Regular Expression Matching with Multi-Strings and Intervals

Philip Bille*
phbi@imm.dtu.dk

Mikkel Thorup
mthorup@research.att.com

Abstract

Regular expression matching is a key task (and often computational bottleneck) in a variety of software tools and applications. For instance, the standard `grep` and `sed` utilities, scripting languages such as `perl`, internet traffic analysis, XML querying, and protein searching. The basic definition of a regular expression is that we combine characters with union, concatenation, and kleene star operators. The length m is proportional to the number of characters. However, often the initial operation is to concatenate characters in fairly long strings, e.g., if we search for certain combinations of words in a firewall. As a result, the number k of strings in the regular expression is significantly smaller than m. Our main result is a new algorithm that essentially replaces m with k in the complexity bounds for regular expression matching. More precisely, after an $O(m \log k)$ time and $O(m)$ space preprocessing of the expression, we can match it in a string presented as a stream of characters in $O(k\frac{\log w}{w} + \log k)$ time per character, where w is the number of bits in a memory word. For large w, this corresponds to the previous best bound of $O(m\frac{\log w}{w} + \log m)$. Prior to this work no $O(k)$ bound per character was known. We further extend our solution to efficiently handle character class interval operators $C\{x, y\}$. Here, C is a set of characters and $C\{x, y\}$, where x and y are integers such that $0 \leq x \leq y$, represents a string of length between x and y from C. These character class intervals generalize variable length gaps which are frequently used for pattern matching in computational biology applications.

1 Introduction

1.1 The Problem A regular expression specifies a set of strings formed by characters combined with concatenation, union (`|`), and kleene star (`*`) operators, e.g., `(a|(ba))*` is a string of `a`s and `b`s where every `b` is followed by an `a`. Given a regular expression R and a string Q the regular expression matching problem is to decide if Q matches any of the strings specified by R. This problem is a key primitive in a wide variety of software tools and applications. Standard tools such as `grep` and `sed` provide direct support for regular expression matching in files. The scripting language `perl` [30] is a full programming language that is designed to easily support regular expression matching. In large scale data processing applications, such as internet traffic analysis [14, 31], XML querying [18, 21], and protein searching [25], regular expression matching is often the main computational bottleneck.

Historically, regular expression matching goes back to Kleene in the 1950s [16]. It became popular for practical use in text editors in the 1960s [28]. Compilers use regular expression matching for separating tokens in the lexical analysis phase [2].

1.2 Previous Work Let R be a regular expression of length m and let Q be a string of length n. Typically, we have that $m \ll n$. The alphabet is denoted Σ. All of the bounds mentioned below and presented in this paper hold for a standard unit-cost RAM with w-bit words with standard word operations including arithmetic and logical operations. This means that the algorithms can be implemented directly in standard imperative programming languages such as C [15] or C++ [27]. We note that most open source implementations of algorithms for regular expression, e.g., `grep`, `sed` and `perl`, are written in C. An index into Q can be stored in a single word and therefore $w \geq \log n$. The space complexity is the number of words used by the algorithm, not counting the input which is assumed to be read-only.

The classical textbook solution to the problem by Thompson [28] from 1968 takes $O(nm)$ time. It uses a standard state-set simulation of a nondeterministic finite automaton (NFA) with $O(m)$ states and transitions produced from R. Using the NFA, we scan Q. Each of the n characters is processed in $O(m)$ time by a standard state-set simulation of the NFA.

[*] Supported by the Danish Agency for Science, Technology, and Innovation.

In 1985 Galil [10] asked if a faster algorithm could be derived. Myers [23] met this challenge in 1992 with an $O(\frac{nm}{\log n} + (n+m)\log n)$ time solution, thus improving the time complexity of Thompson algorithm by a $\log n$ factor for most values of n and m. Bille and Farach-Colton [5] improved the space usage of Myers' algorithm. Recently, the authors obtained an algorithm using $O(\frac{nm \log\log n}{\log^{1,5} n} + n + m)$ time [6], further improving Thompson's algorithm by a $\frac{\log^{1,5} n}{\log\log n}$ factor. The space complexity is $O(n^{\varepsilon} + m)$. All of these improvements decompose the NFA from Thompson's algorithm into micro NFAs and tabulate information for these to speedup the simulation of the NFA on Q. In [6] the tabulation is 2-dimensional, covering micro NFAs together with string segments. All these tabulation-based approaches leads to a time-space tradeoff depending on the amount of space we can use for tables.

An incomparable approach of Bille [4] is to simulate the micro NFAs using standard word operations to work on several states in parallel. This is ideal for a small space streaming algorithm since it only uses $O(m)$ space. After an $O(m \log m)$ time and $O(m)$ space preprocessing, he can process each character of the input string in $O(m\frac{\log w}{w} + \log m)$ time. Note that this is better than the bound from [6] if $w \geq \log^{1.5} n$. Bille also has a constant bound per character for very small patterns with $m \leq \sqrt{w}$.

1.3 Our Results

We identify and address two basic questions for regular expression matching.

1.3.1 Multi-Strings

In the above applications, we often see that regular expression are based on a comparatively small set of strings, directly concatenated from characters. As an example, consider the following regular expression used to detect a Gnutella data download signature in a stream [26]:

```
(Server:|User-Agent:)( |\t)*(LimeWire|
BearShare|Gnucleus|Morpheus|XoloX|
gtk-gnutella|Mutella|MyNapster|Qtella|
AquaLime|NapShare|Comback|PHEX|SwapNut|
FreeWire|Openext|Toadnode)
```

Here, the total size of the regular expression is $m = 174$ (the `\t` is a single tab character) but the number of strings is only $k = 21$.

Our first question is if we can design a more efficient algorithm that exploits $k \ll m$. No previous paper has addressed this issue, so it was not even known if we can improve Thompson [28] classic $O(nm)$ bound. We show the following result.

