TCS-TR-A-10-47

TCS Technical Report

Fast Bit-Parallel Matching for Network and Regular Expressions

by

Yusaku Kaneta, Shin-ichi Minato and Hiroki Arimura

Division of Computer Science

Report Series A

November 12, 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

Fast Bit-Parallel Matching for Network and Regular Expressions

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 extend the SHIFT-AND approach by Baeza-Yates and Gonnet (CACM 35(10), 1992) to the matching problem for network expressions, which are regular expressions without Kleene-closure and useful in applications such as bioinformatics and event stream processing. Following the study of Navarro (RECOMB, 2001) on the extended string matching, we introduce new operations called Scatter, Gather, and Propagate to efficiently compute ε-moves of the Thompson NFA using the Extended SHIFT-AND approach with integer addition. By using these operations and a property called the bi-monotonicity of the Thompson NFA, we present an efficient algorithm for the network expression matching that runs in $O(ndm/w)$ time using $O(dm)$ preprocessing and $O(dm/w)$ space, where m and d are the length and the depth of a given network expression, n is the length of an input text, and w is the word length of the underlying computer. Furthermore, we show a modified matching algorithm for the class of regular expressions that runs in $O(ndm \log(m)/w)$ time.

1 Introduction

Recent emergence of massive text and sequence data in networks has attracted much attention to string processing technologies [1, 3, 4, 11, 14, 16, 18]. In this paper, we study the *regular expression matching problem*, which is one of the most important problems in string processing. Especially, for the last decades, approaches based on efficient NFA simulation have been extensively studied for restricted subclasses of regular expressions, namely, the four-russian approach for the class REG of regular expressions [4, 11]; the SHIFT-AND approach for the class STR of strings [3, 18], and the SHIFT-ADD approach for the classes of k-mismatch string patterns [3, 8]. In particular, Navarro and Raffinot [13, 14] presented efficient bit-parallel approach, called *Extended SHIFT-AND approach* tailored to a restricted but useful subclass EXT of *extended string patterns*, which are regular expressions in linear form, such as $R_0 = (\texttt{[AB]}^+)(\texttt{B.}\{\texttt{1,3}\})(\texttt{[BC]?})(\texttt{.}^*)\texttt{C}$, that consists of letters $a \in \Sigma$, wildcards ".", classes of letters $\alpha = [ab\cdots]$, optional letters $\alpha?$, bounded repeats $\alpha\{x, y\}$, and unbounded repeats α^* and α^+, where $\alpha \subseteq \Sigma$. In this approach, Navarro and Raffinot [14] nicely extended the original approach of [3, 18] by introducing a new bit-parallel simulation technique, called the propagation, with the use of integer addition "+" (or subtraction "−") in addition to usual Boolean operations on RAM to deal with a special case of ε-closure caused by optional letters $\alpha?$ and bounded repeats $\alpha\{x, y\}$ in extended string patterns as well as unbounded repeats α^* with the use of an extended letter mask.

In this paper, inspired by the work by Navarro and Raffinot [14], we study the pattern matching problem for a special class NET of regular expressions, called *network expressions*, which are introduced in Myers [12]. A network expression (over strings) in NET is a regular expression without Kleene-closure, that is, an expression constructed recursively from strings in STR applying ε-edges, concatenation,

and union. Similarly, we can define the class EXNET of *extended network expressions*, which are network expressions over extended string patterns in EXT. For example, $R_1 = \mathtt{A(BA|CD)(CD|AB)B}$ and $R_2 = \mathtt{A(AB|B?)(B?.^*|AB)C}$ are examples of expressions in NET and EXNET, respectively. Network expressions and extended network expressions are widely used in applications in the various fields including such as bioinformatics [12], event stream processing [1], and network intrusion detection systems [16].

As main results in this paper, we show the followings. Let $RAM(op)$ denote a unit-cost random access machine equipped with a set op of arithmetic operations in addition to the standard Boolean operations "&", "|", "$\sim$", and "$\oplus$". We present an efficient algorithm that solves the regular expression matching problem for the classes NET and EXNET in $O(nd\lceil m/w\rceil)$ time using $O(dm + |\Sigma|\lceil m/w\rceil)$ preprocessing and $O(d\lceil m/w\rceil + |\Sigma|\lceil m/w\rceil)$ space on $RAM(+)$, where Σ is a fixed alphabet, m and d are the length and the depth of an input expression R, and n is the length of an input text T over Σ. Furthermore, we show that the regular expression matching problem for the full class REG can be solved in $O(nd\lceil m/w\rceil \log m)$ time using $O(dm \log m + |\Sigma|\lceil m/w\rceil)$ preprocessing and $O(d\lceil m/w\rceil \log m + |\Sigma|\lceil m/w\rceil)$ space on $RAM(+)$. If we allow the reversal of bitmasks inv as a primitive, then the problem can be solved in the same time, preprocessing, and space complexities as NET and EXNET on $RAM(+, inv)$.

