# Faster Pattern Matching Algorithms for Massive Regular Expression Matching and Its Applications

（大規模正規表現照合のための高速なパターン照合アルゴリズムとその応用）

Hirohito Sasakawa

February, 2016

# Abstract

By rapid development of network infrastructure and sensing technologies, massive amount of streaming and unstructured data have emerged in various fields. The regular expression matching is the one of the most important task to extract desired information from the data. The importance of regular expression matching is have been increasing because of the emergence of massive data.

However, the new types of problems of regular expression matching in practical application is appeared. We especially tackle to following three problems of them in this thesis: (1) The massive regular expression matching which process large number of regular expression patterns against an input text. This is quite CPU intensive and difficult to process in single CPU. (2) The regular expression matching on time series multi-dimension numerical data. It is difficult to apply existing pattern matching method to this problem efficiently. (3) The regular expression matching on sensitive data. We need to consider the privacy leakage through data analysis on sensitive data.

Our research goal is to develop an efficient regular expression matching applicable to the real world application. To achieve this goal, we tackle the three specific problems of regular expression matching described above, and develop efficient algorithms based on *bit-parallel approach*. In this study, we utilize bit-parallel NFA simulation as a key to solve these problems.

In Chapter 3, we study massive regular expression matching problem, which is problem of, given a set of input regular expressions $\mathcal{P} = \{P_1, \cdots, P_k\}$ and an input

text $T$, finding all occurrences of each regular expression of $\mathcal{P}$ in $T$. We use GPU (graphic processing unit) as target hardware, which is a parallel hardware accelerator equipped with several hundreds of processing cores and large bandwidth memory in this chapter. As a main result, we propose the hierarchically decomposed finite automaton called *hierarchical non-deterministic finite automaton* (*HFA* for short), and its parallel simulation algorithm on GPU based on Extended SHIFT-AND algorithm proposed by Navarro and Raffinot in 2003. Experimental results showed that our algorithm works faster than PFAC algorithm proposed by Lin et al. in 2010 by a factor of 150.

In Chapter 4, we study regular expression matching problem on trajectory data of moving objects. The problem is, given a trajectory regular expression $P$ of size $m$, text trajectory $T$ of size $n$, and threshold parameter $r \geq 0$, to find all position $j$ in $T$ that every subtrajectory $T[i..j]$ satisfies the similarity condition defined in the chapter. As a main result, we first formulate regular expression matching problem on trajectory data under similarity condition based on $L_\infty$ distance. Then, we give the efficient online bit-parallel pattern matching algorithm by combining simple spatial index and bit-parallel NFA simulation runs in $O(mn \log\log n/logn)$ time using $O(m)$ space on unit cost word RAM. Furthermore, our algorithm supports popular geometric query such as OD queries and its variant widely used in geographic information systems (GIS for short).

In Chapter 5, we study secure regular expression matching problem. In this problem, there are two participants Alice and Bob. Alice has a input regular expression $R$, and Bob has a text $T$ as their secret information. The problem is to report existence of the occurrence of $R$ in $T$ to Alice nothing to Bob without leakage of their secrets. As a main result, we propose an efficient two party secure protocol for NFA evaluation called *oblivious NFA evaluation (ONE)* runs in $O(nm^2)$ time, $O(nm^2)$ communication and $O(nm)$ round using $O(m^2)$ space, where $m$ is the size of a regular expression $R$ and $n$ is the length of text $T$. Finally, we give the conclusion of this thesis and discuss

future works in Chapter 6.

# Acknowledgements

First of all, I would like to express my appreciation to my supervisor Professor Hiroki Arimura. He gave me plenty of advice. I would not accomplish this work without his kind support. I am deeply grateful to Professor Takuya Kida for his great support in my life in Information Knowledge Network Laboratory.

I would like to thank Professor Shin-ichi Minato and Thomas Zeugmann for their fruitful discussion and valuable comments. I would would like to express my gratitude to Professor Jun Sakuma, Professor Koji Tsuda, Dr. David duVerle and Mr. Hiroki Harada. They directed me to an interesting field of privacy preserving computation.

I would be grateful to all colleagues and secretary, especially Ms. Yu Manabe, Dr. Yusaku Kaneta, Dr. Satoshi Yoshida, Mr. Kunihiro Wasa, Mr. Kei Sekine, and Mr. Takuya Takagi, Mr. Takuya Masaki, Mr. Masahiro Yamamoto, and Mr. Kazuhiro Kurita in Information Knowledge Network laboratory, and Mr. Ryutaro Kurai in Knowledge Media Laboratory. I could do nothing without their help. I enjoyed my life in the laboratory.

Finally, I would like to thank my father and my mother for all their support.

# Contents

**Abstract** **i**

**Acknowledgements** **v**

**1 Introduction** **1**

**2 Preliminaries** **7**

2.1 Basic Definitions . . . . . 7

2.2 Regular Expressions . . . . . 7

2.3 Non-deterministic and Deterministic Finite Automata . . . . . 9

2.4 Model of Computation . . . . . 10

2.5 Bit-parallel NFA Simulation Algorithms . . . . . 10

**3 Faster Massive Pattern Matching System on GPU** **17**

3.1 Background . . . . . 17

3.2 Preliminaries . . . . . 20

3.3 Pattern Matching System on GPUs . . . . . 21

3.4 Time and Space Complexity . . . . . 25

3.5 Experiments . . . . . 30

3.6 Chapter Summary . . . . . 34

**4 Efficient Complex Temporal Pattern Detection over Trajectory Streams 39**

4.1 Background . . . . . 40

4.2 Related work . . . . . 40

4.3 Preliminaries . . . . . 42

4.4 The Proposed Algorithm . . . . . 47

4.5 Chapter Summery . . . . . 56

**5 Oblivious Evaluation of Non-deterministic Finite Automata with Application to Privacy-Preserving Virus Genome Detection 59**

5.1 Background . . . . . 60

5.2 Matrix-based NFA Evaluation . . . . . 64

5.3 Privacy Model and Problem Statement . . . . . 66

5.4 Oblivious NFA evaluation . . . . . 69

5.5 Oblivious Ukkonen NFA Evaluation . . . . . 79

5.6 Experiments . . . . . 86

5.7 Related Work and Discussion . . . . . 92

5.8 Chapter Summary . . . . . 95

**6 Concluding Remarks 97**

6.1 Summary of the Results . . . . . 97

6.2 Future Researches . . . . . 98

**A Construction of Thompson NFA 101**

# Chapter 1

# Introduction

By rapid development of network infrastructure and sensing technologies, massive amount of streaming and unstructured data, such as network access log, GPS trajectory of moving objects, and human genomes, have emerged in various fields. Extracting desired information from such massive data stream is the basic and important task. For example, *network intrusion detection system (NIDS)*, *GPS trajectory analysis*, and *clinical diagnosis* based on patient's genomic data are real world applications in information extraction on streaming and unstructured data.

*Regular expression matching* is given a regular expression $R$ and a text $T$, to find all occurrences of $R$ in $T$. Regular expression matching is a long-standing field of computer science with numerous theoretical and practical studies [10,12,20,44,61]. By the recent emergence of massive data, regular expression matching technologies are becoming increasingly important. However, the new types of problems of regular expression matching in practical application for massive data is appeared. We especially focus on following three problems of them in this thesis:

- The problem called massive regular expression matching which process large number of regular expression patterns against an input text. is one of the new task of pattern matching applications. This task requires very expensive cost on CPU

and thus it is difficult to solve the problem efficiently for an software on single processing core.

- There have been increasing demand on information extraction from time series multidimensional numerical value data such as network access log, stock price data, and GPS trajectory data. However, it is difficult to apply existing regular expression matching methods directly to such time series multidimensional numerical values.

- Pattern matching on genomic data to evaluate the patient's risk of getting specific diseases is attracting much attention by recent advancing of biomedical technology. In such application, we need to consider the problem of privacy leakage of contributing parties.

Our research goal is to develop an efficient regular expression matching technology that applicable to the real world application. To achieve this goal, we tackle the three specific problems of regular expression matching described above and develop efficient algorithms based on *bit-parallel approach*. In this study, we utilize the flexibility and efficiency of bit-parallel non-deterministic finite automata (NFA for short) simulation to develop efficient algorithms.

Bit-parallel approach is a method of designing efficient algorithm developed in early 1990s. This approach utilizes parallelism of Boolean and arithmetic operations on computer words. SHIFT-AND algorithm proposed by Beaza-Yates and Gonnet [4] as well as Wu and Manber [69] is a efficient pattern matching algorithm for exact string pattern based on bit-parallel approach. In this algorithm, in the preprocessing, we construct NFA for short, and transform its transition table into a set of bitmasks. In the searching, we represent a state set of the NFA as a bitvector in registers. Then, we simulates NFA transition efficiently by using Boolean operation and shift operations between stateset bitvector and a set of bitmasks constructed in preprocessing.

Moreover, Navarro and Raffinot proposed Extended SHIFT-AND algorithm [45], an extension of SHIFT-AND for more complex regular expression, including class of characters, optional character, bounded and unbounded repeats, and wildcard operators. This algorithm efficiently simulates a consecutive $\varepsilon$-transitions in an NFA by using arithmetic operations.

The rest of this thesis is organized as follows. In chapter 2, we give the basic notations and definitions used in this thesis. In Chapter 3, we study *large-scale regular expression matching problem*, which is problem of, given a set of input regular expressions $\mathcal{P} = \{P_1, \cdots, P_k\}$ and an input text $T$, finding all occurrences of each regular expression of $\mathcal{P}$ in $T$. We use *graphic processing unit* (*GPU* for short) as target hardware, which is one of the most popular parallel hardware accelerators in data engineering [19, 72], and high performance computing field [49, 62] in this chapter. As a target pattern class, we consider *extended string patterns* [44] which are useful subclass of regular expressions in linear form consists of letters, classes of letters, optional letters, and bound and unbounded repeats of letters. Contrary to exact string matching, this class of regular expression has a problem that sequences of $\varepsilon$-transitions invoke costly $\varepsilon$-closure calculations. To solve this problem, we propose the hierarchically decomposed finite automaton called *hierarchical non-deterministic finite automaton* (*HFA* for short), and give its parallel simulation algorithm on GPU based on Extended SHIFT-AND algorithm. A key to our algorithm is HFA structure to reduce the number of synchronization between GPU cores by localizing $\varepsilon$-closure calculation in each computer words. Our algorithm is fast when average size of input patterns is longer than $w$, the bit length of a computer word. Experimental results showed that our algorithm works faster than PFAC algorithm [35] proposed by Lin et al. by a factor of 150.

In Chapter 4, we study regular expression matching problem on trajectory data of moving objects. The problem is, given a trajectory regular expression $P$ of size $m$,

text trajectory $T$ of size $n$, and threshold parameter $r \geq 0$, to find all position $j$ in $T$ that every subtrajectory $T[i..j]$ satisfies the similarity condition defined in the chapter. Trajectory data which is time series of coordinates of moving objects is increasing rapidly by advancing sensing technology and the wide spread of mobile GPS devices, and it has been attracted match attention [48, 71]. In this chapter, we first formulate regular expression matching problem on trajectory data under similarity condition based on $L_\infty$ distance. Then, we give the efficient online bit-parallel pattern matching algorithm for the problem. The difficulty of this problem is to find matched pattern points to the input point of text trajectory $T$ efficiently to simulate NFA transition. This is the bottleneck to break $O(nm)$ worst case time. To overcome this difficulty, we propose simple and compact point location index that answers a set of pattern points matches to the input point. We combine fast table look using this spatial index and bit-parallel NFA simulation to achieve efficient trajectory regular expression matching. The proposed algorithm solves regular expression matching problem on trajectory data in $O(nm \log\log n / \log n)$ time using $O(m)$ space on unit cost word RAM. Furthermore, our algorithm supports popular geometric query such as OD queries and its variant widely used in geographic information systems (GIS for short).

In Chapter 5, we study *secure regular expression matching problem*. In this problem, there are two participants, the one called an *automaton holder* ($AH$) has an automaton $A$ transformed from a input regular expression $R$, and the other called a *text holder* ($TH$) has a text $T$ to be tested. The automaton and the text are secret information for AH and TH, respectively. Secure regular expression matching problem is to report whether there are occurrences of regular expression R in the input text T or not to AH, and nothing to TH, without any leakage of their secret information. As a main result, we propose an efficient two party secure protocol for NFA evaluation called *oblivious NFA evaluation (ONE)* runs in $O(nm^2)$ time, $O(nm^2)$ communication and $O(nm)$ round using $O(m^2)$ space, where $m$ is the size of a regular expression $R$ and $n$

is the length of text $T$. A key to our improvements from the DFA simulation based method [63] is that we extend the idea of bit-parallel NFA simulation to arithmetic operations on encrypted transition table of NFA by additive homomorphic encryption [47] that supports addition of encrypted integers without encryption. Our protocol takes full advantage of NFA's compactness against DFA, making it exponentially faster than DFA simulation based method in worst-case complexity. Our protocol is the first polynomial time protocol for secure regular expression matching. We also consider secure approximate string matching problem, and propose the modified protocol of ONE that runs in linear time with respect to $m$. The algorithms we proposed in this chapter are secure against a semi-honest model. Experimental results showed that our protocols are significantly faster against the DFA based method proposed by Troncoso et al. [63] on artificial data and real world DNA short read data. In Chapter 6, we finally give the conclusion and future studies.

# Chapter 2

# Preliminaries

In this chapter, we give the basic notations and definitions on strings, regular expressions, finite automata, the model of computation, and basic bit-parallel NFA simulation algorithm used in this thesis.

## 2.1 Basic Definitions

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

## 2.2 Regular Expressions

In this section, we give the definition of the regular expressions and its subclass according to [44]. We also define the regular expression matching problem. First, we define the regular expressions:

**Definition 1.** *For an alphabet $\Sigma$, the regular expressions $R$ on $\Sigma$ is defined by the*

*following recursive rules: An empty string* $\alpha = \varepsilon$ *is a regular expression; A letter* $\alpha \in \Sigma$ *is a regular expression; If* $R_1$ *and* $R_2$ *are regular expressions, the concatenation* $R_1 \cdot R_2$*, the union* $R_1|R_2$*, and the Kleene-closure* $R_1^*$ *are regular expressions. Then, the language* $L(R) \subseteq \Sigma^*$ *specified by regular expression* $R$ *is recursively defined as follows:* $L(\alpha) = \{\alpha\}$*,* $L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$*,* $L(R_1|R_2) = L(R_1) \cup L(R_2)$*, and* $L(R_1^*) = L(R_1)^*$*.*

The *size* of $R$ is defined by the number of symbols from $\Sigma \cup \{\varepsilon, \cdot, |, *\}$ appearing in $R$. We may omit the symbol "$\cdot$" and parentheses if they are clear from context.

Next, we define the extended string pattern [44], the subclass of regular expression.

**Definition 2.** *The extended string pattern* $R$ *on* $\Sigma$ *and its language are definede as follows: An extended string pattern* $R = a_1 \cdots a_m\, (m \geq 0)$*, where for each* $1 \leq i \leq m$*,* $a_i$ *is an expression, called a component, with one of the following forms:*

1. *A letter* $a_i = a \in \Sigma$ *is a component with the language by* $L(a) = a$*.*
2. *A don't care* $a_i = .$ *is a component with the the language by* $L(a) = \Sigma$*. This matches any letter in* $\Sigma$*.*
3. *A class of letters* $a_i = \beta$ *is a component, where* $\beta \subseteq \Sigma$*, with the language* $L(\beta) = \beta$*. As notation, we write* $[ab\cdots]$ *for* $\beta = \{a, b, \ldots\}$*. Note that a letter* $a \in \Sigma$ *and a don't care symbol '.' are a class of letters.*
4. *An optional letter* $a_i = \beta?$ *is a component, where* $\beta \subseteq \Sigma$ *is a class of letters, and* $\beta? \equiv (\beta|\varepsilon)$
5. *Bounded repeats* $a_i = \beta\{x, y\}$*,* $a_i = \beta\{, y\}$*,* $a_i = \beta\{x\}$ *are components with equivalence* $\beta\{x, y\} \equiv (\beta?)^{y-x}\beta^x$*,* $\beta\{y\} \equiv (\beta?)^y$*,* $\beta\{x\} \equiv (\beta)^x$*, respectively, where* $\beta \in \Sigma$ *and* $x \leq y$*.*
6. *Unbounded repeats* $a_i = \beta^*$ *and* $a_i = \beta^+$ *are components, where* $\beta \in \Sigma$ *is a class of letters, and* $\beta^+ = \beta\beta^*$*.*

*For $R = a_1 \cdots a_m$, we define its language by $L(R) = L(a_1) \cdots L(a_m)$ and the size of $R$ is defined as $m$.*

**Example 1.** *For finite alphabet $\Sigma = \{A, B, C\}$, $P_1 = AB^+A?B?C?CB?C?A?$ is an example of the extended pattern.*

Finally, we give the definition of regular expression matching problem. We say that a regular expression $R$ *occurs* in a text $T$ if there exists some substring $T[i..j]$ that matches $R$ , i.e., $T[i..j] \in L(R)$ for some $0 \leq i \leq j \leq n$ where $n$ is the length of $T$. Then, the index $j$ is called an *end-position* of $R$ in $T$. Now we state the regular expression matching problem as follows:

**Definition 3.** *The regular expression matching problem is, given a regular expression $R$ and a text $T \in \Sigma^*$, to report all the end-positions of $R$ in $T$.*

## 2.3 Non-deterministic and Deterministic Finite Automata

In this section, we introduce the definition of finite automata. A *non-deterministic finite automaton* (*NFA* for short) $N = (V, E, \theta, \phi)$ is an edge-labeled directed graph, where $V = \{1, \ldots, m\}$ denotes a set of nodes called *states*, $E \subseteq V \times (\Sigma \cup \{\varepsilon\}) \times V$ is a set of labeled directed edges between states called *transitions*, and $\theta$ and $\phi$ represent the sets of *start* and *accept state*, respectively. In a transition $(j, \sigma, k)$, we call $j$ the *source state* and $k$ the *target state*.

A *deterministic finite automaton* (*DFA* for short) $D$ is an NFA that have no transitions labeled by $\varepsilon$ and all outgoing transitions labeled by different letters on $\Sigma$.

We say that automaton $N$ *accepts* a string $T \in \Sigma^*$ if there is a path from $\theta$ to $\phi$ such that the concatenation of edge labels on the path spells out $T$.

## 2.4 Model of Computation

In the following in this thesis, we assume a *unit-cost word RAM* [3] with word length $w$ as the model of computation. For any bitmask length $s$, a *bitmask* is a bit sequence $X = b_s \cdots b_1 \in \{0,1\}^s$. For bitmask with the length $\ell \leq w$, we assume that the following Boolean and arithmetic operations are performed in $O(1)$ time:

- *bitwise AND* "&",
- *bitwise OR* "|",
- *bitwise NOT* "$\sim$",
- *bitwise XOR* "$\wedge$",
- *left shift* "<<",
- *right shift* ">>",
- *integer addition* "+",
- *integer subtraction* "$-$".

The space complexity is measured in the number of words.

## 2.5 Bit-parallel NFA Simulation Algorithms

Bit-parallelism is a faster calculation method using in-resister parallelism, which takes advantage of the intrinsic parallelism of the bitwise and arithmetic operations inside a computer word. It is also called SWAR (SIMD Within A Register), or broadword computation [65]. In this section, we introduce NFA simulation algorithms based on bit-parallelism.

### 2.5.1 SHIFT-AND Algorithm

The SHIFT-AND algorithm proposed by Beaza-Yates and Gonnet [4] as well as Wu and Manber [69] is a pattern matching algorithm for *exact string pattern*, where an input pattern $R = r_1 r_2 \cdots r_m \in \Sigma^m$ is a string. Algorithm 1 shows pseudocode of the algorithm.

In the preprocessing of this algorithm, we first construct an linear form NFA $N(R) = (V, E, \Theta, \Phi)$. To simulate this NFA, we use $m$-bit bitmask $Ch[\alpha]$ , $I$, and $Acc$ defined as follows:

- Character mask $Ch[\alpha] \in \{0,1\}^m \quad (\alpha \in \Sigma)$ is the $m$-bit mask. For each $0 \leq i \leq m$ and $\alpha \in \Sigma$, $Ch[\alpha][i] = 1$ indicates a set of transition $E$ contains the transition $(i, \alpha, i+1)$. In other words, $Ch[\alpha]$ is the bitvector representation of transitions labeled by $\alpha$.

- Start state mask $I$ is the $m$-bit mask that sets 1 at the first bit.

- Accept state mask $Acc$ is the $m$-bit mask that sets 1 at the $m$th bit.

In the searching phase, SHIFT-AND algorithm updates the stateset bitvector $S$ using following update formula (Algorithm 1, line 16). We note that $t_i \in \Sigma$ is the current letter of the input text.

$$S \leftarrow ((S \ll 1) \mid I) \;\&\; Ch[t_i] \tag{2.1}$$

The algorithm check the accept state by following code (Algorithm 1, line 17 – 18):

$$\textbf{if} \quad S \;\&\; Acc \neq 0^m \quad \textbf{then} \quad \text{report occurrence} \tag{2.2}$$

### 2.5.2 Extended SHIFT-AND Algorithm

The Extended SHIFT-AND algorithm, proposed by Navarro and Raffinot [45], is a pattern matching algorithm for an extended string pattern. In the preprocessing, we

**Algorithm 1** SHIFT-AND Algorithm

```
1: procedure SHIFT-ANDmain(P, T)
2:     Ch ← Preprocess(P)
3:     Seach(T, Ch)
4: end procedure
5: procedure Preprocess(P)
6:     I ← Bit(0)
7:     Acc ← Bit(m)
8:     for i ← 1 to m do
9:         Ch[P[i]] ← Ch[P[i]] | Bit(i)
10:    end for
11:    return Ch
12: end procedure
13: procedure Search(Ch, T)
14:    S ← 0^m
15:    for i ← 1 to n do
16:        S ← ((S ≪ 1) | I) & Ch[T[i]];
17:        if S & Acc ≠ 0^m then
18:            report an occurrence at i;
19:        end if
20:    end for
21: end procedure
```

construct NFA $N(R) = (V, E, \Theta, \Phi)$ corresponding to input extended string pattern $R$. Figure 2.1 is an NFA of the input pattern $P_1$ on Example 1. Let $B \subset V$ is a set of states that induces a maximal connected component consisting only of $\varepsilon$-transitions $\varepsilon$-transitions. The NFA in Figure 2.1 has two $\varepsilon$-blocks $B_1 = \{2, 3, 4, 5\}$, and $B_2 = \{6, 7, 8, 9\}$.

Then, constructs a set of bitmasks defined as follows.

- $Ch[c]$ is a $w$-bit mask indicates that indicates all position of forword letter transition labeled with $c$ i.e. for each $0 \leq i \leq m$ and $c \in \Sigma$, $Ch[c][i] = 1$ means that a set of transitions $E$ contains the transition $(i, c, i + 1)$.

- $Rep[c]$ is the $w$-bit mask that indicates all positions of self-loop transition labeled with $c$ i.e. $Rep[c][i] = 1$ means that a set of transitions $E$ contains the transition $(i, c, i)$.

