2018

staDFA: An Efficient Subexpression Matching Method

Mohammad Imran Chowdhury

FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

staDFA: AN EFFICIENT SUBEXPRESSION MATCHING METHOD

By

MOHAMMAD IMRAN CHOWDHURY

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

2018

Mohammad Imran Chowdhury defended this thesis on July 20, 2018.
The members of the supervisory committee were:

Robert A. van Engelen
Professor Directing Thesis

David Whalley
Committee Member

An-I Andy Wang
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

To my beloved parents, who always believed in me even when I didn't

ACKNOWLEDGMENTS

First of all, I would like to give my gratitude to my creator almighty ALLAH for whom what I am today. Next to my parents, for their endless support and encouragement, which makes me hopeful and confident. My thanks also go to my professor and thesis advisor, Prof. Dr. Robert van Engelen. With every stage of my thesis work, he helped me a lot and provided valuable resources. Without his help I would not be here today. I would also like to give thanks to all of my committee members, Prof. Dr. David Whalley, and Prof. Dr. An-I Andy Wang.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

The following short list of symbols are used throughout the document.

RE	Regular Expression
NFA	Nondeterministic Finite Automata
DFA	Deterministic Finite Automata
TNFA	Tagged Nondeterministic Finite Automata
TDFA	Tagged Deterministic Finite Automata
staDFA	store-transfer-accept DFA
MPS	Memory Positions Store
$\|\|R\|\|$	Alphabetic width (length) of an RE R
w	Word or string
$\|w\|$	Length of word w
t	Marker or tag
MPS	Memory Positions Store
$M[t]$	MPS memory with position for marker t
m_i^t	MPS memory cell at position (or address) i for marker t
$\not{/}^t$	Mark-first for marker t
$\not{\backslash}^t$	Mark-last for marker t
$\not{/}_f^t R$	Leftmost-first marker of R for marker t
$\not{\backslash}_l^t R$	Leftmost-last marker of R for marker t
S_i^t	The store command using marker t at address location i
$T_{i,j}^t$	The transfer command using marker t from address location j to i
A_i^t	The accept command using marker t at address location i

ABSTRACT

The main task of a *Lexical Analyzer* such as Lex [20], Flex [26] and RE/Flex [34], is to perform *tokenization* of a given input file within reasonable time and with limited storage requirements. Hence, most lexical analyzers use *Deterministic Finite Automata* (*DFA*) to tokenize input to ensure that the running time of the lexical analyzer is linear (or close to linear) in the size of the input.

However, *DFA* constructed from *Regular Expressions* (*RE*) are inadequate to indicate the positions and/or extents in a matching string of a given *subexpression* of the regular expression. This means that all implementations of *trailing contexts* in *DFA*-based lexical analyzers, including Lex, Flex and RE/Flex, produce incorrect results. For any matching string in the input (also called the *lexeme*) that matches a *token* is regular expression pattern, it is not always possible to tell the position of a part of the lexeme that matches a subexpression of the regular expression. For example, the string `abba` matches the pattern `a b*/b a`, but the position of the trailing context `b a` of the pattern in the string `abba` cannot be determined by a *DFA*-based matcher in the aforementioned lexical analyzers. There are algorithms based on *Nondeterministic Finite Automata* (*NFA*) that match subexpressions accurately. However, these algorithms are costly to execute and use backtracking or breadth-first search algorithms that run in non-linear time, with polynomial or even exponential worst-case time complexity. A tagged *DFA*-based approach (*TDFA*) was pioneered by Ville Laurikari [15] to efficiently match subexpressions. However, *TDFA* are not perfectly suitable for lexical analyzers since the tagged *DFA* edges require sets of memory updates, which hampers the performance of *DFA* edge traversals when matching input. I will introduce a new *DFA*-based algorithm for efficient subexpression matching that performs memory updates in *DFA* states.

I propose the *Store-Transfer-Accept Deterministic Finite Automata* (*staDFA*). In my proposed method, the subexpression matching positions and/or extents are stored in a *Marker Position Store* (*MPS*). The *MPS* is updated while the input is tokenized to provide the positions/extents of the sub-match.

Compression techniques for *DFA*, such as Hopcroft's method [14], default transitions [18, 19], and other methods, can be applied to *staDFA*. For an instance, this thesis provide a modified Hopcroft's method for the minimization of *staDFA*.

CHAPTER 1

INTRODUCTION

Regular Expressions (*RE*) are typically used to define patterns to match (parts of) the text of a given input. There are many *regex-engines* such as Perl [24], PCRE2 [13], RE2 [9], Flex [26], Lex [20] and RE/Flex [34] that use *RE* to match or tokenize some given input. All *regex-engines* represent *RE* either in an *NFA*-like or in a *DFA* form or a combination, because of performance considerations [7]. Algorithms to translate *RE* to *DFA* are well known [1, 2, 4]. But these algorithms are not adequate to tell the position and extent of a subexpression within a given *RE*, when we have a matching string.

Ville Laurikari [15] proposed a *tagged-DFA* (*TDFA*) to solve this problem. In [15] Ville Laurikari, introduced the concept of *tag subexpressions* by inserting tags t_x in regular expressions. For example $\texttt{a}^*\, t_1\, \texttt{b}$ has a tag t_1 that marks the transition from $\texttt{a}^*$ to `b`. The tagged regular expression is converted to a*tagged-DFA* (*TDFA*) via an intermediate transformation to a *tagged-NFA* (*TNFA*). The *TDFA* is used to update a *tag map* by a string matcher. In this case, matching with a *TDFA* returns position 2 for tag t_1 in the accepted string `aab`. Tagging of, and matching with, *TDFA* is *transition-oriented* since edges are marked.