To obtain above results, we devise the following techniques to achieve efficient bit-parallel simulation of Thompson NFA (TNFA, for short) for classes NET and EXNET. A key of NFA simulation for the full class REG is an efficient simulation of ε-closure in TNFA as mentioned in the previous works [4, 11]. Hence, by extending the previous SHIFT-AND [3, 18] and Extended SHIFT-AND [14] approaches, we introduce a set of new bit-parallel simulation operations, called Scatter, Gather and Propagate operations to deal with the long succession and the branching of ε-edges caused by concatenation and union in network expressions in NET and EXNET. Furthermore, we also devise a transformation technique of a given TNFA into a special form of NFA that satisfies a property called "*bi-monotonicity*" of ε-moves by attaching new ε-edges to all subexpressions whose initial and final states are ε-reachable in the original expression. Furthermore, we introduce the *barrel shifter* technique for implementing backward ε-edges for REG based on a well-known technique in the VLSI circuit design.

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 TNFAs and does not use complex module decompositions as in [4], it is particularly efficient for regular expressions with small depth. (ii) Hardware friendly: Since it uses only simple bit-operations and addition/subtraction and avoids the heavy use of table-lookup, it has potential to be implemented on modern parallel hardwares with simple structure, such as GPGPUs or FPGAs. To confirm the above observations, we developed a hardware implementation of a multiple regular expression matching system on FPGA based on the proposed algorithm. The experimental results showed that the system could match 256 patterns at the same time against a text stream with throughput of 1.6Gbps and 0.5Gbps in total for NET and EXNET, respectively.

Related works. There are a number of researches on the regular expression matching problem for REG other than the Extended SHIFT-AND approach. In the Table-Lookup approach, Myers [11] developed an $O(nm/\log n)$ time and space algorithm. Improving the space complexity of [11], Bille and Thorup [5] presented $O(nm(\log\log n)/(\log n)^{3/2} + n + m)$ time and $O(m)$ space algorithm. For DFA simulation by Brute force determinization, Navarro and Raffinot [15] proposed an $O(n)$ time and $O(m2^m)$ bits space algorithm using DFA simulation of Glushkov's NFAs, while Wu and Manber [18] presented an $O(n)$ time and $O(m2^{2m})$ bits space algorithm based on the DFA simulation of Thompson's NFAs. Champarnaud *et al.* [7] improves this result by obtaining an expected exponential reduction of the space complexity. Papers [6, 12, 13] study pattern matching with bounded and unbounded gaps.

Organization of this paper. In Section 2, we give basic definitions and notations. In Section 3, we present our algorithm for the class NET of network expressions as well as extended network expressions EXNET. In Section 4, we give a modified algorithm for the full class REG of regular expressions. In Section 5, we show experimental results on the hardware implementation of the proposed algorithms. In Section 6, we conclude this paper.

2 Preliminary

In this section, we give basic definitions and notations in the regular expression matching problem according to [2, 11, 14].

Regular expression matching problem. Let $\mathbf{N} = \{0, 1, 2, \ldots\}$. For $i \leq j$, we define $[i..j] = \{i, i+1, \ldots, j\}$. Let Σ be a finite alphabet of *letters*. A *string* on Σ is a sequence $S = s_1 \cdots s_n$ of letters, where $s_i \in \Sigma$ for every i. For every $1 \leq i \leq j \leq n$, We denote by $S[i] = s_i \in \Sigma$, by $S[i..j]$ the substring $s_i \cdots s_j$, and by ε the *empty string*. If $i > j$, we define $S[i..j] = \varepsilon$. For a string S, we denote by $|S|$ the *length* (or the *size*) of S.

The class REG of *regular expressions* on Σ is defined recursively as follows: (1) If $a \in \Sigma \cup \{\varepsilon\}$ then $a \in \mathsf{REG}$. (2) If $R_1, \ldots, R_n \in \mathsf{REG}$ then $(R_1 \cdots R_n)$, $(R_1| \cdots |R_n)$, $(R_1)^* \in \mathsf{REG}$. In this paper, regular expressions are *unbounded*, i.e., $n \geq 1$, while $n = 2$ in the standard definition [4, 14]. The *length* (or the *size*) of R is defined by the number $||R||$ of symbols from $\Sigma \cup \{\varepsilon, \cdot, |, *\}$ appearing in R. For a regular expression R, the *parse tree* T_R, the *language* $L(R) \subseteq \Sigma^*$, and the *depth* (or the *height*) $d(R)$, respectively, are defined in the standard way [14]. Let $\mathcal{C} \subseteq \mathsf{REG}$ be any subclass of REG. A *pattern* is any regular expression $R \in \mathcal{C}$ and a *text* is a string $T \in \Sigma^*$ over Σ. We say that a regular expression R of length m *occurs* in a text T of length n 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. Now, we state our problem below. *The regular expression matching problem for a class* $\mathcal{C} \subseteq \mathsf{REG}$ is, given a regular expression $R \in \mathcal{C}$ of length m and an input text T of length n, to output the set of all end positions of R in T.