THEOREM 1.1. *Let R be a regular expression of length m containing k strings and let Q be a string of length n. On a unit-cost RAM we can solve regular expression matching in time*

$$O\left(n\left(k\frac{\log w}{w} + \log k\right) + m\log k\right) = O(nk + m\log k)\,.$$

This works as a streaming algorithm that after an $O(m \log k)$ time and $O(m)$ space preprocessing can process each input character in $O(k\frac{\log w}{w} + \log k)$ time.

Compared to the algorithm by Bille [4] we replace m with k.

1.3.2 Character Class Intervals

We consider an extension of the standard set of operators in regular expressions (concatenation, union, and kleene star). Given a regular expression R and integers x and y, $0 \leq x \leq y$, the *interval operator* $R\{x, y\}$ is shorthand for the expression

$$\overbrace{R\cdots R}^{x}\overbrace{(R|\varepsilon)\cdots(R|\varepsilon)}^{y-x}\ ,$$

that is, at least x and at most y concatenated copies of R.

We are interested in the important special case of *character class intervals* defined as follows. Given a set of characters $\{\alpha_1, \ldots, \alpha_c\}$ the *character class* $C = [\alpha_1, \ldots, \alpha_c]$ is shorthand for the expression $(\alpha_1|...|\alpha_c)$. Thus, the *character class interval* $C\{x, y\}$ is shorthand for

$$C\{x,y\} = \overbrace{(\alpha_1|\ldots|\alpha_c)\cdots(\alpha_1|\ldots|\alpha_c)}^{x}\cdot$$
$$\overbrace{(\alpha_1|\ldots|\alpha_c|\varepsilon)\cdots(\alpha_1|\ldots|\alpha_c|\varepsilon)}^{y-x}\ .$$

Hence, the length of the short representation of $C\{x, y\}$ is $O(|C|)$ whereas the length of $C\{x, y\}$ translated to standard operators is $\Omega(|C|y)$.

Our second question is if we can design an algorithm for regular expression matching with character class intervals that is more efficient than simply using the above translation with an algorithm for the standard regular expression matching problem. Again, no previous paper has addressed this issue. We show the following generalization of Theorem 1.1.

THEOREM 1.2. *Let R be a regular expression of length m containing k strings and character class intervals and let Q be a string of length n. On a unit-cost RAM we can solve regular expression matching* *in time*

$$O\left(n\left(k\frac{\log w}{w} + \log k\right) + X + m\log m\right) .$$

Here, X is the sum of the lower bounds on the lengths of the character class intervals. This works as a streaming algorithm that after an $O(X + m\log m)$ time and $O(X + m)$ space preprocessing can process each input character in $O(k\frac{\log w}{w} + \log k)$ time.

Even without multi-strings this is the first algorithm to efficiently support character class intervals in regular expression matching.

The special case $\Sigma\{x, y\}$ (we identify character classes by their corresponding sets) is called a *variable length gap* since it specifies an arbitrary string of length at least x and at most y. Variable length gaps are frequently used in computational biology applications [8,9,22,24,25]. For instance, the PROSITE data base [7, 13] supports searching for proteins specified by patterns formed by concatenation of characters, character classes, and variable length gaps.

In general the variable length gaps may be long compared to the length of the expression. For instance, the pattern

$$\texttt{MT}\cdot\Sigma\{115, 136\}\cdot\texttt{MTNTAYGG}\cdot\Sigma\{121, 151\}\cdot\texttt{GTNGAYGAY}$$

appears in Morgante et al. [20].

For various restricted cases of regular expressions some results for variable length gaps are known. For patterns composed using only concatenation, character classes, and variable length gaps Navarro and Raffinot [25] gave an algorithm using $O(\frac{m+Y}{w}+1)$ time per character, where Y is the sum of the upper bounds on the length of the variable length gaps. This algorithm encodes each variable length gap using a number of bits proportional to the upper bound on the length of the gap. Fredriksson and Grabowski [8, 9] improved this for the case when all variable length gaps have lower bound 0 and identical upper bound y. They showed how to encode each such gaps using $O(\log y)$ bits [8] and subsequently $O(\log\log y)$ bits [9], leading to an algorithm using $O(m\frac{\log\log y}{w}+1)$ time per character. The latter results are based on maintaining multiple counters efficiently in parallel. We introduce an improved algorithm for maintaining counters in parallel as a key component in Theorem 1.2. Plugging in these counters in the algorithms of Fredriksson and Grabowski [8, 9] we obtain an algorithm using $O(\frac{m}{y} + 1)$ per character for this problem. With our counters we can even remove the restriction that each variable length gap should have the same upper bound. However, when $k \ll m$ the bound of Theorem 1.2 for the more general problem is better.

1.4 Technical Outline We will first show how to get an $O(nk + m\log k)$ bound for regular expression matching using the classic multi-string matching algorithm of Aho and Corasick [1] inside Thompson's [28] classic regular expression matching algorithm. In order to achieve the speed up we show how to efficiently handle multiple bit queues of non-uniform length and combine these with the algorithm of Bille [4]. For the character class intervals we further extend our algorithm to efficiently handle multiple non-uniform counters. We believe that our techniques for non-uniform bit queues and counters are of independent interest. We do not see a way of benefitting from a small $k \ll m$ in tabulation based methods like that in [6].

2 Basic Concepts

We briefly review the classical concepts used in the paper.

2.1 Regular Expression and Finite Automata

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*[1]. 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

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

ϵ-transitions, and all outgoing transitions of any state have different labels. Otherwise, A is a *non-deterministic automaton* (NFA). When we deal with multiple automatons, we use a subscript $_A$ to indicate information associated with automaton A, e.g., θ_A denotes 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 say that A *accepts* a string Q if there is a path in A from θ to ϕ that matches Q. Otherwise A *rejects* Q. Let S be a state-set in A and let α be a character from Σ, and define the following operations.

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

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

One may use a sequence of Move and Close operations to test if A accepts a string Q of length n as follows. First, set $S_0 := \mathsf{Close}(\{\theta\})$. For $i = 1, \ldots, n$ compute $S_i := \mathsf{Close}(\mathsf{Move}(S_{i-1}, Q[i]))$. It follows inductively that S_i is the set states in A reachable by a path from θ matching the ith prefix of Q. Hence, Q is accepted by A if and only if $\phi \in S_n$.