The *TDFA* approach has two challenges to incorporate them in lexical analyzers. First, because *TDFA* are transition-oriented they are not perfectly suitable for lexical analyzers since the tagged edges require sets of memory updates, hampering the performance of edge traversals while matching input. Second, *TDFA* relies on a conversion algorithm to obtain a *TDFA* from a *tagged-NFA* (*TNFA*), where the *TNFA* was constructed from an *RE*. Third, a *TDFA* cannot be minimized by a *DFA* minimization algorithm such as Hopcroft's method [14], because of the presence of tagged edges.

By contrast *staDFA* removes these limitations. Loosely speaking, *staDFA* can be considered Moore machines with operations in states versus the *TNFA* and *TDFA* automata that can be considered Mealy machines with operations on transitions.

1.1 A Motivating Example

Suppose we have a regular expression $R = (\mathtt{a} \mid \mathtt{a}\,\mathtt{b}) \,/\, \mathtt{b}\,\mathtt{a}$ and the following accepted input strings `aba` and `abba`, where / indicates a trailing context `b a` for the sub-expression $R_1 = (\mathtt{a} \mid \mathtt{a}\,\mathtt{b})$. For this example, Lex [20] fails to match sub-strings `a` and `ab` against R_1, respectively, according to Appel [4]. Flex [26] uses an ad-hoc approach to fix some of these limitations. However, Flex fails in the general case. Take for example the regular expression such as $R = \mathtt{a}\,\mathtt{b}^*/\mathtt{b}\,\mathtt{a}^*$ on input strings `aba` and `abba` Flex fails to identify substrings `a` or `ab` due to the overlapping patterns $\mathtt{b}^*$ at the end of the sub-expression and `b` at the start of the trailing context.

The subexpression matching problem can be explained as follows. Let's assume a regular expression R consists of several sub-expressions R_i such that $R = \ldots(R_1)\ldots(R_2)\ldots(R_3)\ldots$. The task at hand is to construct an automaton that can efficiently tell the positions n_i of these subexpressions for the accepted input strings w of length (say, l) in preferably $\mathcal{O}(|w|)$ time:

$$\overbrace{w[0]\ldots\underbrace{w[n_4]\ldots w[n_9]}_{\text{matches } R_1}\ldots\underbrace{w[n_{11}]\ldots w[n_{15}]}_{\text{matches } R_2}\ldots\underbrace{w[n_{17}]\ldots w[n_{20}]}_{\text{matches } R_3}\ldots w[\ell]}^{\text{matches } R}$$

An automaton alone is not sufficient enough to track positions n_i in submatch problem definition. Thus, a memory-based mechanism is essentially needed to track the string positions n_i.

1.2 Thesis Statement

This thesis presents an *staDFA* constructed directly from *RE*. The biggest advantage compared to *NFA-based* methods and *TDFA* is that we can efficiently find sub-matches and trailing-contexts, lookaheads, capturing groups, back-references, while providing both the leftmost, rightmost, first, and last match policies. TDFA as described in [15] is not POSIX compliant, i.e. does not offer a guaranteed leftmost longest matching policy.

The concept of *staDFA* is introduced in this thesis with Store, Transfer and Accept commands associated with *DFA* states and an algorithm to convert a regular expression with marked subexpressions directly into *staDFA*.

The *staDFA* concept offers the following properties:

- An algorithm for *staDFA* construction from a *RE* such that subexpressions are identified by their position and extent in accepted strings, Section 4.2.1
- An efficient *staDFA* matching algorithm, such that the *staDFA* matching algorithm returns well-formed marker positions in accepted strings, Section 4
- *staDFA* permits strategies to disambiguate marked subexpressions by resolving conflicts between commands in the *staDFA*, Section 2
- *staDFA* permits optimization with Hopcroft's *DFA* minimization, Section 4.5

The remainder of this thesis work is organized as follows. Chapter 2 introduces the preliminaries about regular expression, *NFA*s/*DFA*s, etc. With respect to my proposed approach, the related-work are summarized on Chapter 3. In Chapter 4, my proposed approach is presented thoroughly. The performance evaluation of my approach is demonstrated in Chapter 5. The conclusion and future work is compiled on Chapter 6.

CHAPTER 2

PRELIMINARIES

In this chapter I present regular expressions and some of their applications. Also, I demonstrate how these regular expressions are used by *NFAs* and *DFAs* for text matching. Section 2.1 shows regular expressions and their real-world applications. Section 2.2 describes basic terminology about *NFAs* and their use of regular expression. The basic principles of *DFAs* and their application to pattern-matching based on regular expression are presented on Section 2.3.

2.1 Regular Expressions

Loosely speaking, regular expressions (RE) represent a language L containing a set of words (or strings) $w \in \Sigma^*$ for some alphabet Σ.

Syntax of Regular Expressions. Regular expressions have three meta operations. These are alternation, concatenation and repetition. Optionally, it has other parts such as Anchors, and Character Sets.

Anchors. Usually there are two anchor characters $\wedge$ and \$. Here, $\wedge$ represents the starting anchor that identifies the start position of the matched-pattern and \$ represents the ending anchor that identifies the end position of the matched-pattern. Therefore, $\wedge$`A`* matches the input of text at the starting position and matching `A` zero or more times. Similarly, if we have regular expression like `A`*\$ which means match the input lines of text that has the ending position and matching `A` zero or more times.