Subclasses of regular expressions. We introduce the classes STR, EXT, NET, and EXNET of string patterns, extended string patterns, network expressions, and extended network expressions, respectively, as follows. A *string pattern* over Σ is a string $R \in \Sigma^*$. An *extended string pattern* [13] over Σ is a regular expression $R = r_1 \cdots r_m$ $(m \geq 0)$, where for every $1 \leq i \leq m$, r_i is one of the following forms:

(i) letters $a \in \Sigma$, (ii) wildcards ".", (iii) classes of letters $\alpha = [ab\cdots]$, (iv) optional letters $\alpha?$, (v) bounded repeats $\alpha\{x, y\}$, and (vi) unbounded repeats α^* and α^+, where $\alpha \subseteq \Sigma$. The semantics of the additional operations is given by the notational equivalence: "." $\equiv \Sigma$, $\alpha? \equiv (\alpha|\varepsilon)$, $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$, and $\alpha^+ \equiv (\alpha\,\alpha^*)$.

A *network expression* (over strings) in NET [12] is a regular expression over strings, that is, a regular expression obtained from strings, ε-edges, concatenation, and union. An *extended network expression* in EXNET [12] is a network expression over extended string patterns in EXT. For example, $R_0 = (\texttt{[AB]}^+)(\texttt{B.}\{1,3\})(\texttt{[BC]?})(.^*)\texttt{C}$, $R_1 = \texttt{A(BA|CD)(CD|AB)B}$, and $R_2 = \texttt{A(AB|B?)(B?.*|AB)C}$ are examples of expressions over $\Sigma = \{\texttt{A}, \texttt{B}, \texttt{C}\}$ in EXT, NET, and EXNET, respectively.

Model of computation. As the model of computation, we assume a *unit-cost RAM* with word length w [2]. For any bitmask length $L \geq 0$, A *bitmask* is a vector $X = b_L \cdots b_1 \in \{0, 1\}^L$ of L bits. for a bit $b \in \{0, 1\}$, we denote by b^k the bitmask consisting of k copies of b. For bitmasks with $L \leq w$, we assume that the following Boolean and arithmetic operations are executed in $O(1)$ time: *bitwise* AND "&", *bitwise* OR "|", *bitwise* NOT "$\sim$", *bitwise* XOR "$\oplus$", *left shift* "$\ll$", *right shift* "$\gg$" on $RAM()$, *integer addition* "+" and *integer subtraction* "$-$" on $RAM(+)$. The space complexity is measured in the number of words.

3 Fast Bit-parallel Algorithm for Extended Network Expressions

In this section, we present an efficient algorithm that receives any input extended network expression R in NET or EXNET with length m and depth d and an input text T on Σ with length n, and finds all the occurrences of R in T in $O(nd\lceil m/w \rceil)$ time using $O(dm + |\Sigma|\lceil m/w \rceil)$ preprocessing and $O(d\lceil m/w \rceil + |\Sigma|\lceil m/w \rceil)$ space on $RAM(+)$. In what follows, we assume an input regular expression R with length m and depth d and the input text T with length n.

3.1 Basic NFA simulation algorithm

We show the outline of our algorithm BP-Match. First, in the preprocessing phase, we construct a set of the bitmasks M_R from a given extended network expression $R \in$ EXNET, and then, in the runtime phase, we search for all the end positions of R in an input text T based on NFA simulation of N_R.

Algorithm BP-Match($T \in \Sigma^*$: an input text, $R \in$ EXNET: an extended network expression)

PREPROCESS:

(1) Transform R to its expanded form Expand(R).
(2) Construct the TNFA N_R from Expand(R).
(3) Construct a set M_R of the bitmasks from N_R.

RUNTIME:

(4) Simulate N_R on T by using M_R

Transformation of a regular expression to its expanded form. As preprocessing, we first expand all the occurrences of bounded repeats $\alpha\{x, y\}$ and

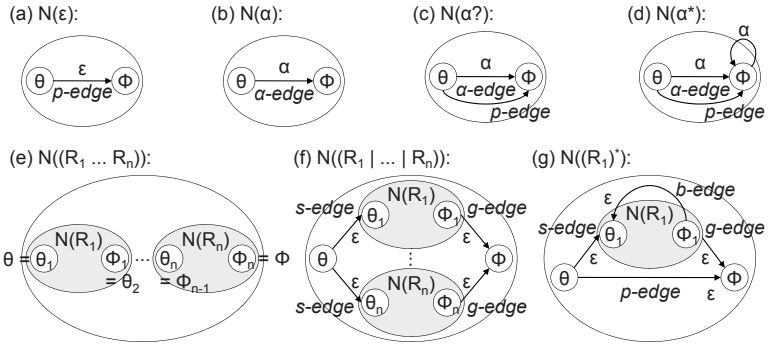

Fig. 1. The construction of Thompson automata (TNFAs).

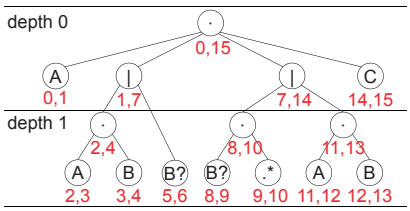

Fig. 2. The parse tree of an extended network expression $R_2 = \texttt{A(AB|B?)(B?.*|AB)C}$.

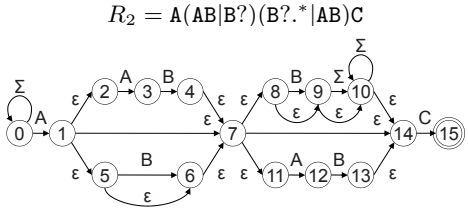

Fig. 3. An extended network expression R_2 and its TNFA $N_2 = N(R_2)$.