Given a regular expression R, an NFA A accepting precisely the strings in $L(R)$ can be obtained by several classic methods [11, 19, 28]. In particular, Thompson [28] gave the simple well-known construction in Figure 1. We will call an automaton constructed with these rules a *Thompson NFA* (TNFA). Fig. 2(a) shows the TNFA for the regular expression $R = \mathtt{(aba|a)^*ba}$.

A TNFA $N(R)$ for R has at most $2m$ states, at most $4m$ transitions, and can be computed in $O(m)$ time. We can implement the Move operation on $N(R)$ in $O(m)$ time by inspecting all transitions. With a breadth-first search of $N(R)$ we can also implement the Close operation in $O(m)$ time. Hence, we can test acceptance of a string Q of length n in $O(nm)$ time. This is Thompson's algorithm [28].

2.2 Multi-String Matching

Given a set of pattern strings $P = \{P_1, \ldots, P_k\}$ of total length m and a text Q of length n the *multi-string matching problem* is to report all occurrences of each pattern string in Q. Aho and Corasick [1] generalized the classical Knuth-Morris-Pratt algorithm [17] for single string matching to multiple strings. The *Aho-Corasick automaton* (AC-automaton) for P, denoted $\mathrm{AC}(P)$, consists of the trie of the patterns in P. Hence, any path from the root of the trie to a state s corresponds to a prefix of a pattern in P. We denote this prefix by $\mathrm{path}(s)$. For each state s there is also a special *failure transition* pointing to the unique state s' such that $\mathrm{path}(s')$ is the longest prefix of a pattern in P matching a proper suffix of $\mathrm{path}(s)$. Note that the depth of s' in the trie is always strictly smaller for non-root states than the depth of s.

Finally, for each state s we store the subset $\mathrm{occ}(s) \subseteq P$ of patterns that match a suffix of $\mathrm{path}(s)$. Since the patterns in $\mathrm{occ}(s)$ share suffixes we can represent $\mathrm{occ}(s)$ compactly by storing for s the index of the longest string in $\mathrm{occ}(s)$ and a pointer to the state s' such that $\mathrm{path}(s')$ is the second longest string if any. In this way we can report $\mathrm{occ}(s)$ in $O(|\mathrm{occ}(s)|)$ time.

The maximum outdegree of any state is bounded by the number of leaves in the trie which is at most k. Hence, using a standard comparison-based balanced search tree to index the trie transitions out of each state we can construct $\mathrm{AC}(P)$ in $O(m \log k)$ time and $O(m)$ space. Fig. 2(c) shows the AC-automaton for the set of patterns $\{\mathtt{aba}, \mathtt{a}, \mathtt{ba}\}$. Here, the failure transitions are shown by dashed arrows and the non-empty sets of occurrences are listed by indices to the three strings.

For a state s and character α define the *state-transition* $\Delta(s, \alpha)$ to be the unique state s' such that $\mathrm{path}(s')$ is the longest suffix that matches $\mathrm{path}(s) \cdot \alpha$. To compute a state-set transition $\Delta(s, \alpha)$ first test if α matches the label of a trie transition t from s. If so we return the child endpoint of t. Otherwise, we recursively follow failure transitions from s until we find a state s' with a trie transition t' labeled α and return the child endpoint of t'. If no such state exists we return the root the trie. Each lookup among the trie transitions uses $O(\log k)$ time. One may use a sequence of state transitions to find all occurrences of P in Q as follows. First, set s_0 to be the root of $\mathrm{AC}(L)$. For $i = 1, \ldots, n$ compute $s_i := \Delta(s_{i-1}, Q[i])$ and report the set $\mathrm{occ}(s_i)$. Inductively, it holds that $\mathrm{path}(s_i)$ is the longest prefix of a pattern in P matching a suffix of the ith prefix of Q. For each failure transition traversed in the algorithm we must traverse at least as many trie transitions. Therefore, the total time to traverse $\mathrm{AC}(P)$ and report occurrences is $O(n \log k + \mathrm{occ})$, where occ is the total number of occurrences.

Hence, the Aho-Corasick algorithm solves multi-string matching in $O((n + m) \log k + \mathrm{occ})$ time and $O(m)$ space.

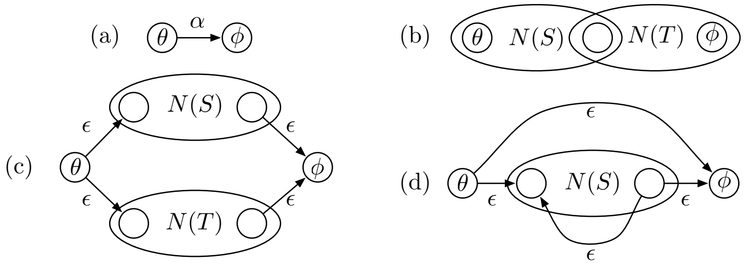

Figure 1: Thompson's recursive NFA construction. The regular expression for a character $\alpha \in \Sigma$ corresponds to NFA (a). If S and T are regular expressions then $N(ST)$, $N(S|T)$, and $N(S^*)$ correspond to NFAs (b), (c), and (d), respectively. In each of these figures, the leftmost node θ 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).

3 A Simple Algorithm

In this section we present a simple comparison-based algorithm for regular expression matching using $O(nk + m \log k)$ time. In the following section we give a faster implementation of the algorithm leading to Theorem 1.1.

3.1 Preprocessing Recall that R is a regular expression of length m containing k strings $L = \{L_1, \ldots, L_k\}$. We will preprocess R in $O(m)$ space and $O(m \log k)$ time as described below. First, define the *pruned regular expression* $\widetilde{R}$ obtained by replacing each string L_i by its index. Thus, $\widetilde{R}$ is a regular expression over the alphabet $\{1, \ldots, k\}$. The length of $\widetilde{R}$ is $O(k)$.