Character Sets and Classes. The common character sets are: [ ] known as positive character group, [^] known as negative character group, $\cdot$ (dot) any character. Similarly, the most common character classes are: \w known as word character, \W known as non-word character, \s white-space character, \S non-white-Space character, and character class subtraction such as [`a` $-$ `z` $-$ $-$[`aeiou`]].

2.2 Non-Deterministic Finite Automata

Firstpos is the set of positions that match the first symbols of accepted strings w by an regular expression R:

$$\mathit{firstpos}(R) = \begin{cases} \mathit{firstpos}(R_1) \cup \mathit{firstpos}(R_2) & \text{if } R = R_1 \mid R_2 \\ \mathit{firstpos}(R_1) \cup \mathit{firstpos}(R_2) & \text{if } R = R_1\, R_2 \text{ and } \mathit{nullable}(R_1) = \text{true} \\ \mathit{firstpos}(R_1) & \text{if } R = R_1\, R_2 \text{ and } \mathit{nullable}(R_1) = \text{false} \\ \mathit{firstpos}(R_1) & \text{if } R = {R_1}^* \\ \{a_i\} & \text{if } R = a_i \text{ and } a \in \Sigma \\ \varnothing & \text{if } R = \varepsilon \end{cases}$$

where $\mathit{nullable}(R)$ is defined as:

$$\mathit{nullable}(R) = \begin{cases} \mathit{nullable}(R_1) \vee \mathit{nullable}(R_2) & \text{if } R = R_1 \mid R_2 \\ \mathit{nullable}(R_1) \wedge \mathit{nullable}(R_2) & \text{if } R = R_1\, R_2 \\ \text{true} & \text{if } R = {R_1}^* \\ \text{false} & \text{if } R = a_i \text{ and } a \in \Sigma \\ \text{true} & \text{if } R = \varepsilon \end{cases}$$

Lastpos is the set of positions that match the last symbols of accepted strings w by an regular expression R:

$$\mathit{laspos}(R) = \begin{cases} \mathit{laspos}(R_1) \cup \mathit{laspos}(R_2) & \text{if } R = R_1 \mid R_2 \\ \mathit{laspos}(R_1) \cup \mathit{laspos}(R_2) & \text{if } R = R_1\, R_2 \text{ and } \mathit{nullable}(R_2) = \text{true} \\ \mathit{laspos}(R_2) & \text{if } R = R_1\, R_2 \text{ and } \mathit{nullable}(R_2) = \text{false} \\ \mathit{laspos}(R_1) & \text{if } R = {R_1}^* \\ \{a_i\} & \text{if } R = a_i \text{ and } a \in \Sigma \\ \varnothing & \text{if } R = \varepsilon \end{cases}$$

Followpos is the set of edges between positions of matching symbols and is defined as follows:

$$\mathit{followpos}(R) = \mathit{edges}(R) \cup \{a_i \in \mathit{lastpos}(R) \colon (a_i, A)\}$$

where the set $\mathit{edges}(R)$ of a regular expression R contains directed edges (a_i, b_j) from position a_i to position b_j such that b_j may match symbol $w[k+1]$ of a string $w \in \Sigma^*$ after successfully matching a_i with symbol $a = w[k]$ at position k in w:

$$\mathit{edges}(R) = \begin{cases} \mathit{edges}(R_1) \cup \mathit{edges}(R_2) & \text{if } R = R_1 \mid R_2 \\ \mathit{edges}(R_1) \cup \mathit{edges}(R_2) \cup \{a_i \in \mathit{laspos}(R_1), b_j \in \mathit{firstpos}(R_2) \colon (a_i, b_j)\} & \text{if } R = R_1\, R_2 \\ \mathit{edges}(R_1) \cup \{a_i \in \mathit{laspos}(R_1), b_j \in \mathit{firstpos}(R_1) \colon (a_i, b_j)\} & \text{if } R = {R_1}^* \\ \varnothing & \text{otherwise} \end{cases}$$

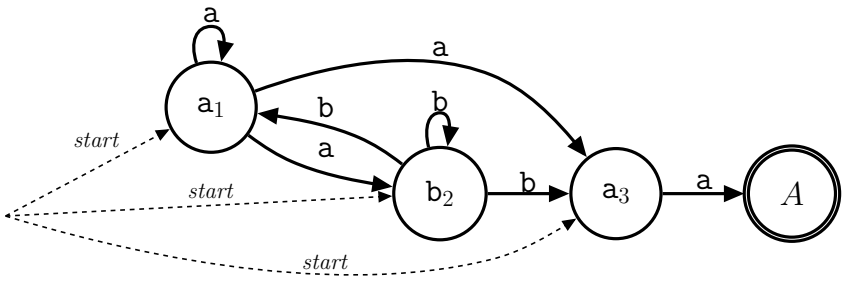


Figure 2.1: A position *NFA* of regular expression $(\mathtt{a}_1 \mid \mathtt{b}_2)^* \, \mathtt{a}_3$.

Based on the above definitions of *firstpos(R)*, *lastpos(R)*, and *followpos(R)* a position *NFA* is formed, where from each state on a particular input string symbol, a position *NFA* can move to a combination of states at a time.