- $Eblk$ is the $w$ bit mask that sets 1 at all positions of $\varepsilon$-blocks.

- $Ebeg$ is the $w$ bit mask that sets 1 at the all previous positions of the lowest position of every $\varepsilon$-transition.

- $Eend$ is the $w$ bit mask that sets 1 at the highest position of every $\varepsilon$-block.

In the searching phase, we search for all the occurrence of $R$ in $T$ using bit parallel simulation of NFA transition. First, the forword and self-loop letter transitions simulates by following code:

$$S \leftarrow(((S \ll 1) \mid I) \ \& \ Ch[c]) \tag{2.3}$$

$$\mid (S \ \& \ Rep[c]) \tag{2.4}$$

Line 2.3 which is the same code of SHIFT-AND makes the forword letter transition, and line 2.3 makes self-loop transitions. Then, following three lines of code makes

$\varepsilon$-transitions:

$$High \leftarrow S \mid Eend \tag{2.5}$$

$$Low \leftarrow High - Ebeg \tag{2.6}$$

$$S \leftarrow (Eblk \ \& \ ((\sim Low) \oplus High)) \tag{2.7}$$

$$\mid S \tag{2.8}$$

At line 2.5, we set the end bit of $\varepsilon$-block in $S$, and store it in temporary mask $High$. At line 2.6, we flip all bits lower than or equal to the lowest 1 bit of all and the previous bits of each $\varepsilon$-block in $High$, and store it in temporary mask $Low$. At line 2.7, we generate an bitmask that set 1 at all positions properly higher than the lowest 1 bit of all and the previous bits of each $\varepsilon$-block in $S$, and we add the change caused by $\varepsilon$-transitiont $S$ at line 2.8. Finally, we check the acceptance by performing the same acceptance test as SHIFT-AND method.

The extended SHIFT-AND algorithm runs in $O(m+|\Sigma|)$ time for the preprocessing phase, $O(n\lceil m/w \rceil)$ time for the searching phase, and it consumes $(2|\Sigma| + 4)\lceil m/w \rceil$ word for memory space, where $w$ is the bit length of registers and $m$ is the number of components of $P$.

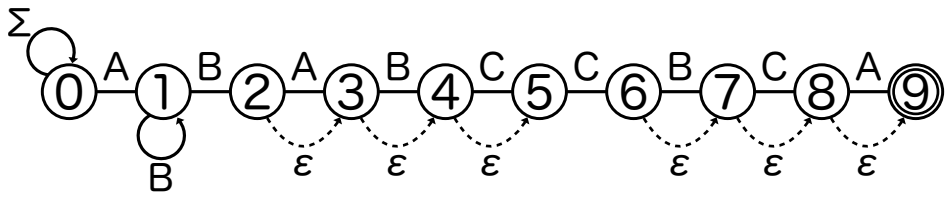


Figure 2.1: The NFA of $P_1$ on example 1.

# Chapter 3

# Faster Massive Pattern Matching System on GPU

In this chapter, we study the multiple regular expression matching problem for the class of extended string pattern, which is a problem of, given a set of extended string pattern $\mathcal{P} = \{P_1, \cdots, P_k\}$ and a text $T$, finding all occurrences of $P_i$ in $T$ for $1 \leq i \leq k$ in $T$.

This chapter is based on the papers [55] and [53]. Detail of the organization is as follows: The hierarchical NFA and theoretical analysis of its simulation algorithm described in Section 3.3.2 are based on the result from the paper [55]. GPU implementation of HFA and experimental results are based on the paper [53].

## 3.1 Background

By rapid growth of network infrastructure and sensor technologies, massive amount streaming and unstructured data have emerged in various fields. The *massive online pattern matching* is one of the most important problems in where a pattern matching system has to work with a large number of input pattern against a huge amount of

data. For instance, in network intrusion detection systems (NIDS) such as Snort[1], a system must process real time pattern matching of several hundreds (or thousands) of complex regular expressions against data streams.

In spite of the importance of this problem, it is a quite CPU-intensive task, and it is difficult for a software on CPU to efficiently perform massive pattern matching for data streams in real time. For this reason, researches working on programmable multi-core hardware have attracted increasing attention for recent years.

A GPU (Graphics Processing Unit) is a special hardware, originally designed for graphics processing, equipped with several thousands of general purpose processing units and high memory band width. Because of its highly parallel processing architecture, GPU is recently used for general purpose (non-graphics) computation to achieve fast processing of CPU-intensive tasks. This technique is called *general purpose computing on GPUs (GPGPU)* and there are intensive research on high performance computing field [31, 49, 62].

### 3.1.1 Main results

In this chapter, we propose fast pattern matching system on GPU **HFA-BP** designed for performing massive pattern matching of a large number of very long patterns against a data streams. As the target class of patterns, we consider a subclass of regular expression, called *extended string patterns* [44], each of which has a linear form and consists of letters and popular matching operators such as letter classes, don't cares, optional letters, and bounded and unbounded repeats of letter. Such class of regular expressions are widely used in bioinformatics, complex event processing (CEP), and network intrusion detection systems.

For example, "$P_0 = [\mathtt{AB}]^{+}\mathtt{B}.\{\mathtt{1},\mathtt{3}\}[\mathtt{BC}]?.^{*}\mathtt{C}$" is an extended strings over alphabet $\{A, B, C\}$ in the proposed class, which represents any string starting from more than

[1] https://www.snort.org

one repeat of $A$ and $B$, and followed by the followings: letter $B$, a gap of length from one to three, optionally one of $B$ or $C$, then a gap of length zero or more, and finally, terminal letter $C$.

A basic idea of the proposed system is to transform a given extended string pattern into an equivalent NFA (non-deterministic finite automaton), and then simulate its matching efficiently on a GPU by using inter-core and intra-register parallelism of the hardware. An obvious approach to massive string matching first transforms a pattern string into a DFA (deterministic finite automaton) and then makes its deterministic simulation on a hardware, say, an ordinary CPU. For our target class of regular expressions, however, this obvious transformation does not work at all since this transformation from a regular expression to a DFA causes well-known exponential brow-up of the number of states of a DFA [23].

The key of our approach is to use a compact representation of a NFA suitable to efficient parallel execution on GPU. In this representation, called the *hierarchically decomposed NFA* (or *HFA*, for short), a long NFA is decomposed into a collection of small NFAs, each of which is encoded by a bit-vector stored in a computer word, and organized into a hierarchical structure that consists of an upper module and a set of lower modules. In running time, we perform efficient simulation of the NFA over a set of registers using the collection of bitmasks on GPU. This simulation is done by combining fast bit-parallel simulation [45] of the state transition of small NFAs using intra-register parallelism in lower modules, and fast state aggregation and propagation using inter-core parallelism in an upper module. We have two subtypes of our system, called HFA-BP-AS and HFA-BP-BS, depending on the implementation of state aggregation. In addition, we give theoretical analysis of computational complexity of the proposed system.

Finally, we ran experiments on the amino-acid sequence data to evaluate efficiency of the proposed system. In the experiments, we observed that the proposed pattern

matching system showed linear scale-up property when the input and pattern sizes increase as expected by theoretical analysis. The proposed system on GPU was also two order of magnitude faster than a conventional pattern matching algorithm on CPU. We also observed that the subtype HFA-BP-AS is slightly faster than another subtype HFA-BP-BS.

### 3.1.2 Related works

We explain a recent research of massive pattern matching problem. Lin and Tsai [35] proposed an extension of Aho-Corasick algorithm [2] on GPU. This algorithm supports exact string matching for multiple patterns. Wu, Diao, and Rizvi [68] gave pattern matching algorithm for complex event processing. Margara and Cugola [37] proposed complex event processing engine on GPU.

## 3.2 Preliminaries

In this section, we introduce our pattern matching problem our target hardware.

### 3.2.1 Multiple regular expression matching

Let $R$ be a *regular expression* and on finite alphabet $\Sigma$ and a *text* $T \in \Sigma^*$ is a sequence of letters on $\Sigma$. We say that a regular expression $R$ *occurs* in $T$ of length $n$ if there exists some $1 \leq i \leq j \leq n$ such that $T[i..j] \in L(R)$ holds. Now, we state the multiple regular expression matching problem as follows:

**Definition 4.** *Multiple regular expression matching problem is, given a set of regular expressions* $\mathcal{P} = \{P_1, P_2, \cdots, P_k\}$*, and a input text* $T$*, to report the occurrence of* $P_i$ *in* $T$ *for all* $1 \leq i \leq k$*.*

### 3.2.2 GPU

*Graphics processing unit* (GPU) is a special hardware for graphics processing. It has several sets of processing unit, called *stream multi-processor* (SM), and each SM has $Q$ SIMD processing core and *shared memory* of size $M$ which can access in low-latency. For example, a high-end GPU, Nvidia GeForce GTX 480 released in 2011, has 15 SMs with $Q = 32$ cores and shared memory of $M = 64$ KB, totally equipped with 480 cores.

A GPU achieves fast graphics processing by running hundreds of threads using SMs with shared memory. The use of a GPU instead of a CPU to perform general purpose computation is known as *GPGPU (General Purpose computing on GPUs, or GPU)* and its researches have attracted much attention in recent years, especially in high performance computing fields [6, 36, 66].

## 3.3 Pattern Matching System on GPUs

In this section, we present our pattern matching system on GPU based on simulation of NFAs using **HFA-BP** algorithm.

### 3.3.1 Architecture of HFA-BP family

The proposed pattern matching system, called HFA-BP system, is a core-parallel implementation of hierarchical NFA (HFA) with bit-parallel extended SHIFT-AND simulation [45] on bit-based lower modules. The hardware configuration of the system consists of two parts: *host* side computing using CPU and *device* side computing using GPU. Overview of our system works is as follows:

1. Construct a set of bitmasks for each pattern $P_i \in \mathcal{P}$ on the host side.

2. Transport sets of bitmasks to the device side.

3. Run pattern matching algorithm in parallel using sets of bitmasks on the device side.

4. Return the results of pattern matching to the host side.

The HFA-BP system is further classified into the following two sub-types according to the implementation of an upper module:

- *HFA-BP-BS*: A variation based on Bit-Based upper module with Bit-Serial aggregation and Bit-Parallel propagation.

- *HFA-BP-AS*: A variation based on Array-Based upper module with Core-Parallel Collision-Free Concurrent-Write aggregation and serial propagation.

### 3.3.2 The outline of HFA-BP simulation algorithm

We show the outline of our bit-parallel pattern matching algorithm for extended pattern matching problem. In this algorithm, we use the special NFA tailored for efficient parallel computation of NFA transitions, called *hierarchical NFA* (*HFA* for short). The HFA is a set of NFAs, where each of NFA is stored into a $w$-bit register, and is consists of two layers: a *upper module* $UM$ and *lower modules* $LM = \{LM_1, \ldots, LM_k\}(k > 0)$. For every lower module $LM_i \in LM$, it stores NFA of length $w$ and an upper module stores NFA of length $k$. Figure 3.1 shows the HFA of Example 2.1 with $w = 4$.

Algorithm 2 shows the outline of HFA simulation on input text $T$. The algorithm assigns GPU threads to each lower modules and a upper module. The HFA is simulated by following steps.

Step 1: Distribute the active state from $UM$ to $LM$s (Line 6).

Step 2: Simulate partial NFA transition in parallel by a letter $T[i]$ on each $LM$s using bit-parallel NFA simulation (Line 7).

Step 3: Aggregate the active state from $LM$s to $UM$ (Line 8).

Step 4: Propagate active states on $UM$ (Line 10).

Details of the algorithm are described in following subsections.

### 3.3.3 NFA transitions on the lower Modules

In lower modules of HFA, we use extended Shift-AND method proposed by Navarro and Raffinot [45] to simulate NFA transition in each lower modules. For every $0 \leq i \leq k$, a lower module $LM_i$ can be represented as a triplet of two types of bitmasks $LM_i = \langle S, I, BM \rangle$, where $S$ and $I$ are dynamic bitmasks and $BM = \langle Ch, Rep, Ebeg, Eblk, Eend \rangle$ is a set of static bitmasks. $S \in \{0,1\}^w$ represents a set of state set, where $S[j] = 1$ indicates $j$-th state of the lower module NFA is active. $I \in \{0,1\}$ has a information of $i$-th state of upper module NFA. The a set of static bitmasks $BM$ defined as follows:

- $Ch[c]$ is a $w$ bit mask that indicates all positions of letter move labeled with a letter $c$ in input pattern $P$. That is, $Ch[c][i] = 1$ iff the $i$-th state of NFA has a outgoing letter move edge labeled with $c \in \Sigma$.
- $Rep[c]$ is the $w$ bit mask that indicates all positions of self-loops labeled with a letter $c$ in input pattern $P$. That is, $Rep[c][i] = 1$ iff the $i$-th state has a self-loop labeled with $c \in \Sigma$.
- $Ebeg$ is the $w$ bit mask that sets 1 at the previous position of the lowest position of every $\varepsilon$-block.
- $Eblk$ is the $w$ bit mask that sets 1's at all positions of in every $\varepsilon$-block.
- $Eend$ is the $w$ bit mask that sets 1 at the highest position of every $\varepsilon$-block.

In the preprocessing, the algorithm construct a set of static bitmasks $BM$ from an input extended string pattern $P$.

After the calculation of a set of bitmasks, we simulate transitions using runtime algorithm RunExShiftAND shown in Algorithm 3. In Algorithm 3, we simulate NFA transitions on input text letter using bit-parallel technique. At first, the algorithm propagate upper module information to $S$ (line 2), then it perform the character transition (line 4) and $\varepsilon$-closure (line 5). $\varepsilon$-closure is compute by Algorithm 4 using carry propagation by performing integer subtraction.

### 3.3.4 Aggregation and propagation on a upper module

After computing in lower module, the algorithm aggregate the results of lower modules and compute transitions in the upper module. In aggregation phase, the algorithm aggregates the last states of NFA in a set of lower modules using Algorithm 6. All of the threads assigned to lower module attempt to write their information to the upper module. Then, the algorithm propagates active states in the upper module. We propose two different implementation of the upper module to achieve efficient parallel aggregation and propagation describe as follows.

#### Bit-based upper module implementation

First, we introduce the bit-based upper module implementation. In this implementation, the upper module is implemented by a $w$ bit register. In aggregation, since all of the threads cannot write to single register simultaneously, the access by each lower modules is serialized by atomic operation provided by GPU programming environment. After aggregation phase, the algorithm computes $\varepsilon$-closure on upper module by using bit-parallel operation described in Algorithm 4. In aggregation step, it consumes $O(k)$ time because the access by whole threads are serialized and in propagation step, it consumes $O(1)$ time per register.

**Array-Based Upper Module Implementation**

Next, we describe the array-based implementation. In this implementation, the upper module is implemented by a array with $w$ cells. In aggregation, all of the threads can write their information to the upper module simultaneously because the destinations of each threads are different cells of the array. The propagation in upper module, however, the algorithm have to scan the array and perform $\varepsilon$-closure operation in serial manner. In this approach, the aggregation step consumes $O(1)$ time and the propagation step consumes $O(k)$ time.

### 3.3.5 Distribution from an upper module to lower modules

Finally, we describe distribution step from an upper to lower modules shown in Algorithm 5. In this step, threads assigned to lower modules access an upper module. We utilize the property of *shared memory multi-casting* [25], all of the threads assigned to lower modules can read the value of upper module in parallel.

## 3.4 Time and Space Complexity

In this section, we analyze space and time complexity of our GPU pattern matching system. Since the complexities of HFA-BP-BS and HFA-BP-AS are the same, we describe the complexity of HFA-BP-BS only. Let $k = \lceil \ell/w \rceil$ be the number of computer words necessary to store the state set of an NFA with $\ell$ states.

### 3.4.1 Space complexity

In HFA-BP algorithm, HFA has one upper module and $k$ lower modules, where $k = \lceil \ell/w \rceil$. For a upper module, it consists of state set mask $S$ and three $\varepsilon$-transition masks $Ebeg, Eblk, Eend$ of length $w$. For a lower module, it consists of state set mask

**Algorithm 2** The proposed GPU-based pattern matching algorithm for simulating HFA on text $T$.

**shared memory variables:**

Registers $UM$ for upper module, and $LM_1, \ldots, LM_k$ for lower modules.

1: **procedure** HFA-BP(T: text)
2: $n \leftarrow$ the length of text $T$;
3: Set the initial state of the upper module to be active;
4: Simulate $\varepsilon$-transitions in the upper module by EpsClo($UM$);
5: **for** $i = 0$ **to** $n$ **do**
6: **par** $j = 0$ **to** $k - 1$ **do**
7: Distribute active states from the upper module to lower modules by UpperToLower($UM, LM_j$);
8: Simulate transition of $LM_j$ by RunExShiftAnd($LM_j, T[i]$);
9: **for** $j = 0$ **to** $k - 1$ **do**
10: Aggregate active states from lower modules to the upper module by LowerToUpper($UM, LM_j$);
11: **end for**
12: Simulate $\varepsilon$-transitions in the upper module by EpsClo($UM$);
13: **if** the final state of the upper module is active **then**
14: report an occurrence at i;
15: **end if**
16: **end for**
17: **end procedure**

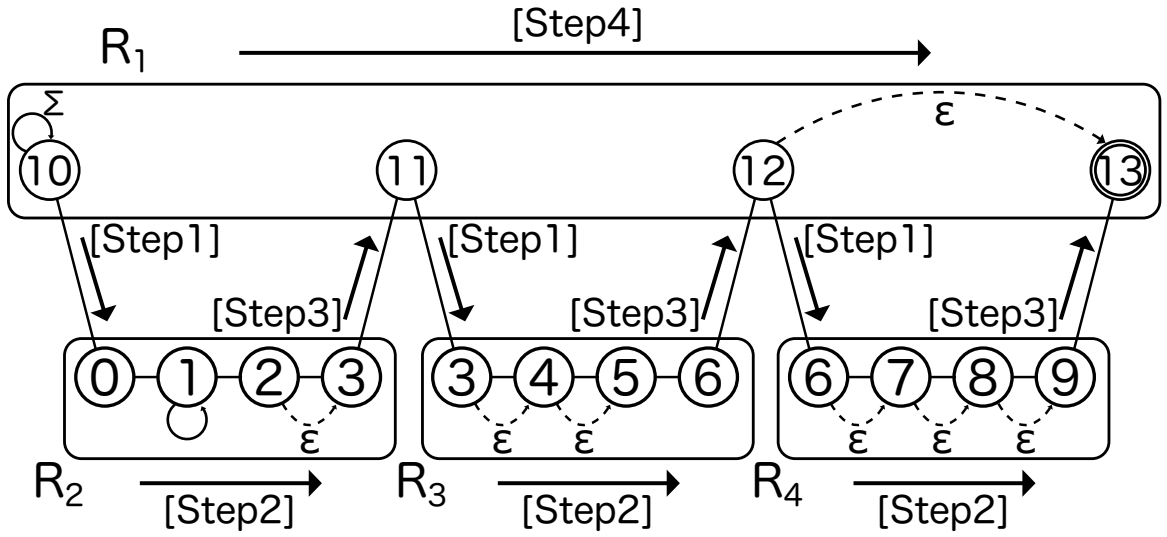


Figure 3.1: The hierarchical NFA constructed from $P_1 = AB^+A?B?C?CB?C?A?$ (Example 1) and How to simulate the transitions of the HFA.

**Algorithm 3** The algorithm that simulates an NFA stored in a register $LM$ by extended Shift-AND method.

1: **procedure** RunExShiftAnd($LM = \langle S, BM \rangle$: module, $T[i]$: letter)
2: $S \leftarrow S \mid I$;
3: EpsClo($LM$);
4: $S \leftarrow (((S << 1) \mid I) \,\&\, Ch[T[i]]) \mid (S \,\&\, Rep[T[i]])$;
5: EpsClo($LM$);
6: **end procedure**

**Algorithm 4** The algorithm that computes $\varepsilon$-closure of an NFA stored in a lower module $LM$.

1: **procedure** EpsClo($M = \langle S, BM \rangle$: module)
2: $Tmp \leftarrow S \mid Eend$
3: $S \leftarrow S \mid (Eblk \,\&\, ((\sim (Tmp - Ebeg)) \wedge Z))$
4: **end procedure**

$S$, three $\varepsilon$-transition masks, and $|\Sigma|$ arrays of bitmasks for character transition and self-loop transition. Therefore, the memory usage of RunHFA is,

$$\begin{aligned} S_{all} &= (2|\Sigma| + 4)\lceil \ell/w \rceil + 4 \\ &= O(|\Sigma|\lceil \ell/w \rceil)\,(\text{words}) \end{aligned}$$

The details are given in Table3.1.

For *multi-stream pattern matching* which is single pattern matching on $s$ streams, our algorithm can deal the task using one set of bitmasks (same as signal stream pattern matching) and $s$ state set mask. In this situation, we consume

$$\begin{aligned} S_{all} &= s(\lceil \ell/w \rceil + 1) + (2|\Sigma| + 3)\lceil \ell/w \rceil + 3 \\ &= O((s + |\Sigma|)\lceil \ell/w \rceil)\,(\text{words}) \end{aligned}$$

memory space.

### 3.4.2 Time complexity

If the NFA size is $\ell$, our algorithm runs in $O(\ell)$ time for preprocessing phase. The RunHFA shown in Fig. 2 simulates NFA transition for each letter $T[i]$ for $i = 1, \ldots n$. In the calculation of NFA transition (line 5 to 12), the procedure UpperToLower (Fig. 5), RunExShiftAnd (Fig. 3), and LowerToUpper (Fig. 6) are performed on every lower module $LM_j (0 \leq j \leq k-1)$, each procedure runs in $O(1)$ time, the total time on lower modules is $O(k) = O(\lceil \ell/w \rceil)$. For the upper module, the procedure EpsClo is performed on upper module runs in $O(1)$ time. Therefore, the total time of NFA simulation of each character in the input text is

$$O(\lceil \ell/w \rceil) + O(1) = O(\lceil \ell/w \rceil)\,(\text{time}). \tag{3.1}$$

From discussions above, we have following theorem:

**Algorithm 5** The algorithm that distributes information of the upper module to the $i$-th lower module $LM_i$.

```
1: procedure UpperToLower(UM, LM_i: modules)
2:     if UM.S[i] = 1
3: then LM.I ← 1;
4:     else LM.I ← 0;
5: end procedure
```

**Algorithm 6** The algorithm that aggregates the result of $i$-th lower module $LM_i$ transition to the upper module.

```
1: procedure LowerToUpper(UM =, LM_i: modules)
2:     if LM_i.S[w] = 1
3: then UM.S[j] ← 1;
4:     else UM.S[j] ← 0;
5: end procedure
```

Table 3.1: Space complexity of RunHFA algorithm.

| module | $S$ | $\varepsilon$-move | char-move | #module | total |
|---|---|---|---|---|---|
| Upper | 1 | 3 | 0 | 1 | 4 |
| Lower | 1 | 3 | $2\|\Sigma\|$ | $\lceil \ell/w \rceil$ | $(4+2\|\Sigma\|)\lceil \ell/w \rceil$ |

**Theorem 1.** *Our proposed pattern matching algorithm HFA-BP solves extended pattern matching problem in* $O(n\lceil \ell/w \rceil)$ *time using* $O(\ell)$ *time for preprocessing and* $O(|\Sigma|\lceil \ell/w \rceil)$ *space, where* $n$ *is the length of the input text,* $\ell$ *is the size of NFA of input pattern,* $w$ *is the computer word length.*

## 3.5 Experiments

To evaluate the performance of our proposed system, we conducted computational experiments.