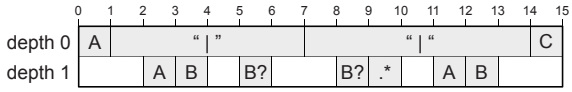

Fig. 4. The bit-position assignment for subexpressions of $\mathsf{Expand}(R_2)$, where the depth is compressed by ignoring internal nodes labeled with concatenation "·".

unbounded repeats α^+ in an input expression R using the equivalence $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$ and $\alpha^+ \equiv (\alpha\alpha^*)$, respectively. Furthermore, we apply the operation, called *bypassing*, that replaces all the subexpressions S in R such that $\varepsilon \in L(S)$ with the expression $S' \equiv (S \mid \varepsilon)$. This bypassing does not change the language $L(R)$. We denote by $\mathsf{Expand}(R)$ the resulting extended network expression. The properties of $\mathsf{Expand}(R)$ will be examined later.

Construction of TNFA. We construct the parse tree T_R of R as shown in Fig. 2. By the construction in Fig. 1, we compute the Thompson NFA (TNFA, for short) $N(R) = (V, E, \theta, \phi)$ of $\mathsf{Expand}(R)$ as shown in Fig. 3, where $V = \{0, \ldots, L\}$ for $L \geq 0$. As a special case, for the subexpression $S' \equiv (S|\varepsilon)$ introduced by the bypassing, we add the ε-edge to S directly connecting from θ_S to ϕ_S instead of rule (f). In Fig. 3, we show an example of TNFA for $R_2 = \texttt{A(AB|B?)(B?.*|AB)C}$. For each node v of T_R, let $S = S(v)$ be the subexpression of $\mathsf{Expand}(R)$ associated with v and $N_S = N(S) = (V_S, E_S, \theta_S, \phi_S)$ be its corresponding TNFA, called the *component TNFA* for v, with a state set V_S, an edge set E_S, initial and final states θ_S and ϕ_S. By depth-first search of T_R from left to right, we assign the set $V(v) = \{\theta_S, \phi_S\} \subseteq [0..L]$ of the initial and final states of S to each node v of T_R as in Fig. 2, and define the depth $d(v)$ by the number of non-concatenation nodes on the path from the root to v. For each $x \in V(v)$, we define $d(x) = d(v)$, and for each subexpression $S = S(v)$, we associate the interval $I_S = [\theta_S..\phi_S] \subseteq [0..L]$. In Fig. 4, we show a bit-position assignment related to T_R in Fig. 2. A labeled edge $e = (u, \beta, v) \in E_S$ is an *α-edge* if $\beta \subseteq \Sigma$, and is an *ε-edge* if $\beta = \varepsilon$.

Algorithm RunTNFA($T = t_1 \cdots t_n$: an input text, $N(R)$: a TNFA)

```
 1: D ← Init_N; //initial state set
 2: D ← EpsClo_N(D); //ε-closure
 3: for i ← 1, ..., n do
 4:    if D ∩ Accept_N ≠ ∅ then
 5:       report "match at i − 1";
 6:    D ← Move_N(D, t_i); //α-edges
 7:    D ← EpsClo_N(D); //ε-closure
 8: end for
 9:
10:
```

Fig. 5. The algorithm RunTNFA for NFA simulation in the runtime phase.

Procedure $\mathsf{EpsClo}_N(D$: the state set for a TNFA $N(R))$

```
 1: for k ← d(R), ..., 1 do
 2:    D ← Propagate(D, k);
 3:    D ← Gather(D, k − 1);
 4: end for
 5: D ← Propagate(D, 0);
 6: for k ← 1, ..., d(R) do
 7:    D ← Scatter(D, k − 1);
 8:    D ← Propagate(D, k);
 9: end for
10: return D;
```

Fig. 6. The procedure EpsClo_N for computing ε-closure.

Efficient NFA simulation. Next, we describe the standard *NFA simulation* method developed by Thompson [2, 17] that most of the previous regular expression matching algorithms [3, 4, 11, 14, 18] employ. In Thompson's algorithm [17], the current status of the TNFA $N_R = (V_R, E_R, \theta_R, \phi_R)$ is represented by a set $D \subseteq V_R$ of *active states*. Then, we define the following operations: Init_N returns the set $\{\theta_R\}$; Accept_N returns the set $\{\phi_R\}$; For any letter $c \in \Sigma$, $\mathsf{Move}_N(D, c)$ returns the set $\{y \in V_R \mid y$ is reachable from some $x \in D$ by exactly one α-edge such that $c \in \alpha\}$; $\mathsf{EpsClo}_N(D)$ returns the set $\{y \in V_R \mid y$ is reachable from some $x \in D$ by zero or more ε-edges $\}$, called the *ε-closure* of D.

In Fig. 5, we show the algorithm RunTNFA that simulates the computation of the TNFA N_R on an input text T. We can show the following lemma [17].

Lemma 1 (Thompson [17]). *For any input text T, the algorithm* RunTNFA *in Fig. 5 correctly solves the regular expression matching problem for* REG.