Definition 1. *Given a regular expression R over the alphabet Σ, the position* NFA *of R is a 5-tuple $\langle Q, \Sigma, \delta, q_0, A \rangle$, where*

- *Q is the finite set of states formed by $\{(a_i, b_j) \in \mathit{followpos}(R) \colon a_i\} \cup \{A\}$,*
- *Σ is an* alphabet *defined over a finite set of symbols,*
- *$\delta \colon Q \times \Sigma \to 2^Q$ is the* transition function *where, $\delta(a_i, a) = \{(a_i, b_j) \in \mathit{followpos}(R) \colon b_j\}$,*
- *q_0 is the set of* initial states *where, $q_0 \in \mathit{firstpos}(R)$,*
- *A is the* accepting state.

For example, we have the regular expression $R = (\mathtt{a}_1 \mid \mathtt{b}_2)^* \, \mathtt{a}_3$ which accepts strings $w \in \Sigma^*$ with $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. The position *NFA* of R is $\mathit{NFA}(R) = \langle \{\mathtt{a}_1, \mathtt{b}_2, \mathtt{a}_3, A\}, \{\mathtt{a}, \mathtt{b}\}, \delta, \{\mathtt{a}_1\}, A \rangle$, where δ is the transition function depicted by the labeled edges in Fig. 2.1, which shows the positions *NFA* constructed for $R = (\mathtt{a}_1 \mid \mathtt{b}_2)^* \, \mathtt{a}_3$.

2.3 Deterministic Finite Automata

A *DFA* from each state on a particular input string symbol, can move to only one state at a time.

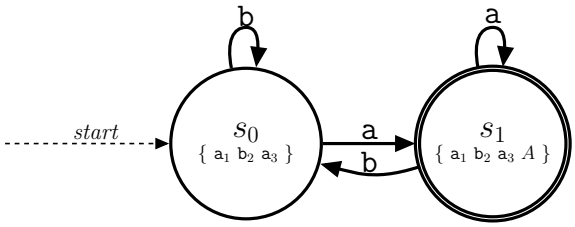


Figure 2.2: A *DFA* for regular expression $R = (\mathtt{a}_1 \mid \mathtt{b}_2)^* \, \mathtt{a}_3$ constructed from the position NFA of R by subset construction.

Definition 2. *Given a regular expression R over the alphabet Σ, the* DFA *of R constructed from the position* NFA(R) *by* subset construction *is a 5-tuple $\langle S, \Sigma, \delta, s_0, F \rangle$, where*

- $s \in S$ *is the finite set of states formed by $S \subseteq 2^Q$, where Q is the states of* NFA(R),
- Σ *is an* alphabet *defined over a finite set of symbols,*
- $\delta \colon S \times \Sigma \to S$ *is the* transition function, *where $\delta(s, a) = \bigcup_{a_i \in s} \{(a_i, b_j) \in \mathit{followpos}(R) \colon b_j\}$,*
- s_0 *is the* initial state, *where* $s_0 = \begin{cases} \mathit{firstpos}(R) \cup \{A\} & \text{if } \mathit{nullable}(R) = \mathit{true} \\ \mathit{firstpos}(R) & \text{otherwise} \end{cases}$
- $F = \{s \in S : A \in s\}$ *is the set of* accepting states, *where A is the accepting state of* NFA(R).

Let's consider the same regular expression $R = (\mathtt{a}_1 \mid \mathtt{b}_2)^* \, \mathtt{a}_3$, which accepts strings $w \in \Sigma^*$ with $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. The construction of *DFA* that supports the above same regular expression is $\mathit{DFA}(R) = \langle \{\mathtt{a}_1, \mathtt{b}_2, \mathtt{a}_3, A\}, \{\mathtt{a}, \mathtt{b}\}, \delta, \{\mathtt{a}_1\}, A \rangle$, where δ is the transition function depicted as labeled edges. The complete *DFA* for the above regular expression is shown in Fig. 2.2.

2.4 Extended Regular Expressions Forms

In real-world applications of pattern matching, extended forms of regular expressions are often used. Following are the commonly used few types of extended regular expressions defined by R. Cox [9]:

Submatch extraction. Efficient submatch extraction of the parsed input strings is for example a regular expression $R = ([0-9]^+/[0-9]^+/[0-9]^+)([0-9]^+ \colon [0-9]^+)$ which may be used to match the strings (say, date and time). Therefore, to efficiently find out which part of accepted

strings w were matched by the sub-expression (i.e. R_1, R_2) of this full regular expression R is crucial.

ESCAPE SEQUENCES. Regular expression containing escape sequences are common in real world applications program. Since, these escape sequences can appear either as a meta-characters (i.e. $\wedge, (,), \backslash, \{, \}, \$$ etc.) or as control sequences such as combination of $\backslash n$ stands for newline, and combination of $\backslash w$ stands for a word character.

COUNTED REPETITION. Used as range quantifiers for example, regular expression containing $a\{n, m\}$ means to match n to m times the letter a. There are several variations of counted repetition in use.

UNANCHORED MATCHES. Another interesting form of extended regular expression, used to do partial-matches on input strings. For example, unanchored "ab" will find matches on both input strings "abc", and "1ab" whereas anchored $\wedge ab\$$ will match only "ab". Note that by default patterns are considered as unanchored.

NON-GREEDY OPERATORS. In traditional or POSIX-mode the commonly used operators *, +, ?, {...} are greedy operators. However, Perl introduced a new version of above operators known as non-greedy operators. These are represented as $*?, +?, ??$.