### 3.5.1 Experimental settings

The experiments were performed on workstation equipped with a host CPU 2.8GHz Intel Core i7 processor with 12GB RAM and a device GPU Nvidia GTX 480 in Fermi micro-architecture, operating on Debian GNU/Linux 6.0.5. We used g++ 4.6.3 and CUDA 5.0 as the compiler.

In the experiments, we used 10 MB of amino-acid sequence data whose alphabet size is $|\Sigma| = 20$ from ExPASy : SIB Bioinformatics Resource Portal[2] as the input text and random generated extended string regular expressions as the input pattern. For Exp.4, we also obtained 771 real PROSITE patterns from ExPASy.

We compared following methods:

- **HFA-BP-BS**: the GPU implementation of the proposed bit-parallel HFA simulation algorithm of Sec. 3.3 in CUDA language[3].

- **HFA-BP-AS**: the GPU implementation of the proposed bit-parallel HFA simulation algorithm, where the upper module is implemented by array.

[2] `http://expasy.org/`

[3] `https://developer.nvidia.com/category/zone/cuda-zone`

- **TFA-AP**: the GPU implementation of Thompson's NFA simulation algorithm [61], it consumes $O(n\ell)$ time and $O(\ell)$ word space using integer array of length $\ell$ for representing a state set. In pattern matching, GPU threads are assigned to each state array and computes transitions of each state. Therefore total number of threads is $\ell$.

We also implemented non-parallel version of HFA-BP-BS running on CPU, called HFA-BP-BS-CPU, using C++ language.

### 3.5.2 Exp. 1: Comparison against the size of the text

In this experiment, we measured running time of algorithms by varying the size of the input text from 1MB to 10MB. We fix the input pattern length as $\ell = 130$ and the number of input pattern as 100. Fig. 3.2 shows the running time of three algorithms described above. In Fig. 3.2, the running time of all the algorithm grew linear with respect to the size of the input text. HFA-BP-BS is approximately 20 times faster than TFA-AP and approximately 1.3 times faster than HFA-BP-AS.

### 3.5.3 Exp. 2: Comparison against the number of the patterns

In this experiment, we measured running time of algorithms by varying the number of the input patterns from 100 to 1000. We fix the input pattern size as $\ell = 130$ and the size of the text as 1MB. Fig. 3.3 shows the result. TFA-AP algorithm couldn't run with 500 patterns or more because the memory shortage in shared memory. HFA-BP-BS and HFA-BP-AS could run up to 1000 patterns. This result shows us the advantage of compact representation of NFA on lower modules.

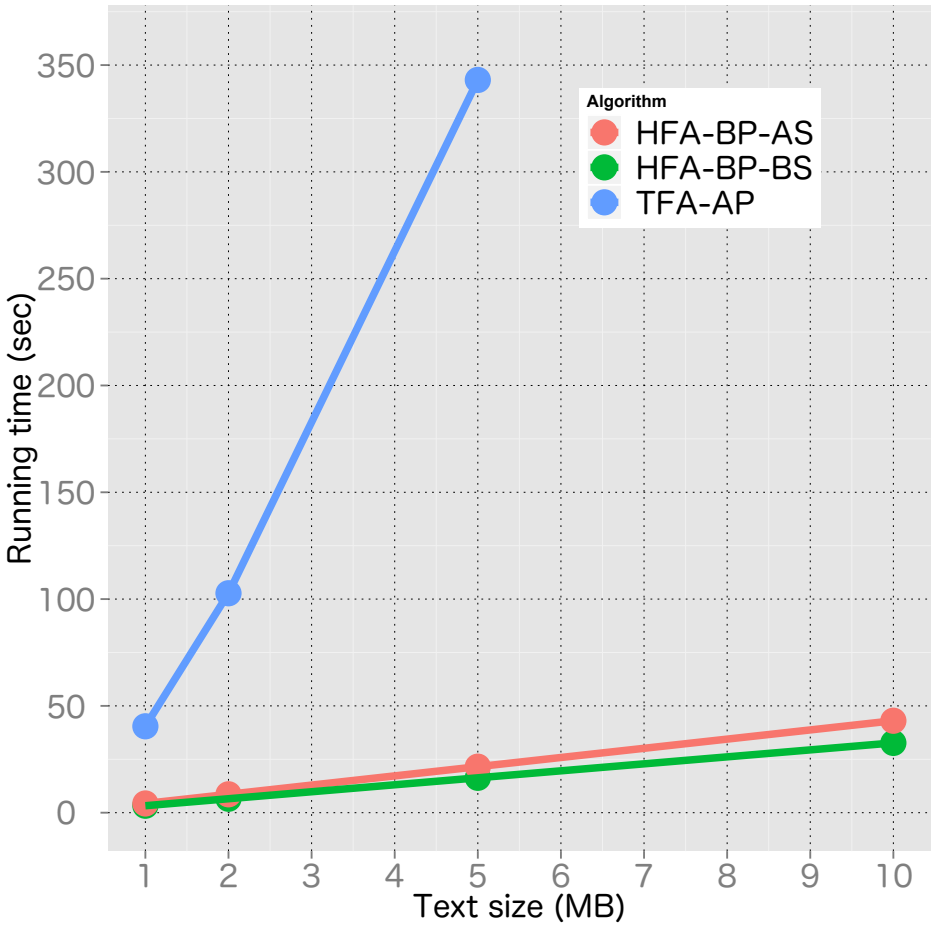


Figure 3.2: The running time of the HFA-BP-BS, HFA-BP-AS, and TFA-AP by varying the size of the input text from 1MB to 10MB. The number of threads invoke of HFA-BP-BS and HFA-BP-AS is $\lceil \ell/w \rceil + 1$. The number of threads for TFA-AP is $\ell$.

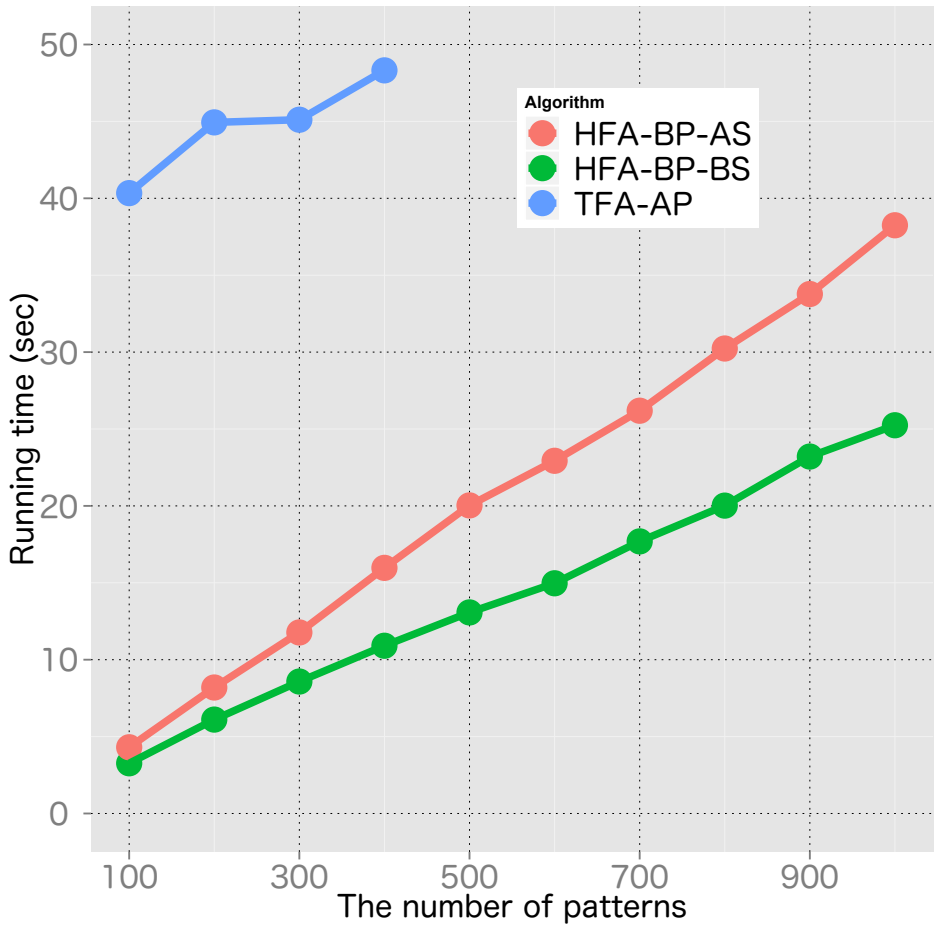


Figure 3.3: The running time of the HFA-BP-BS, HFA-BP-AS, and TFA-AP by varying the number of patterns from 100 to 1000.

### 3.5.4 Exp. 3: Comparison against input pattern length

We compared the running time of HFA-BP-BS-GPU, HFA-BP-BS-CPU, and HFA-BP-AS-GPU by varying the length of input pattern. The input pattern length measured by the number of words $L = \lceil \ell/w \rceil$. The pattern length varied from 1 to 15 and we fix the number of patterns as 100. In Fig. 3.4, we observe the running time of GPU implementation HFA-BP-BS-GPU and HFA-BP-AS-GPU is faster than HFA-BP-BS-CPU when $L = 2$ or longer and We also observe that HFA-BP-BS-GPU is two times faster than HFA-BP-BS-CPU when $L = 12$.

### 3.5.5 Exp. 4: Benchmarking on real pattern dataset

We compared the running time of HFA-AP-BS-GPU, and PFAC algorithm on 771 PROSITE patterns and 2MB of amino acid text. The implementation of PFAC is obtained from the author's site[4]. To apply PFAC algorithm to the PROSITE pattern, we expand each PROSITE pattern into its language, and then search against the text. The specifications of the PROSITE patterns and the expanded strings is shown in table 3.2. Table 3.3 shows that the proposed algorithm is 156 times faster than PFAC algorithm.

## 3.6 Chapter Summary

In this chapter, we propose fast massive pattern matching system on GPU. In the proposed system, we give the GPU extension of bit-parallel pattern matching algorithm HFA-BP based on hierarchical module decomposition of NFA. The key of the proposed algorithm is In the experiment, our algorithm achieved twenty times faster than straight forward Thompson NFA implementation on GPU. The key of fast pattern matching is the compact representation of NFA using bit-parallel technique and

[4] `https://github.com/pfac-lib/pfac`

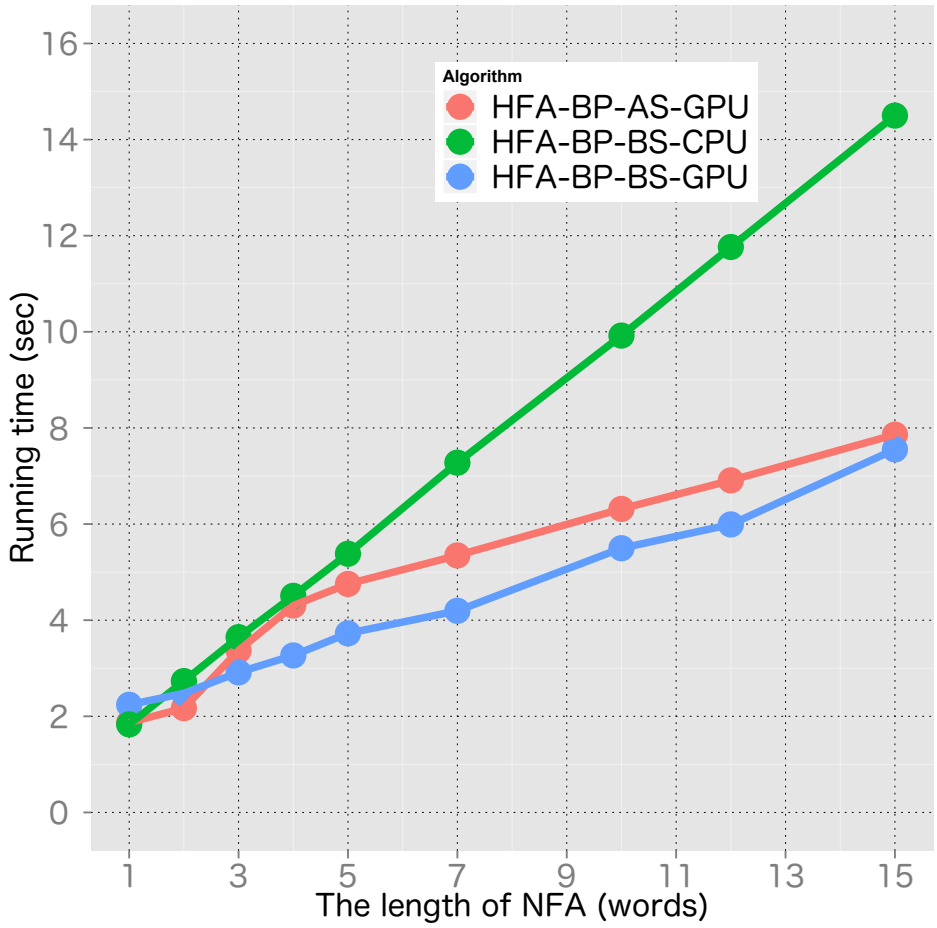


Figure 3.4: The running time of HFA-BP-BS-GPU, HFA-BP-BS-CPU, and HFA-BP-AS-GPU by varying the input pattern length.

Table 3.2: The specifications of the PROSITE patterns and its expanded strings.

| | #patterns | size |
|---|---|---|
| PROSITE | 771 | 18KB |
| expanded | 417,991,018 | 5.3GB |

module decomposition for GPU parallel execution.

In our algorithm, the aggregation step is the bottle neck of computing HFA transition. As the future work, we will consider efficient parallel aggregation method. It is also an interesting problem to extend our system to real world stream data such as GPS trajectories.

Table 3.3: The running time and speedup of GPU implementaions.

| | PFAC | HFA-AP-BS | speedup |
|---|---|---|---|
| PROSITE | 1567.7 sec | 10.8 sec | 156 times |

# Chapter 4

# Efficient Complex Temporal Pattern Detection over Trajectory Streams

In this chapter, we study the regular expression matching problem on trajectory data. First, we formulate the regular expression matching on trajectory data under $L_\infty$ distance extended from the problem on strings. Then, we present efficient bit-parallel algorithm for the online trajectory regular expression matching problem that runs in $O(nm \log\log n / \log n)$ time using $O(m)$ space and $O(m \log\log n)$ preprocessing time on unit-cost word RAM with $w = \Theta(\log n)$ bit computer word, where $n$ and $m$ are the size of an input text trajectory and an input trajectory regular expression. The key to our algorithm is combining bit-parallel NFA simulation for calculating efficient NFA transitions and the simple point location data structure for detecting the matched pattern point with input text trajectory point. In practice, this algorithm expect to work efficiently on the machine equipped with long computer word such as GPU and hardware accelerator (e.g. Intel Xeon Phi). This chapter is based on the papers [56,57].

## 4.1 Background

Recent advancing of sensing technology and the wide spread of mobile GPS devices, huge amount of trajectory of moving object have emerged. Since trajectory data is time series $d$-dimensional numerical data, it is difficult to apply the regular expression matching technique to this problem directly. In this chapter, we propose a new approach to this problem, called geometric bit-parallel matching, by combining efficient point location data structures and bit-parallel simulation of non-deterministic finite automata (NFA for short), which has advantages in flexibility and efficiency compared to the previous approaches. Based on this approach, we introduce a subclass of geometric temporal patterns, called extended geometric sequence patterns, which are simple regular expressions consisting of concatenation, bounded and unbounded gaps, and repetition over rectangular regions with size $r$. This class can represent popular geometric queries including bounded and unbounded duration OD queries, by-way-of OD queries, and trajectory pattern queries widely used in geographic information systems (GIS for short). In the case of $L_\infty$ distance over points and any fixed dimension $d$, we present an efficient online algorithm that detects all occurrences of a given pattern of size $m$ in a continuous stream of $n$ points in $O(nm \log\log n / \log n)$ time, while the previous fastest algorithms have worst-case time complexity $O(nm)$ for exact pattern matching in 1-dim. Hence, this is the first result achieving sub-quadratic time in geometric temporal pattern matching for a subclass of regular expressions. Finally, we also show computational experiments to evaluate the usefulness of the proposed algorithm.

## 4.2 Related work

Information retrieval on trajectory data have been studied intensively since 1990s in data engineering and geometric information systems (GIS) field. Faloutsos et al. [15] proposed the index structure based on R*-tree for subsequence matching on one-

dimensional trajectory data.

For online trajectory stream processing, a clustering algorithm for anomaly detection is proposed by Piciarelli et al. [48]. To the best of our knowledge, there is no studies about online pattern matching on multidimensional trajectory data. The online trajectory matching algorithm proposed by Sasakawa et al. [52] is approximate trajectory matching algorithm on encoded trajectory by the encoding scheme also proposed in the paper. In this algorithm, the trajectory similarity precision is depends on coding scheme, we cannot set similarity parameter arbitrarily without decoding.

In complex event processing (CEP) and event stream processing (ESP), there are studies on pattern matching for time series 1-dimensional numerical value sequence data. Sadri et al. [50] proposed the pattern matching algorithm inspired by Knuth-Morris-Pratt (KMP) algorithm [29]. Harada [21] proposed the pattern matching algorithm inspired by Boyer-Moore (BM) algorithm [10]. Saito et al. [51] proposed the pattern matching algorithm inspired by SHIFT-AND algorithm [4, 69]. From the view of practical aspect, these algorithms show good performance on the computational experiments, but from the view of theoretical aspect, the worst case complexity of these algorithms are $O(nm)$ time that is same as naive method, where $n$ and $m$ are the size of input text trajectory and pattern trajectory.

**Recent breakthrough: The problem of computing discrete Frechet distance** Agarwal et al. [1] consider the decision problem of discrete Frechet distance problem. For the problem, the algorithm runs in $O(nm)$ time based on dynamic programming is known in previous studies. Discrete Frechet distance is informally, given input trajectory A and B of length $m$ and $n$, the minimum leash length to connect two frogs, jump along with $n$ and $m$ specific stones whether without backtracking on trajectory A and B, respectively. The decision version of this problem [14] is, give trajectory $A$ and $B$, and non-negative real number $r$, decide whether the discrete Frechet distance between

$A$ and $B$ is shorter than $r$ or not. Agarwal et al. give the algorithm based on following idea; first, we compile trajectory $A$ into a deterministic finite automaton (DFA), and then it runs on trajectory $B$; To accelerate calculate distance between points, it use the point location data structure [13]. The algorithm is first subquadratic algorithm for the decision problem of discrete Frechet distance between two trajectories that runs in $O(mn \log\log n/(\log n)^2)$ time. using $O(m+n)$ space.

On the other hand, it is difficult to apply this algorithm to online version of this problem because of the memory consumption. In the preprocessing of this algorithm, for the parameter $0 < c < 1$, we divide the trajectory $B$ into blocks whose size is $x = c \log n$, and construct a table of $O(2^x) = O(n^c) = O(n)$ space. We look up this table using constant time throughout the algorithm. In online case, we cannot perform preprocessing because determine the length of trajectory $B$ Therefore, we cannot apply this algorithm to trajectory stream searching.

## 4.3 Preliminaries

In this section, we give the basic definitions and notations on our geometric temporal pattern matching problem.

### 4.3.1 Basic definitions

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of all non-negative integers, and $\mathbb{R}$ be the set of all real numbers. For integers $i \leq j$, $[i..j]$ denotes the discrete interval $\{i, i+1, \ldots, j\} \subseteq \mathbb{N}$, while for real numbers $a \leq b$, $[a, b]$ denotes the closed interval $\{\, x \in \mathbb{R} \mid a \leq x \leq b \,\} \subseteq \mathbb{R}$. Similarly, we define intervals $(a, b)$, $(a, b]$, and $[a, b)$ as usual. For a set $A$ of elements, we denote by $|A|$ the *cardinality* of $A$, and by $A^*$ the set of all sequences over $A$. For a set $S \subseteq A^*$ of sequences over $A$, we denote by $||S|| = \sum_{s \in S} |s|$ the *total length* of $S$.

We prepare some notions in geometry. Let $d \geq 0$ be a positive integer called the

*dimension* and $q \in \mathbb{N} \cup \{\infty\}$ be a norm parameter. Let $\mathbb{R}^d$ be a *d-dimensional space*, where each element $p = (p_1, \ldots, p_d)$ of $\mathbb{R}^d$ is called a *point* in $\mathbb{R}^d$, where $p_i \in \mathbb{R}$ is the $i$-th coordinate of $p$. We often denote the universe $\mathbb{R}^d$ of points by $\Sigma$. As the distance between two points $p$ and $q$ in $\mathbb{R}^d$, we use $L_\infty$ distance defined by

$$\delta(p, q) = \delta_\infty(p, q) = \max_{i \in [1..d]} \{|p_i - q_i|\}. \tag{4.1}$$

On the other hand, we can naturally define the problem with $L_2$ distance, $\delta_2(p, q) = \{\sum_{i \in [1..d]} |p_i - q_i|^2\}^{1/2}$. instead of $L_\infty$ distance, but we don't discuss the problem in this chapter.

A *trajectory* in $\mathbb{R}^d$ is a sequence $S = p_1 \ldots p_k$ of length $k \geq 0$, where $S[i] = p_i \in \mathbb{R}^d$ is a point in $\mathbb{R}^d$ for $i \in [1..n]$. For every $1 \leq i \leq j \leq k$, we denote by $S[i..j]$ the *subtrajectory* $p_i p_{i+1} \ldots p_j$ for positions from $i$ to $j$, and by $\varepsilon$ the *empty trajectory* of length zero. If $i > j$, we define $S[i..j] = \varepsilon$.

### 4.3.2 The class of extended trajectory patterns

Our target class of patterns is the class of *extended trajectory patterns* defined as follows, which are regular expressions over elementary region expressions, which will be defined below. In what follows, we assume $L_\infty$-distance if it is clear form context.

First of all, we introduce elementary region expressions as basic components of our pattern language. Let $q \in \mathbb{N} \cup \{\infty\}$ be a norm parameter.

**Definition 5.** An *elementary region expression* in $\mathbb{R}^d$ is a pair $e = (c, r)$ of a point $c \in \mathbb{R}^d$ and a positive number $r > 0$ that represents the closed sub-region of $\mathbb{R}^d$

$$R_q(c, r) = \{\, p \in \mathbb{R}^d \mid \delta_q(c, p) \leq r \,\} \subseteq \mathbb{R}^d. \tag{4.2}$$

consisting of all points $p$ in $\mathbb{R}^d$ whose distance from $c$ is at most $r$.