Given R we compute $\widetilde{R}$, the TNFA $N(\widetilde{R})$, and $\mathrm{AC}(L)$ using $O(m \log k)$ time and $O(m)$ space. For each string $L_i \in L$ define $\mathrm{start}(L_i)$ and $\mathrm{end}(L_i)$ to be the startpoint and endpoint, respectively, of the transition labeled i, in $N(\widetilde{R})$. For any subset of strings $L' \subseteq L$ define $\mathrm{start}(L') = \bigcup_{L_i \in L'} \mathrm{start}(L_i)$ and $\mathrm{end}(L') = \bigcup_{L_i \in L'} \mathrm{end}(L_i)$. Fig. 2(b) shows $\widetilde{R}$ and $N(\widetilde{R})$ for the regular expression $R = \texttt{(aba|a)*ba}$. The corresponding AC-automaton for $\{\texttt{aba}, \texttt{a}, \texttt{ba}\}$ is shown in Fig. 2(c).

To represent the interaction between $N(\widetilde{R})$ and $AC(L)$, we construct and maintain a set of FIFO queues $F = \{F_1, \ldots, F_k\}$ associated with the strings in L. Queue F_i stores a sequence of $|L_i|$ bits. Let $S \subseteq \mathrm{start}(L)$ and define the following operations on F.

$\mathsf{Enqueue}(S)$: For each queue F_i enqueue into the back of F_i a 1-bit if $\mathrm{start}(L_i) \in S$ and otherwise a 0-bit.

Front: Return the set of states $S \subseteq \mathrm{end}(L)$ such that $\mathrm{end}(L_i) \in S$ iff the front bit of F_i is 1.

Using a standard cyclic list for each queue we can support the operations in constant time per queue. Both of these queue operations take $O(k)$ time.

All in all, we use $O(m)$ space and $O(m \log k)$ time for preprocessing.

3.2 Matching We show how to match R to Q in $O(k)$ time per character of Q. We compute a sequence of state-sets $S_0, \ldots, S_n$ in $N(\widetilde{R})$ and a sequence of states $s_0, \ldots, s_n$ in $\mathrm{AC}(L)$ as follows. Initially, set $S_0 := \mathsf{Close}(\{\theta\})$, set s_0 to be the root of $\mathrm{AC}(L)$, and initialize the queues with all 0-bits. At step i of the algorithm, $1 \leq i \leq n$, we perform the following steps.

1. Compute $\mathsf{Enqueue}(S_{i-1} \cap \mathrm{start}(L))$.
2. Set $s_i := \Delta(s_{i-1}, Q[i])$.
3. Set $S' := \mathsf{Front}(F) \cap \mathrm{end}(\mathrm{occ}(s_i))$
4. Set $S_i := \mathsf{Close}(S')$.

Finally, we have that $Q \in L(R)$ if and only $\theta \in S_n$. Note the correspondence between the sequence of bits in the queue and the states representing the strings in R within $N(R)$. The queue and set operations take $O(k)$ time and traversing the AC-automaton takes $O(n \log k)$ time in total. Hence, with preprocessing the entire algorithm uses $O(nk + (n + m) \log k) = O(nk + m \log k)$ time. Note that all computation only

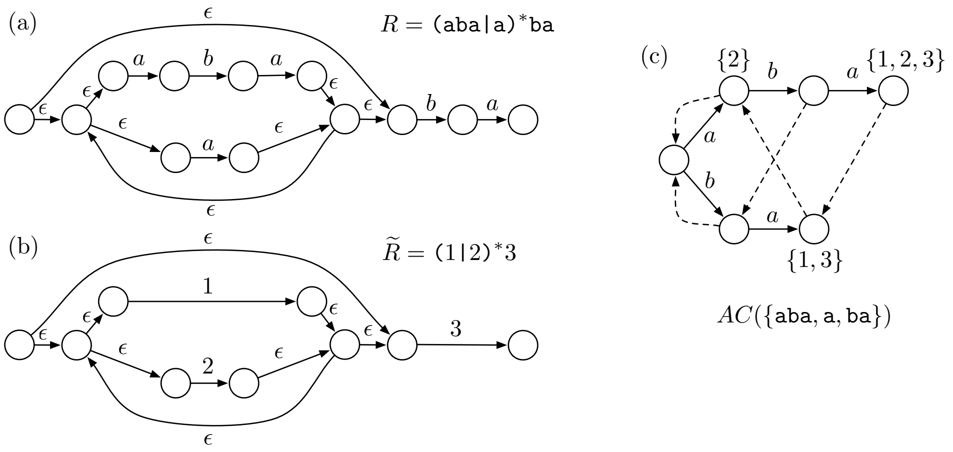

Figure 2: (a) $N(R)$ for $R =$ (aba|a)*ba. (b) $N(\widetilde{R})$ where $\widetilde{R} =$ (1|2)*3 (c) AC-automaton for the strings in R.

requires a comparison-based model of computation. Hence, we have the following result.

Theorem 3.1. *Given a regular expression of length m containing k strings and a string of length n, we can solve regular expression matching in a comparison-based model of computation in time $O(nk + m \log k)$ and space $O(m)$.*

4 A Faster Algorithm

We show how to speed up the simple algorithm from the previous section to do matching in $O(k \log w/w + \log k)$ time per character. We first give a high-level description of the algorithm by Bille [4] that uses $O(m \log w/w + \log m)$ per character. Then, we show how to modify it to obtain our new bound. Essentially, we will be able to apply Bille's Close operation directly to the TNFA $N(\widetilde{R})$ of the pruned regular expression. The more interesting part will be to speed-up the FIFO bit queues of non-uniform length that represent the interaction between the TNFA and the multi-string matching.

4.1 Bille's Speed-up for Thompson's Algorithm We now describe the key features of the basic algorithm of Bille [4]. Later we show how to handle strings more efficiently as we did above for Thompson's algorithm. Initially, we assume $m \geq w$.