ASSERTIONS. There are several forms of assertions for extended-regular expression in Perl-compatible mode. The most common form are $\backslash b, (? = re), (?! re), (? \leq re)\, and\, (? <! re)$. The assertion $\backslash b$ is called word boundary, the forms (?=re), (?! re) are known as lookahead assertions and the forms $(? \leq re)$ and $(? <! re)$ are called lookbehind assertions.

BACKREFERENCES. An important feature of extended-regular expressions is backreferences. Normally, backreferences require backtracking operations to match the same pattern again which was already matched by a capturing group. Thus, this leads to *NFA* implementations as common choice. Backreferences represented as $\backslash m$, where m refers to the matched-pattern captured by m^{th} capturing-group. For example, a simple regular expression $R = (\texttt{a} \mid \texttt{b})\, \texttt{c}\, \backslash 1$, where, $\backslash 1$ refers to match the same pattern again which is already captured by first capturing group $(\texttt{a} \mid \texttt{b})$.

In the next chapter, I will present some related-work along with its limitations on efficient subexpressions matching. Also, I will present why my proposed *staDFA* algorithm improves over

existing work as well. I will also show how my proposed approach can resolve the limitations of existing approaches on efficient subexpressions matching.

CHAPTER 3

RELATED WORK

Algorithms for translating *Regular Expressions* (*RE*) to *Deterministic Finite Automata* (*DFA*) are well known [1, 2]. But, these algorithms are not adequate to tell the position and extent of a subexpression within a given *RE*, when we have a matching string. Ville Laurikari [15] proposed a *tagged-DFA* (*TDFA*) to solve this problem efficiently. However, *TDFA* has some practical challenges when using the method for lexical analysis. There is no direct conversion algorithm to obtain the *TDFA* from a regular expression. A *tagged-NFA* (*TNFA*) is constructed first and then a *TDFA* is constructed, which requires two graph constructions whereas methods based on the positions *NFA* require only one construction step. Second, *TDFA* are transition-oriented which is not perfectly suitable for lexical analyzers since the tagged edges require sets of memory updates when a transition is made on an input symbol, which requires edge-annotated tabular representations of the DFA (i.e. encoding the δ transition function in tables) thereby hampering the performance. Third, a *TDFA* cannot be minimized by a *DFA* minimization algorithm such as Hopcroft's method [14], because of the presence of tagged edges. In contrast, my proposed *staDFA* does not have these limitations, which is demonstrated in details on Chapter 4.

Ulya [16] proposed *TDFA*(1) instead of using original TDFA proposed by [15]. The author improved Laurikari's algorithm by using *one-symbol lookahead* concept on original *TDFA* (i.e. called *TDFA*(0)). Results show that significant reduction of tag variables is obtained, thus reducing potential tag conflicts while performing the subexpressions matching. As a consequence, *TDFA*(1) is faster than the original *TDFA*(0). Also, this lookahead-aware *TDFA*(1) is smaller in size than the baseline *TDFA*. These *TDFA*(0) and *TDFA*(1) loosely resemble LR(0) and LR(1) parsers. Now, the interesting part is, we know in LR(1) an item $[A \rightarrow \alpha \cdot \beta, a]$ contains a lookahead terminal `a`, meaning α is already on top of the stack, expecting to see βa. Hence, for an item of the form $[A \rightarrow \alpha \cdot, a]$ the lookahead `a` is used to reduce $A \rightarrow \alpha$ if and only if the next input is `a`. Anyways, if $\beta \neq \epsilon$, then for the above form $[A \rightarrow \alpha \cdot \beta, a]$ lookahead has no effect.

Similarly, on *TDFA*(1) when the next processing input symbol matches the *one-symbol lookahead*, then the *TDFA*(1) removed the transitions containing the tag variables and operations, which is supposed to have conflicts with the next tag variable and its operation in order to make the baseline *TDFA* perform better and reduce its size. For an example, let's assume we have a regular expression, $R = (\,\texttt{b}\,|\,\texttt{bb}\,|\,\texttt{bbb}\,|\,\texttt{bbbb})^*$ and input string "bbbb". Using *TDFA* concept if we rewrite the regular expression, $R = (\,\texttt{tag1 b}\,|\,\texttt{tag2 bb}\,|\,\texttt{tag3 bbb}\,|\,\texttt{tag4 bbbb})^*$ and then if we try to match it for the input string "bbbb" then, tag4 will give the extent of it and at the same time will give the result such that "bbb" of "bbbb" matched the extent of tag3, "bb" of "bbbb" matched the extent of tag2, and finally"b" of "bbbb" matched the extent of tag1. I believe that this crucial aspect is completely overlooked by the *TDFA*(1) approach proposed in [16]. Thus, this may make the original concept of efficient subexpressions matching worse. Even though initially it apears that *TDFA*(1) reduces the tag overheads introduced by the original *TDFA* concept, but it is not in a real sense.

In this way one can make the original *TDFA* faster and smaller in size, but losing the original principal of efficient subexpression matching. However, using the *staDFA* matcher we can even resolve this tags overhead without losing its original principle, which is efficient subexpression matching. Since *staDFA* matcher is state-based, unlike *TDFA*(0) or *TDFA*(1) which is transition-based, we can easily resolve the above tags conflict by resolving the store-transfer-accept conflicts within the states in *staDFA* as explained in Section 4.3. Thus, the *staDFA* Matcher ensures better performance than other existing approaches, such as *TDFA*(0), *TDFA*(1). Also, another big advantage of the *staDFA* matcher is that same marker $\texttt{t} = \texttt{1} \in M[1 : ||R||]$ can be placed into several positions in *RE*.