Specifically, $L_\infty(c, r)$ is the axis-aligned box $\prod_{i=1}^d [p_i - 0.5r, p_i + 0.5r] \subseteq \mathbb{R}^d$ in $L_\infty$-distance, while $L_2(c, r)$ is the disk with radius $r$ centered at $c$ in $L_2$-distance. In what

follows, $\mathcal{R}$ denotes the class of all elementary region expressions defined in this way. If no confusion arises, we also denote by $\mathcal{R}$ the class of regions represented by elementary regions in $\mathcal{R}$.

The class of *extended trajectory patterns* $E$ in $\mathbb{R}^d$, denoted by $\mathsf{EXTP}$, is a subclass of regular expressions defined over elementary region expressions as follows. In what follows, "$\equiv$" means the notational equivalence and we use exponentiation of elements to denote repetition, i.e. $a^3 = aaa$.

**Definition 6.** *(Extended trajectory pattern)* An *extended trajectory pattern* is a sequence of some components $E = e_1 \dots e_m (m \geq 0)$, where for each $1 \leq i \leq m$, $e_i$ is an expression, called a *component*, with associated language $L(e_i)$ in one of the following forms:

1. An *elementary region expression* $e_i = (c, r) \in \mathbb{R}^d$ is a component with the language $L(e_i) = R_q(c, r) \subseteq \mathbb{R}^d$.

2. A *don't care* $e_i = \Sigma$ is a component with the language $L(\Sigma) = \mathbb{R}^d$.

3. A *finite class* $e_i = (e_{i1}| \dots |e_{ik})$, $k \geq 1$, is a component that represents as the language the disjunction $L(e_i) = L(e_{i1}) \cup \dots \cup L(e_{ik})$, where $e_{i1}, ..., e_{ik}$ are elementary region expressions in $\mathbb{R}^d$. Note that a letter and a don't care are a finite class of points.

4. An *optional expression* $e_i = \alpha?$ is a component that represents as the language the optional components $L(\alpha?) = L(\alpha) \cup \{\varepsilon\}$, meaning either putting the element $\alpha$ itself or skipping it.

5. A *bounded repeat* $e_i = \alpha\{x, y\}$ is a component, where $0 \leq x \leq y$, with equivalence $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$ that represents the repeat of sub-expression $\alpha$ with some number of times between $x$ and $y$. If $\alpha$ is a don't care '$\Sigma$', then $e_i$ is called a *bounded length gap*.

6. An *unbounded repeat* $e_i = \alpha^*$ and $e_i = \alpha^+$ are components that represent the set of all sequences with length $\geq 0$ and all sequences of length $\geq 1$ of elements from $L(\alpha)$, where $\alpha^+ \equiv \alpha\alpha^*$. If $\alpha$ is a don't care '$\Sigma$', then $e_i$ is called a *variable length don't care* (*VLDC* for short) or *unbounded length gap*.

In the above definition, $\alpha$ denotes either an elementary region expression, a don't care, or a finite class. We sometimes use '.' as a delimiter between consecutive components, not as a wildcard. For an extended trajectory pattern $E = e_1 \cdots e_m$, we define its language by $L(E) = L(e_1) \cdots L(e_m)$. Each member $R$ in $L(E)$ is called an *instance* of $E$, which is a sequence of elementary regions $R = R_1 \cdots R_m$ by definition. The *size* (or the *length*) of the pattern $E$ is measured by the number of elementary region expressions and operators appearing in $E$.

**Example 2.** *Let $d = 2$ and $q = \infty$. We show examples of extended trajectory patterns in $\mathbb{R}^2$ with $L_\infty$-distance, where $A, B$, and $C$ are rectangular regions in $\mathbb{R}^2$ and note that '.' is a delimiter, not a wildcard as we defined above.*

- $E_1 = \texttt{ABABC}$.
- $E_2 = [\texttt{AB}]^+.\texttt{B}.\Sigma\{\texttt{1},\texttt{3}\}.[\texttt{BC}]?.\Sigma^*.\texttt{C}$.
- $E_3 = \texttt{A}.[\texttt{BC}]^*.\Sigma\{\texttt{0},\texttt{4}\}.\texttt{D}^+$.

We say that $R$ is an *exact string patterns* (also called a *string pattern*), denoted by $\mathsf{STR}$, if every component $e_i$ of an extended patterns $R = e_1 \cdots e_m$ is a elementary region in $\mathcal{R}$ such as $E_1$ above.

### 4.3.3 Pattern matching problem

The pattern matching problem that we consider in this chapter is obtained from the ordinary regular expression matching problem for strings [43] by replacing the character

matching "$a = b$?" between letters with the containment test "$p \in R$?" between a point $p$ and an elementary region $R \in \mathcal{R}$.

An instance $R = R_1 \cdots R_m \in \mathcal{R}^*$ *matches* a trajectory $T = t_1 \dots t_m \in (\mathbb{R}^d)^*$ of the same length $m$ if the following containment condition is satisfied.

$$t_i \in R_i, \forall i \in [1..m] \tag{4.3}$$

Then, we say that a extended trajectory pattern $E$ *occurs* in trajectory stream $T$ of length $n$ if there exists some positions $1 \leq i \leq j \leq n$ and some instance $R = R_1 \cdots R_m \in L(E)$ of $E$ such that the subtrajectory $T[i..j]$ matches $R$. In this case, we call the position $j$ the occurrence of $E$ in $T$.

Now we state *the geometric temporal pattern matching problem* as follows:

**Definition 7.** *(Extended trajectory pattern matching problem) Given an extended trajectory pattern $R \in \mathsf{EXTP}$ and an input trajectory stream $T$, report all the occurrences of $R$ in $T$.*

### 4.3.4 Applications of extended trajectory patterns

We introduce applications of the trajectory patterns in the context of geographical information systems (GIS). This class of patterns can represent following queries widely used in GIS field:

(1) *Origin-Destination query (OD query)*: An OD query specify only the start point $a$ and end point $b$ in $\mathbb{R}^d$ and don't care about the route where the trajectory go through. In our framework, this query can be written as extended trajectory pattern "$A.^*B$", where regions $A$ and $B \subseteq \mathbb{R}^d$ specify the origin and destination areas and unbounded repeat "$.^*$" of wildcard "." (actually this is the whole space $\mathbb{R}^d$) represents any route of unbounded length between $A$ and $B$.

(2) *Bounded OD query*: An bounded OD query is a OD query that the length of matched trajectory is bounded. This query can be written as "$A \cdot \{x, y\}B$" for $0 \leq x \leq y$ using bounded length gap. This query matches trajectories that start and end point are $A$ and $B$, respectively, and whose length $l$ is bounded in $x + 2 \leq l \leq y + 2$.

(3) *by-way-of query*: By-way-of query that specify the several number of points where the trajectory go though and can also bound the length between the points. We can consider this query as several concatenations of unbounded/bounded OD queries.

## 4.4 The Proposed Algorithm

In this section, we present an efficient pattern matching algorithm ExTM (extended trajectory matcher) for extended trajectory patterns.

### 4.4.1 Outline

The proposed algorithm ExTM is an extension of bit-parallel string matching algorithm, known as Extended SHIFT-AND proposed by Navarro et al. [45] for trajectories in $d$-dimensional trajectory. In Algorithm 7, we give an outline of ExTM.

A basic idea of the algorithm ExTM is to firstly transform a given pattern into a non-deterministic finite automaton (NFA) over elementary region, and then, construct a set of bitmasks

We start with given an extended trajectory pattern

$$E_2 = [\mathtt{AB}]^+ \, \mathtt{B} \, \Sigma\{\mathtt{1}, \mathtt{3}\} \, [\mathtt{BC}]? \, \Sigma^* \, \mathtt{C} \tag{4.4}$$

in EXTP, where $A, B, C$ are elementary region expressions in $\mathcal{R}$.

### 4.4.2 Normalization of a pattern

First, we expand all components defined through "syntax sugar" with the following equalities $\alpha? \equiv (\alpha|\varepsilon)$, $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$, and $\alpha+ \equiv \alpha.\alpha^*$. We also apply $\alpha* \equiv (\alpha|\varepsilon).\alpha^*$ only when $\alpha^*$ does not follow any other $\alpha$. Then, the resulting normal form $\mathsf{Extend}(E) = e_1 \cdots e_m$ of length $m$ contains only region expressions, optional component, and unbounded repeat.

For example, the input pattern $E_2$ above is normalized to

$$\mathsf{Expand}(E_2) = [\mathtt{AB}][\mathtt{AB}]^*\,\mathtt{B}\,\Sigma?\,\Sigma?\,\Sigma\,[\mathtt{BC}]?\,\Sigma^*\,\mathtt{C}. \tag{4.5}$$

by the above expansion.

### 4.4.3 Transformation from a normalized expression to a region NFA

Next, we transform the obtained pattern $\mathsf{Extend}(E) = e_1 \cdots e_m$ of length $m$ into a non-deterministic finite automaton (NFA) in linear shape, defined over an alphabet of elementary regions. The obtained NFA is called a *region NFA* and is represented as $N_E = (\mathcal{Q}_E, \Sigma, \mathcal{R}_E, \mathcal{E}_E, q_0, q_f)$, where $\mathcal{Q}_E = \{0, \ldots, m\}$ is the state set, $\Sigma = \mathbb{R}^d$ is the universe of input points, $q_0 = 0$ and $q_f = m$ are the initial and final states, and $\mathcal{E}_E \subseteq \mathcal{Q}_E \times \mathcal{R}_E \times \mathcal{Q}_E$ is the transition relation. For example, the expanded pattern $\mathsf{Expand}(E_2)$ above is transformed into the NFA shown in Fig. 4.1.

Specifically, for every $1 \leq i \leq m$, the $i$-th component $e_i$ of $\mathsf{Expand}(E)$ is transformed into the $i$-th transition triple $\tau_i = (i-1, e, i)$ according to the type of $e_i$ as follows.

- If $e_i$ is either single region expression $e \in \mathcal{R}$ or a don't care $e = \Sigma$, $\tau_i$ is the directed edge $(i-1, e, i)$.
- If $e_i$ is an optional component $e?$, $\tau_i$ is the pair of parallel directed edges $(i-1, e, i)$ and $(i-1, \varepsilon, i)$.

- If $e_i$ is an unbounded repeat $e^*$, $\tau_i$ is the directed edge $(i-1, e, i)$ with the self-loop $(i, e, i)$ at end point $i$.

In Fig. 4.1, we see the example of transformation for unbounded repeat $e_1 = [\mathtt{A}, \mathtt{B}]^*$, single region $e_2 = \mathtt{B}$, and optional component $e_3 = \Sigma?$, resp., in transitions $\{(0, [\mathtt{A}, \mathtt{B}], 1), (1, [\mathtt{A}, \mathtt{B}], 1)\}$, $\{(1, \mathtt{B}, 2)\}$, and $\{(2, \Sigma, 3), (2, \varepsilon, 3)\}$ of the NFA for $E_2$.

Any directed edge $\tau$ is classified into one of the following three types: an *$\varepsilon$-edge* goes from $i-1$ to $i$ and labeled by $\varepsilon$; a *goto-edge* goes from $i-1$ to $i$ and labeled with one or more regions; a *self-loop* goes from $i$ to itself with one or more regions. By construction of our region NFA, there are at most $m$ edges in each types.

### 4.4.4 Basic simulation method for a region NFA

Let $T = t_1 \cdots t_n \in (\mathbb{R}^d)^*$ be an input trajectory of length $n$.

For ordinary NFAs on strings of letters, there is a well-known method, called the *Thompson NFA simulation* [44], that can simulate the computation of a given NFA with $m$ states on an input string of length $n$ in $O(mn)$ time and $O(m)$ space. We can extend this method by slight modification of the original method as follows.

To apply Thompson's method, we need some preprocessing of an NFA $N_E$. Consider the set $\mathcal{R}_E$ of all elementary regions appearing in the NFA $N_E$as the labels of transition edges. Given any point $t \in \mathbb{R}^d$ as input, a region $R$ in $\mathcal{R}_E$ is *hitting* $t$ if the input point $t$ falls in$R$, i.e., $t \in R$. Let $HitRegion(t) \subseteq \mathcal{R}_E$ be the set of all hitting elementary regions w.r.t. $t$.

Using a set of pebbles to indicate *active states* of $N_E$ at each moment of computation, the modified Thompson simulation method proceeds as follows.

- Initially at stage 0, we put a pebble at the initial state $q_0$, which indicates the only active state. Make all possible $\varepsilon$-transitions on $N_E$.

- For every stage $i = 1, \ldots, n$, we iterate the following processes:

(i) Receive the next input point $t = t_i \in \mathbb{R}^d$.

(ii) Find the set of all elementary regions $HitRegion(t)$ that the input point $t$ falls in.

(iii) For every region $R$ in $HitRegion(t)$ and every transition edge $\tau = (x, R, y) \in \mathcal{E}_E$ labeled by $R$, make a transition on $\tau$ with moving copies of a pebble at source state $x$ to all possible destination states $y$, if possible.

(iv) Finally, make all possible $\varepsilon$-transitions on $N_E$.

(v) If the final state $q_f$ has a pebble, i.e., it is active, then report match at position $i$.

If we use a data structure with $f(m) = O(\log n)$ time and $g(m) = O(m)$ word space to compute the set $HitRegion(t)$, such as ones by Kirkpatrick [7], we can implement the above modification of Thompson's NFA simulation method with $O(n\{m + \log m\}) = O(nm)$ time and $O(m)$ word space for the class of extended trajectory patterns.

This complexity is still same to the naive matching algorithm for STR and smaller than the backtracking regular expression matching algorithm with exponential time. In the following subsections, we describe how to achieve speed-up and compaction of memory.

### 4.4.5 Partition of NFA into $O(\log n)$-sized blocks

After this transformation, the states set of $N_E$ is then partitioned into blocks $N_1, \ldots, N_{\lceil m/b \rceil}$ of size $h = O(\log n)$. For each $1 \leq k \leq \lceil m/b \rceil$, the block NFA

$$N_k = (\mathcal{Q}^{(k)}, \Sigma, \mathcal{R}^{(k)}, \mathcal{E}^{(k)}, q_0^{(k)}, q_f^{(k)}), \tag{4.6}$$

where the $k$-th state set $\mathcal{Q}^{(k)}$ consists of at most $b$ states from $b(k-1)$ to $bk$, at most $b$ directed edges, and at most $b$ elementary regions. Then, we preprocess the block NFA $N_k$ to build the following two structures:

- A point location index $\mathcal{I}_k$ that stores a collection $\mathcal{R}_k$ of elementary regions contained in the block NFA $N_k$. This is used to find the set of all elementary regions in $\mathcal{R}_k$ in which a given input point $p$ falls.

- A collection $\mathcal{B}_k$ of bit-masks that succinctly encodes the transition relation $\mathcal{E}_k$ of the block NFA $N_k$. This is used to simulate the state transition of the block NFA $N_k$ on an input trajectories.

Details of these structures described in following subsections.

### 4.4.6 A simple point location index

Let $A(R_k)$ is an *arrangement* [13] that is the partition of the plane formed by $R_k$. Let $R_k$ be a pattern trajectory of size $h = O(\log n)$ and $\mathcal{A}(R_k)$ be an arrangement determined by $R_k$. It is known that the number of faces in $\mathcal{A}(R_k)$ are $O(h^2)$ [13]. $\mathcal{F}(R_k)$ denotes a set of faces determined by the arrangement $\mathcal{A}(R_k)$, and $\mathcal{F}(R_k) = \{f_1, \ldots, f_\ell\}$, where $\ell = O(h^2)$). Figure 4.2 is the example of arrangement $A(P)$, where $P = p_1 p_2 \cdots p_6$ and $F(P) = \{f_1, f_2, \cdots, f_{13}\}$.

Now, we describe simple point location index $\mathcal{I}_\parallel$ answers the set of all elementary region in $R_k$ in which a given input point $p$ falls. Let $Seg_i(R_k)$ is a collection of segments defined by projecting edges of the each elementary region $R \in R_k$ onto $i$-th axises. Figure 4.3 is the example of projection onto x-axis (1st-axis) in 2-dimensional case. In this example, $Seg_1(P) = \{[x_1, x_2), [x_2, x_3), \cdots, [x_{11}, x_{12})\}$.

The basic idea to implement this index is to represent the face $f_j \in F(R_k)$ disjoint union of product of $Seg_i(R_k)$ for $1 \leq i \leq d$. For each $i = 1, 2 \cdots, d$, we compute $Seg_i(R_k) = \{[x_i, x_{i+1}) | i = 0, \cdots, k\}$ by sorting the coordinates of projection of edges in $O(h \log h)$ time, where $x_0 = -\infty$, and $x_{k+1} = \infty$. Next, for each segment $s \in Seg_i(R_k)$, we store a set of the elementary region intersect with $s$ as bitmask representation $BM_i[s] \in \{0, 1\}^h$. We can store a bitmask of length $h$ using $\lceil h/m \rceil$ words. In Figure 4.3,

for the segment $s = [x_7, x_8)$, $BM_1[s]$ stores the bitmask representation of $\{B_3, B_4, B_5\}$. The size of $Seg_i(R_k)$ is $O(h)$ and for each dimension $1 \leq i \leq d$ and for each segment $s \in Seg_i(R_k)$, the size of $BM_i[s]$ is $O(h) \times O(h/w) = O(h^2/w)$. Assuming that $w = O(logn)$ from word RAM assumption, and $h = O(\log n)$, the total space of this index is $O(h^2/w) = O(\log n)$.

In the querying, we search the segment $s_i$ that a given input point $p$ falls by binary search on $Seg_i(R_k)$ for $1 \leq i \leq d$ using $O(\log h)$ time, and then, we get bitmasks $BM_i[s_i]$. We finally get the bitmask representation of the set of the set of all elementary region in $R_k$ contains an input point $p$ by taking bitwise AND of $BM_i[s_i]$ for $1 \leq i \leq d$ using $O(h/w)$ time. From discussion above, we have following lemma.

**Lemma 1.** *Let $R$ be a set of elementary regions of size $h = O(\log n)$. Given an input point $p$, there exits a point location index answers the bitmask representation of the set of the set of all elementary region in $R$ contains in $O(\log \log n)$ time using $O(\log n \log \log n)$ preprocessing time and $O(\log n)$ space on the word RAM.*

### 4.4.7 Bit-parallel simulation of NFA in ExTM

We give the efficient NFA simulation algorithm Seach. Algorithm 8 shows our NFA simulation. To simulate region NFA efficiently, for each $1 \leq k \leq b$, and a given input point $p$, we use a collection of bitmasks $\mathcal{B}_k$ includes following set of bitmasks of length $h$:

- *Eblk* is the bitmask that sets 1 at all positions of $\varepsilon$-transition starts.
- *Ebeg* is the bitmask that sets 1 at the previous position of the lowest position of every $\varepsilon$-block.
- *Eend* is the $h$ bit mask that sets 1 at the highest position of every $\varepsilon$-block.

We also define bitmasks for forward transition.

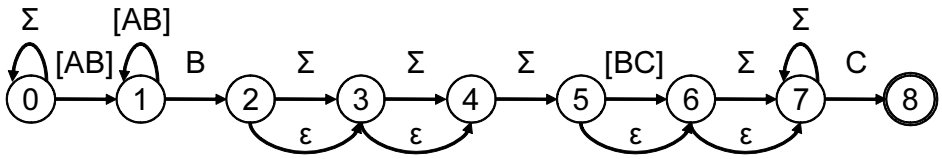


Figure 4.1: The extended trajectory pattern NFA of $E_2 = [\mathtt{AB}]^+.\mathtt{B}.\Sigma\{1,3\}.[\mathtt{BC}]?.\Sigma^*.\mathtt{C}$.

**Algorithm 7** ExTM$(P, T)$

`Preprocessing:`

1: Construct an NFA $N_E$ accepts an input pattern $P$.

2: Compute a set of bitmasks $B$ representing NFA $N_E$.

3: Construct a spatial index $\mathcal{I}$.

4: Store a set of bitmasks into $\mathcal{I}$ for each block.

`Searching:`

5: Simulate NFA on $T$ using point location index $\mathcal{I}$.

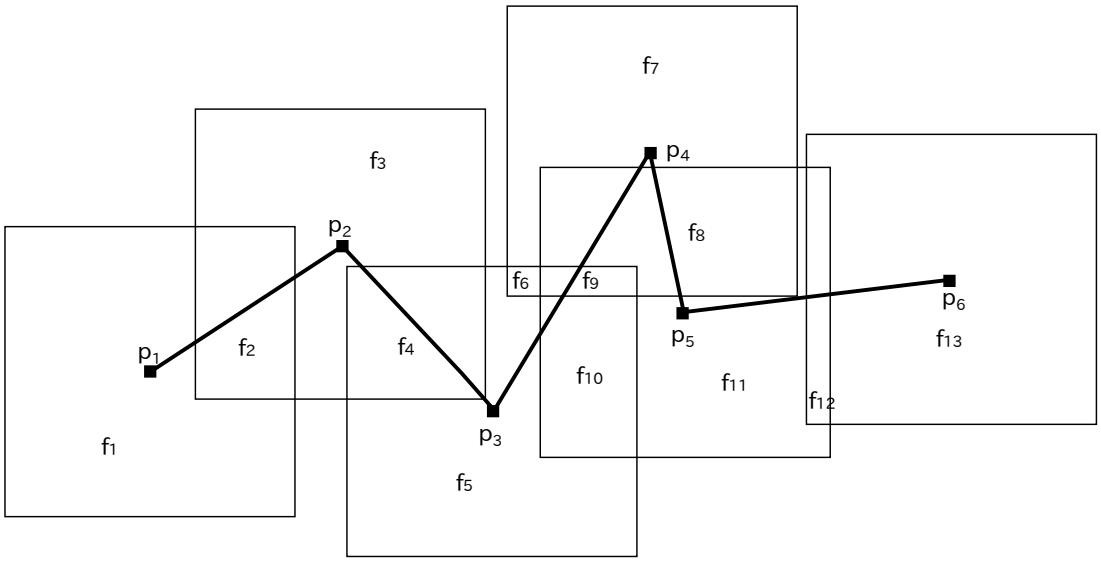


Figure 4.2: The example of arrangement.