Fine classification of ε-moves. It is not hard to efficiently implement Move_N either by using *table-lookup* [11] or SHIFT-AND approach [3, 18], while it is not straightforward to efficiently implement EpsClo_N since we have to compute ε-closure. The key of our algorithm is an efficient implementation of EpsClo_N based on a set of new bit-parallel operations Scatter, Gather, and Propagate defined as follows.

In the construction (a)–(g) of TNFA in Fig. 1, we categorize ε-edges in a component TNFA $N(S)$ into four types: (i) $e = (\theta, \varepsilon, \theta_i)$ in (f) or (g) is a *scatter edge* (s-edge) with depth $d(\theta)$, (ii) $e = (\phi_i, \varepsilon, \phi)$ in (f) or (g) is a *gather edge* (g-edge) with $d(\phi)$, (iii) $e = (\theta, \varepsilon, \phi)$ in (a), (c), (d), or (g) is a *propagate edge* (p-edge) with $d(\theta) = d(\phi)$, and (iv) $e = (\phi_i, \varepsilon, \theta_i)$ in (g) is a *back edge* (b-edge) with $d(\theta_i) = d(\phi_i)$, where θ and ϕ are the initial and the final states of $N(S)$. We classify the ε-edge introduced by bypassing as a propagate edge. For scatter, gather, propagate, and back edges in $N(S)$, we assign the depth of the outermost node θ or ϕ of $N(S)$. The next lemma gives a characterization of ε-edges.

Lemma 2. *If $e = (u, \varepsilon, v)$ is an ε-edge in the TNFA N_R for $R \in$ EXNET, then $\Delta = d(v) - d(u) \in \{+1, 0, -1\}$ holds. Moreover, (i) if $\Delta = +1$, e is a scatter edge, (ii) if $\Delta = -1$, e is a gather edge, and (iii) if $\Delta = 0$, e is a propagate edge.*

For any set $D \subseteq V$ and any $k = 0, \ldots, d(R)$, we define $\mathsf{Scatter}(D, k)$ (or $\mathsf{Gather}(D, k)$) the sets of states from some states in D reachable by exactly one scatter edge (or one gather edge, resp.) with depth k. On the other hand, the set $\mathsf{Propagate}(D, k)$ is defined by the ε-closure of D restricted by the propagate edges with depth k. For any component TNFA S, an *ε-block* $B \subseteq V_S$ is a set of states that induces a maximal connected component consisting only of propagate edges. By construction of TNFA and bypassing, we can see that any such ε-block forms a chain. Clearly, all states in B have the same depth d, which is called the *depth* of B. For example, an expression $R_2 = \mathtt{A}(\mathtt{AB}|\mathtt{B}?)(\mathtt{B}?.^*|\mathtt{AB})\mathtt{C}$ in Fig. 3 has three ε-blocks, $B_1 = \{1, 7, 14\}$, $B_2 = \{5, 6\}$, and $B_3 = \{8, 9, 10\}$.

Now, we show the key lemma, called the bi-monotonicity lemma on bypassing transformation. For any $d, d' \in \mathbf{N}$, we define $d \leq_1 d'$ if $d' - d \leq 1$ holds. For any states x, y in TNFA $N(R)$, a ε-path $\pi = (x_1 = x, \ldots, x_n = y) \in (V_R)^*$ is said to be *bi-monotone* if there exists some state x_k $(1 \leq k \leq n)$ such that $d(x_1) \geq_1 \cdots \geq_1 d(x_k)$ and $d(x_k) \leq_1 \cdots \leq_1 d(x_n)$ hold, that is, the depth sequence for the first half is non-decreasing and the latter half is non-increasing. By induction on the construction of TNFA, we can show the next lemma.

Lemma 3 (bi-monotonicity lemma). *Let x, y be any states in $\mathsf{Expand}(R)$. If π be any ε-path from x to y, then there also exists some bi-monotone ε-path from x to y in $\mathsf{Expand}(R)$.*

Based on the bi-monotonicity of an expanded version of TNFA, we present in Fig. 6 the procedure EpsClo_N that computes the ε-closure for EXNET.

Lemma 4. *Suppose that* Scatter*,* Gather*, and* Propagate *operations are correctly implemented for $R \in$ EXNET with depth $d(R)$. Then, the algorithm* EpsClo *in Fig. 6 correctly computes the ε-closure $\mathsf{EpsClo}_{N(R)}(D)$ of any state set D.*

Proof. The soundness is obvious from construction. The completeness follows that if a state y is ε-reachable from a state x, then the applications of operators in the order of the regular expression $(\mathsf{Propagete.Gather})^*$ Propagete $(\mathsf{Scatter.Propagete})^*$ moves x to y by the existence of a bi-monotone ε-path by Lemma 3. Since this is what EpsClo_N does, the lemma is proved. □

3.2 Bit-parallel implementation

To simulate the TNFA N_R for an extended network expression $\mathsf{Expand}(R)$, we use a set M_R of bitmasks of L bits $\mathtt{CHR}[c]$, $\mathtt{REP}[c]$, $\mathtt{BLK}_\tau[k]$, $\mathtt{SRC}_\tau[k]$, and $\mathtt{DST}_\tau[k] \in \{0, 1\}^L$, for every $c \in \Sigma$, $0 \leq k \leq d(R)$, and $\tau \in \{\mathtt{S}, \mathtt{G}, \mathtt{P}\}$, where L is the number of the states of N_R. Then, by further generalizing the Extended SHIFT-AND approach, we simulate the ε-closure operations Scatter, Gather, and Propagate as follows. Let $N_S = (V, E, \theta, \phi)$ be any component TNFA in depth k.