In [41] Schwarz et al. proposed an algorithm to efficiently extract the full parse tree of an *RE* containing capture groups. In the traditional algorithm used in POSIX-compatible matching, it initially produces partial parse trees instead of full parse trees for the pattern specified by *RE* matching the input strings. This leads to a higher time and space complexity. Therefore, Schwarz et al. [41] proposed an algorithm that can efficiently generate the full parse trees for input strings matched by pattern specified by *RE*. This approach, provides better time and space complexity. Their solution is based on parsing the input strings matching the *regular expression with capture groups* into an abstract syntax tree (AST). Second, they transform the AST to a *TNFA* proposed

in Laurikari [15] and complete all the steps specified in [15]. However, the proposed algorithm in Schwarz et al. [41] is only valid for *RE* with capture groups while having all the same limitations as Ville Laurikari [15].

Parsing input strings and generating parse trees by which we can tell which substrings matched with which *sub-expression*, is another approach. However, this parsing techinque has one major problem which is ambiguity, as presented in [37, 38]. It is common to get two distincts parse tree for the same input strings pattern matching the *sub-expression*. Therefore, if we have an ambiguous *RE* which provides more than one parse trees matching the same input string pattern, then there are two common approaches that exist to perform disambiguation and which is presented by Angelo Borsotti et al. [37] in detail. One is generating parse tree based on POSIX [11] and another one is greedy [13]. However, most implementations of these approaches are buggy specifically all POSIX implementations. Therefore Martin Sulzmann et al. in [38] mainly worked on POSIX-based disambiguation on the Brzozowski's regular expression derivatives in [6].

Disambiguation plays a crucial part to determine which substrings matched with which *sub-expression*. Hence, *staDFA* also does disambiguation, but not in the same manner presented in [11, 13, 37, 38]. Conflicts are resolved by giving the priorities over the combination of *store-transfer-accept* operations on *marked sub-regular expression position* matching the sub-strings in the accepted strings, which I describe in detail on Chapter 4. Thus, having this unique disambiguation technique, we can even match the longest leftmost/rightmost with fast/last matching criteria regardless of classical disambiguation techniques in [11, 13, 37, 38].

Michela and Patrick [5] proposed an extended finite automaton to efficiently support Perl-compatible requlars expressions. They worked to support counting constraints and handling back-references in *RE*. To support the back-references in *RE* they used ad-hoc techniques to augment the corresponding *NFA*. Also, matching sub-strings in actual input strings do not reflect the captured parentheses perfectly within the main *RE*, which I explained with the help of an example in detail in next section. Another drawback of their approach is that they tried to support counting constraints and back-references by augmenting the *NFA*, whereas I did it based on *staDFA* which is also an augmented DFA while having fewer limitations.

In [22] Nakata et al. proposed an algorithm to the submatch problem by adding semantic rules to the *RE* and then converting them to equivalent *DFA*. However, this approach has several limitations.

One major problem is when the *RE* is ambiguous, their algorithm does not work properly for the efficient submatch problem. Another problem is that it can not handle the situations where more than one application of the same semantic rule should be stopped at a certain point. In addition, for a simple *RE* it may need to add several productions to represent the equivalent language, which is undesirable. In contrast, my proposed *staDFA* approach does not have such limitations while handling ambiguous markers.

S. Kumar et al. [17] presented limitations of traditional *DFA* based Network intrusion detection system (*NIDS*), where *RE* matching is the core of *NIDS*. However, the traditional *DFA* used in *NIDS* are suffering from inefficient partial matching signatures, and is thus incapable of efficiently keep the track of matching counts, which causes a potential security vulnerability. Therefore, to ensure security and efficient signatures matching one could use the *staDFA* as an application for the common known security threats and viruses.

I. Nakata [25] used string matching and repeated pattern matching even though better forms of extended regular expression exists. The most difficult situations arise when we need to define the lexical syntax of the C programming language comments. However, they introduced a new special symbol called *any-symbol* • in normal regular expression to generalize the problems of pattern matching. Also, in *TDFA* the same *any-symbol*, • was used in [15] in Section 5.2 to search for matching substrings corresponding to a regular expression R. However, the interesting thing is that their complete work can be replaced by *staDFA* matcher without introducing any special symbol like *any-symbol* in *RE*. In *staDFA* matcher we do have the freedom to give the different prioritization technique over store-transfer-accept which is described in detail in Section 4.3. Thus, this leads to the different longest matching behaviors (e.g. either leftmost or rightmost), and in case of subexpressions repetitions can have the matching criteria such as the first or last substring matching. The *staDFA* matcher can easily be used to further extend the searching algorithm used in both [25, 27] to address the substrings pattern matching problems. Thus, *staDFA* ensures a better working solution approach than [25, 27].

To achieve efficient subexpressions matching for any *RE*, one can use the algorithms proposed in [28, 29], but preprocessing of the search-text [15] have to be performed, which is not possible in the context of lexical analyzers. Alternatively, preprocessing of regular grammars in [30, 31] have to be done before performing the actual subexpressions matching for any *RE*, in order to get an

expected linear time complexity [15]. Another, interesting aspect of using *staDFA* matcher is that it does not require any preprocessing of such techniques for the text to be searched while providing an expected (but not necessarily the worst-case) linear time complexity $\mathcal{O}(|w| \cdot (n + k))$, where $|w|$ is the length of the input strings, n is $||R||$ the alphabetic length of R, and k is the fixed number of repetition operators in regular expression R.