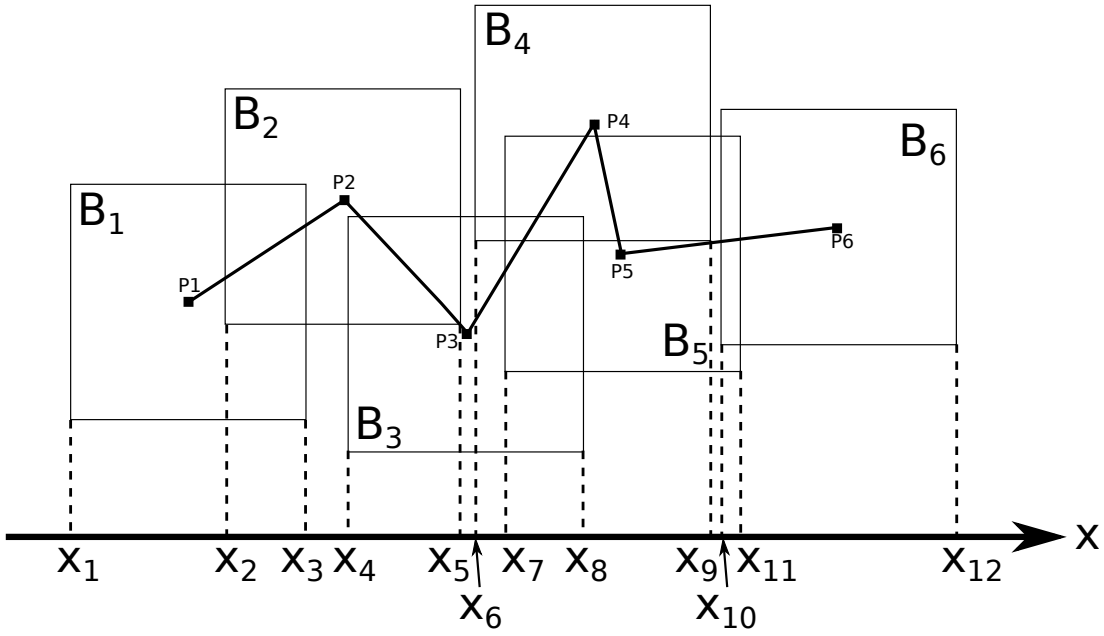


Figure 4.3: Example of projection onto x-axis.

**Algorithm 8** EXTM search($\mathcal{I}$, $T$)

1: **for** $i = 1$ to $n$ **do**
2: **for** $j = 1$ to $b$ **do**
3: $CHR[T[i]] \leftarrow 0^m$
4: $REP[T[i]] \leftarrow 0^m$
5: $CHR_j[T[i]], REP_j[T[i]] \leftarrow \mathcal{I}_j(T[i])$
6: append $CHR_j[T[i]]$ to $CHR[T[i]]$
7: append $REP_j[T[i]]$ to $REP[T[i]]$
8: **end for**
9: $S \leftarrow (((S \ll 1)\ \&\ CHR[T[i]]) \mid 1) \mid (S\ \&\ REP[T[i]])$
10: $Tmp \leftarrow S \mid Eend$
11: $S \leftarrow S \mid (Eblk\ \&\ ((\sim (Tmp - Ebeg)) \oplus Z))$
12: **end for**

- $CHR[p]$ is the bitmask representation of a set of elementary region of forward edge that contains an input point $p$. This mask is calculated by $\mathcal{I}_k$.

- $REP[p]$ is the bitmask representation of a set of elementary region of self-loop edge that contains an input point $p$. This mask is calculated by $\mathcal{I}_k$.

Next, we describe bit-parallel NFA simulation. In this simulation, first, for each $1 \leq j \leq b$, we query by an input point $p$ to point location index to get $CHR_j[p]$, and $REP_j[p]$, then, we concatenate each of them to get bitmasks $CHR[p]$ and $REP[p]$ of total length $m$. Secondly, the forward and self-loop letter transitions by an input point $p$ simulates by following update formula.

$$S \leftarrow (((S \ll 1) \,\&\, CHR[p]) \mid 1) \mid (S \,\&\, REP[p]) \tag{4.7}$$

Thirdly, following three lines of code makes $\varepsilon$-transitions:

$$High \leftarrow S \mid Eend \tag{4.8}$$

$$Low \leftarrow High - Ebeg \tag{4.9}$$

$$S \leftarrow (Eblk \,\&\, ((\sim Low) \oplus High)) \tag{4.10}$$

$$\mid S \tag{4.11}$$

Finally, we check the acceptance by following code:

$$\textbf{if} \quad S \,\&\, 10^{m-1} \neq 0^m \quad \textbf{then} \quad \text{report occurrence} \tag{4.12}$$

This NFA simulation is almost same as Extended SHIFT-AND algorithm [45].

### 4.4.8 Complexity analysis

Combining the point location index introduce in Section 4.4.6 and bit-parallel NFA simulation, we have following theorem.

**Theorem 2.** *The algorithm ExTM solves the extended trajectory pattern matching problem in $O(nm \log\log n/\log n)$ time using $O(m)$ space and $O(m \log\log n)$ preprocessing time.*

*Proof.* In preprocessing, we construct region NFA from an input pattern and divide into $h = O(\log n)$ sized blocks. For each block, we construct a set of bitmasks and a point location index. From lemma 1, this phase consumes $O(\log n \log\log n)$ time. The number of blocks is $\lceil m/h \rceil = O(m/\log n)$. Therefore, the total time of the preprocessing is

$$O(m/\log n) \times O(\log n \log\log n) = O(m \log\log n) \tag{4.13}$$

In searching, for each point of the input text trajectory $T$, at line 2 – line 7 in Algorithm 8, we consumes $O(\log\log n)$ time for each block, and the number of blocks is $O(m/\log n)$. Then, the total time is

$$O(m/\log n) \times O(\log\log n) = O(m \log\log n/\log n). \tag{4.14}$$

At line 9 – line 11, We simulate transitions of NFA by bit-parallel NFA simulation runs in $O(m/w)$ time. Therefore, the total time of the searching phase is $O(nm \log\log n/\log n)$ time.

For the space complexity, a set of bitmask use $O(m/w) = O(m/\log n)$ word. The point location index consumes $O(\log n)$ space for each block. Therefore the total space complexity is

$$O(m/\log n) \times (O(\log n) + O(m/\log n)) = O(m). \tag{4.15}$$

□

## 4.5 Chapter Summery

In this chapter, we considered trajectory regular expression matching problem. We proposed efficient bit-parallel matching algorithm for the problem runs in $O(nm \log\log n/\log n)$

time using $O(m)$ space and $O(m \log \log n)$ preprocessing time where $n$ and $m$ are the size of an input text trajectory and an input trajectory regular expression.

# Chapter 5

# Oblivious Evaluation of Non-deterministic Finite Automata with Application to Privacy-Preserving Virus Genome Detection

By recent emergence of huge amount of data (as know as *big data*), data analysis and its utilization have attracted huge attentions. There are a number of studies for real world applications of big data analysis, e.g. web advertising [11], traffic accidents prediction [24], and personal healthcare [40]. However, the privacy concerns of big data analysis have also been emerged. For example, Sweeney suggests in [60] that we can re-identify Governor of Massachusetts' health record from anonymized public medical dataset by combining several public data. We cannot leave the privacy problem to utilize big data.

In this chapter, we study pattern matching problem for analyzing sensitive data. We

especially consider pattern matching performing between two parties, Alice and Bob, the one party has secret pattern and the other party has secret text. The problem is to report the existence of pattern in the text to Alice (or Bob) without any information leakage about the secret pattern and text each other.

As main results, we propose efficient oblivious automata evaluation protocols for regular expression and approximate string matching based on NFA evaluation. These protocols are first polynomial time protocols for oblivious automata evaluation with respect to the size of input regular expression. Experimental results demonstrate that our method can be orders of magnitude faster than DFA-based methods, making it applicable to real-life scenarios, such as privacy-preserving detection of viral infection using genomic data. This chapter is based on the paper [54].

## 5.1 Background

String matching is a long-standing field of discrete algorithms with numerous applications such as genome analysis [20], network surveillance [30] and cloud computing [34], where the need for privacy preservation is rapidly mounting.

In medical genomics, for example, oblivious string matching can be crucially relevant to clinical diagnoses based on sequenced data: recent work [5] has shown that the analysis of DNA or RNA reads, such as obtained through next-generation sequencing techniques, can provide a quick and relatively inexpensive way to test for the presence of specific pathogens in a patient's sample. In a number of cases, however, part or all the information needed to conduct the test, as well as the result itself, may need to be kept secret from the contributing parties. For commercial or safety reasons (e.g. bio-terrorism), the testing authority may need to keep the exact pathogen sequence private, while patients may not wish to trust a third-party conducting the test with their own genomic data or the status of their diagnosis. In such a scenario, the privacy

must be kept rigorously on both sides, and cryptographic methods must be employed despite increased computational and communication costs. In cloud computing [34], an administrator may need to audit cloud access logs to detect malicious accesses. At the same time, it is not desirable to provide the administrator with access to users' private information. Oblivious string matching allows the administrator to audit the logs with complete privacy protection.

In string matching, an *automaton holder* has an automaton representing a regular expression while a *text holder* keeps a text string that needs to be tested. The text holder submits the text to the automaton holder letter by letter, triggering state transitions in the automaton, and finally obtains a notification of acceptance or rejection. Encryption allows us to perform the procedure obliviously, that is, with proof that the automaton holder has gained no knowledge about the text and the text holder no knowledge about the automaton and state transitions that occurred. Typically, several rounds of communication involving decryption and re-encryption are required to make a transition happen. An oblivious string matching method is evaluated by its computational cost for the automaton and text holders, communication cost and number of communication rounds.

A regular expression can be represented either by a DFA or an NFA. Troncoso et al. [63] and Frikken [16] designed oblivious string matching algorithms based on DFAs. An advantage of DFAs is that the active state is always unique and easy to keep track of. NFAs, on the other hand, have multiple active states leading to inferior run-time efficiency in non-oblivious settings. The size of NFAs, however, can be exponentially smaller than DFAs. In this chapter, we present an NFA-based oblivious string matching algorithm and prove it has better computational complexity bounds than DFA-based methods (Table 5.1). Our algorithm, termed ONE (Oblivious NFA evaluation), despite requiring a larger number of rounds, takes full advantage of NFA's compactness, making it exponentially faster than DFA-based algorithms in worst-case

complexity. Furthermore, our algorithm allows us to trade some privacy for speed by selectively disclosing a part of the graph structure of NFA.

Approximate string matching (as know as sequence alignment) is often implemented via dynamic programming (cf. Needleman-Wunsch and Smith-Waterman algorithms [43]). Such methods fill up a table with intermediate costs, where the computation of a cost involves integer comparison that cannot be implemented obliviously without Yao's garbled circuits [70]. We propose a novel oblivious technique based on Ukkonen NFAs [64]: a special grid-structured NFA. Existing algorithms either convert the NFA to a DFA [64] or require various manipulations beyond addition and multiplication [69]. ONE can be applied to Ukkonen NFA as well, but we will show that a protocol specially designed for Ukkonen NFA to improve computation and communication complexity by utilizing its regular grid structure. Both of our algorithms are secure against a semi-honest model.

Table 5.1: Comparison of the complexity of oblivious NFA evaluation (ONE) to DFA counterparts. Here $n$ denotes the input text length, $|\Sigma|$ the size of alphabet, and $m$ the number of NFA states. The exponential factor $2^m$ is due to the fact that the number of DFA states can be exponentially larger than that of NFAs.

| Method | Preprocess | | | Runtime | | |
|---|---|---|---|---|---|---|
| | Computation | Communication | Round | Computation | Communication | Round |
| Troncoso+ [63] | $O(2^m)$ | $O(\|\Sigma\|)$ | $O(1)$ | $O(n(2^m + \|\Sigma\|))$ | $O(n(2^m + \|\Sigma\|))$ | $O(n)$ |
| Frikken [16] | $O(n2^m\|\Sigma\|)$ | $O(n2^m\|\Sigma\|)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ |
| ONE | $O(m^2\|\Sigma\| + m^3)$ | $O(m^2\|\Sigma\|)$ | $O(1)$ | $O(nm^2)$ | $O(nm^2)$ | $O(nm)$ |

## 5.2 Matrix-based NFA Evaluation

Additive homomorphic cryptosystems [47] allow us to perform addition of two integers without decryption. Oblivious multiplication protocols can be designed based on the cryptosystem, if several communication rounds are allowed [46]. When evaluating a NFA, active states are updated by following directed edges [64], typically implemented as pointers. However, following pointers in an oblivious manner is more difficult than performing addition and multiplication. Thus we propose the following NFA evaluation algorithm consisting solely of matrix calculations.

In this chapter, we assume that the input NFA is a *transitive closure NFA*: if state $k$ is $\varepsilon$-reachable from state $j$, i.e., there exists a path connecting $j$ and $k$ consisting of $\varepsilon$-transitions only, then the automaton contains an $\varepsilon$-transition from $j$ to $k$. If the given NFA is not a transitive closure NFA, it can be computed in $O(m^3)$ time and $O(m^2)$ space [43].

Initially, only start states are active. As a text, $T = T[1], \ldots, T[n]$, is given letter by letter, active states transition by following edges with the corresponding letter. Thus, when evaluating an NFA, one must track multiple active states simultaneously. All states connected to an active state with $\varepsilon$-transitions immediately turn active, and all start states always stay active. When at least one of the accept states is active at time $i$, the prefix $T[1..i]$ is accepted. Language $L(F)$ is the set of strings represented by all paths from start states to accept states. If $T[1..i]$ is accepted, a suffix of $T[1..i]$ belongs to $L(F)$.

Let us define a *mask matrix* $M[\sigma]$ as follows. If $(j, \sigma, k) \in E$, $M[\sigma]_{j,k} = 0$ and $M[\sigma]_{j,k} = 1$ otherwise. It is the complement of the adjacency matrix restricted to letter $\sigma$. $M[\varepsilon]$ is a mask matrix for $\varepsilon$-edges. The active state array is defined as $S \in \mathbb{Z}^m$, where $S[j] = 0$ implies that the $j$-th state is active and otherwise inactive.

Then, the state transition induced by letter $t$ is described as

$$\hat{S}[k] \leftarrow \prod_{j=1}^{m} (S[j] + M[t]_{j,k}), \tag{5.1}$$

for $k = 1, \ldots, m$. It implies that $\hat{S}[k] = 0$ if and only if there exists $j$ such that $S[j] = 0$ and $M[t]_{j,k} = 0$. Similarly, the $\varepsilon$-transition is realized as

$$\hat{S}[k] \leftarrow \prod_{j=1}^{m} (S[j] + M[\varepsilon]_{j,k}). \tag{5.2}$$

Using the above update formulas, we describe the NFA evaluation method as follows. Initially, the active state array $S$ is set as

$$S[j] = \begin{cases} 0 & \text{if } j \in \Theta \\ 1 & \text{otherwise.} \end{cases} \tag{5.3}$$

Before reading out the text, $\varepsilon$-transition is performed once. At time $i$, the $i$-th letter $T[i]$ is given. In each iteration, the normal transition is applied and then $\varepsilon$-transition is applied (eq. 5.1). All start states are reset to active, $\hat{S}[j] = 0$ for $j \in \Theta$. If $\hat{S}[\phi] = 0$ for some accept state $\phi \in \Phi$, the acceptance of the prefix is reported. This algorithm finishes in $O(m^2 n)$ time and $O(m^2)$ space.

We note that if this algorithm does not necessarily evaluate NFAs correctly unle ss all $\varepsilon$-transitions defined with the $\varepsilon$-closure are performed. Even if $\varepsilon$-transitions of $F$ is replaced with the $\varepsilon$-closure thereof, the language accepted by the NFA is unchanged. Unless otherwise noted, we suppose $\varepsilon$-transitions of $F$ is replaced with the $\varepsilon$-closure thereof before evaluation in what follows.

Interestingly, active states in our model are represented as zero, which allows us to turn any state active by multiplying it by zero. Non-transition is implemented as a multiplication by a non-zero integer that keeps active states unchanged. Notice that our algorithm is inspired by bit-parallel string matching techniques such as presented in [4, 69]. In our case, however, we do not need to restrict ourselves to a binary active state array.

In a non-oblivious settings, where pointer-based algorithms are more efficient, such an approach would not be useful. It does, however, turn out to produce a competitive protocol in combination with homomorphic encryption.

## 5.3 Privacy Model and Problem Statement

### 5.3.1 Privacy model of NFAs

Two stakeholders appear in our oblivious NFA evaluation problem: the *text holder* (TH) and *automaton holder* (AH). The text holder possesses a private text that cannot be shared with the automaton holder; the automaton holder possesses an NFA that must remain private too. The oblivious NFA evaluation (ONE) problem supposes that both parties cannot share their private information mutually, but they wish to jointly simulate the running of the NFA over the text. After evaluation, only the automaton holder (or the text holder) should learn the evaluation result, meanwhile, nothing but what can be inferred from the evaluation result should be learned from the protocol execution. Thus, the oblivious NFA evaluation problem can be defined as an instance of secure multiparty computation.

The notion of privacy for the text holder is obvious: any substring of the text should not be leaked by NFA evaluation. Privacy for the automaton holder, on the other hand, is subject to discussion. In the pioneering work of [63], where DFAs are used, the privacy of DFAs is defined as the initial state, the set of accept states, and the transition function. We extend this privacy definition of DFA to NFAs.

We can show that the privacy of an NFA, $F = (V, E, \Theta, \Phi)$, is only dependent on $E$, the labeled edges: no matter how the state indices, $V$, are set, the texts accepted by the NFA is unchanged. In other words, the language $L(F)$ is independent of $V$, and its disclosure does not leak any information on the NFA except its size, $|V|$. For the same reason, disclosure of the set of initial states $\Theta$ and accept states $\Phi$ do not leak

any information about the NFA[1].

Since the privacy of NFAs fully relies on the labeled edges, $E$, we define the privacy of $E$ by partitioning the mask matrix of labeled edges into a public and a private part:

$$M[\sigma]^{\text{pub}}_{j,k} = \begin{cases} 0 & \text{if } e = (j, \sigma, k) \in E \text{ is public} \\ 1 & \text{if } e = (j, \sigma, k) \notin E \text{ is public} \\ priv & \text{otherwise.} \end{cases} \tag{5.4}$$

$$M[\sigma]^{\text{priv}}_{j,k} = \begin{cases} 0 & \text{if } e = (j, \sigma, k) \in E \text{ is private} \\ 1 & \text{if } e = (j, \sigma, k) \notin E \text{ is private} \\ pub & \text{otherwise.} \end{cases} \tag{5.5}$$

where $j, k \in V$ and $\sigma \subseteq \Sigma \cup \{\epsilon\}$. In eq. 5.4 and eq. 5.4.2, "public" means that anyone, text holder included, can learn the presence or absence of edge $e = (j, \sigma, k)$, "private" means that only the automaton holder knows the presence or absence of $e$.

For example, $M[\sigma]^{\text{pub}}_{j,k} = 1$ indicates that the absence of edge $e = (j, \sigma, k)$ is publicly known, while $M[\sigma]^{\text{pub}}_{j,k} = priv$ indicates that the text holder is not allowed to know the presence or absence of $e$ and $M[\sigma]^{\text{priv}}_{j,k}$ always takes 0 or 1.

### 5.3.2 Problem statement

With this matrix representation of privacy, the problem of oblivious NFA evaluation is immediately defined:

**Definition 8.** *A text holder (TH) holds a private text of length $n$, $T \in \Sigma^n$. An automaton holder (AH) holds an NFA $F = (V, E, \Theta, \Phi)$ where $M[\sigma]^{priv}$ is the mask matrix representing the private partition of labeled edges in $F$. After execution of the*

[1]More precisely, disclosure of $\Phi$ reveals the size of the accept state sets $|\Phi|$. However, by adding a special state that aggregates occurrence of acceptance, any NFA can be modified so that $|\Phi| = 1$ always holds. So we can consider that nothing is revealed by disclosing $\Phi$.

*protocol for oblivious NFA evaluation, AH learns the text size $n$ and whether $T[1..i]$ is accepted by $F$ for $i = 1, \ldots, n$, but nothing else. TH learns the NFA size $|V|$ but nothing else.*

We denote secure two-party computation $f$ by

$$f(X_1, X_2) \longrightarrow (Y_1, Y_2) \tag{5.6}$$

where $X_1$ and $X_2$ are private inputs from the first and second party, respectively; $Y_1$ and $Y_2$ are private outputs to the first and second party, respectively. Public inputs are omitted in this notation. With this notation, oblivious NFA evaluation can be written as:

$$\mathrm{ONE}(T, \{M[\sigma]^{\mathrm{priv}}\}_{\sigma \in \Sigma \cup \{\varepsilon\}}) \longrightarrow (|V|, (\{b_i\}_{i=1}^{n}, n)) \tag{5.7}$$

where $b_i \in \{true, false\}$ denotes whether text $T[1..i]$ is accepted by NFA $F$.

As shown in the definition, we are not hiding the sizes of the NFA, $|V|$, or the text, $n$. Because our protocols need to run on computers with specifically allocated physical memory space, it is difficult to conceal these sizes. By adding dummy states, obfuscation of such information is possible, it does however still give hints on $|V|$ or $n$, since the memory consumption cannot be made less than $O(|V|)$ or $O(n)$ for any obfuscation. If AH wishes to perfectly conceal all information about the input NFA, they can set all elements in $M[\sigma]^{\mathrm{pub}}$ to private: $M[\sigma]^{\mathrm{pub}}_{j,k} = priv$ for every $j, k \in V$.

We can easily modify the model to the model for the decision version of the problem. In this model, after execution of the protocol, AH learns only the fact that there are some substring of $T$ accepted by $F$ or not. This modification restricts reveal of information about $T$ to AH.

The privacy model in this setting is almost equivalent to the one introduced in [63] with two differences: First, their approach basically defines privacy for DFAs only. NFAs are transformed into DFAs and the privacy is defined for the transformed DFAs. Second, our model allows the automaton holder to specify a private set of edges and

edge labels of the NFA while the remaining public part can be published. In some NFAs with specific functionalities, such as Thompson NFAs or Ukkonen NFAs, the edges are known to form certain types of graphs. We will show that computation and communication complexity for oblivious evaluation of such NFAs can be greatly improved if the NFA's public part is appropriately specified.

## 5.4 Oblivious NFA evaluation

We introduce a secure computation that privately simulates NFAs, following the privacy model presented in the previous section. In this section, we assume that all elements of the input NFA are private, and introduce a naive protocol for oblivious protocol evaluation.

We initially introduce two cryptographic primitives: homomorphic addition and a one-round multiplication protocol with homomorphic encryption. Our oblivious NFA evaluation protocol is realized by simulating matrix-based NFA evaluation using the properties of homomorphic encryption.

### 5.4.1 Cryptographic primitives

#### Additively homomorphic encryption

In a public key cryptosystem, encryption uses a *public key* that can be known to everyone, while decryption requires knowledge of the corresponding *private key*. Given such a pair $(\mathrm{sk}, \mathrm{pk})$ of private and public keys and a message $m$, then $c = \mathrm{Enc}_{\mathrm{pk}}(m; \ell)$ denotes a (random) encryption, or ciphertext, of $m$, and $m = \mathrm{Dec}_{\mathrm{sk}}(c)$ denotes decryption. If a public key cryptosystem guarantees that any two ciphertexts are indistinguishable, then this cryptosystem provides *semantic security* (*IND-CPA security*). An *additive homomorphic cryptosystem* allows addition operations on encrypted values without knowledge of the secret key. Specifically, there exists an operator "$\circ$" such that for any

plaintexts $a_1$ and $a_2$:

$$\mathrm{Enc}_{\mathrm{pk}}(a_1 + a_2; \ell) = \mathrm{Enc}_{\mathrm{pk}}(a_1; \ell_1) \circ \mathrm{Enc}_{\mathrm{pk}}(a_2; \ell_2), \tag{5.8}$$

where $\ell$ is uniformly random provided that at least one of $\ell_1$ and $\ell_2$ is. On the basis of this property, it follows that given a constant $a_2$ and the encryption $\mathrm{Enc}_{\mathrm{pk}}(a_1; \ell)$, we can compute multiplications by $a_2$ via repeated application of "$\circ$". This also enables a re-randomization property, which allows the computation of a new random encryption $c' = \mathrm{Enc}_{\mathrm{pk}}(a; \ell')$ of $a$ from an existing encryption $c = \mathrm{Enc}_{\mathrm{pk}}(a; \ell)$ of $a$, again without knowledge of the private key or of $a$, as follows:

$$\mathrm{Enc}_{\mathrm{pk}}(a; \ell') = \mathrm{Enc}_{\mathrm{pk}}(a; \ell_1) \circ \mathrm{Enc}_{\mathrm{pk}}(0; \ell_2). \tag{5.9}$$

In the rest of the chapter, we omit the random number $\ell$ from our encryptions for simplicity.

