TCS-TR-A-10-40

TCS Technical Report

An Efficient Matching Algorithm for Acyclic Regular Expressions with Bounded Depth

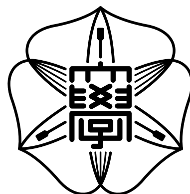
by

Yusaku Kaneta, Shin-ichi Minato, and Hiroki Arimura

Division of Computer Science

Report Series A

February 23, 2010



Hokkaido University
Graduate School of
Information Science and Technology

Email: arim@ist.hokudai.ac.jp

Phone: +81-011-706-7680
Fax: +81-011-706-7680

An Efficient Matching Algorithm for Acyclic Regular Expressions with Bounded Depth

Yusaku Kaneta, Shin-ichi Minato, and Hiroki Arimura

Hokkaido University, N14, W9, Sapporo 060-0814, Japan
{y-kaneta,minato,arim}@ist.hokudai.ac.jp

Abstract. In this paper, we study the regular expression matching problem for a subclass of regular expressions of small depth. A regular expression is *acyclic* if it is over the basis $\Sigma \cup \{\cdot, |\}$. By extending the SHIFT-AND approach by Wu and Manber (JACM 39(2),1992), we present an efficient algorithm that solves the regular expression matching problem for acyclic regular expressions with length m and depth d and an input text of length n in $O(nmd/w)$ time using $O(md)$ preprocessing and $O(md/w)$ space in words on unit-cost RAM model with word length w. When the depth d is constant, typical in real applications, our algorithm runs in $O(nm/w)$ time and $O(m/w)$ space, while the algorithm by Bille (ICALP, 2006) runs in $O(nm \log w/w)$ time and $O(m \log w/w)$ space. Hence, this achieves $O(\log m)$ and $O(\log w)$ speed-ups in *small automata* ($m \leq w$) and in *large automata* ($m > w$) cases, resp., still keeping $O(m/w)$ space.

1 Introduction

Recent emergence of massive text and sequence data in networks has attracted much attention to researches on efficient string processing technologies [1, 3, 4, 6, 7, 9, 12, 13]. In this paper, we study the *regular expression matching problem*, which is one of the most important problems in string processing. This problem is stated as follows: Given a regular expression R of length m and an input text T of length n, to determine if there exists some substring of T that belongs to the language defined by R.

For the regular expression matching problem, in 1968, Thompson [11] devised an efficient algorithm using the idea of simulating a special NFA called a TNFA (Thompson NFA), which is obtained from an input regular expression by a transformation (the 2nd column of Fig. 1). This gives an $O(nm)$ time and $O(m)$ space algorithm for the problem.

In 1992, Myers [6] gave the first improvement of $O(nm)$ time complexity of Thompson's algorithm by presenting an $O(nm/w) = O(nm/\log n)$ time algorithm on the $(\log n)$-uniform RAM model, where $w = O(\log n)$ by using so-called *four Russians technique*, which is a combination of modular decomposition of a TNFA and its simulation using table lookup (the 3rd column of Fig. 1). The space complexity of the algorithm is $O(nm/\log n)$ space (in words), which is still dependent of the text size n [1].

Then, Bille [4] breaks the $O(m)$ space barrier of space complexity by giving algorithms of $O(nm \log w/w)$ time for large automata and $O(n \log m)$ time for small automata using $O(m)$ words space (the 4th column of Fig. 1). His algorithm is based on a hierarchical decomposition of a given TNFA to a balanced binary tree with $O(\log m)$ depth, and simulates the TNFA level by level in a topdown manner using bit-parallelism with Boolean operations and subtraction.

In this paper, we study efficient regular expression matching for unbounded regular expressions with simple structure, large branching, and small depth, typically appearing

[1] For any fixed $0 < \epsilon \leq 1$, we can derive an algorithm with space $O(m2^x) = O(m^{1+\epsilon})$ bits by setting the block length $x = \epsilon \log m$. However, its time complexity become $O(\frac{1}{\epsilon}(nm/\log m))$ with factor $\frac{1}{\epsilon}$.

Algorithm	Pattern	Time	Preprocess	Space (in words)
Thompson [11]	$\mathsf{REG}(\Sigma, \varepsilon, \cdot, \mid, *)$	$O(nm)$	$O(m)$	$O(m)$
Myers [6]	$\mathsf{REG}(\Sigma, \varepsilon, \cdot, \mid, *)$	$O(nm/w)$	$O(nm/w)$	$O(nm/w)$
Bille [4] $(m > w)$	$\mathsf{REG}(\Sigma, \varepsilon, \cdot, \mid, *)$	$O(nm \log w/w)$	$O(m \log w)$	$O(m \log w/w)$
$(m \leq w)$		$O(n \log m)$	$O(m \log m)$	$O(\log m)$
Ours $(m > w)$	$\mathsf{REG}(\Sigma, \cdot, \mid)$	$O(nmd/w)$	$O(md)$	$O(md/w)$
Sec. 3 $(m \leq w)$		$O(nd)$	$O(md)$	$O(d)$

Fig. 1. Summary of the results on regular expression matching algorithms in terms of time T, space S (in words), and preprocessing P for unbounded acyclic regular expressions over constant alphabets in text size n, pattern size m, pattern depth d, and word length w.