Pedro Garca et al. in [35] proposed a modified algorithm which actually reduced the number of states in *NFA* representation of the equivalent regular expression by using both the concept of partial derivatives automaton proposed in [3] and follow automaton proposed in [33]. Unfortunately, it still has quadratic time complexity instead of preferably linear time as *staDFA*. Also, in [32], the authors introduced derivatives and partial derivatives of *RE* for submatching. All of these approaches in [3, 32, 35] use partial derivatives, thus avoiding the ϵ-transitions from the *NFA*. As a result, they must consider the $\mathcal{O}(n^2)$ transitions unlike $\mathcal{O}(n)$ transitions in Thompson's *NFA*. Therefore, none of them can do better than $\mathcal{O}(|w| \cdot n^2)$ which is quadratic time complexity instead of preferable linear time complexity. Also, [3, 32, 33, 35] are based on *NFAs*, whereas *staDFA* is completely *DFA*-based, thus providing the advantages of speed and determinism.

The most common *DFA* minimization technique is Hopcroft's minimization algorithm in [14]. I propose a modified Hopcroft's minimization algorithm by adjusting it to *staDFA*. The minimization technique has two steps. First, we remove all unnecessary store, transfer, and accept operations generated by the compilation function. Next we, apply Hopcroft's minimization algorithm on the resultant *staDFA*. These two steps are described in detail in Chapter 4. Therefore, we can say that, this *staDFA* approach can provide a faster *DFA* based solution while having no storage complication.

In Chapter 4, I will present my proposed staDFA in detail. That includes, the construction of *staDFA* from the *RE* which can be ambiguous, the disambiguation technique, and a modified Hopcroft's minimization algorithm compatible with my proposed algorithm on efficient subexpression matching.

CHAPTER 4

STORE-TRANSFER-ACCEPT DFA

In this Chapter, I present the concept of *staDFA* in detail, including the construction of an *staDFA* from an *RE*. I will discuss ambiguity and *staDFA* conflict resolution for efficient subexpression matching, comparing and contrasting *tagged-RE* with *staDFA*. I will present a minimization technique for *staDFA*, and finally present some examples of real-world applications using *staDFA*.

4.1 staDFA Definition

This section defines *staDFA* and the marker positions store (*MPS*) used to match input.

4.1.1 Basic Definitions

Given, a regular expression R accepting input string w a *Store-Transfer-Accept DFA* (staDFA) is a *DFA* with commands $\mathtt{C}$ associated with states $s \in S$ as defined as follows:

Definition 3. *An* staDFA *is a 6-tuple* $\langle S, \Sigma, \delta, s_0, F, \mathtt{C} \rangle$*, where*

- S *is a finite set of* states*,*
- Σ *is an* alphabet *defined over a finite set of symbols.*
- $\delta\colon S \times \Sigma \to S$ *is the* transition function*,*
- s_0 *is the* initial state*,*
- F *is the set of* accepting states*,*
- $\mathtt{C}$ *is a set of commands* S_i^t (store)*,* $T_{i,j}^t$ (transfer)*, and* A_i^t (accept) *for markers* $t = 1, \ldots, n$ *with bounded subscripts* $0 \leq i \leq ||R||$*,* $0 \leq j \leq ||R||$*.*

Note that, the commands $\mathtt{C}$ of an *staDFA* are executed on the (*MPS*) by the *staDFA* matcher.

4.1.2 The Marker Positions Store

The *MPS* operations are kept separate from the actual *staDFA* states. The *MPS* operations are executed by the commands `C` associated with a state of a *staDFA*. More generally, the *MPS* consists of two arrays: an array of $M[1:n]$ size containing $t = 1, \ldots, n$ finite set of markers that are used to define the *sub-expression* positions and/or their extent and an array m_i^t indexed by marker t and position i. In practice this array m_i^t is quite sparse, meaning a tabular 2D array has many unused entries thus a compact data structure should be used similar to TNFA and TDFA tag maps. Note that $t \leq n \leq ||R||$ where $||R||$ is the alphabetic length of R. Here, we assumed the *linearized version* [3, 40] of regular expressions to determine positions. A marker $t = 1, \ldots, n$ is a position that marks the *sub-expression* matching the sub-strings of the accepted input strings pattern. For example, substring `abb` on the accepted string $w =$ `abbbcc` starting from 0, matches the *sub-expression* $R_1 = \mathtt{a}_1\,\mathtt{b}_2{}^*$ on RE $R = \mathtt{a}_1\,\mathtt{b}_2{}^*\,\mathtt{b}_3\,\mathtt{c}_4{}^*$. Now, place marker $t = 1$ at address $\mathtt{b}_3$ to identify the position. Hence, the marked RE is $R = \mathtt{a}_1\,\mathtt{b}_2{}^*\,{}^1\!/\,\mathtt{b}_3\,\mathtt{c}_4{}^*$. At the end of the actual *staDFA* match algorithm, array M is returned. Therefore, we will see that $M[1] = 3$ is returned for the used marker $t = 1$ at address $\mathtt{b}_3$ on *RE* R, which accurately identified the positions and extent of subexpression R_1.