We require a cryptosystem that provides semantic security (under appropriate computational hardness assumptions), re-randomization, and the additive homomorphic property, such as the Paillier cryptosystem [47]. In addition, we make use of the following specific property of the Pailler encryption in our prptocol construction:

$$\mathrm{Enc}_{\mathrm{pk}}(a_1; \ell)^{-a_2} = \mathrm{Enc}_{\mathrm{pk}}(-a_1 a_2; \ell). \tag{5.10}$$

### Multiplication protocol

Paillier encryption only allows multiplication of two integers when either of the integers is not encrypted, as pointed out in section 5.4.1. Multiplication of two encrypted values by a party without the decryption key (referred to as *sender*) can be achieved if one round of communication with the party having the decryption key (referred to as *evaluator*) is allowed:

$$\mathrm{MULT}((\mathrm{Enc}_{\mathrm{pk}}(x_1), \mathrm{Enc}_{\mathrm{pk}}(x_2)), \mathrm{sk}) \rightarrow (\mathrm{Enc}_{\mathrm{pk}}(x_1 x_2 \mod N), \emptyset). \tag{5.11}$$

where sk is the decryption key corresponding to the Paillier encryption scheme used for the input. We call this problem MULT. A solution commonly used (e.g.: [46]) to problem MULT is as follows:

1. The sender computes $\mathrm{Enc}_{\mathrm{pk}}(x')$ and $\mathrm{Enc}_{\mathrm{pk}}(y')$, and sends them to the evaluator:

$$\mathrm{Enc}_{\mathrm{pk}}(x') = \mathrm{Enc}_{\mathrm{pk}}(x) \circ \mathrm{Enc}_{\mathrm{pk}}(u) \text{ where } u \in_r \mathbb{Z}_N$$
$$\mathrm{Enc}_{\mathrm{pk}}(y') = \mathrm{Enc}_{\mathrm{pk}}(y) \circ \mathrm{Enc}_{\mathrm{pk}}(v) \text{ where } v \in_r \mathbb{Z}_N$$

2. The evaluator decrypts $\mathrm{Enc}_{\mathrm{pk}}(x')$ and $\mathrm{Enc}_{\mathrm{pk}}(y')$ and sends $z' = x'y' \mod N$ to the sender.

3. The sender receives $z'$, computes $\mathrm{Enc}_{\mathrm{pk}}(z)$ as follows, and outputs $\mathrm{Enc}_{\mathrm{pk}}(z)$.

$$\begin{aligned}\mathrm{Enc}_{\mathrm{pk}}(z) &= \mathrm{Enc}_{\mathrm{pk}}(z') \circ \mathrm{Enc}_{\mathrm{pk}}(y)^{-u} \\ &\quad \circ \mathrm{Enc}_{\mathrm{pk}}(x)^{-v} \circ \mathrm{Enc}_{\mathrm{pk}}(u \cdot v)^{-1} \\ &= \mathrm{Enc}_{\mathrm{pk}}(xy \mod N).\end{aligned}$$

The evaluator can decipher all encrypted messages but they have all been obfuscated with random values; the sender only deals with encrypted messages that it cannot decipher. It is thus easy to see that neither the sender nor evaluator gain any information about $x_1$ and $x_2$ from the protocol execution.

### 5.4.2 Oblivious NFA evaluation protocol

We employ Paillier encryption as the encryption function in our protocol. When $A$ is an array, the $i$th element is denoted by $A[i]$. $\mathrm{Enc}_{\mathrm{pk}}(A)$ denotes element-wise encryption of array $A$; the $i$th element of $\mathrm{Enc}_{\mathrm{pk}}(A)$ denotes $\mathrm{Enc}_{\mathrm{pk}}(A[i])$.

AH can arbitrarily specify the private part of NFA $F$ by setting $M[\sigma]^{\mathrm{pri}}$ according to eq. ,while public information is set in $M[\sigma]^{\mathrm{pub}}$ through eq. 5.4.

**Algorithm 9** Oblivious NFA evaluation protocol

-public input: $\Sigma, \Theta, \Phi, M[\sigma]^{\mathrm{pub}}$

-AH's private input: $M[\sigma]^{\mathrm{priv}}$

-AH's private output: $\{true, false\}^n$

-TH's private input: $T \in \Sigma^n$

-TH's private output: $\emptyset$

1: AH generates key pair $(pk, sk)$ and sends the public key $pk$ to TH. Then, AH evaluates $M^{\mathrm{pub}}$ by eq. 5.4.

2: For each $\sigma \in \Sigma \cup \{\varepsilon\}$, AH evaluates $\mathrm{Enc}_{\mathrm{pk}}(M[\sigma])$ and sends them to TH.

$$\mathrm{Enc}_{\mathrm{pk}}(M[\sigma]_{j,k}) = \begin{cases} \mathrm{Enc}_{\mathrm{pk}}\left(M[\sigma]^{\mathrm{pub}}_{j,k}\right) & \text{if } M[\sigma]^{\mathrm{pub}}_{j,k} = 0, \\ \mathrm{Enc}_{\mathrm{pk}}\left(M[\sigma]^{\mathrm{priv}}_{j,k}\right) & \text{if } M[\sigma]^{\mathrm{priv}}_{j,k} = 0 \text{ or } 1. \end{cases}$$

3: For each $1 \leq j \leq m$, AH initializes the array of the encrypted state variables $\mathrm{Enc}_{\mathrm{pk}}(S)$ and $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}\right)$ as follows and sends them to TH.

$$\mathrm{Enc}_{\mathrm{pk}}(S[j]) = \begin{cases} \mathrm{Enc}_{\mathrm{pk}}(0) & (j \in \Theta), \\ \mathrm{Enc}_{\mathrm{pk}}(1) & (\text{otherwise}). \end{cases} \tag{5.12}$$

$$\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[j]\right) = \mathrm{Enc}_{\mathrm{pk}}(1). \tag{5.13}$$

4: ▷ ——————$\varepsilon$-transition——————

5: **for** $(j,k) \in \{\, (j,k) \mid j,k \in V, M[\epsilon]^{\mathrm{pub}}_{j,k} \neq 1 \,\}$ **do**

6: TH evaluates the following formula.

$$tmp = \mathrm{Enc}_{\mathrm{pk}}(S[j]) \circ \mathrm{Enc}_{\mathrm{pk}}(M[\varepsilon]_{j,k}) \tag{5.14}$$

$$\mathrm{MULT}((\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[k]\right), tmp), \mathrm{sk}) \rightarrow (\mathrm{Enc}_{\mathrm{pk}}\left(\hat{\mathrm{S}}[\mathrm{k}]\right), \emptyset) \tag{5.15}$$

7: **end for**

8: **for** $i = 1$ to $n$ **do**

9: For each $1 \leq j \leq m$, TH initializes $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}\right)$ by $\hat{S}[j] = \mathrm{Enc}_{\mathrm{pk}}(1)$.

10: ▷ ——————-letter-transition——————

11: **for** $(j,k) \in \{\, (j,k) \mid j,k \in V, M[T[i]]^{\mathrm{pub}}_{j,k} \neq 1 \,\}$ **do**

12: TH evaluates following formula.

$$tmp = \mathrm{Enc}_{\mathrm{pk}}(S[j]) \circ \mathrm{Enc}_{\mathrm{pk}}(M[T[i]]_{j,k}) \tag{5.16}$$

$$\mathrm{MULT}((\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[k]\right), tmp), \mathrm{sk}) \rightarrow (\mathrm{Enc}_{\mathrm{pk}}\left(\hat{\mathrm{S}}[\mathrm{k}]\right), \emptyset) \tag{5.17}$$

13: **end for**

14: ▷ ——————-$\varepsilon$-transition——————

15: **for** $(j,k) \in \{\, (j,k) \mid j,k \in V, M[\epsilon]^{\mathrm{pub}}_{j,k} \neq 1 \,\}$ **do**

16: TH evaluates following formula.

$$tmp = \mathrm{Enc}_{\mathrm{pk}}(S[j]) \circ \mathrm{Enc}_{\mathrm{pk}}(M[\varepsilon]_{j,k}) \tag{5.18}$$

$$\mathrm{MULT}((\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[k]\right), tmp), \mathrm{sk}) \rightarrow (\mathrm{Enc}_{\mathrm{pk}}\left(\hat{\mathrm{S}}[\mathrm{k}]\right), \emptyset) \tag{5.19}$$

17: **end for**

18: For all $\phi \in \Phi$, TH sends $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\phi]\right)^{r_0}$ where $r_0 \in_r \mathbb{Z}^*_N$ to AH

19: AH decrypts $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\phi]\right)^{r_0}$ and outputs

$$b_i = \begin{cases} true & (\text{if } r_0 \hat{S}[\phi] = 0) \\ false & (\text{otherwise}) \end{cases} \tag{5.20}$$

20: **end for**

Algorithm 9 gives a solution for problem ONE. First, the encrypted state variables are initialized (line 1-3): this step activates states contained in the set of initial states before NFA evaluation (eq. 5.12 in the algorithm corresponds to eq. 5.3). Then, initial $\varepsilon$-transitions are performed (line 5-7).

Next, the text gets processed letter-by-letter (line 8-20). First, letter transitions are processed for all $(j, k)$ s.t. $M[T[i]]^{\text{pub}}_{j,k} \neq 1$ (line 11-13). Note that $M[T[i]]^{\text{pub}}_{j,k} = 1$ indicates that the absence of labeled edge $(i, T[i], j)$ is publicly known and, in such a case, the corresponding letter transition can be skipped. Then, $\varepsilon$-transitions are processed in the same manner for all $(j, k)$ s.t. $M[T[i]]^{\text{pub}}_{j,k} \neq 1$ (line 15-17). The encrypted state array is updated by iterating over these two steps alternately for each letter.

In the following subsections, we describe both of these transition steps in greater details:

### Letter transition

Letter transitions are performed in the for-loop on line 11: TH updates state variables with eq. 5.16 and eq. 5.17 for edges $(j, T[i], k)$ s.t. $M[T[i]]^{\text{pub}}_{j,k} \neq 1$. This ensures that the letter transition is performed only for $T[i]$-labeled edges whose presence is known publicly or edges whose presence (or absence) is kept private.

In addition, due to the homomorphic property in eq. 5.16, the state $j$ is made active only when (1) state $k$ is active and (2) there exists a letter transition from state $k$ to state $j$. The result of this update is preserved in a temporary variable *tmp*. Note that this computation is oblivious to TH. No matter if there exists a (private) edge between states $k$ and $j$, this update correctly evaluates the letter transition.

Next, using multiplication with protocol MULT, the state $j$ is made active if (1) *tmp* is active or (2) state $k$ is already active.

**$\varepsilon$-transition**

$\varepsilon$-transitions are performed in the for-loop at lines 5-15. Same as for the letter transitions, $\varepsilon$-transitions are only performed for $\varepsilon$-labeled edges whose absence is not known publicly. The mechanism of the update is exactly the same as with letter transitions.

### 5.4.3 Correctness and security of oblivious NFA evaluation

In this subsection, we show that Algorithm 9 solves the oblivious NFA evaluation problem securely and correctly.

**Theorem 3.** *Algorithm 9 correctly evaluates NFA $F$.*

Intuitively, the theorem is proved by showing that active (zero) or inactive (non-zero) states, as updated by Algorithm 9 are always equivalent to the states computed by matrix-based NFA evaluation in Section 5.2. The values computed by Algorithm 9 are not exactly the same as the values computed by the matrix-based NFA evaluation. However, noting that acceptance is determined by the state (zero or non-zero), the proof simply shows that Algorithm 9 and the matrix-based NFA evaluation behave equivalently in the way states' active statuses are updated.

**Theorem 4.** *If Paillier encryption is semantically secure and the text holder and automaton holder behave semi-honestly, Algorithm 9 is secure in the sense of Definition 8.*

We used the simulation-based standardized proof methodology in the semi-honest model. Intuitively, what $AH$ observes in the middle of the protocol execution is all additively or multiplicatively randomized; the TH's privacy is information-theoretic. What $TH$ observes in the middle of the protocol execution is all encrypted; since the encryption is semantically secure by assumption, AH's privacy is secure against probabilistic polynomial adversaries.

### 5.4.4 Complexity analysis

This subsection discusses the complexity analysis of the ONE protocol in detail.

During initialization, Algorithm 9 encrypts the mask matrix $M[\sigma]$ in $O(m^2)$ time (line 2), and initializes encrypted state variables in $O(m)$ time (line 3).

We introduce the notation

$$d = |\{ (j,k) \,|\, \forall \sigma \in \Sigma, M[\sigma]_{j,k} \neq 1 \}| \,, \tag{5.21}$$

which denotes the number of transitions whose presence is public or whose presence (or absence) is kept private. Noting that TH can skip letter transitions and $\varepsilon$-transitions when their absence is public, the protocol for ONE requires $O(d)$ transitions times per letter. A single transition invokes multiplications of encrypted values (addition in plaintext) and protocol MULT (multiplication in plaintext) $O(1)$ times. Letter- and $\varepsilon$-transition cost $O(d)$ computation time and $O(d)$ communication rounds with communication of $O(d)$ words per letter. The protocol for ONE thus requires $O(nd)$ computation time, $O(nd)$ communication rounds, and communication of $O(nd)$ words to process a $n$-length text.

We note that transitions having different destination states can be processed independently. If we process transitions having different target states in one round, we can reduce communication rounds to $O(\ell)$ where $\ell$ is the maximum number of source states that a single state can have.

The upper bound of $d$ is $m^2$; this occurs in the perfect privacy setting (presence or absence of all transitions is kept private) defined by [63], yielding $O(nm^2)$ time, $O(nm)$ communication rounds, and $O(nm^2)$ word communication. If we can reveal the absence of transitions for a limited portion of all transitions, $d$ can be greatly reduced.

### 5.4.5 Oblivious regular expression matching

We introduce an application of Algorithm 9 for the regular expression matching problem. The regular expression matching problem is given a regular expression $R$ and a text $T$, to report all occurrences of $R$ in $T$.

The Thompson NFA (TNFA for short) is a subclass of NFA proposed by Thompson [61]. Since TNFAs can be constructed from any regular expression using Thompson's construction algorithm, TNFAs are commonly used for general string matching. The detail of the construction algorithm of TNFA is shown in the appendix A. It is well known that every regular language is accepted by some TNFA [43].

In TNFAs, the letter transitions leaving state $j$ are always connected to state $j+1$ for every $j = 1, \ldots, m$, whereas $\varepsilon$-transitions can connect any two states. Given $R$, it is well known that the total number of states of the TNFA is linear in the size of $R$ [41]. These properties of TNFAs allow us to develop efficient evaluation protocol by reducing the number of expensive oblivious multiplication protocols.

**Privacy Model of TNFA:** Privacy model of TNFAs for oblivious evaluation is stated as follows. Let $F = (V, E, \Theta, \Phi)$ be the TNFA constructed from an input regular expression. Suppose the automaton holder holds a private TNFA and the text holder knows that the text is evaluated with a given TNFA (e.g., a given regular expression). Since letter transitions leaving state $j$ are always connected to state $j+1$ in a TNFA, it is obvious that letter transitions connecting state $j$ and state $k \neq j+1$ do not exist and the updates corresponding to these transitions can be skipped in our oblivious evaluation.

Formally, the privacy model of TNFAs is stated as follows. For every $\sigma \in \Sigma$ and every $j, k \in V$, the mask matrices $M^{\mathrm{pub}}[\sigma]$ and $M^{\mathrm{priv}}[\sigma]$ for the letter transitions of

TFNAs are given by:

$$M^{\mathrm{pub}}[\sigma]_{j,k} = \begin{cases} priv & \text{if } k = j+1 \\ 1 & \text{otherwise.} \end{cases} \tag{5.22}$$

$$M^{\mathrm{priv}}[\sigma]_{j,k} = \begin{cases} 0 & \text{if } (j,\sigma,k) \in E \\ 1 & \text{otherwise.} \end{cases} \tag{5.23}$$

The mask matrix $M^{\mathrm{pub}}[\varepsilon]$ and $M^{\mathrm{priv}}[\varepsilon]$ for $\varepsilon$-transitions can be arbitrary determined depending on the privacy requirements of the automaton holder.

**Protocol Description:** Noting that letter transitions are only connecting state $j$ and state $j+1$, the letter transition part, lines 5.16-5.17 in Algorithm 9, can be modified as follows:

$$\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[j+1]\right) \leftarrow \mathrm{Enc}_{\mathrm{pk}}\left(S[j]\right) \circ \mathrm{Enc}_{\mathrm{pk}}\left(M[T[i]]_{j,j+1}\right) \tag{5.24}$$

for $j = 1, \ldots, m$. Since this update does not invoke the oblivious multiplication MULT, communication rounds required for letter transitions are reduced from $m^2$ to 0.

If the automaton holder requires perfect privacy for $\varepsilon$-transitions, the communication rounds still remains $O(m^2)$ to process private $\varepsilon$ transitions. However, they can be reduced to $O(m)$ by assuming state $j$ is connected by $\varepsilon$-transitions to at most a single state $k \neq j$. NFAs satisfying this condition belong to a subclass of TNFAs that does not necessarily support the full regular expression class, but contains useful subclasses of regular expression such as *extended string patterns* or *PROSITE patterns* [43].

**Modification to the decision problem:** The readers may be concerned that this algorithm reveals too much information to NFA holder. For this problem, we can modify the algorithm to the decision problem of regular expression matching by making small change to our algorithm. This problem is to report whether there are occurrences of regular expression $R$ in the input text $T$ or not. To solve this problem, we first add a self-loop transition labeled by all letters to the accepting state of the NFA, and then change the algorithm to output the encrypted accepting state $\hat{S}[\phi]$ only once at the end

of computation. It is not hard to see that this modification yields that the NFA owner learns the result only at the end without the text owner's revealing any information about the input. This modification does not change the time and space complexity of our algorithm.

## 5.5 Oblivious Ukkonen NFA Evaluation

In this section, we introduce another useful class of string matching problem, called *approximate string matching problem*, which is extensively studied in text retrieval and bioinformatics [43, 45]. For this problem, we propose a modification of the oblivious NFA evaluation protocol in section 5.4 to handle the oblivious approximate string matching problem. Using a class of NFAs with special structure, called Ukkonen NFAs [64, 69], this protocol drastically reduces the complexity of oblivious evaluation for approximate string matching.

### 5.5.1 Approximate string matching

The *edit distance* between two strings $X$ and $Y$ is the minimum number of *edit operations* (a letter insertion, deletion, or substitution) required to convert $X$ into $Y$ or vice versa [33]. We denote the edit distance between $X$ and $Y$ as $ed(X, Y)$. The approximate string matching problem can be stated as follows:

**Definition 9.** *The approximate string matching problem is, given a pattern $R \in \Sigma^m$, a text $T \in \Sigma^n$, and a threshold $k \geq 0$, to report all the end-position $j$ in $T$ of every substring $T[i..j]$ that satisfies $ed(R, T[i..j]) \leq k$.*

A well-known approach to solve the approximate string matching problem is to use dynamic programming, known as *Smith-Waterman* algorithm [59] or *DP-match* algorithm [43]. In this approach, the algorithm constructs an $(m + 1) \times (n + 1)$ table

$D = (D_{ij})$, where each cell $D_{ij}$ contains the edit distance $ed(P[1..i], T[1..j])$ between the prefix $P[1..i]$ of the pattern and the prefix $T[1..j]$ of the text. Such a table can be efficiently constructed in $O(nm)$ time and space using dynamic programming, and then the match can be detected if the cell $D_{mn}$ contains any value no more than $k$.

The DP-based approach solves the problem of approximate string matching in $O(nm)$ space and time. In general, secure comparison is expensive compared to secure arithmetic operations and unfortunately DP-based approach contains $O(nm)$ comparisons. Thus, secure multiparty computation for approximate string matching based on the DP approach can be inefficient, particularly when the input text or pattern is large.

An alternative approach is to use the Ukkonen NFA (or UNFA) [64,69]. The UNFA is an NFA that recognizes all substrings of an input text $T$ with edit distance at most $k$ against a pattern $R$. UNFA forms the regular $(k + 1) \times (m + 1)$-grid structure, where $k$ is the maximum number of error allowed and $m$ is the length of a pattern $R$. In Figure 5.1, we show the Ukkonen NFAs for pattern $R = ababb$ with edit distance $k = 2$.

Ukkonen [64] presented an algorithm that solves the approximate string matching problem in $O(n)$ time by converting Ukkonen NFA to DFA, and then by evaluating DFA on an input text. The drawback of this algorithm is space complexity caused by the exponential number $O(\min(3^m, m(2m|\Sigma|)^k))$ of DFA states [43]. Later, Wu an Manber [69] proposed an algorithm that directly simulates UNFA using bit-parallelism. We use Wu and Manber's NFA evaluation algorithm [69] as the basis of our protocol.

### 5.5.2 Ukkonen NFA

The Ukkonen NFA is a grid-structured NFA. Every row $\ell$ represents the number of errors, and every column $j$ represents the prefix of $P$. We denote by $U_{\ell,j}$ the state in row index $\ell$ and column index $j$, where $0 \leq \ell \leq k$ and $0 \leq j \leq m$. We call each edge

labeled with $\Sigma$ a $\Sigma$-transition, which represents a letter transition with any letter of $\Sigma$.

As shown in Figure 5.1, a UNFA has the following classes of transitions each of which corresponds to one of the three edit operations:

1. Each horizontal letter transition (solid arrow with label $\sigma \in \Sigma$) from $U_{\ell,j}$ to $U_{\ell,j+1}$ represents matching of a letter $\sigma$ that is the $j$-th letter of $P$. We call this transition the *h-trans*.

2. Each vertical $\Sigma$-transition (solid arrow with label $\Sigma$) from $U_{\ell,j}$ to $U_{\ell+1,j}$ inserts any letter to $P$ between the $(j-1)$-th and $j$-th letters. We call this transition the *v-trans*.

3. Each diagonal $\Sigma$-transition (solid arrow with label $\Sigma$) from $U_{\ell,j}$ to $U_{\ell+1,j+1}$ substitute some letter in $\Sigma$ for the $j$-th letter of $P$. We call this transition the $\Sigma$*-d-trans*.