in practical applications such as NIDS [9, 10]. As a main result, for the class AREG of unbounded acyclic regular expressions over the basis of letters in Σ, dot '$\cdot$', and union '$|$', we present an algorithm that solves the regular expression matching problem in $O(nmd/w)$ time using $O(md + m|\Sigma|/w)$ preprocessing and $O(md/w + m|\Sigma|/w)$ space in words, where $d \geq 0$ is the depth of a given regular expression. When the depth d is constant, this result gives an $O(\log w)$ time speed up of the $O(nm \log w/w)$ algorithm by Bille [4] still keeping $O(m/w)$ space bound. The key of our algorithm is efficient simulation of the ϵ-closure on TNFA tailored for acyclic regular expressions with small depth. Since our TNFAs are of unbounded branching, we cannot directly apply Bille [4]'s simulation technique. Hence, we introduce new simulation techniques based on an extension of the SHIFT-AND bit-parallel technique [7, 12], called *scatter and gather operations* using word-level parallelism.

The advantages of our approach to regular expression matching are summarized as follows. (i) Simple and efficient: Since our algorithm naturally exploits the composition structure of TNFA and does not use complex module decomposition as in [4] (in the small automata case), it is particularly efficient for regular expressions with small depth. (ii) Hardware friendly: Since it uses only simple bit-operations and addtion/subtraction and avoids the use of table-lookup, it has potential to be implemented on modern parallel hardwares with simple structure, such as GPGPUs or FPGAs. We can even implement these instructions directly on hardwired circuits.

The organization of this paper is as follows. In Section 2, we give basic definitions. In Section 3 and Section 4, we present our algorithm *BPD* (Algorithm with Bit-Parallel Distribution Techniques) for $\mathsf{AREG} = \mathsf{REG}(\Sigma, \cdot, |)$. In Section 5, we report our experimental results and, in Section 6, we conclude.

2 Preliminary

In this section, we give basic definitions and notations in the regular expression matching problem according to [4, 6]. For detailed definitions, please consult text books [2, 5, 7].

2.1 Regular expressions and the matching problem

Basic definitions. Let Σ be a finite alphabet of *letters*. A *string* on Σ is a sequence $S = s_1 \cdots s_n$ of letters, where $S[i] = s_i \in \Sigma$ for every $1 \leq i \leq n$. We denote by $S[i..j]$ the substring $s_i \cdots s_j$ for every $i \leq j$, and by ε the *empty string*. For a set $S \subseteq \Sigma^*$ of strings, we denote by $|S|$ the cardinality and by $||S|| = \sum_{s \in S} |s|$ the total size of S. We denote by Σ^* and $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$ the sets of all strings and all non-empty strings on Σ.

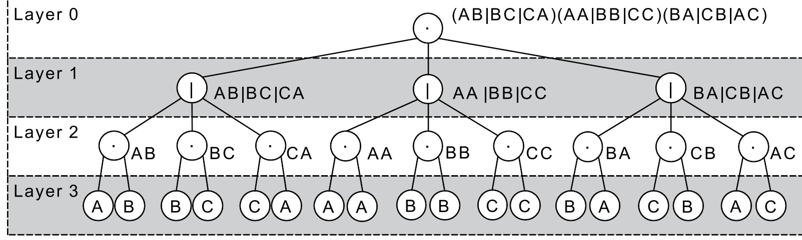


Fig. 2. The parse tree for an acyclic regular expression $R_1 = (AB|BC|CA)(AA|BB|CC)(BA|CB|AC)$.

Regular expressions. The class of *regular expressions* on Σ is defined recursively as follows: (1) If $a \in \Sigma \cup \{\varepsilon\}$ then a is a regular expression. (2) If $R_1, \ldots, R_n$ are regular expressions then $(R_1 \cdots R_n)$, $(R_1| \cdots |R_n)$, $(R_1)^+$, and $(R_1)^*$ are regular expressions. In the above definition, regular expressions are *unbounded*, i.e., $n \geq 1$ can be any integer, while $n = 2$ in the standard definition [2, 4, 7]. The *length* (or the *size*) of R is defined by the number of symbols of $\Sigma \cup \{\varepsilon, \cdot, |, *, +\}$ appearing in R. A regular expression R defines the *language* $L(R) \subseteq \Sigma^*$ in the standard way [2, 7].

Acyclic regular expressions. As notation, for any subset $\mathcal{O} \subseteq \{\Sigma, \varepsilon, \cdot, |, *, +\}$, we denote by $\mathsf{REG}(\mathcal{O})$ the subclass of REG of all regular expressions constructed from the letters and the operators in $\mathcal{O}$. Then, we denote by $\mathsf{REG} = \mathsf{REG}(\Sigma, \varepsilon, \cdot, |, *) = \mathsf{REG}(\Sigma, \varepsilon, \cdot, |, *, +)$ the class of regular expressions. In this paper, we introduce a subclass of REG, the class $\mathsf{AREG} = \mathsf{REG}(\Sigma, \cdot, |)$ of *acyclic* regular expressions[2], The *depth* of R, $d(R)$, is defined by: $d(a) = 0$ for $a \in \Sigma \cup \{\varepsilon\}$. $d((R_1 \cdots R_n)) = d((R_1| \cdots |R_n)) = 1 + \max_i d(R_i)$, and $d((R_1)^+) = d((R_1)^*) = 1 + d(R_1)$. For every $d \geq 0$, we denote by AREG^d the subclass consisting only of expressions with depth no more than d.

