.
22. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2004.
23. B. Stroustrup. *The C++ Programming Language: Special Edition (3rd Ed.)*. Addison-Wesley, 2000. First edition from 1985.
24. K. Thompson. Regular expression search algorithm. *Comm. ACM*, 11:419–422, 1968.
25. L. Wall. *The Perl Programming Language*. Prentice Hall Software Series, 1994.
26. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX*, pages 153–162, 1992.
27. F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proc. ANCS*, pages 93–102, 2006.

A Handling Arbitrary Alphabets

We extend the result of Sec. 3 to larger alphabets. The basic idea is to work locally with much smaller alphabets with $O(\log\log t)$-bit characters where t as in the previous sections is the bound on the resources we are devoting to the global tables. We follow an idea of Bille and Farach-Colton [5] for the string edit distance problem. Note that for all our bounds, we can assume $t \geq m$, for if $t < m$, we can create global tables as part of the $O(m)$ time and space spent on the concrete matching.

In the previous section, we assumed constant alphabet size. We will now incorporate the effect of a non-constant alphabet size σ, hence $\lceil \log \sigma \rceil$-bit characters. The consequence for our encoding is that we now need $O(x \log \sigma)$ bits to represent a subautomaton of size x, and $O(y \log \sigma)$ bits to represent a substring q of length y. However, the larger alphabet does not affect the representation of indices into q. In particular, for SS_A, we still need less than y^2 bits to represent all pairs of indices marking the start and end of substrings of q accepted by $A(q)$. It follows that we can represent every index and entry of the global tables in $O(x \log \sigma + y^2 + y \log \sigma)$ bits. We will end up using an alphabet of size at most $(\log t)^2$. Setting $x = \varepsilon \log t / \log\log t$ and $y^2 = \varepsilon \log t$ for some sufficiently small constant $\varepsilon > 0$, we will be able to construct the tables in $O(t)$ time and space like in the previous sections. With alphabet size bounded by $(\log t)^2$, the matching time becomes

$$O\left(m + (m/x + y) \cdot n/y\right) = O(n + m + nm(\log\log t)/(\log^{3/2} t)).$$

Reducing the alphabet size We now have to reduce the alphabet size. First we consider the "normal" case where both n and m are at least $(\log t)^2$. This means that our matching time is dominated by the second degree term $O(nm/(xy)) = O(nm(\log\log t)/(\log t)^{3/2})$.

Recall that in the processing of the regular expression, we enumerated the subautomatons $A \in AS$ in bottom order, using this order directly in bottom-up traversals (phase 1 and 2), and inversely in top down traversals (phase 3 and 4). Continuing the processing of the regular expression, we group the subautomatons into segments of y consecutive subautomatons. For each group, we construct a deterministic dictionary over all the characters in the subautomatons in the group. We have at most xy such characters, and in addition, we need an error character. The dictionary maps any character from the original alphabet into $\{0, ..., xy\}$ in constant time. The group characters are mapped 1-1 into $\{1, ..., xy\}$, and all non-group characters are mapped to 0. Having constructed a group dictionary, we use it to convert all characters in the automatons in the group. We have $xy = \varepsilon \log t / \log\log t \cdot \sqrt{\varepsilon \log t} \ll \log^2 t$, so our global tables apply to these converted automatons.

Each group has at most xy characters. A group dictionary using $O(xy)$ space can be constructed deterministically in $O(xy \log(xy)) = O(xy \log\log t)$ time [11]. The character conversion within the group is done in $O(xy)$ time. The number of subautomaton groups is $O(m/(xy))$, so the group dictionaries and the converted

subautomatons are constructed in $O(m \log\log t)$ time and $O(m)$ space. We are now done with the processing of the pattern. Note that since $m, n > \log^2 t$ we have that $O(m \log\log t) = o(nm(\log\log t)/(\log t)^{3/2})$.

We will now process the string, processing one substring Q_i of length y at the time. To compute the state transition $S_i = \delta_{N(R)}(S_{i-1}, Q_i)$, in each phase, every time we get to a new automaton group, we use the group dictionary to convert the characters of Q_i into those of the group, in $O(y)$ time. Each group is used once in each of four phases, and there are $m/(xy)$ groups, so the total character conversion time is $O(ym/(xy)) = O(m/x)$, matching the time we use for the table based state transition with the converted characters. Thus, using the global tables, we end up performing the whole regular expression matching in $O(nm/(xy)) = O(nm(\log\log t)/(\log t)^{3/2})$ time.