4. Each diagonal $\varepsilon$-transition (dashed arrow with label $\varepsilon$) from $U_{\ell,j}$ to $U_{\ell+1,j+1}$ deletes the $j$-th letter of $P$. We call this transition the $\varepsilon$*-d-trans*.

The rightmost states are accepting states. If the $j$th accepting state reports acceptance, if indicates that the number of error is $i$.

### 5.5.3 Oblivious Ukkonen NFA evaluation protocol

In this subsection, we present an oblivious NFA evaluation protocol using the Ukkonen NFA.

**Privacy Model of Oblivious UNFA:** The privacy model of oblivious Ukkonen NFA evaluation is stated. Suppose the automaton holder holds a private UNFA and the text holder knows that the automaton holder evaluates the text with a certain UNFA (e.g., approximate string matching with a certain string). In this case, if the pattern length $m$ and maximum error threshold $k$ is public, the edge relations (not labels of edges) of the

Ukkonen NFA is uniquely determined. The labels except h-transes are also uniquely determined. In other words, if the automaton holder reveals that his automaton is the Ukkonen NFA (e.g., the automaton holder evaluates approximate string matching) with some $k$ and $m$, it follows that no edge relations (not labels of edges) need to be kept private any longer. However, we note that the automaton holder's pattern to be matched need to be kept private.

Formally, the privacy model UNFAs is stated as follows. Let $F = (V, E, \Theta, \Phi)$ be UNFA constructed from input pattern $R$. From the structure of UNFA, for every $\sigma \in \Sigma \cup \{\varepsilon\}$ and $i, j \in V$, we set mask matrix $M^{\mathrm{pub}}[\sigma]$ and $M^{\mathrm{priv}}[\sigma]$ as follows:

$$M^{\mathrm{pub}}[\sigma]_{j,k} = \begin{cases} priv & \text{if } (j, \sigma, k) \text{ is h-trans,} \\ 0 & \text{if } (j, \sigma, k) \text{ is v-trans or } \Sigma- \text{ or } \varepsilon\text{-d-trans,} \\ 1 & \text{otherwise.} \end{cases} \tag{5.36}$$

$$M^{\mathrm{priv}}[\sigma]_{j,k} = \begin{cases} 0 & \text{if } (j, \sigma, k) \in E, \\ 1 & \text{otherwise.} \end{cases} \tag{5.37}$$

In oblivious evaluation of Ukknen NFAs, only the letters attached to h-trans from $U_{\ell,j}$ to $U_{\ell,j+1}$ are private and the remaining transitions and letters are all public.

**Protocol Description:** In Algorithm 10, we show the modified oblivious NFA evaluation protocol for Ukkonen NFA. The procedure basically follows Algorithm 9 except the update. The mask matrices are encrypted at line 2. The encrypted state variables are initialized at line 3.

States in the first row of the UNFA are updated only by h-trans. This is processed by eq. 5.27. States not in the first row of the UNFA are updated by $v$-trans, $v$-trans, $\Sigma - d$-trans, and $\varepsilon - d$-trans in turn at lines 8-12. In the algorithm, the results of update by h-trans (eq. 5.28) and v-trans (eq. 5.29) are integrated into $C1$ by protocol MULT (eq. 5.32). Then, the results of update by $\Sigma - d$-trans (eq. 5.30) and $\Sigma - \epsilon$-trans (eq.

**Algorithm 10** ONE-Ukkonen protocol

-public input: $\Sigma, \Theta, \Phi, M[\sigma]^{\mathrm{pub}}, k$

-AH's private input: $M[\sigma]^{\mathrm{priv}}$

-AH's private output: $\{true, false\}^n$

-TH's private input: $T \in \Sigma^n$

-TH's private output: $\emptyset$

1: AH generates key pair $(pk, sk)$ and sends the public key $pk$ to TH. Then, AH evaluates $M^{\mathrm{pub}}$ by eq. 5.4.

2: For each $\sigma \in \Sigma \cup \{\varepsilon\}$, AH evaluates $\mathrm{Enc}_{\mathrm{pk}}(M[\sigma])$ and sends them to TH

$$\mathrm{Enc}_{\mathrm{pk}}(M[\sigma]_{j,k}) = \begin{cases} \mathrm{Enc}_{\mathrm{pk}}\left(M[\sigma]^{\mathrm{pub}}_{j,k}\right) & \text{if } M[\sigma]^{\mathrm{pub}}_{j,k} = 0, \\ \mathrm{Enc}_{\mathrm{pk}}\left(M[\sigma]^{\mathrm{priv}}_{j,k}\right) & \text{if } M[\sigma]^{\mathrm{priv}}_{j,k} = 0 \text{ or } 1. \end{cases}$$

3: For each $0 \leq j \leq m$ and $0 \leq \ell \leq k$, AH initializes the array of the encrypted state variables $\mathrm{Enc}_{\mathrm{pk}}(S)$ and $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}\right)$ as follows and sends them to TH.

$$\mathrm{Enc}_{\mathrm{pk}}(S[\ell][j]) = \begin{cases} \mathrm{Enc}_{\mathrm{pk}}(0) & \text{if } j = \ell = 0, \\ \mathrm{Enc}_{\mathrm{pk}}(1) & \text{otherwise.} \end{cases} \tag{5.25}$$

$$\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\ell][j]\right) = \mathrm{Enc}_{\mathrm{pk}}(1). \tag{5.26}$$

4: **for** $i = 1$ to $n$ **do**

5: TH evaluates $\mathrm{Enc}_{\mathrm{pk}}(S[0][0]) = \mathrm{Enc}_{\mathrm{pk}}(0)$.

6: **for** $j = 1$ to $m$ **do**

7: TH evaluates following formulas.

$$\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[0][j]\right) \leftarrow \mathrm{Enc}_{\mathrm{pk}}(S[0][j-1]) \circ \mathrm{Enc}_{\mathrm{pk}}\left(M[T[i]]_{U_{0,j},U_{0,j+1}}\right) \tag{5.27}$$

8: **end for**

9: **for** $\ell = 1$ to $k$ **do**

10: **for** $j = 1$ to $m$ **do**

11: TH evaluates following formula.

$$B1[j] \leftarrow \mathrm{Enc}_{\mathrm{pk}}(S[\ell][j-1]) \circ \mathrm{Enc}_{\mathrm{pk}}\left(M[T[i]]_{U_{\ell,j},U_{\ell,j+1}}\right) \tag{5.28}$$

$$B2[j] \leftarrow \mathrm{Enc}_{\mathrm{pk}}(S[\ell-1][j]) \tag{5.29}$$

$$B3[j] \leftarrow \mathrm{Enc}_{\mathrm{pk}}(S[\ell-1][j-1]) \tag{5.30}$$

$$B4[j] \leftarrow \mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\ell][j-1]\right) \tag{5.31}$$

$$\mathrm{MULT}((B1[j], B2[j]), \mathrm{sk}) \rightarrow (\mathrm{C1[j]}, \emptyset) \tag{5.32}$$

$$\mathrm{MULT}((B3[j], B4[j]), \mathrm{sk}) \rightarrow (\mathrm{C2[j]}, \emptyset) \tag{5.33}$$

$$\mathrm{MULT}((C1[j], C2[j]), \mathrm{sk}) \rightarrow (\mathrm{Enc}_{\mathrm{pk}}\left(\hat{\mathrm{S}}[\ell][\mathrm{j}]\right), \emptyset) \tag{5.34}$$

12: **end for**

13: **end for**

14: For all $\phi \in \Phi$, TH sends $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\phi]\right)^{r_0}$ where $r_0 \in_r \mathbb{Z}_N^*$ to AH.

15: AH decrypts $\mathrm{Enc}_{\mathrm{pk}}\left(\hat{S}[\phi]\right)^{r_0}$ and outputs

$$b_i = \begin{cases} true & \text{if } r_0 \hat{S}[\phi] = 0, \\ false & \text{otherwise.} \end{cases} \tag{5.35}$$

16: **end for**

5.31) are integrated into $C2$ by protocol MULT (eq. 5.33). Finally, results $C1$ and $C2$ are integrated in eq. 5.34. The calculations eq. 5.32 and eq. 5.33 can be performed in one round, and then perform eq. 5.34. By doing so, the number of rounds invoked by MULT is reduced from three to two.

**Security:** The security proof can be shown by similar arguments to that used in Sec. 5.4. The primitives are homomorphic addition and MULT protocol only. The TH's privacy is information theoretic and the AH's privacy is secure against probabilistic polynomial adversaries.

### Complexity analysis

The complexities of the proposed protocol are proved by the following theorem.

**Theorem 5.** *For any input pattern $R$ and text $T$ on alphabet $\Sigma$, and threshold $k$, ONE-Ukkonen in Algorithm 10 solves the oblivious approximate string matching problem in $O(knm)$ time and $O(|\Sigma|m^2)$ preprocessing and space using $O(knm)$ communication and $O(kn)$ rounds, where $m$ and $n$ are the pattern and text length, and $\Sigma$ is an alphabet.*

*Proof.* First, we show the space complexity. The mask matrices $M[\sigma]^{\mathrm{pub}}$ and $M[\sigma]^{\mathrm{prib}}$ require $O(|\Sigma|m^2)$ space and the state array $S$ requires $O(km)$ space. Thus, this algorithm consumes $O(|\Sigma|m^2)$ space and preprocessing in total. Next, we consider the update of each row of UNFA, which consists of $m$ states. From Algorithm 10, we observe that the update of a row for $\ell \geq 1$ requires one homomorphic additions and three MULT protocol invocation per state. The update for $\ell = 0$ requires only one homomorphic addition per state. Furthermore, for each $\ell = 0, \ldots k$ we see that only the $m$ MULT require communication, and they can be overlapped in one round. Combining the above discussion, we can update a row in $O(m)$ time, $O(m)$ communication, and one round. Since an UNFA consists of $k + 1$ rows and a text contains $n$ letters, the result immediately follows. □

**Security proof**

The security proof can be shown by similar arguments to that used in Sec. 5.4. The primitives used here are homomorphic addition and MULT protocol only. The TH's privacy is information theoretic and the AH's privacy is secure against probabilistic polynomial adversaries as well as the protocol ONE.

## 5.6 Experiments

This section includes benchmarking using simple regular expressions, and a medical genomic application to virus detection from short DNA reads.

### 5.6.1 Benchmarking

We benchmarked the communication and computational costs of ONE and ONE-Ukkonen in comparison to DFA-based approaches. For DFA-based approaches, we converted the NFAs into DFAs and used the DFAs for evaluation purposes. Frikken's method [16] is theoretically appealing but consumes huge memory space due to the exponential number of states necessary to implement Yao's garbled circuit [70]. We opted to compare ONE to the method proposed by Troncoso et al. [63] (hereafter noted as Troncoso).

Our Java implementations of ONE and Troncoso used Paillier cryptosystem with 1024 bit keys. The automaton holder and text holder resided in different PCs connected by 100Mbit ethernet. As shown in Section 5.3, the privacy of each transition can be defined separately. In the following, all transitions are assumed to be private so as to match Troncoso's privacy model. Table 5.2 describes the sizes of DFA and NFA that represents a regular expression $R = (a|b|c)^*a(a|b)^t$ for different values of $t$ [2]. It represents a set of strings of length $t + 1$ whose first letter is $a$ and remaining

[2]We use exponentiation to denote repetition, i.e. $a^3 = aaa$.

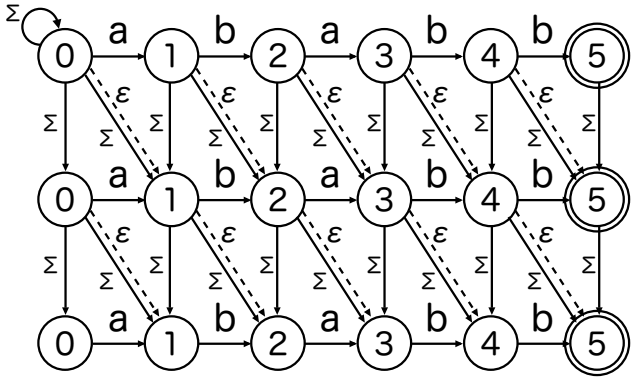


Figure 5.1: The Ukkonen NFA of $R = ababb$, $k = 2$.

Table 5.2: Sizes of DFA and NFA representing the regular expression $R = (a|b|c)^*a(a|b)^t$.

| t | 0 | 1 | 5 | 10 | 15 |
|---|---|---|---|---|---|
| NFA size | 2 | 3 | 7 | 12 | 17 |
| DFA size | 9 | 12 | 65 | 2,063 | 68,328 |

letters are either $a$ or $b$. The large difference in DFA and NFA sizes is reflected in computational and communication costs (Figure 5.2): Troncoso shows superexponential growth in the computational time for the automaton holder, while ONE's growth is subexponential, resulting in huge differences in total time (aggregated communication and computational time).

Next, we tested the performance of ONE-Ukkonen presented in Section 5.5. A random text is matched against a regular expression $R = \{ab(a|b)^*b\}^p$ allowing at most two mismatches. Figure 5.3 shows communication and computational costs per letter ($p = 1, \ldots, 4$). Overall, ONE-Ukkonen performed exponentially better than Troncoso.

### 5.6.2 Short read matching

As mentioned previously, the field of medical genomics offer a number of real-life scenarios where some form of substring matching may need to be conducted in a privacy-preserving framework.

With the advent of next-generation sequencing techniques [58], gathering whole-genome sequence information for individual patients is fast becoming a practical and financial reality. Such data opens the door to quantities of applications in personalized medicine and fast clinical diagnoses [38].

For instance, it was shown that the genome of certain RNA viruses (Hepatitis C and Norovirus) could be recovered from blood and fecal samples of infected patients [5]. So-called RNA shotgun sequencing produces a "genomic snapshot" of a sample, in the form of a large number of short RNA sequences short reads, that can be subsequently mapped (allowing for edit operations substitutions, insertions or deletions) to a reference genome sequence in order to detect small variations or, as in the present case, the presence of a specific pathogen. Following a similar pattern, it is possible to diagnose virtually any viral infection by a DNA or RNA virus by testing for the presence of a

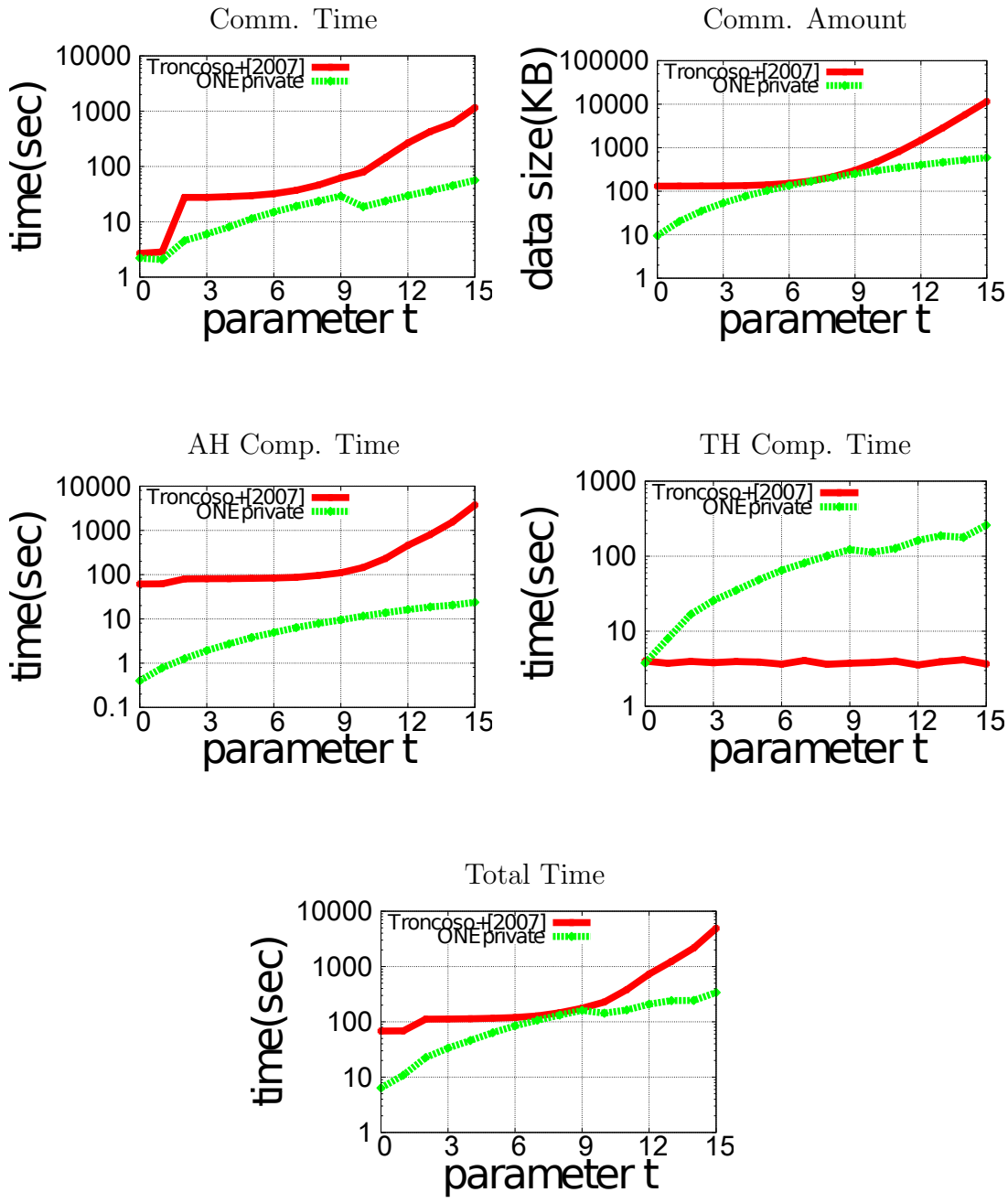


Figure 5.2: Computational and communication cost per text letter for regular expression $R = a(a|b)^t$. Green and red lines corresponds to ONE and Troncoso, respectively.

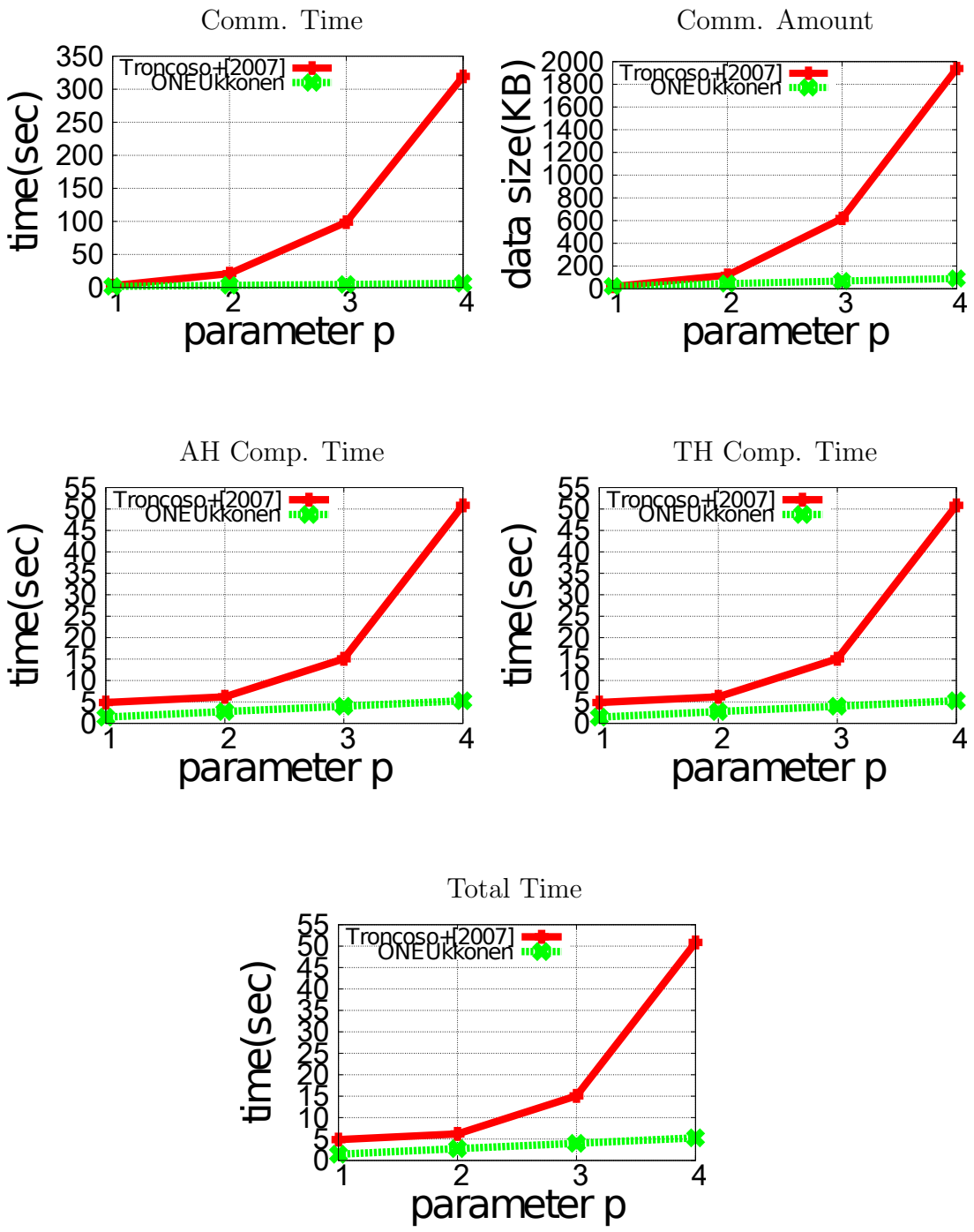


Figure 5.3: Computational and communication cost per text letter for approximate matching to regular expression $R = \{ab(a|b) * b\}(p)$. Green and red lines corresponds to ONE-Ukkonen and Troncoso, respectively.

significant number of short reads that map to the virus' genome (allowing for small variations in the sequence due to naturally-occurring mutations).

However, in the case of a new, potentially deadly, viral strain, health authorities may not want to disclose the exact sequence being tested, so as to avoid abuse by malicious parties. Meanwhile, patients may be reluctant to disclose their personal genomic information, as well as the result of the test itself, to a third-party conducting the test. Such a scenario is a perfect application for our oblivious approximate string matching algorithm (using Ukkonen NFA) presented in section 5.5.

Using short read data from [5], we simulated the oblivious detection of Norovirus genomic material in three samples from infected patients, using an additional non-infected sample for control. Each sample consists of a set of many millions short reads sequence (see table 5.3), each on average 150 letter-long, (over the 5-letter alphabet {A, C, G, T, N}, where 'N' represents a sequencing error), from which a small fraction can be mapped to a reference genome for human Norovirus GII (strain GII.4, Uniprot reference number D0EK97): that is, on average 1.175% and 0.013% (using approximate and exact matching, respectively) of all short read sequences are substrings of the whole Norovirus genome sequence (7646 letters, read in either direction).