Example 1. $R_1 = ((AB|BC|CA)(AA|BB|CC)(BA|CB|AC)) \in \mathsf{AREG}^3$ is an acyclic regular expression with depth $d = 3$. $R_2 = (A|B|C|D) \in \mathsf{AREG}^1$ and $R_3 = ((A|B)|(C|D)) \in \mathsf{AREG}^2$ are acyclic regular expression with depth 1 and depth 2, respectively.

Parse trees. The *parse tree* for a regular expression R is a labeled, rooted, and ordered tree T_R with node set $dom(T_R)$ satisfying: (1) The leaves are labeled with symbols $lab(v)$ from $\Sigma \cup \{\varepsilon\}$. (2) The internal nodes are labeled with operators $lab(v)$ from $\{\cdot, |, *, +\}$. (3) The internal nodes labeled with $*$ or $+$ have one child, and those labeled with . and $|$ have unbounded number of children. We denote the *depth* of node v by $d(v)$ and the *depth* of T_R by $d(T_R)$. Parse trees have one-to-one correspondence to regular expressions in a standard manner [5]. Fig. 2 shows an example of the parse tree T_{R_1} for R_1 in Example 1.

Non-deterministic finite automata. We use non-deterministic finite automata for recognizing the strings in a regular language. A *non-deterministic finite automaton* (*NFA*, for short) is a quadruple $N = (V, E, \theta, \phi)$, consists of the followings: (1) A set V of nodes, called *states*. (2) A set $E \subseteq V \times (\Sigma \cup \{\varepsilon\}) \times V$ of labeled, directed *edges*. (3) Distinguished nodes θ and $\phi \in V$, called the *source* and the *sink* of N. Each element $(x, a, y) \in E$, where $a \in \Sigma \cup \{\varepsilon\}$, is called an *a-edge* or a *a-transition* from state x to state y. We define a *path* as a connected sequence π of edges. A path π *spells* a string s if s is obtained by concatenating the labels of the edges in π. The *language accepted by* N is the set $L(N) \subseteq \Sigma^*$ of all strings spelled on the paths from the source θ to the sink ϕ in N.

[2] Note that this is different from the notion of *acyclic regular languages* [8], which is the class of languages defined by *extended* regular expressions without "$*$" and "$+$", but with complement $\overline{\cdot}$.

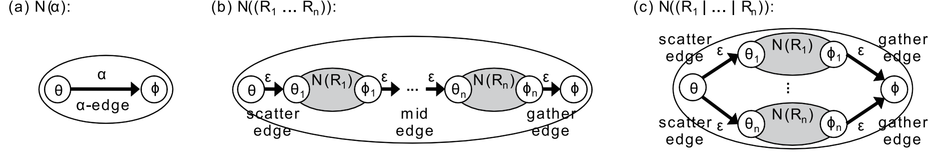


Fig. 3. Construction of Thompson automata (TNFAs).

2.2 Regular expression matching problem

A *pattern* is any regular expression and a *text* of length n is a string T over Σ. We say that a regular expression R *occurs* in a text T if there exist some $i \leq j$ such that $T[i..j] \in L(R)$ holds. Then, the index j is called the *end position* of R in T. Let $\mathcal{C} \subseteq \mathsf{REG}$ be any subclass of REG. Now, we state our problem below.

Definition 1. *The regular expression matching problem for* $\mathcal{C}$ is, given a regular expression $R \in \mathcal{C}$ of length m and a text $T = t_1 \cdots t_n$ of length n, to output the set of all end positions of R in T.

As the model of computation, we assume the *uniform-cost RAM* with word length $w \geq \log n$ [5]. Let $M \geq 0$ be a positive integer. A *bitmask* is a vector of M bits $X = b_M \cdots b_1 \in \{0,1\}^M$. If $M \leq w$ we say the *small automata case*, and if $M > w$ we say the *large automata case* [4]. For a bit $b \in \{0,1\}$, b^k denotes the bitmask consisting of m copies of b. We assume that for bitmasks with $M \leq w$, Boolean and arithmetic operations *bitwise-and* " & ", *bitwise-or* " | ", *bitwise-not* "∼", *left-shift* "<<", *right-shift* ">>", *addition* "+" and *subtraction* "−" are executed in $O(1)$ time. The space complexity is measured in the number of computer words.

3 Efficient State-set Simulation of Thompson Automata

In this section, we review Thompson's algorithm for regular expression matching based on the state-set simulation of a special class of automata, called TNFAs. Next, we discuss how to speed-up the simulation of TNFA.

3.1 Construction of Thompson automata

Thompson [11] gave a method to construct a NFA in a special form, called a TNFA, from a regular expression R that exactly accepts the language $L(R)$. The classification of ε-edges into scatter, gather, mid, and back edges below is not contained in the original [11] and newly introduced in this paper.