Simulation of Move operation. *Preprocess*: Let $e = (\theta, \alpha, \phi) \in E$ be an α-edge of N_S, where $\alpha \subseteq \Sigma$. To implement the Move operation, we precompute the following bitmasks. For every letter $c \in \alpha$ and every N_S, we define: (M.1) $\mathtt{CHR}[c]$ has 1 in the bit-position $j = \phi$. (M.2) $\mathtt{REP}[c]$ has 1 in the bit-position $j = \phi$ such that $\theta = \phi$ holds, that is, an α-edge e is a self-loop, equivalently, either $S = \alpha^*$ or $S = \alpha^+$.

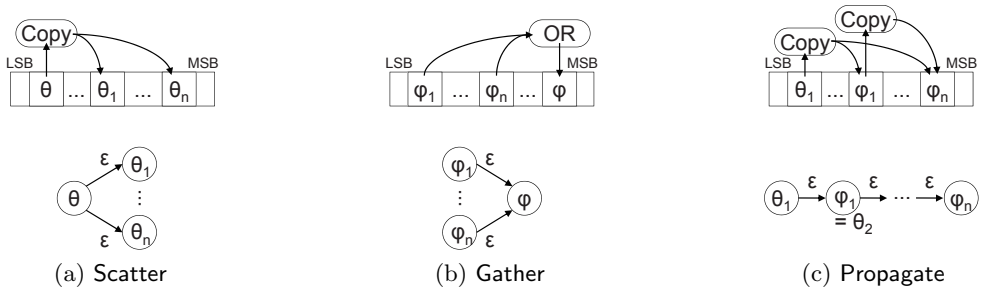

Fig. 7. The bit-operations and the corresponding parts of TNFAs

Runtime: To simulate the $\mathsf{Move}(D, t_i)$, we perform

$$D \leftarrow (((D \ll 1)\ \&\ \mathtt{CHR}[t_i])\mid 1)\mid (D\ \&\ \mathtt{REP}[t_i]); \tag{1}$$

where $t_i \in \Sigma$ be an input letter. This code is the same as the code for α-moves in the Extended SHIFT-AND approach [14]. For the details, see [14].

Simulation of Scatter operation. *Preprocess*: Let $e = (\theta, \varepsilon, \theta_i) \in E$ be a scatter edge of N_S. To implement the Scatter operation, we precompute the following bitmasks. For every depth k and every N_S, we define: (S.1) $\mathtt{BLK_S}[k]$ has 1 in the bit-position $j = \phi - 1$. (S.2) $\mathtt{SRC_S}[k]$ has 1 in the bit-position $j = \theta$. (S.3) $\mathtt{DST_S}[k]$ has 1 in the bit-position j iff $j = \theta_i$ for all i (Fig. 7).

Runtime: To simulate the $\mathsf{Scatter}(D, k)$, we perform

$$D \leftarrow D \mid ((\mathtt{BLK_S}[k] - \{D\ \&\ \mathtt{SRC_S}[k]\})\ \&\ \mathtt{DST_S}[k]); \tag{2}$$

Firstly, by the formula $(D\ \&\ \mathtt{SRC_S}[k])$, we extract the values of source bits from D. Then, by subtracting the values from $\mathtt{BLK_S}[k]$, all the destination bits are set to 1 if the source bits is 1, and all to 0 otherwise. Note that this is done by carry propagation of subtraction "$-$". Finally, we extract the destination bits by AND-ing the result with $\mathtt{DST_S}[k]$, and put all the destination bits to D.

Simulation of Gather operation. *Preprocess*: Let $e = (\phi_i, \varepsilon, \phi) \in E$ be a gather edge of N_S. For every depth k and every N_S, we define: (G.1) $\mathtt{BLK_G}[k]$ has 1 in the bit-position $j \in [\theta + 1..\phi - 1]$. (G.2) $\mathtt{SRC_G}[k]$ has 1 in the bit-position j iff $j = \phi_i$ for all i. (G.3) $\mathtt{DST_G}[k]$ has 1 in the bit-position $j = \phi$ (Fig. 7).

Runtime: To simulate the $\mathsf{Gather}(D, k)$, we do the following

$$D \leftarrow D \mid ((\mathtt{BLK_G}[k] + \{D\ \&\ \mathtt{SRC_G}[k]\})\ \&\ \mathtt{DST_G}[k]); \tag{3}$$

Since this code is similar to one of Scatter except that Gather uses addition, while Scatter uses subtraction, we omit the details.