Next we prepare for the extreme case where either the regular expression or the string is of size less than $(\log t)^2$. An easy theoretical solution is to employ atomic heaps of Fredman and Willard [8]. As part of the $O(t)$ time and space preprocessing, we get atomic heaps with capacity for up to $(\log t)^2$ elements. These are very general, but we will just use them as dynamic dictionaries supporting both updates and queries in constant amortized time. If, say, the regular expression R is of size $m < (\log t)^2$, using an atomic heap, we create a dictionary over the characters in R in $O(m)$ time, and use it to convert all characters in both the regular expression and in the string Q in linear time. The case where the sting Q is of size $n < (\log t)^2$, is symmetric. In both cases, we pay linear time to get a converted alphabet of size at most $(\log t)^2$. We have thus shown the unit-cost RAM part of Theorem 1:

Theorem 2. *On a unit-cost RAM with word length w and standard instruction set, for any parameter $t < 2^w$, we can do a general preprocessing for regular expression matching using $O(t)$ time and space. Subsequently, given any regular expression of length m and string of length n, we can perform the matching in*

$$O\left(\frac{nm \log\log t}{(\log t)^{3/2}} + n + m\right)$$

time using $O(t + m)$ space. Our matching only makes a single pass through the string, which may hence be presented as a stream.

A.1 An alternative for large alphabets

We will add a simple alternative to Theorem 2 for large alphabets.

Theorem 3. *Given any regular expression of length m and string of length n and some $t \geq m^{1+\varepsilon}$ for constant $\varepsilon > 0$, we can solve the regular expression matching problem in*

$$O\left(t + \frac{nm}{(\log t)^{3/2}} + n\right)$$

time using $O(t)$ space. Our matching only makes a single pass through the string, which may hence be presented as a stream.

Contrasting Theorem 2, this new result does not benefit from a global tabulation. It does avoid the unappealing factor $\log\log t$ but has the unappealing restriction that $t \geq m^{1+\varepsilon}$. However, this seems the better choice when a fairly small regular expression is matched against a large string.

The idea is very simple. Instead of encoding a subautomaton A using $O(x \log\log t)$ bits, we just enumerate the automatons. There are only $O(m/x)$ of them, so their numbers need $\log m = (1-\varepsilon)\log t$ bits. They remaining part of the indices uses $O(x + y^2 + y\log\log t)$ bits. We can now use $x = y^2 = \varepsilon' \log t$ for some small enough $\varepsilon' > 0$ as we did for constant size alphabets, and thus get the same total running time as we had there. However, we cannot use global tabulation, for the subautomaton with number i is not known until we have decomposed the regular expression.

B External Memory

We will now modify our techniques for external memory model with block size B and internal memory size $M \geq B$. Here in external memory we follow the usual indivisibility assumption that characters and pointers are atomic units, and that size is measured as the number of these units. Then sorting s characters requires ext-sort$(s) = \Theta(s/B \cdot \log_{M/B}(s/B))$ I/O operations [1]. We are not going to use any of the tabulation from our unit RAM solution. In external memory, the point of packing hard subproblems is not to solve them by an expensive table look-up. Rather, on a much larger scale, the point is to cluster hard subproblems so that they can be quickly read into and processed in internal memory.

If $m = o(M)$, the state-set simulation of $N(R)$ can be carried out within internal memory, and then it is trivial to perform regular expression matching in $O(n/B + m/B)$ I/O operations. Hence we assume $m = \Omega(M)$.

Performing each step in internal memory Our basic idea is to implement our algorithm from Section 4 in such a way that each step can be completed in internal memory. For some small constant ε, we use subautomaton size $x = \varepsilon M$ and substring size $y = \sqrt{\varepsilon M}$. The input and output of each step is of size $O(x + y^2) < M$ for sufficiently small ε.

For example, when computing the accepted substrings, the input of a step is a subautomaton A of size x, a substring q of size y, and for each of at most two children C of A, the set SS_C of index pairs (i, j) such that $\overline{C}$ accepts $q[i, j]$. The output is $A(q) = (A, \{\mathrm{SS}_C \mid C \text{ child of } A\}$ as well as the set SS_A of index pairs (i, j) such that $A(q)$ accepts $q[i, j]$.

Assuming that each subautomaton is stored with related information in consecutive blocks, we can read the input A, q, and SS_C for each child of A in $O(M/B)$ I/O operations. We now need to show that we can compute SS_A without leaving internal memory. For each pair (i, j), $1 \leq i \leq j \leq k$, one at the time, we need to parse $q[i, j]$ checking if it is accepted by $A(q)$.

So far we have said that $A(q)$ represented that NFA obtained from A by expanding the pseudo-edge of each child C to an NFA accepting the at most

y^2 substrings of q indexed by SS_C. However, the sum of the lengths of these substrings may be $\Theta(y^3)$ so the resulting NFA could have that many states. Since $M = O(y^2)$, this would not fit internal memory.