Let $R \in \mathsf{AREG}$ be any acyclic regular expression and T_R be the parse tree for R. For each node v, let $R(v)$ be the regular expression that corresponds to the subtree $T_R(v)$ rooted at v. The following method recursively constructs the TNFA $N(v) = N_R = (V(v), E(v), \theta(v), \phi(v))$ for each node v by traversing the parse tree T_R in the post-order. During the traversal, we construct the following auxiliary information for each node v: the source $\theta(v)$ and the sink $\phi(v)$; the sets of edges $LE(v), SE(v), GE(v)$, and $ME(v)$ whose members are called *letter, scatter, gather*, and *mid* edges; First, we suppose that v is a leaf of T_R.

(a) For $R(v) = \alpha \in \Sigma$ with $lab(v) = \alpha$, we build TNFA $N(v) = N_\alpha = (\{\theta, \phi\}, LE(v), \theta, \phi)$ by connecting $\theta = \theta(v)$ and $\phi = \phi(v)$ with letter-edge in $LE(v) = \{(\theta, \alpha, \phi)\}$ as in (a) of Fig 3.

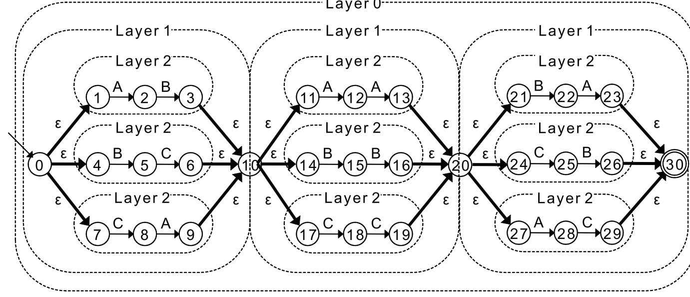


Fig. 4. The TNFA for an acyclic regular expression $R_1 = (AB|BC|CA)(AA|BB|CC)(BA|CB|AC) \in$ AREG.

Next, we suppose that v is an internal node with n children $v_1, \ldots, v_n$ for $n \geq 1$. For every $1 \leq i \leq n$, we assume that a TNFA $N_{R_i} = (V_i, E_i, \theta_i, \phi_i)$ with source $\theta_i = \theta(v_i)$ and sink $\phi_i = \phi(v_i)$ is already built for $R_i = R(v_i)$.

(b) For concatenation $R(v) = (R_1 \cdots R_n)$ with $lab(v)$ = '$\cdot$', we build TNFA $N(v) = N_{(R_1 \cdots R_n)}$ by adding new source $\theta = \theta(v)$ and sink $\phi = \phi(v)$, and by connecting subautomata $R_1, \ldots, R_n$ in serial by edges in $SE(v) = \{(\theta, \varepsilon, \theta_1)\}$, $ME(v) = \{ (\phi_i, \varepsilon, \theta_{i+1}) \,|\, 1 \leq i \leq n-1 \}$, and $GE(v) = \{(\phi_n, \varepsilon, \phi)\}$ as in (b) of Fig 3.

(c) For union $R(v) = (R_1|\cdots|R_n)$ with $lab(v)$ = '|', we build TNFA $N(v) = N_{(R_1|\cdots|R_n)}$ by adding new source $\theta = \theta(v)$ and sink $\phi = \phi(v)$, and by connecting subautomata $R_1, \ldots, R_n$ in parallel by edges in $SE(v) = \{ (\theta, \varepsilon, \theta_i) \,|\, 1 \leq i \leq n \}$ and $GE(v) = \{ (\phi_i, \varepsilon, \phi) \,|\, 1 \leq i \leq n \}$ as in (c) of Fig 3.

(d) In the above definition, if any of the sets $LE(v)$, $SE(v)$, $GE(v)$, $ME(v)$, and $BE(v)$ is undefined, then we define that it is empty. We also define $\Psi(v) = \{\theta(v), \phi(v)\}$ and $\Delta(v) = LE(v) \cup SE(v) \cup GE(v) \cup ME(v) \cup BE(v)$. Then, we define the node set by $V(v) = \Psi(v) \cup (\bigcup_{i=1}^{n} \Psi(v_i))$ and the edge set by $E(v) = \Delta(v) \cup (\bigcup_{i=1}^{n} E_i)$.

In Fig. 4, we show the TNFA for the acyclic regular expression R_1 of Example 1.

3.2 Efficient State-set Simulation

Most of the previous regular expression matching algorithms [3, 4, 6, 7, 12] employ the standard *state-set simulation* method developed by Thompson [5, 11]. In this method, an algorithm first builds TNFA N_R of a given regular expression R, and then simulates the computation by N_R on text T using an appropriate data structure.

Let R be a regular expression, and let $N_R = (V_R, E_R, \theta_R, \phi_R)$ be the TNFA for R constructed in Section 3.1. In Thompson's state-set simulation [11], the current status of the TNFA N_R is represented by a set $S \subseteq V_R$ of *active states*. Let $N = N_R$ be the TNFA and $V = V_R$ be its state set. For any set $S \subseteq V$ of active states and any letter $\alpha \in \Sigma$, we define the following operations:

- Init_N returns the initial state set $\{\theta_R\}$ and Accept_N returns the sink state set $\{\phi_R\}$.
- $\mathsf{Move}_N(S, \alpha)$ returns the set of all states that is reachable from a current active state by a single α-transition in E. That is, $\mathsf{Move}_N(S, \alpha) = \{ y \in V \,|\, x \in S, (x, \alpha, y) \in E_R \}$.
- $\mathsf{EpsClo}_N(S)$ returns the set of all states that is reachable from a current active state by one or more ε-transitions in E. That is, $\mathsf{EpsClo}_N(S) = \{ y \in V \mid x \in S, (x_1, \varepsilon, x_2), (x_2, \varepsilon, x_3), \ldots, (x_{n-1}, \varepsilon, x_n) \in E_R \}$, and called the ε-*closure* of S.

```
algorithm RunTNFA(T = t_1 ··· t_n: a text, N: a TNFA)
1: X ← Init_N; //The initial state set
2: X ← EpsClo_N(X); //Simulate ε-transition
3: for i ← 1, ..., n do
4:   X ← Move_N(X, t_i); //Simulate α-transition
5:   X ← EpsClo_N(X); //Simulate ε-transition
6:   if X ∩ Accept_N ≠ ∅ then
7:     output Match; //A match found
8: end for
```

Fig. 5. The algorithm RunTNFA for scanning phase.

```
procedure EpsClo(S : state set):
1: for k ← d(R), ..., 0 do S ← Apply[GE[k]](S);
2: for k ← d(R), ..., 0 do S ← Apply[ME[k]](S);
3: for k ← 0, ..., d(R) do S ← Apply[SE[k]](S);
4: return X;
```

Fig. 6. The algorithm EpsClo for computing ε-closure for acyclic regular expressions

In Fig. 5, we show an algorithm RunTNFA that simulates the computation of TNFA N on an input text T. We can show that the following lemma [11].

Lemma 1 (Thompson [11]). *For any input text $T = t_1 \cdots t_n$, the algorithm* RunTNFA *in Fig. 5 correctly solves the regular expression matching problem for* REG.

Efficient implementation of Move_N is not hard either by using *table lookup* [6] or using *SHIFT-AND* bit-parallel technique [12]. However, an efficient implementation of EpsClo_N is not straightforward since we have to compute the transitive closure of ε-edges.

The key of our algorithm is an efficient implementation of EpsClo_N by hierarchical decomposition. We partition the ε-edges of TNFA N_R into a collection of mutually disjoint layers as follows. Let $V_R = V[1] + \cdots + V[k]$ be the partition of the node set V_R by their depths, where $V[k] = \{ v \in V_R \mid d(v) = k \}$ for every *level* $1 \le k \le d(T_R)$. $V[k]$ is called the k-th layer of V_R. We define the sets $LE[k], SE[k], GE[k], ME[k]$, and $BE[k]$ for level k as follows. For $Z \in \{LE, SE, GE, ME, BE\}$, the k-th layer $Z[k]$ of the edge set Z is define by $Z_k = \{ e \mid e \in Z(v), v \in V[k] \}$. For any subset $F \subseteq E_R$ of ε-edges and any set $S \subseteq V_R$ of active states, we define the operation $\mathsf{Apply}[F](S)$ by the set of all states reachable from a current active state by zero or one ε-transition in F. In Fig. 6, we present an algorithm that computes ε-closure for acyclic regular expressions in $\mathsf{AREG} = \mathsf{REG}(\Sigma, \varepsilon, \cdot, |)$.

Lemma 2. *Let R be any unbounded acyclic regular expression with depth $d = d(R)$ and N_R be its TNFA. For any state set $S \subseteq V_R$, the algorithm* EpsClo *of Fig. 6 correctly computes the ε-closure* $\mathsf{EpsClo}(S)$ *of S on N_R.*

Proof. We can see that if R is acyclic then for any subexpression R' with source θ' and sink ϕ', there is no ε-path from θ' to ϕ'. Furthermore, we can see that if π is any ε-path in the TNFA then it has the form $\pi = (g_p, g_{p-1}, \ldots, g_r, m_r, s_r, s_{r+1} \ldots, s_q)$, first going up by gather edges $g_p, g_{p-1}, \ldots, g_r \in GE[k]$, reaching a mid edge $m_r \in ME[r]$, and then going down by scatter edges $s_r, s_{r+1} \ldots, s_q \in SE[k]$, such that their depths satisfies $0 \le r \le \min\{p, q\} \le d(R)$. Then, we see that EpsClo simulates the transition along π. □

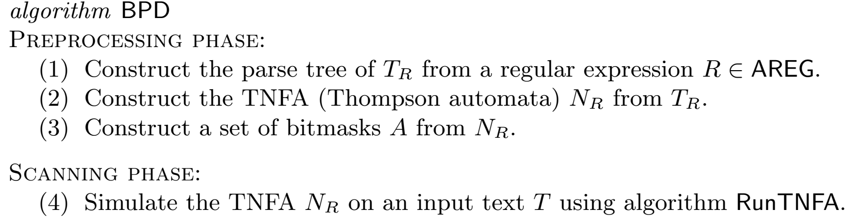
4 An efficient algorithm for acyclic regular expressions

4.1 Outline of the algorithm

In this section, we present an algorithm BPD (*Bit-Parallel Algorithm using Bit-Distribution*) that efficiently solves the regular expression matching problem for acyclic regular expressions in AREG. For an acyclic regular expression $R \in \mathsf{AREG}$ with length m and depth d and

a text T with length n, this algorithm runs in $O(nmd/w)$ time using $O(md)$ preprocessing and $O(md/w)$ space in words on unit-cost RAM with word length w.