First, we *decompose* $N(R)$ into a tree AS of $O(\lceil m/w \rceil)$ micro TNFAs, each with at most w states. For 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. 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. Similar decompositions are used in most of the fast algorithms [4–6, 23].

Secondly, we store a state-set S as a set of local state-set S_A, $A \in AS$. We represent S_A compactly by mapping each state in A to a position in the range $[0, O(w)]$ and store S_A by the bit string consisting of 1s in the positions corresponding the states in S_A. For simplicity, we will identify S_A with the bit string representing it. The mapping is constructed based on a balanced separator decomposition of A of depth $O(\log w)$, which combined with some additional information allows us to compute local Move_A and Close_A operations efficiently. To implement Move_A we ensure that endpoints of transitions labeled by characters are mapped to consecutive positions. This is always possible since the startpoint of such a transition has outdegree 1 in TNFAs. We can now compute $\mathsf{Move}_A(S_A, \alpha)$ in constant time by shifting S_A and &'ing the result with a mask containing a 1 in all positions of states with incoming transitions labeled α. This idea is a simple variant of the classical Shift-Or algorithm [3]. To implement Close_A we use properties of the separator decomposition to define a recursive algorithm on the decomposition. Using precomputed mask combined with the mapping the recursive algorithm can be implemented in parallel for each level of the recursion using standard word operations. Each level of the recursion is done in constant time leading to an $O(\log w)$ algorithm for Close_A.

Finally, we use the local Move_A and Close_A algorithm to obtain an algorithm for Move and Close on $N(R)$. To implement Move we simply perform Move_A independently for each micro TNFA $A \in AS$ using $O(|AS|) = O(m/w)$ time. To implement Close from Close_A we use the result that any cycle-free path of ε-transitions in a TNFA uses at most one of the back transitions we get from the kleene star operators [23]. This implies that we can compute the Close in two depth-first traversals of the decomposition using constant time per micro TNFA. Hence, we use $O(|AS| \log w) = O(m \log w/w)$ time per character. For the case of $m < w$ we only have one micro TNFA and the depth of the separator decomposition is $\log m$. Hence, we spend $O(\log m)$ time per character. In general, we use $O(m \log w/w + \log m)$ per character.

4.2 Speeding Up the Simple Algorithm

We now show how to improve the simple algorithm to perform matching in $O(k \log w/w + \log k)$ time per character. Initially, we assume that $k \geq w$.

We now consider the TNFA $N(\widetilde{R})$ for the pruned regular expression $\widetilde{R}$ that we defined in the previous section. We decompose $N(\widetilde{R})$ into a tree $\widetilde{AS}$ of $O(k/w)$ micro TNFAs each with at most w states. We represent state-sets as in the algorithm by Bille [4] and we will use the same algorithm for Close operation in step 4 of our simple algorithm. Hence, step 4 takes $O(|\widetilde{AS}| \log w) = O(k \log w/w)$ time as desired. The remaining step steps only affect the endpoints of non-ε transitions and therefore we can perform them independently on each $A \in \widetilde{AS}$. We show how to do this in $O(\log w)$ amortized time leading to the overall $O(k \log w/w)$ time bound per character.

Let $A \in \widetilde{AS}$ be a micro TNFA with at most w states. Let L_A be the set of $k_A \leq w$ strings corresponding to the non-ε labeled transitions in A. First, construct $\mathrm{AC}(L_A)$. For each state $s \in \mathrm{AC}(L_A)$ we represent the set $\mathrm{end}(\mathrm{occ}(s))$ as a bit string of length x with a 1 in each position corresponding to the states $\mathrm{end}(\mathrm{occ}(s))$ in A. Each of these bit strings is stored in a single word. Since the total number of states for all AC-automata in $\widetilde{AS}$ is $O(m)$ the total space used is $O(m)$ and the total preprocessing time is $O(m \log \max_{A \in \widetilde{AS}} k_A) = O(m \log w)$. Traversing $\mathrm{AC}(L_A)$ on the string Q takes $O(\log k_A)$ time per character. Furthermore, we can retrieve $\mathrm{end}(\mathrm{occ}(s))$ in step 3 in constant time. We can compute the two $\cap$'s in steps 1 and 3 in constant time by a bitwise & operation. It remains to implement the queue operations. In the following section we show how to implement these in $O(\log k_A) = O(\log w)$ amortized time, giving us a solution using $O(k/w \log w)$ time per character.

4.2.1 Fast Bit Queues

Let F_A be the set of $k_A \leq w$ queues for the strings L_A in A and let $\ell_1, \ldots, \ell_{k_A}$ be the lengths of the queues. We want to support local queue operations with input and output state-sets represented compactly as a bit string. For simplicity, we will identify $\mathrm{start}(L_A)$ with $\mathrm{end}(L_A)$ such that the input and output of the queue operation correspond directly. This is not a problem since the states are consecutive in the mapping and we can therefore translate between them in constant time. If we use the implementation of the queue operations from Section 3 we get a solution using $O(w)$ time per operation. Our goal in this section is to improve this to the following.

LEMMA 4.1. *For a set of $k_A \leq w$ bit queues we can support* Front *in $O(1)$ time and* Enqueue *in $O(\log k_A) = O(\log w)$ amortized time.*

To prove Lemma 4.1 we divide into k'_A short queues of length less than $2k_A$ and k''_A long queues of length at least $2k_A$.

Case 1: Short Queues Let $\ell < 2k_A \leq 2w$ be the maximal length of a short queue. Each short queue fits in a double word. The key idea is to first pretend that all the short queues have the same length ℓ and then selectively move some bits forward in the queues to speed up the shorter queues appropriately. We first explain the algorithm for a single queue implemented within a larger queue and then show how to implement it efficiently in parallel for $O(k_A)$ queues. We assume w.l.o.g. that ℓ is a power of 2.