Simulation of Propagate operation. *Preprocess*: Let $e = (\theta, \varepsilon, \phi) \in E$ be a propagate edge of N_S. For every k and ε-block B with depth k, we define: (P.1) $\mathtt{BLK_P}[k]$ has 1 in the bit-position $i \in B$. (P.2) $\mathtt{SRC_P}[k]$ has 1 in the bit-position $i = min(B)$. (P.3) $\mathtt{DST_P}[k]$ has 1 in the bit-position $i = max(B)$ (Fig. 7).

Runtime: To simulate the $\mathsf{Propagate}(\mathsf{D}, k)$, we perform the following

$$A \leftarrow (D\ \&\ \mathtt{BLK_P})\mid \mathtt{DST_P}[k]; \tag{4}$$

$$D \leftarrow D \mid (\mathtt{BLK_P}[k]\ \&\ ((\sim (A - \mathtt{SRC_P}[k])) \oplus A)); \tag{5}$$

The above code works since any ε-block is a chain in the same depth, and thus the propagation of the carry bits in the subexpression $(A - \mathtt{SRC_P}[k])$ correctly implements the ε-closure on a chain as shown in [13, 14].

3.3 Main results

From Navarro and Raffinot (Sec. 1.3.1, [14]), we know that the integer addition and subtraction can be executed in $O(\lceil m/w \rceil)$ time and space by simulating carry propagation. Combining this and the arguments in the previous section, we have the following lemma.

Lemma 5. *By the above construction, the* $\mathsf{Move}(D, c)$, $\mathsf{Scatter}(D, k)$, $\mathsf{Gather}(D, k)$, *and* $\mathsf{Propagate}(D, k)$ *operations for* $N(R)$ *are correctly implemented to run in* $O(\lceil m/w \rceil)$ *time on* $RAM(+)$, *where* $c \in \Sigma$, $0 \leq k \leq d(R)$, D *is any* m*-bit mask,* m *is the number of states in* $N(R)$ *and* w *is the word length.*

From Lemma 5, in the *large automata* case with $m > w$, we can use inexpensive simulation of primitive operations on $RAM(+)$ instead of expensive module decomposition technique used tabling-based algorithms as in [4, 11]. This will be an advantage of our algorithm in implementing it on parallel hardwares such as GPGPUs and FPGAs. Now, we show the main result of this paper.

Theorem 1. *The algorithm* BP-Match *solves the regular expression matching problem for* NET *and* EXNET *of network and extended network expressions in* $O(nd\lceil m/w \rceil)$ *time using* $O(dm + |\Sigma|\lceil m/w \rceil)$ *preprocessing and* $O(d\lceil m/w \rceil + |\Sigma|\lceil m/w \rceil)$ *space, where* $n = |T|$, $m = ||R||$, $d = d(R)$, w *is the word length.*

Proof. The correctness follows from Lemma 1, Lemma 4, and Lemma 5. Then, the result immediately follows from that the for-loop is executed at most $d(R)$ times and each code can be executed in $O(\lceil m/w \rceil)$ time from Lemma 5 □

4 Extension for general regular expressions

To generalize our algorithm in Sec. 3 for the full class REG in the Extended SHIFT-AND approach, we need to simulate backward ε-edges corresponding to the Kleene-closure "$*$". However, the backward ε-edges from lower to higher bits seems hard to compute on $RAM(+)$. To overcome this difficulty, we introduce a technique called *barrel shifter* as follows.

The idea is to decompose each backward ε-moves from higher to lower bits having the length J bits into a series of right-shifts "$\gg$" having the widths $2^0 = 1, 2^1 = 2, \ldots, 2^\ell$, where $\ell = \lceil \log \delta \rceil$ and $\delta = O(m)$ is the maximum length of the backward ε-edges in TNFA. More precisely, for each back edge e in a certain depth of R, if the edge e has the width $J \geq 0$, we have the unique binary expansion $bin(J) = J_{\ell-1} \cdots J_0 \in \{0, 1\}^L$ such that $J = \sum_{i=0}^{\ell-1} J_i 2^i$. For each $k = 0, \ldots, d(R)$ and $i = 0, \ldots, \ell - 1$, the bitmask $\mathtt{BLK_B}[k][i]$ is defined by: for each back edge $e = (\phi_i, \varepsilon, \theta_i)$ in depth k, we fill the interval $I_e = [\theta_i..\phi_i]$ with 1's if $J_k = 1$ and with 0's if $J_k = 0$. In run-time, we set $jmp \leftarrow 1$, and repeatedly perform $D \leftarrow (D \;\&\; \sim \mathtt{BLK_B}[k][i]) \mid ((D \;\&\; \mathtt{BLK_B}[k][i]) \gg jmp); jmp \leftarrow jmp \ll 1$. From the construction, this operation can be implemented in $O(d\lceil m/w \rceil \log m)$ time using $O(dm \log m)$ preprocessing and $O(d\lceil m/w \rceil \log m)$ space.

Table 1. Summary of experimental results on the hardware implementation, where #op, #add, #reg, #bram, and #slice, are the numbers of 32 bit operations, 32 bit integer additions, registers, block RAM lines, resp., per PMM. #pat and #char are the number and the total size of input patterns, resp.