Our matching algorithm BPD consists of a preprocessing phase that transform an input regular expression $R \in$ AREG into a number of tables followed by a scanning phase that search for the occurrences of R in an input text T in the following steps.



algorithm BPD

PREPROCESSING PHASE:

(1) Construct the parse tree of T_R from a regular expression $R \in$ AREG.
(2) Construct the TNFA (Thompson automata) N_R from T_R.
(3) Construct a set of bitmasks A from N_R.

SCANNING PHASE:

(4) Simulate the TNFA N_R on an input text T using algorithm RunTNFA.

Fig. 7. The outline of our regular expression matching algorithm BPD

4.2 Preprocessing phase: Computation of bitmasks

In this subsection, we will describe the preprocessing phase. The key of our algorithm is efficient bit-parallel simulation of ε-closure $\mathsf{EpsClo}_A(X)$. Given a regular expression R, the preprocessing phase first builds the parse tree T_R, and then the TNFA N_R from T_R. The algorithm then builds a set of bitmasks from N_R using the construction described below.

Numbering of states. For the TNFA N_R with M states, our algorithm uses M-bit integer $X \in \{0,1\}^M$ as a bitmask with length M to encode any subset of V_R. To do this, we assign a unique bit-position $1 \le Bit(x) \le M$ to each state $x \in V_R$ by the depth-first traversal of D_R, where the source $\theta(v)$ and the sink $\phi(v)$ are numbered at node v in the pre-order and the post-order, respectively. For any state set $S \subseteq V_R$, define $BIT(S) = \{ Bit(x) \mid x \in S \}$. For any subset $X \subseteq [1, M]$, define $NUM(X)$ to be the bitmask that represents X. Then, it is ensured that for each node v, the states subautomata $N(v)$ occupies a continuous interval $I(v) = [Bit(\theta(v)), Bit(\phi(v))] \subseteq [1, M]$, in a bitmask. For distinct $u, v \in V[k]$ in the same level k, their intervals do not overlap, i.e., $I(u) \cap I(v) = \emptyset$.

Construction of bitmasks. For any acyclic regular expression $R \in$ AREG with depth $d = d(R) = d(T_R)$ and its TNFA N_R, we construct a number of bitmasks in the preprocessing phase as follows.

(1) *Alpha-edge transition*: For $\alpha \in \Sigma$,
$$Chr[\alpha] = |_{v \in dom, lab(v) = \alpha} NUM(Bit(\phi(v))).$$

(2) *Scatter-edge transition*: For $k \in [0, d]$,
$$SSrc[k] = |_{v \in V[k]} NUM(Bit(\theta(v))).$$
$$SDst[k] = |_{v \in V[k], (\theta = \theta(v), \varepsilon, \theta') \in SE(v)} NUM(Bit(\theta')).$$
$$SBlk[k] = |_{v \in V[k]} NUM(Bit(\phi(v))).$$

(3) *Gather-edge transition*: For $k \in [0, d]$,
$$GSrc[k] = |_{v \in V[k], (\phi', \varepsilon, \phi = \phi(v)) \in GE(v)} NUM(Bit(\phi')).$$
$$GDst[k] = |_{v \in V[k]} NUM(Bit(\phi(v))).$$
$$GBlk[k] = |_{v \in V[k]} NUM([Bit(\theta(v)), Bit(\phi(v)) - 1]).$$

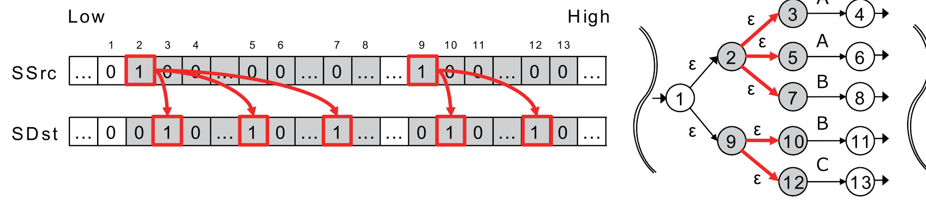


Fig. 8. Scatter transitions.

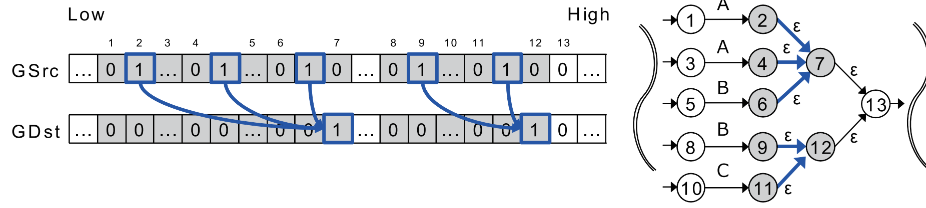


Fig. 9. Gather transitions.

(4) *Mid-edge transition*: For $k \in [0, d]$,

$$Mid[k] = |_{v \in V[k], (\theta', \varepsilon, \phi') \in ME(v)} NUM(Bit(\phi')).$$

4.3 Scanning phase: Simulating TNFA in bit-parallel computation

The scanning phase simulate the behavior of the TNFA N_R in a standard state-set simulation by bit-parallel computation using previously constructed bitmasks. This is done as follows. Let S be any state set and $X = NUM(BIT(S))$ be the corresponding bitmask. Then, we define a number of procedures for state transition as follows.