Consider a single bit queue of length $\ell_j \leq \ell$ represented by a queue of length ℓ. We want to move all bits inserted into the queue forward by $\ell - \ell_j$ positions. To do so we insert $\log \ell$ *jump points* $J^1, \ldots, J^{\log \ell}$. If the ith least significant bit in the binary representation of $\ell - \ell_j$ is 1 the ith jump point J^i is *active*. When we insert a bit b into the back of queue we first move all bits forward and insert b at the back. Subsequently, for $i = \log \ell, \ldots, 2$, if J_i is active we move the bit at position 2^i to position 2^{i-1}. By the choice of jump points it follows that a bit inserted into the queue is moved forward by $\ell - \ell_i$ positions during the lifetime of the bit within the queue.

Next we show how to implement the above algorithm in parallel. First, we represent all of the k'_A bit queues in a combined queue F of length ℓ implemented as a cyclic list. Each entry in F stores a bit string of length k'_A where each bit represents an entry in one of the k'_A bit queues. Each k'_A-bit entry is stored "vertically" in a single bit string. We compute the jump points for each queue and represent them vertically using $\log \ell$ bit strings of length k'_A with a 1-bit indicating an active jump point. Let F^i and F^{i-1} be entries in F at position 2^i and 2^{i-1}, respectively. We can move all bits in parallel from queues with active jump point at position 2^i to position 2^{i-1} as follows.

$$\begin{aligned} Z &:= F^i \mathbin{\&} J^i \\ F^i &:= F^i \mathbin{\&} \neg J^i \\ F^{i-1} &:= F^{i-1} \mid Z \end{aligned}$$

Here, we extract the relevant bits into Z, remove them from F^i, and insert them back into the queue at F^{i-1}. Each of the $O(\log \ell)$ moves at jump points takes constant time and hence the Enqueue operation takes $O(\log \ell) = O(\log k_A)$ time. The Front operation is trivially implemented in constant time by returning the front of F.

Case 2: Long Queues We now consider the k''_A long queues, each of length at least $2k_A \geq 2k''_A$. The key idea is to represent the queues horizontally with additional buffers of size k''_A in the back and front of the queue represented vertically. The buffers allows us to convert between horizontal and vertical representations every k''_A Enqueue operations which we can do efficiently using a fast algorithm for transposition by Thorup [29].

First, we take each long queue L_i and make an individual horizontal representation as a bit string over $\lceil \ell_i / w \rceil$ words. As usual, the representation is cyclic. Second, we add two buffers, one for the back of the queue and one for the front of the queue. These are cyclic vertical buffers like those we used for the short queues, and each has capacity for k''_A bits from each string. When we start, all bits in all buffers and queues are 0. The first k''_A Enqueue operations are very simple. Each just adds a new word to the end of the vertical back buffer. We now apply the word matrix transposition of Thorup [29] so that we for each queue L_i get a word with k''_A bits for the back of the queue. Since $k''_A \leq w$, the transposition takes $O(k''_A \log k''_A)$ time. We can now in $O(k''_A)$ total time enqueue the results to the back of each of the horizontal queues L_i.

In general, the system works in periods of k''_A Enqueue operations. The Front operation simply returns the front word in the front buffer. This front word is removed by the Enqueue operation which also adds a new word to the end of the back buffer. When the period ends, we move the back buffer to the horizontal representation as described above. Symmetrically, we take the front k''_A bits of each L_i and transpose them to get a vertical representation for the front buffer. The total time spent on a period is $O(k''_A \log k''_A)$ which is $O(\log k''_A)$ time per Enqueue operation.

To show Lemma 4.1 we simply partition F_A into short and long queues and apply the algorithm for the above cases. Note that the space for the queues over all micro TNFAs is $O(m)$.

4.3 Summing Up In summary, our algorithm for regular expression matching spends $O(m \log k)$ time and $O(m)$ space on the preprocessing. Afterwards, it processes each character in $O(|\widetilde{AS}| \log w) = O(k/w \log w)$ time. So far this assumes $k \geq w$. However, if $k < w$, we can just use k bits in each word, and with these reduced words, the above bound becomes $O(\log k)$. This completes the proof of Theorem 1.1.

5 Regular Expressions with Black-Boxes

Before addressing character class intervals we first discuss how to extend the special treatment of strings to more general automata within the same time bounds.

Let $B_1, \ldots, B_k$ be a set of black-box automata. Each black-box is represented by a transition in the pruned TNFA and each black-box accepts a single bit input and output. In Section 4.2 each black-box was an automata that recognized a single string and we gave an algorithm to handle $k_A \leq w$ such automata in $O(\log k_A)$ time using our bit queues and a multi-string matching algorithm. In this setting none of the black-boxes accepted the empty string and since all black-boxes were of the same type we could handle all of them in parallel using a single algorithm.

To handle general black-box automata we need to modify our algorithm slightly. First, if B_i accepts the empty string we add a direct ε-transition from the startpoint to the endpoint of the transition for B_i in the pruned TNFA. With this modification the $O(k \log w / w + \log k)$ time Close algorithm from the previous section works correctly. With different types of black-boxes we create a bit mask for the startpoint and endpoints of each type. We can then extract the inputs bits for a given type independently of the other types. As before, the output bit positions can be obtained by shifting the bit mask for the input bits by one position.

With these fixes we can now implement general black-boxes within our algorithm from Section 4.2 efficiently. Specifically, let A be a micro TNFA with $k_A \leq w$ black-box automata of a constant number of types. If we can simulate each type in amortized $O(\log k_A)$ time per operation the total time for regular expression matching with the black-boxes will be dominated by the time for the Close operation and we get the same time complexity as in Theorem 1.1.

In the following section we show how to handle $\leq w$ character class intervals in amortized constant time. Using these as black-boxes as described above this leads to Theorem 1.2.

6 Character Class Intervals

We now show how to support character class intervals $C\{x, y\}$ in the black-box framework described in the previous section. For succinctness we will abbreviate character class intervals by intervals in the following. It is convenient to define the following two types: $C\{= x\} = C\{x, x\}$ and $C\{\leq x\} = C\{0, x\}$. We have that $C\{x, y\} = C\{= x\}C\{\leq y - x\}$.