Class	#op	#add	#reg	#bram	#slice	frequency	throughput	load time	#pat	#char
STR	5	0	3	256	54	363 MHz	2.9 Gbps	0.182 ms	256	8,192
EXT	11	1	6	512	123	202 MHz	1.6 Gbps	0.328 ms	128	4,096
EXNET	20	9	24	512	736	65 MHz	0.5 Gbps	1.055 ms	128	4,096

Theorem 2. *The regular expression matching problem for the class* REG *can be solved in* $O(nd\lceil m/w\rceil \log m)$ *time using* $O(dm \log m + |\Sigma|\lceil m/w\rceil)$ *preprocessing and* $O(d\lceil m/w\rceil \log m + |\Sigma|\lceil m/w\rceil)$ *space.*

As an alternative, if there are at most constant number of back edges with mutually distinct lengths, then we can replace the $O(\log m)$ term with $O(1)$. As other option, if the $O(1)$-bit-reversal inv is available, we can also replace the $O(\log m)$ term with $O(1)$ on $RAM(+, inv)$ by simulating backward ε-moves by Scatter (or Gather) and inv. Thus, we obtain the same complexity as Theorem 1.

5 Experimental results

To evaluate the performance, we implemented our regular expression matching algorithm on *FPGA* in Verilog-HDL for STR, EXT, and EXEXT. We designed the algorithm as a collection of up to 256 pattern matching modules (PPMs) working simultaneously [10], where the word length is $w = 32$ bits, and masks are stored in block RAMs and a set of registers. We used the Xilinx ISE Design Suite 10.1 and Synopsys VCS development tools. Having targetted an *FPGA* device, Xilinx Virtex-5 LX330 with -1 speed grade, which had 51,840 slices and 288 block RAMs with 36 Kbits, we could install up to 256 PPMs. For more details of the experiments, see the companion paper [10]. Table. 1 shows the summary of the experimental results on our hardware. The #bram is given by the number of block RAMs times $|\Sigma| = 256$. Then, we can observe that our hardware achieves the high throughput of 0.5 Gbps for the class EXNET and of 1.6 Gbps for the class EXT, which is hard to achieve by software implementation on the current general CPUs. Hence, our algorithm is suitable to hardware implementation.

6 Conclusion

In this paper, we presented an efficient bit-parallel algorithm that solves the regular expression matching problem for the class EXNET of extended network expressions in $O(nd\lceil m/w\rceil)$ time using $O(d\lceil m/w\rceil)$ space and $O(dm)$ preprocessing by extending the Extended SHIFT-AND approach [14]. Furthermore, we show that the problem for the full class REG of regular expressions is solvable in $O(nmd \log w/w)$ time on $RAM(+)$. Experiments on its hardware implementation showed that the proposed algorithm is suitable to parallel execution on hardwares. Other advantage is the guaranteed worst-case time complexity. Thus, it may be useful as a base algorithm for other approaches such as filtration as mentioned in [8, 14]. Application of the Extended SHIFT-AND to tree and XML matching [9] will be an interesting future research.

Acknowledgements

The authors would like to thank Osamu Watanabe, Ryuhei Uehara, Takashi Horiyama, Yoshio Okamoto, Thomas Zeugmann, Shinobu Nagayama, Ayumi Shinohara, Masayuki Takeda, for their discussions and valuable comments, and to Shingo Yoshizawa, and Yoshikazu Miyanaga for their warm encouragements and constant supports on VLSI circuit design. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, and interdisciplinary research project "*high performance FPGA-based string matching hardwares*" under MEXT/JSPS Global COE Program, FY2007–2011.

References

1. J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *Proc. ACM SIGMOD'08*, 147–160, 2008.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
3. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10), 74–82, 1992.
4. P. Bille. New algorithms for regular expression matching. In *Proc. ICALP'06*, 643–654, 2006.
5. P. Bille and M. Thorup. Faster regular expression matching. In *Proc. ICALP'09*, 171–182, 2009.
6. P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. SODA'10*, 1297–1308, 2010.
7. J. -M. Champarnaud, F. Coulon, and T. Paranthoën. Compact and fast algorithms for safe regular expression search. *Int. J. Comput. Math.*, 81(4), 383–401, 2004.
8. S. Grabowski and K. Fredriksson. Bit-parallel string matching under Hamming distance in $O(n\lceil m/w \rceil)$ worst case time. *Inf. Proccess. Lett.*, 105(5), 182–187, 2008.
9. Y. Kaneta and H. Arimura. Faster bit-parallel algorithm for unordered pseudo-tree matching and tree homeomorphism. In *Proc. of IWOCA'10*, Jul. 2010. (also appeared as Hokkaido U., TCS-TR-A-10-43, May 2010.)
10. Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga. Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching. Hokkaido U., TCS-TR-A-10-45, June 2010 (submitting).
11. E. W. Myers. A four-Russian algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2), 430–448, 1992.
12. E. W. Myers. Approximate matching of network expessions with spacers. *J. Computational Biology*, 3(1), 33–51, 1996.
13. G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proc. RECOMB'01*, 231–240, 2001.
14. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge, 2002.
15. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2), 89–116, 2005.
16. R. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. IEEE FCCM'01*, 227–238, 2001.
17. K. Thompson. Programming techniques: regular expression search algorithm. *Communications of the ACM*, 11(6), 419–422, 1968.
18. S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10), 83–91, 1992.