(1) *Alpha-edge transition*: For $\alpha \in \Sigma$, define $\mathsf{Alpha}(X, \alpha)$ by

$$X \leftarrow (X << 1) \ \& \ Chr[\alpha] \tag{1}$$

This is same as the SHIFT-AND method for exact string matching [3, 12]. For any letter-edge $(x, \alpha, y) \in LE(v)$, $Bit(y) = Bit(x) + 1$ by construction, Thus, the above code correctly implements $\mathsf{Move}(S)$.

(2) *Scatter-edge transition*: For $k \in [0, d]$, define $\mathsf{Scatter}(X, d)$ by

$$X \leftarrow (X \mid (SBlk[d] - (X \ \& \ SSrc[d])) \ \& \ SDst[d]) \tag{2}$$

We explain the behavior of $\mathsf{Scatter}(X, d)$ for the $N(v)$ for each $v \in V[k]$. Let $SE(v) = \{\ (\theta, \varepsilon, \theta_i) \mid 1 \le i \le n\ \}$ for some $n \ge 1$. Firstly, $Y = X \ \& \ SSrc[d]$ obtains the bit at the source position $Bit(\theta)$, which is the lowest in $I(v)$, in the current state set X. $SBlk[d]$ is the bitmask that only the highest bit of $I(v)$ is set to on and the others remain off. Subtracting Y from $SBlk[d]$, $Z = SBlk[d] - Y$, results that all the bits in $[Bit(\theta) + 1, Bit(\phi) - 1]$ turn on if the source bit $Bit(\theta)$ of Y is on, and all bits turn off otherwise. Thus, taking destination mask $SDst[d]$ and then take bitwise-or to X, $X' = (X \mid (Z \ \& \ SDst[d]))$, turns all bits but the destination bits off. In summary, the above steps correctly copy the source bit in X to all destination bits in X'.

(3) *Gather-edge transition*: For $k \in [0, d]$, define $\mathsf{Gather}(X, d)$ by

$$X \leftarrow (X \mid (GBlk[d] + (X \,\&\, GSrc[d])) \,\&\, GDst[d]) \quad (3)$$

Since this is similar to the case for Scatter except that Gather uses addition, while Scatter uses subtraction, we omit the details.

(4) *Mid-edge transition*: For $k \in [0, d]$, define $\mathsf{Mid}(X, d)$ by

$$X \leftarrow (X \mid (X << 1) \,\&\, Mid[k]) \quad (4)$$

For mid edge $(\phi_i, \varepsilon, \theta_{i+1}) \in ME(v)$, $Bit(\theta_{i+1}) = Bit(\phi_i) + 1$, and it works correctly.

We can show the next lemma that says that the above bit-parallel method correctly implements α-transitions and ε-transitions of all types necessary for our similation of $TNFA$.

Lemma 3. *Let $R \in \mathsf{AREG}$ and N_R be the TNFA for R. For any state set S, the bitmask $X = NUM(BIT(S))$ for S, and for any $1 \leq d \leq d(R)$: (1) $X = \mathsf{Alpha}(X, \alpha)$ if and only if $S = \mathsf{Move}(S, \alpha)$. (2) $X = \mathsf{Scatter}(X, d)$ if and only if $S = \mathsf{Apply}[SE[d]](S)$. (3) $X = \mathsf{Gather}(X, d)$ if and only if $S = \mathsf{Apply}[GE[d]](S)$. (3) $X = \mathsf{Mid}(X, d)$ if and only if $S = \mathsf{Apply}[ME[d]](S)$. Furthermore, if $M \leq w$ then the procedures Alpha, Scatter, Gather, and Mid run in $O(1)$ time on uniform-cost RAM.*

4.4 Correctness and complexities

Now, we have the algorithm BPD of Fig. 7 combined with RunTNFA of Fig. 5, EpsClo of Fig. 6, and codes for Alpha, Scatter, Gather, and Mid in the previous subsection. In the followings, $R \in \mathsf{AREG}$ is an acyclic regualr expression with length m and depth d, T is an text with length n, M is the length of bitmasks, and w is the bit-width of a computer word.

In the *small automata* case with $M \leq w$, we have the main results of this paper from Lemma 1, Lemma 2, and Lemma 3.

Theorem 1 (case for small automata). *Suppose that $m \leq w$. Given acyclic regular expression $R \in \mathsf{AREG}$ and input text T with length n, The algorithm BPD of Fig. 7 solves the regular expression matching problem in $O(nd)$ time using $O(md + |\Sigma|)$ preprocessing and $O(d + |\Sigma|)$ space in words.*

In the case *large automata* case with $M > w$, we can show the following theorem by combining our Theorem 1 and Lemma 5 of [4].

Corollary 1 (case for large automata). *Suppose that $m > w$. For acyclic regular expressions in AREG, the regular expression matching problem can be solved in $O(nmd/w)$ time using $O(md + m|\Sigma|/w)$ preprocessing and $O(md/w + m|\Sigma|/w)$ space in words.*

Corollary 2 (bounded depth). *For every $d \geq 0$, the regular expression matching problem for the class AREG^d of acyclic regular expressions with depth d is solved in $O(nm/w)$ time using $O(m)$ preprocessing and $O(m/w)$ space in words, where w is the bit-width of a word.*

5 Experimental Results

In this section, we give the experimental results for the algorithm *BPD*.