Recall that we identify each character class C by the set it represents. It is instructive to consider how to implement some of the simple cases of intervals. It is easy to handle $O(w)$ character classes, i.e., intervals of the form $C\{= 1\}$, in constant time [3]. For each character we store a bit string indicating which character classes it is contained in. We can then & this bit string with the input bits and copy the result to the output bits.

Intervals of the form $\Sigma\{=x\}$ are closely related to our algorithm for multi-strings. In particular, suppose that we have a string matcher for the string L_i that determines if the current suffix of Q matches L_i and a black-box for $\Sigma\{=|L_i|\}$. To implement the black-box for L_i we copy the input to $\Sigma\{=|L_i|\}$ and pass the character to the string matcher for L_i. The output for L_i is then 1 if the output of $\Sigma\{= |L_i|\}$ and the string matcher is 1 and otherwise false. In the previous section we implemented $\Sigma\{= |L_i|\}$ and the string matcher using our bit queues of non-uniform length and a standard multi-string matching algorithm, respectively. Hence, to implement $\Sigma\{= x\}$ we may simply use a bit queue of length x.

To implement $C\{= x\}$ we will use a similar approach as above. The main idea is to think of $C\{=x\}$ as the combination of $\Sigma\{=x\}$ and "the last x characters all are in C". To implement the latter we introduce an algorithm for maintaining generic non-uniform counters below. These will also be useful for implementing $C\{\leq x\}$.

Define a *trit counter with reset* (abbreviated trit counter) T with *reset value* ℓ as follows. The counter T stores a value in the range $[0, \ell]$. We want to determine when $T > 0$ under a sequence of the following operations: A *reset* sets $T := \ell$, a *cancel* sets $T := 0$, and a *decrement* sets $T := T - 1$ if $T > 0$.

To implement multiple trit counters using word-level parallelism we code the inputs with trits $\{-1, 0, 1\}$, where 1 is reset, 0 is decrement and -1 is cancel. Each of these trits is coded with 2 bits. The output is given as a bit string indicating which trit counters are positive. We will show how to implement $O(w)$ non-uniform trit counters in constant amortized time in the next section. The main challenge here is that the counters have individual reset values. Before doing this, we first show how implement the character class intervals using this result.

For $C\{\leq x\}$ we use a trit counter T with reset value $x - 1$. For each new character α we give T an input -1 if $\alpha \notin C$. Otherwise, we input the start state of $C\{\leq x\}$ to T. The output of $C\{\leq x\}$ is the output of T. Hence, we output a 1 iff we input a 1 no more than x characters ago and we have not had a character outside of C.

To finish the implementation of $C\{= x\}$ we need to implement "the last x characters all are in C". For this we use a trit counter T with reset value $x - 1$. For each new character α we input a 1 if $\alpha \notin C$ and otherwise we input a 0. The output for $C\{= x\}$ is 1 if the output from $\Sigma\{= x\}$ is 1 and $T = 0$. Otherwise, the output is 0. In other words, the output from $C\{= x\}$ is 1 iff we input a 1 x characters ago and all of the last x characters are in C.

Using standard word-level parallelism operations it straightforward to implement the required interaction between $O(w)$ bit queues and trit counters in constant time. Combined with our amortized constant time algorithm for maintaining $O(w)$ trit counters presented in the next section we obtain an algorithm for regular expression matching with character class intervals using $O(k \log w/w + \log k)$ per character.

6.1 Fast Trit Counters with Reset

Let T_A be a set of k_A trit counters with reset values $\ell_1, \ldots, \ell_{k_A}$. Our goal is to show the following result.

LEMMA 6.1. *We can maintain a set of $k_A \leq w$ of non-uniform trit counters with reset in constant amortized time per operation.*

The general idea to prove Lemma 6.1 is to split the k_A counters into a sequence of subcounters which we can efficiently manipulate in parallel. We first explain the algorithm for a single trit counter T with reset value ℓ. The algorithm will lend itself to straightforward parallelization leading to Lemma 6.1.

The trit counter T with reset value ℓ is stored by a sequence of w subcounters $t_0, t_1, \ldots, t_{w-1}$, where $t_z \in \{0, 1, 2\}$. Subcounter t_i represents the value $t_i 2^i$ and the value of T is $\sum_{0 \leq i \geq 0} t_i 2^i$. Note that the subcounter representation of a specific value is not unique. We define the *initial configuration* of T to be the unique representation of the value ℓ such that a prefix $t_0, \ldots, t_h$ of subcounters are all positive and the remaining subcounters are 0. To efficiently implement cancel operations T will be either in a *active* or *passive* state. The output of T is 1 if T is active and 0 otherwise. Furthermore, at any given time the be at most one *active subcounter* which can be either *starting* or *finishing*.

In each operation we visit a prefix of the subcounters $t_0, \ldots, t_i$. The length of the prefix is determined by the total number of operations. Specifically, in operation o the endpoint i of the prefix is the position of the rightmost 0 in the binary representation of o, i.e., in operation $23 = 10111_2$ we visit the prefix t_0, t_1, t_2, t_3 since the rightmost 0 is in position 3. Hence, subcounter t_i is visited every 2^i operations and hence the amortized number of subcounters visited per operation is constant.

Consider an concrete operation that visits the prefix $t_0, \ldots, t_i$ of subcounters. Recall that h is the largest nonzero subcounter in the initial configuration and let $z = \min(i, h)$.

We implement the operations as follows. If we get a cancel we simply set T to passive. If we get a reset we reset the subcounters $t_0, \ldots, t_{z-1}$ according to the initial configuration. The new active subcounter will be t_z. If $i < h$ we set t_z to starting; otherwise, $z = h$, and we set it to finishing.

If we get a decrement we proceed as follows. If T is passive we do nothing. If T is active let t_a be the active subcounter. If $i < a$ we also do nothing. Otherwise, we decrement t_a by 1. There are two cases to consider:

1. t_a is starting. The new active subcounter is t_z. If $i < h$ we set t_z to starting and otherwise we set it to finishing.