Rather than testing for the match of each short read as a substring of the virus' genome, we selected a 31-letter substring from the genome, to use as our approximate pattern, against a randomly selected subset of the short reads. While the reads are generally not uniformly mapped on the reference genome, the "minimum coverage" observed with this RNA-Seq technique shows a lower-bound of approximately 0.02% of all reads matching any given position, allowing for up to two mismatches. By excluding known low-coverage areas (such as the extremities of the genome sequence), this lower-bound can be easily doubled. Therefore a pattern of 31 letters judiciously selected from the virus' sequence could be expected to match at least 0.024% of all reads (accounting for the possibility of the pattern overlapping with the read's boundaries). By selecting

a random sample of 30,000 reads out of a maximum of 5,902,290 for the largest sample, the worst-case probability of false negative on the test (failing to approximately match at least one read) is less than 0.074%. Conversely, the length of the pattern, high specificity of viral RNA and background distribution of nucleotides ensure a negligibly low probability of false positive where none of the reads were a match for the viral pattern.

As shown in table 5.4, our implementation of ONE-Ukkonen described in section 5.5, in C++ using Paillier encryption with 512 bit keys, was able to perform on average each approximate substring match in 192.3 s. (allowing for up to two mismatches), using a single Intel Xeon E5540 2.53GHz core. Using a 128-core cluster with parallelization, all 30,000 reads from the subsample were treated in less than 12.5 hours, each time correctly finding all positive matches present all samples. As our experimental results show, in all three infected samples, we were able to detect a significant number of viral RNA subsequences, identifying infected patients without disclosing any genomic data, nor the result of the test, to the other party. We can also note that, while the computing power necessary to conduct such a test with a satisfyingly large subsample of short reads (ensuring a low probability of false negative) is somewhat beyond current personal computing power, it is well within reach of any professional research institute.

## 5.7 Related Work and Discussion

String matching with automaton evaluation covers a wide range of applications, such as exact matching, approximate matching, regular expression, etc [43]. In this chapter, we present a protocol for oblivious NFA evaluation using matrix-based NFA evaluation.

Oblivious DFA evaluation has been extensively studied in various models over the recent years. [63] presented the first two-party oblivious DFA evaluation protocol based on oblivious transfer in the semi-honest model. In the same setting, [16] introduced an

Table 5.3: Total number of reads and size of the subset matching a 31-letter viral pattern, allowing for up to 0, 1 and 2 mismatches respectively (figures in parentheses are averaged numbers of matches on a randomly subsampled set of 30,000 reads).

| Sample | Total reads | Exact Matches | k = 1 | k = 2 |
|---|---|---|---|---|
| Norovirus samples | | | | |
| ERX202484 | 2957446 | 0 (0) | 474 (5.3) | 487 (5.3) |
| ERX202485 | 5902290 | 1078 (5.3) | 1122 (5.3) | 1147 (5.3) |
| ERX202486 | 2671055 | 0 (0) | 0 (0) | 1276 (14) |
| Control sample | | | | |
| ERX202487 | 3526904 | 0 (0) | 0 (0) | 0 (0) |

Table 5.4: Results of our implementation of oblivious approximate substring matching on randomly selected subsets of each sample, allowing for $k = 1, 2$ mismatches (all results are averaged over 3 samples).

| | k = 1 | k = 2 |
|---|---|---|
| Sensitivity (true positives / total positives) | 2/3 | 3/3 |
| Specificity (true negatives / total negatives) | 1/1 | 1/1 |
| CPU time per read (1 core) | 129.2s | 192.3s |
| Time per sample (128 cores) | 8.4h | 12.5h |

oblivious DFA evaluation based on Yao's garbled circuits with improved communication rounds and communication complexity. [8] considered techniques to outsource the computation of oblivious DFA evaluation without additional computational and communication costs. Oblivious DFA evaluation in the face of malicious adversaries was discussed in [39]. Extensions in the outsourced setting with homomorphic encryption was introduced by [67].

The semi-honest adversary model employed by our work follows [63] and [16]. Comparison of the computation, communication, and communication round complexities are summarized in Table 5.1. As already discussed, the computation and communication of existing protocols can be exponential when the target automaton is a NFA. Existing protocols are designed under the assumption that the target NFA can be transformed into a DFA, yet NFA-to-DFA transformation may require exponential computation time, and the resulting DFA may have an exponential number of states [3].

To the best of our knowledge, only ours and [32] provides oblivious NFA evaluation protocol. The NFA evaluation protocol presented in [32] is designed with an abstract computational environment referred to as the arithmetic black box model (ABB). The computation complexity of the protocol is $O(nm^2|\Sigma|)$, which is polynomial in the NFA size, whereas it dependent on the alphabet size. The communication and round complexity of this protocol cannot be specified without realization of the arithmetic black box model.

A number of studies have focused on polynomial-time secure multiparty computation specifically designed for specific string matching tasks; for example: exact string matching [28], string matching with wildcard and approximate string matching [22], sequence alignment by Smith-Waterman algorithm [27], hamming distance evaluation [26], and longest common sequence [18]. Because of the generality of the NFA, it covers a wide range of important string matching tasks, such as regular expression matching and approximate string matching. Our protocol successfully takes advantage

of the generality of the NFA and provides a universal polynomial-time and space secure two-party solutions for these tasks.

Recent advances of secure computation techniques realizes secure computation in a practical time for some specific type of problems. While it is difficult to directly compare the computation or communication complexity of secure computation based on homomorphic encryption with secure circuit evaluation techniques, experimental comparison are needed, which is remained as our future work.

## 5.8 Chapter Summary

In this chapter, we study secure regular expression matching problem. As main results, first, we proposed efficient oblivious evaluation protocol of NFA, called ONE, runs in $O(nm^2)$ time and communication, and $O(nm)$ rounds, where $n$ is the size of input text, and $m$ is the size of input regular expression. This is the first polynomial time oblivious automata evaluation protocol with respect to the size of the input regular expression.

We also proposed efficient secure protocol for approximate string matching, called ONE-Ukkonen, runs in $O(knm)$ time and communication, and $(kn)$ rounds, where $n$ and $m$ same as described above, and $k$ is the maximum number of error allowed. This protocol utilizes the regular grid structure of Ukkonen NFA to reduce the number of expensive multiplication of encrypted values.

# Chapter 6

# Concluding Remarks

## 6.1 Summary of the Results

In this thesis, we studied the regular expression matching problem for real world application. We especially focused on following three problems appeared in practical application:

- The massive regular expression matching problem which prcess several hundreds (or thousands) of pattern against a text.
- The regular expression matching problem on time series of multidimensional numerical value data.
- The secure regualar expression matching problem which perform regular expression matching in privacy-preserving manner.

In Chapter 3, we studied *massive regular expression matching problem*, which is a problem of, given a set of extended string pattern $\mathcal{P} = \{P_1, \cdots, P_k\}$ and a text $T$, finding all occurrences of $P_i$ for $1 \leq i \leq k$ in $T$. As a main result, we proposed the hirarchically decomposed finite automaton, called hirarchical NFA, and its simulation

algorithm on GPU based on Extended SHIFT-AND algorithm. Experimental results showed the proposed algorithm is faster than PFAC algorithm [35] proposed by Lin et al. by factor of 150.

In Chapter 4, we considered regular expression matching problem on trajectory data. As a main result, we developed online trajectory regular expression matching algorithm by combining the spatial index and bit-parallel NFA simulation runs in $O(nm \log\log n/logn)$ time using $O(m)$ space and $O(m \log\log n)$ preprocessing time where $n$ and $m$ are the size of an input text trajectory and an input trajectory regular expression.

In Chapter 5, we have introduced the problem of secure string matching. Then we presented a new oblivious automata evaluation method, called ONE. By theoretical analysis, we have shown that ONE is a first polynomial time algorithm with respect to the size of an input regular expression and an text. Experimental results also showed that that our method can be orders of magnitude faster than DFA-based method [63] proposed by Troncoso on artificail and DNA short read data.

## 6.2 Future Researches

For the regular expression matching on GPU described in Chapter 3, the aggregation procedure in the upper module of HFA is a bottleneck of the algorithm. Therefore, our future work includes to develop fast parallel aggregation method on GPU.

For the trajctory pattern matching, to develop pattern matching algorithms for the problem over containment condition based on $L_1$ and $L_2$ distance distance are the future work. In this problem, it is important problem to develop practically efficient point location index for $L_1$ and $L_2$ distance. The spatial index introduced in Chapter 4 implementing on GPU is also interesting problem.

For the secure regular expression matching problem, one of the problems remaining

unsolved is the number of communication round, which is caused by multiple invocation of the one-round multiplication protocol. Classical homomorphic encryption only supports either additive or multiplicative homomorphism. Following Gentry's breakthrough [17], fully-homomorphic or somewhat homomorphic [9, 42] encryption schemes have been extensively discussed. Exploitation of such recent cryptographic techniques remains as an area to explore in our future work.

# Appendix A

# Construction of Thompson NFA

We introduce the construction algorithm of Thompson NFA [61]. The TNFA $N(R) = (V, E, \Theta, \Phi)$ accepting $L(R)$, where $R$ is input regular expression, is obtained by the following recursive rules:

- For letter $\alpha \in \Sigma$, **(a)** Character transition TNFA $N(\alpha)$ consists of two states $\theta$ and $\phi$ joined together by a letter transition $(\theta, \alpha, \phi)$ as in Rule (a) of Figure A.1.

- Let $N(R_i)$ be the TNFA of regular expression $R_i$ with start and accepting states $\theta_i$ and $\phi_i$ for $i = 1, 2$. Then, TNFA $N(R_1 \cdot R_2)$, $N(R_1|R_2)$, and $N(R_1^*)$ are constructed as follows:

  **(b)** Concatenation $N(R_1 \cdot R_2)$: The two TNFAs $N(R_1)$ and $N(R_2)$ are merged together in series by making the accepting state $\phi_1$ of $R_1$ to be the start state $\theta_2$ of $R_2$ as in Rule (b) of Figure A.1.

  **(c)** Union $N(R_1|R_2)$: The two TNFAs $N(R_1)$ and $N(R_2)$ are merged together in parallel by adding a new start state $\theta_{12}$ with two $\varepsilon$-transitions entering the start states $\theta_1$ and $\theta_2$ and by adding a new accepting state $\phi_{12}$ with two $\varepsilon$-transitions leaving the accepting states $\phi_1$ and $\phi_2$ as in Rule (c) of Figure A.1.

  **(d)** $N(R_1^*)$: The TNFA $N(R_1)$ is made cyclic by adding a backward $\varepsilon$-transition

$(\phi_1, \varepsilon, \theta_1)$, then by adding new states $\theta$ and $\phi$ attached to $\theta_1$ and $\phi_1$, respectively, and finally by connecting start $\theta$ to accepting $\phi$ by a forward $\varepsilon$-transition $(\theta, \varepsilon, \phi)$. as shown in Rule (d) of Figure A.1.

- Finally, we obtain the final automaton $F$ by attaching the self loop $(\theta, \Sigma, \theta)$ to the start state $\theta$ of the top-level TNFA $N(R)$ constructed from an input regular expression $R$.

Fig. A.2 shows the complete example of TNFA $N(R)$ constructed from the regular explession $R = (AA|AT)(AG|AAA)$.

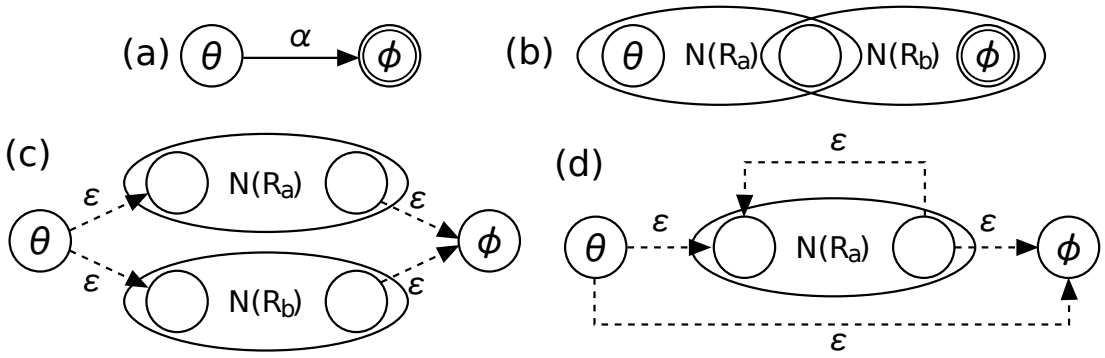


Figure A.1: TNFA construction rules.

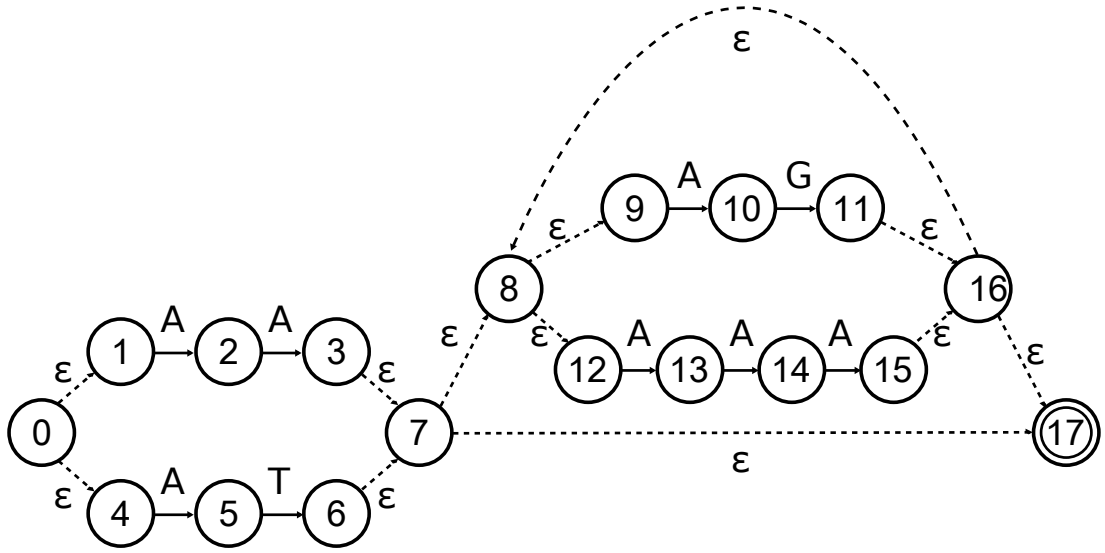


Figure A.2: An example of Thompson NFA $N(R)$ constructed from the regular expression $R = (AA|AT)(AG|AAA)*$.

# Bibliography

[1] Pankaj K Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete fréchet distance in subquadratic time. *SIAM Journal on Computing*, 43(2):429–449, 2014.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communication of the ACM*, 18(6):333–340, June 1975.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[5] Elizabeth M. Batty, T. H. Nicholas Wong, Amy Trebes, Karne Argoud, Moustafa Attar, David Buck, Camilla L. C. Ip, Tanya Golubchik, Madeleine Cule, Rory Bowden, Charis Manganis, Paul Klenerman, Eleanor Barnes, A. Sarah Walker, David H. Wyllie, Daniel J. Wilson, Kate E. Dingle, Tim E. A. Peto, Derrick W. Crook, and Paolo Piazza. A modified RNA-Seq approach for whole genome sequencing of RNA viruses from faecal and blood samples. *PLoS ONE*, 8(6):e66129, June 2013.

[6] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the 22nd International*

*Conference for High Performance Computing Networking, Storage and Analysis (SC'09)*, page 18. ACM, 2009.

[7] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, third edition, 2008.

[8] Marina Blanton and Mehrdad Aliasgari. Secure outsourcing of DNA searching via finite automata. In *Proceedings of the 24th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy XXIV (DESec'10)*, volume 6166 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2010.

[9] Dan Boneh, Craig Gentry, Shai Halevi, Frank Wang, and David J Wu. Private database queries using somewhat homomorphic encryption. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, pages 102–118. Springer, 2013.

[10] Robert S. Boyer and J Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[11] Badrish Chandramouli, Jonathan Goldstein, and Songyun Duan. Temporal analytics on big data for web advertising. In *Proceedings of 28th IEEE International Conference on Data Engineering (ICDE'12)*, pages 90–101. IEEE, 2012.

[12] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

[13] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[14] Thomas Eiter and Heikki Mannila. Computing discrete Fréchet distance. Technical Report Technical Report CD-TR 94/64, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, 1994.

[15] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 419–429, New York, NY, USA, 1994. ACM.

[16] Keith B. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. In *Proceedings of the 23nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security XXIII (DBSec'09)*, pages 81–94. Springer, 2009.

[17] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[18] Mark Gondree and Payman Mohassel. Longest common subsequence as private search. In *Proceedings of 8th Workshop on Privacy in the Electronic Society (WPES'09)*, pages 81–90. ACM, 2009.

[19] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data(SIGMOD'04)*, pages 215–226. ACM, 2004.

[20] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[21] Lilian Harada. Complex temporal patterns detection over continuous data streams. In *Proceedings of the Sixth East European Conference on Advances in Databases and Information Systems (ADBIS'02)*, pages 401–414. Springer-Verlag, 2002.

[22] Carmit Hazay and Tomas Toft. Computationally secure pattern matching in the presence of malicious adversaries. *Journal of cryptology*, 27(2):358–395, 2014.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison-Wesley, 1979.

[24] Weiming Hu, Xuejuan Xiao, Dan Xie, and Tieniu Tan. Traffic accident prediction using vehicle tracking and trajectory analysis. In *Proceedings of 6th IEEE International Conference on Intelligent Transportation Systems (ITSC'03)*, volume 1, pages 220–225, Oct 2003.

[25] Wen-Mei W. Hwu. *GPU Computing Gems Emerald Edition.* Elsevier, 2011.

[26] Ayman Jarrous and Benny Pinkas. Secure hamming distance based computation and its applications. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS'09)*, pages 107–124. Springer, 2009.

[27] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the 29th Symposium on Security and Privacy 2008 (S&P'08)*, pages 216–230. IEEE, 2008.

[28] Jonathan Katz and Lior Malka. Secure text processing with applications to private DNA matching. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, pages 485–492. ACM, 2010.

[29] Donald E. Knuth, James H. Morris, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[30] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communnication Review*, 36(4):339–350, 2006.

[31] Frédéric Kuznik, Christian Obrecht, Gilles Rusaouen, and Jean-Jacques Roux. LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, 59(7):2380–2392, 2010.

[32] Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. Cryptology ePrint Archive, Report 2013/678, 2013.

[33] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[34] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *Proceedings of the 29th IEEE Conference on Computer Communication (INFOCOM'10)*, pages 1–5. IEEE, 2010.

[35] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and Jyuo-Min Shyu. Accelerating string matching using multi-threaded algorithm on GPU. In *Proceedings of the 29th IEEE Global Communications Conference (GLOBECOM'10)*, pages 1–5. IEEE, 2010.

[36] S.A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008.

[37] Alessandro Margara and Gianpaolo Cugola. High performance content-based matching using GPUs. In *Proceedings of the Fifth ACM International Conference on Distributed Event-Based System (DEBS'11)*, pages 183–194. ACM, 2011.

[38] Michael L Metzker. Sequencing technologies, the next generation. *Nature Reviews Genetics*, 11(1):31–46, 2010.

[39] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious DFA evaluation and applications. In *Topics in Cryptology–CT-RSA 2012*, pages 398–415. Springer, 2012.

[40] Travis B. Murdoch and Allan S. Detsky. The inevitable application of big data to health care. *The Journal of American Medical Association*, 309(13):1351–1352, 2013.

[41] Gene Myers. A four Russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(2):432–448, 1992.

[42] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[43] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings — practical on-line search algorithms for texts and biological sequences.* Cambridge, 2002.

[44] Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002.

[45] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003.

[46] Kobbi Nissim and Enav Weinreb. Communication efficient secure linear algebra. In *Theory of Cryptography Conference (TCC'06)*, pages 522–541, 2006.

[47] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology EUROCRYPT*, pages 223–238. Springer, 1999.

[48] Claudio Piciarelli and Gian Luca Foresti. On-line trajectory clustering for anomalous events detection. *Pattern Recognition Letters*, 27(15):1835–1842, 2006.

[49] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[50] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *Proceeding of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'01)*, pages 71–81. ACM, 2001.

[51] Tomoya Saito, Takuya Kida, and Hiroki Arimura. An efficient algorithm for complex pattern matching over continuous data streams based on bit-parallel method. In *IEEE International Workshop on Databases for Next Generation Researchers (SWOD'07)*, pages 13–18. IEEE, 2007.

[52] Hirohito Sasakawa and Hiroki Arimura. Trajectory pattern matching based on bit-parallelism for large gps data. In *IIAI International Conference on e-Services and Knowledge Management (IIAI ESKM 2012)*, pages 66–71. IEEE, 2012.

[53] Hirohito Sasakawa and Hiroki Arimura. Faster multiple pattern matching system on gpu based on bit-parallelism. In *Proceedings of the 18th Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI'13)*, pages 316–321, 2013.

[54] Hirohito Sasakawa, Hiroki Harada, David A. duVerle, Hiroki Arimura, Koji Tsuda, and Jun Sakuma. Oblivious evaluation of non-deterministic finite automata with application to privacy-preserving virus genome detection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society (WPES'14)*, pages 21–30. ACM, 2014.

[55] Hirohito Sasakawa, Yusaku Kaneta, and Hiroki Arimura. Faster pattern matching algorithm for very long extended patterns with applications to large-scale string matching. *DBSJ Journal*, 11(1):55–60, June 2012. In Japanese.

[56] Hirohito Sasakawa, Masahiro Yamamoto, Kazuhiro Kurita, and Hiroki Arimura. Bit-parallel approximate trajectory matching for 2-dimensional trajectory data. In *Proceedings of the 17th Japan Conference on Discrete and Computational Geometry and Graphs (JCDCG$^2$'14)*, 2014.

[57] Hirohito Sasakawa, Masahiro Yamamoto, Kazuhiro Kurita, and Hiroki Arimura. A space efficient algorithm for large-scale trajectory matching based on bit-prallelism. In *Proceedings of the 7th Forum on Web and Database (WebDB Forum'14)*, pages B2–3, 2014. In Japanese.

[58] Stephan C. Schuster. Next-generation sequencing transforms todays biology. *Nature*, 200(8), 2007.

[59] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[60] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[61] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.

[62] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5):232–240, 2010.

[63] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient DNA searching through oblivious automata. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, pages 519–528. ACM, 2007.

[64] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132 – 137, 1985.

[65] Sebastiano Vigna. Broadword implementation of parenthesis queries. *CoRR*, abs/1301.5468, 2013.

[66] Panagiotis D. Vouzis and Nikolaos V. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[67] Lei Wei and Michael K. Reiter. Third-party private dfa evaluation on encrypted files in the cloud. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS'12)*, pages 523–540. Springer, 2012.

[68] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 407–418. ACM, 2006.

[69] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.

[70] Andrew Chi-Chih Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS'82)*, pages 160–164. IEEE, 1982.

[71] Josh Jia-Ching Ying, Wang-Chien Lee, Tz-Chiao Weng, and Vincent S. Tseng. Semantic trajectory mining for location prediction. In *Proceedings of the 19th ACM*

*SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 34–43. ACM, 2011.

[72] Jianting Zhang and Simin You. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the First ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–32. ACM, 2012.