Experimental settings. For input texts, we used a DNA sequence data with length 10MB and alphabet size $|\Sigma| = 4$, which is a copy of a part of E.coli in the Canterbury Corpus [3]. For patterns, we used a regular expression, called (K, L)-CNF-Regexp,

[3] http://corpus.canterbury.ac.nz/

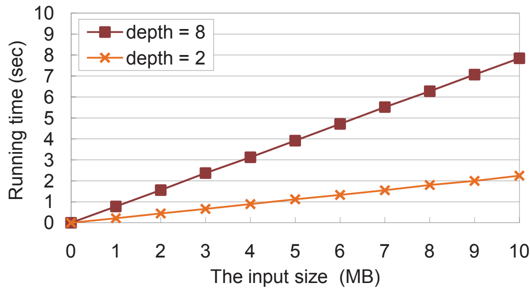


Fig. 10. Running time for the input size.

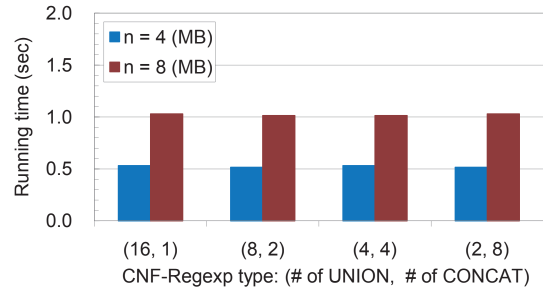


Fig. 11. Running time for CNF-Regexp.

with depth $d = 2$ of the form $R = (R_1^1|\cdots|R_K^1)\cdots(R_1^L|\cdots|R_K^L)$ varying parameters by $(K, L) \in \{(16,1),(8,2),(4,4),(2,8)\}$, where each R_i^j is a string of length 4. The patterns have constant depth d and almost constant size m regardless their parameters (K, L). We implemented the proposed algorithm *BPD* in C++. All experiments were run on a PC (Intel Core 2 Duo CPU P8600, 2.40 GHz, 32bit words, 4GB RAM, Windows Vista, g++4.3.4 on Cygwin), where an input text is stored on the main memory and the cpu time does not include I/O.

Experiment A. Fig. 10 shows the running time of *BPD* by varying the input size $n = 1 \sim 10$MB for patterns with depth $d = 2$ and 8 by explicitly inserting parentheses to the orginal pattern. Then, the running time seems to be linear in the input size n, and also, increased with the depth d.

Experiment B. Fig. 11 shows the running time of *BPD* by varying combination of (K, L) keeping the depth d and the size m of R constant, where the input sizes n are 4MB and 8MB. Then, the running time seems to be independent of (K, L) and the shape of patterns when d and m are constant. These observations are consistent to theoretical analysis of algorithm *BPD* in Section 4.

6 Conclusion

In this paper, we presented a fast matching algorithm for a subclass of the regualr expressions. For the subclass, our algorithm *BPD* solves the regular expression mathcing problem in $T = O(nmd/w)$ time, $S = O(md/w)$ space in words, and $P = O(md)$ preprocessing time with finite alphabet Σ where d is the depth of regular expression w is the bit-width of a computer word. If d, b, and $|\Sigma|$ are constants, *BPD* runs in $O(nm/w)$ time and $O(m/w)$ space in words.

In this paper, we considered a restricted subclass AREG of REG, called acyclic regular expressions without backward edges. Unfortunately, the scatter and gather techniques are not directly applicable to TNFA with backward edges. Extension of the result of this paper for subclasses of regular extensions allowing Kleene-plus and Kleene-star is an interesting research topic. This will be discussed elsewhere.

References

1. J. Agrawal, Y. Diao, D. Gyllstrom, N. Immerman. Efficient pattern matching over event streams. *Proc. ACM SIGMOD'08*, pp. 147–160, 2008.
2. A. V. Aho. Algorithms for Finding Patterns in Strings. *Handbook of Theoretical Computer Science*, Vol. A, pp. 255–300, 1990.
3. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, Vol. 35, No. 10, pp. 74–82, 1992.
4. P. Bille. New algorithms for regular expression matching. *In Proc. of the 33rd International Colloquium on Automata, Languages and Programming(ICALP)*, pp. 643–654, 2006.
5. A. V. Aho, J. E. Hopcroft, J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
6. E. W. Myers. A four-russian algorithm for regular expression pattern matching. *Journal of the ACM*, Vol. 39, No. 2, pp. 430–448, 1992.
7. G. Navarro, and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences.* Cambridge University Press 2002.
8. D. Perrin, Finite Automata, *Handbook of Theoretical Computer Science*, Vol. A, pp. 1–57, 1990.
9. R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, pp. 227–238, 2001.
10. Snort. `http://www.snort.org/`
11. K. Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, Vol. 11, No. 6, pp. 419–422, 1968.
12. S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, Vol. 35, No. 10, pp. 83–91, 1992.
13. H. Yamamoto. A New Recognition Algorithm for Extended Regular Expressions. *Proc. ISAAC 2001*, pp. 257-267, 2001.