We will now show how to run to $A(q)$ within the $O(x + y^2)$ space we have available. First, if C accepts the empty string, we emulate this with a separate ϵ-transition, so we only need to consider non-empty substrings indexed by SS_C. We will grow an index set of the start state θ_C of each child C of A. Here while parsing $q[i, j]$, the index set of a state u in A is the set $I_u = \{k \in [i, j] \mid q[i, k] \in P_{A(q)}(\theta_A, u)\}$. Since $|I_u| \leq y$, it is not a space problem to maintain I_u for the start state of each child.

We will now parse the characters of $q[i, j]$ one at the time. When we get to character $q[\ell]$, $\ell \in [i, j]$, we assume we know the state set S_{k-1} of states u of A such that $q[i, k-1] \in P_{A(q)}(\theta_A, u)$. Here $S_0 = \{\theta_A\}$. Moreover, for each child C, the index set will be filled up to $\ell - 1$, that is, $I_{\theta_C} = \{k \in [i, \ell - 1] \mid q[i, k] \in P_{A(q)}(\theta_A, \theta_C)\}$. We now want to compute the new state set S_k. The important point is that

$$\phi_C \in S_k \iff \exists k \in I_{\theta_C} : [i, \ell] \in \mathrm{SS}_C.$$

This condition is easily checked in $O(y)$ time not using any extra space, and we just have to do it for each of the at most two children C. The rest of the new state set S_k is found using the standard simulation of A in $O(x)$ time. The above is repeated for each $i \leq j \leq k$, so the total time is $O(y^2(x + y))$ while the total space is $O(x+y^2)$ as desired. Afterward we write $A(q)$ and SS_A to external memory using $O(M/B)$ I/O operations.

The phases of the state-set computation are handled similarly. In total, we use $O(m/x)$ steps to deal with each substring of length at most y, so the total number of I/O operations is $O(m/x \cdot n/y \cdot M/B) = O(nm/(\sqrt{M}B))$.

Decomposition in external memory Above we assumed that we had the decomposition AS of $N(R)$ into subautomatons each of size x and stored in consecutive blocks. We show how to do this in $O(\text{ext-sort}(m))$ time. Basically, we have to do a cluster partitioning of the parse tree $T = T(R)$ into clusters of size at most x. Our algorithm is based on local edge contractions. For each node v in T we maintain a weight representing the number of nodes contracted into v. Initially, all nodes are given weight 1 and whenever we contract an edge (v, u) the weight of the new node is $w(v) + w(u)$. We use the following contraction rules. An edge (v, u) where u is the parent of v and $w(v) + w(u) < x$ can be contracted if

(i) v is leaf, or
(ii) v has no siblings.

Let T' be the tree resulting after applying (i) and (ii) greedily until no edges can be contracted according the rules. From the rules it follows that T' is a binary tree containing at most $O(\lceil n/x \rceil)$ nodes and the weight of any node is at most x. Hence, each node corresponds to a cluster in a cluster partition. It is well-known from results on parallel tree contraction algorithms that if we apply the rules in *rounds*, where each round first applies rule (i) greedily and then rule

(ii) greedily, the number of nodes decrease by a constant factor and hence after at most $O(\log m)$ rounds we have computed T'.

To construct the cluster partition in external memory we implement each round by sorting the edges according to parent endpoints. This identifies all children of all nodes. In particular, we can find all leaves and nodes with no siblings. Subsequently, we can contract edges according to rules and update weights accordingly. Hence, each round involves sorting and a constant number of passes over the remaining edges. Since the number of remaining edges decrease by a constant factor in each round the total number of I/O operations is dominated by the number in the first round, which is $O(\text{ext-sort}(m))$.

It follows that we can solve regular expression matching using $O\left(\frac{nm}{\sqrt{M}B} + \text{ext-sort}(m)\right)$ I/O operations. Here in external memory, we did not use any tables. The only space we use beyond the input itself is $O(x + y^2) = O(x)$ per subautomaton in AS, and we have $O(m/x)$ of these, so the total space is $O(m)$. Thus we have proved

Theorem 4. *Let R be a regular expression of length m and let Q be a string of length n. In external memory with block size B and internal memory size M, $B \leq M \leq m$, the regular expression matching problem can be solved in $O(nm/(\sqrt{M}B) + m/B \cdot \log_{M/B}(m/B))$ I/O operations and $O(m)$ space.*

This completes the proof of Theorem 1. In the introduction, we ignored the sorting term $m/B \cdot \log_{M/B}(m/B)$, because it is dominated by the $nm/(\sqrt{M}B)$ term when $n \geq m \geq M$.