2. t_a is finishing. The new active subcounter is t_b, where t_b is the largest nonzero subcounter among $t_0, \ldots, t_a$. We set t_b to finishing. If there is no such subcounter we set the T to passive.

We argue that the algorithm correctly implement a trit counter with reset value ℓ. We first show that the algorithm never decreases a subcounter below 0 in a sequence of decrements following an activation of the counter.

Initially, the first active subcounter is positive since it is in the initial configuration. When we choose an active subcounter in the decrement operation there are two cases to consider. If the current active subcounter t_a is finishing the algorithm selects the next active subcounter to have a positive value. Hence, suppose that t_a is starting. We have that t_h is never starting. Furthermore, if $a = z$ then t_a cannot be starting. Hence, $a < \min(h, i) = z$ and therefore the new active subcounter t_z is positive. Inductively, it follows that we never make a subcounter negative.

The initial configuration represents the value ℓ. Whenever we decrement a subcounter t_i there has been exactly 2^i operations since we last decremented anything. If we do not get cancel operation input the trit counter becomes passive when all subcounters are 0. It follows that the algorithm correctly implements a trit counter.

It remains to implement the algorithm for $k_A \leq w$ trit counters efficiently. To do so we represent each level of the subcounters vertically in a single bit string, i.e., all the k_A subcounters at level i are stored consecutively in a single bit string. Similarly, we represent the state (passive, active, starting, finishing) of each subcounter vertically. Finally, we store the state (passive or active) of the k_A trit counters in a single bit string. In total we use $O(k_A w)$ bits and therefore the space is $O(k_A)$. With this representation we can now read and write a single level of subcounters in constant time.

The input to the trit counters is given compactly as a bit string and the output is the bit string representing the state of all trit counters. Using a constant number of precomputed masks to compare and extract fields from the representation and the input bit string the algorithm is now straightforward to implement using standard word-level parallelism operations in constant time per visited subcounter level. Since we visit an amortized constant number of subcounter levels in each operation the result of Lemma 6.1 follows.

6.2 Summing Up Combining the result of Lemma 6.1 with our black-box extension of our algorithm from Section 4.2 we obtain an algorithm for regular expression matching with character class intervals that uses $O(k \log w/w + \log k)$ time per character as in the bound from Theorem 1.1. We count the extra space and preprocessing needed for the character class intervals.

Each character class interval $C\{x, y\}$ requires a bit queue of length x and a trit counter with reset value $y - x$. The bit queue uses $O(x)$ space and preprocessing time. The trit counter uses only constant space and we can initialize all trit counters in $O(m)$ time.

Given a character we also need to determine which character classes it belong to. To do this we store for each character α a list of the micro TNFAs that contains α in a character class. With each such micro TNFA we store a bit string indicating the character classes that contain α. Each bit string can be stored in a single word. We keep these lists sorted by the traversal order of the micro TNFAs such that we can retrieve each bit string during the traversal in constant time. Note that in the algorithm for $C\{= x\}$ we also need the complement of C which we can compute by negation in constant time.

We only store a bit string for a character α if the micro TNFA contains a character class containing α and therefore the total number of bit strings is $O(m)$. Hence, the total space for all our lists is $O(m)$. If we store the table of lists by a complete table we use $O(|\Sigma|)$ additional space. For small alphabets this gives a simple and practical solution. To get the bound of Theorem 1.2 we use that the total number of different characters in all characters classes is at most m. Hence, only $O(m)$ entries in the table will be non-empty. With deterministic dictionaries [12] we can therefore represent the table with constant time query in $O(m)$ space after $O(m \log m)$ preprocessing.

In total, we use $O(X + m)$ additional space and $O(X + m \log m)$ additional preprocessing. This completes the proof of Theorem 1.2.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

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

[4] P. Bille. New algorithms for regular expression matching. In *Proc. 33rd ICALP*, 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] P. Bille and M. Thorup. Faster regular expression matching. In *Proc. 36th ICALP*, pages 171–182, 2009.

[7] P. Bucher and A. Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Proc. 2nd ISMB*, pages 53–61, 1994.

[8] K. Fredriksson and S. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance. *Inf. Retr.*, 11(4):335–357, 2008.

[9] K. Fredriksson and S. Grabowski. Nested counters in bit-parallel string matching. In *Proc. 3rd LATA*, pages 338–349, 2009.

[10] 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.

[11] V. M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16(5):1–53, 1961.

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

[13] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The prosite database, its status in. *Nucleic Acids Res*, (27):215–219, 1999.

[14] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Monitoring regular expressions on out-of-order streams. In *Proc. 23nd ICDE*, pages 1315–1319, 2007.

[15] B. Kernighan and D. Ritchie. *The C Programming Language (2nd Ed.).* Prentice-Hall, 1988. First edition from 1978.

[16] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies, Ann. Math. Stud. No. 34*, pages 3–41. Princeton U. Press, 1956.

[17] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[18] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. 27th VLDB*, pages 361–370, 2001.

[19] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Computers*, 9(1):39–47, 1960.

[20] M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo. Structured motifs search. *J. Comput. Bio.*, 12(8):1065–1082, 2005.

[21] M. Murata. Extended path expressions of XML. In *Proc. 20th PODS*, pages 126–137, 2001.

[22] E. W. Myers. Approximate matching of network expressions with spacers. *J. Comput. Bio.*, 3(1):33–51, 1992.

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

[24] G. Myers and G. Mehldau. A system for pattern matching applications on biosequences. *CABIOS*, 9(3):299–314, 1993.

[25] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *J. Comput. Bio.*, 10(6):903–923, 2003.

[26] S. Sen, O. Spatscheck, and D. Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *Proc. 13th WWW*, pages 512–521, 2004.

[27] B. Stroustrup. *The C++ Programming Language: Special Edition (3rd Edition)*. Addison-Wesley, 2000. First edition from 1985.

[28] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11:419–422, 1968.

[29] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42(2):205–230, 2002. Announced at SODA 1997.

[30] L. Wall. *The Perl Programming Language.* Prentice Hall Software Series, 1994.

[31] 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.