The *MPS* holds positions M indexed by marker t. To mark positions in RE, the following meta operators are used:

- ${}^t\!/R_1$ refers to mark-first which are positions in *firstpos*(R_1) with marker t.
- $R_1{\backslash\!}^t$ refers to mark-last which are positions in *lastpos*(R_1) with marker t.
- ${}^t\!/\!_f R_1$ refers to leftmost-first marker of R_1 with marker t. More precisely, it is used to match the "first occurrence" of the left-most captured sub-expression R_1.
- $R_1{\backslash\!}_l^t$ refers to leftmost-last marker of R_1 with marker t. Similarly, it is used to match the "last occurrence" of the rightmost captured sub-expression R_1.

Note that if we parse the accepted input strings based on POSIX-mode, then meta operator markers associated with *MPS* operations will be prefixed as **longest**. Otherwise if we do parsing based on Perl-compatible mode, then will be left off as it is.

When states $s \in S$ in staDFA switch from one state to another state during string matching, we need to update the *MPS* associated with each marker t. To update the *MPS*, each time a command `C` is executed to execute a store S, transfer T, or accept A commands defined as follows:

Definition 4. *The basic* Marker Positions Store(*MPS*) *update operation consists of three operations such as store command S_i^t, transfer command $T_{i,j}^t$ and accept command A_i^t, where*

- *the store command S_i^t executes $m_i^t \leftarrow k$, which implies that each marker t stores the current input string position k into the* MPS *memory cell m_i^t at address location i,*
- *the transfer command $T_{i,j}^t$ executes $m_i^t \leftarrow m_j^t$, which implies that each marker t transfers the content of* MPS *memory cell m_j^t from the address location j to* MPS *memory cell m_i^t at address location i,*
- *the accept command A_i^t executes $M[t] \leftarrow m_i^t$, which implies that each marker t assigns the content of* MPS *memory cell m_i^t from address location i to Marker Positions Store(*MPS*) $M[t]$.*

Definition 3 and the working principle of *MPS* provides the basic *staDFA* concept. In the next section I will demonstrate how a *staDFA* can be automatically constructed from a regular expression.

4.2 Converting a Regular Expression to an staDFA

An *staDFA* is constructed using the well-known subset construction method [1, 2, 21]. The *staDFA* construction introduces commands in states which are generated by a transition compilation function denoted by $\mathcal{C}$ and initial state compilation function denoted by $\mathcal{C}_0$.

Definition 5. *Given a regular expression R over alphabet Σ and the set of marked positions $M^t(R)$ in R with markers $t = 1, \ldots, n$, the staDFA(R) $= \langle S, \Sigma, \delta, s_0, F, \mathtt{C} \rangle$ of the regular expression R is the staDFA defined by:*

- *The set of states $S = \{s_0 \rightarrow^* s \colon s\}$ is generated by the reflexive transitive closure $\rightarrow^*$ of the transition relations $s_i \rightarrow_a s_j$ from source state s_i to target state s_j on symbols $a \in \Sigma$ defined by $\delta(s_i, a) = s_j$ of the staDFA(R).*
- *The transition function of the staDFA(R) is defined by:*

  $$\delta(s, a) = \bigcup_{a_i \in s, (a_i, b_j) \in followpos(R)} \mathcal{C}(a_i, b_j),$$

  where the transition compilation function *$\mathcal{C}(a_i, b_j)$ is defined by:*

  $$\mathcal{C}(a_i, b_j) = \begin{cases} \mathcal{T}(b_j) \cup \{b_{j_i^t}, T_{i,h}^t\} & \text{if } a_i = a_{i_h^t} \text{ for some } t \text{ and } h \\ \mathcal{T}(b_j) & \text{if } \mathcal{T}(b_j) \neq \varnothing \\ \{b_j\} & \text{otherwise;} \end{cases}$$

where the target position b_j is compiled with

$$\mathcal{T}(b_j) = \bigcup_{t' \in T, b_j \in M^{t'}(R)} \{b_{j_i^{t'}}, S_i^t\}$$

with $T = \{1, \ldots, n\}$ the set of markers used in the regular expression. Note that $a_i \in s, (a_i, b_j) \in followpos(R)$ in the definition of δ means that we take positions a_i (including marked positions $a_{i_j^t}$) that match symbol a only, thus ignoring all commands in s. Further note that $a_i = a_{i_h^t}$ in the definition of $\mathcal{C}$ means that the source position a_i is marked as $a_{i_h^t}$. When the target position b_j is the special accepting position A, $\mathcal{C}(a_i, A)$ translates A with $\mathcal{T}(A)$ to an accept operation A_i^t for each marker $t \in T$, if $A \in M^t(R)$ or if a_i is marked as $a_{i_h^t}$.

- *The initial state of the staDFA(R) is defined by:*

$$s_0 = \bigcup_{a_i \in firstpos(R)} \mathcal{C}_0(a_i),$$

where the initial state compilation function $\mathcal{C}_0(a_i)$ *is defined by:*

$$\mathcal{C}_0(a_i) = \begin{cases} \mathcal{I}(a_i) & \text{if } \mathcal{I}(a_i) \neq \varnothing \\ \{a_i\} & \text{otherwise.} \end{cases}$$

where the initial position a_i is compiled with

$$\mathcal{I}(a_i) = \bigcup_{t' \in T, a_i \in M^{t'}(R)} \{a_{i_0^{t'}}, T_{i,0}^{t'}\}$$

When a position a_i is the special accepting position A, $\mathcal{C}_0(A)$ translates A to an accept operation A_0^t for each marker $t \in T$ with $T = \{1, \ldots, n\}$, if $A \in M^t(R)$.

- *$F = \{s \in S : A \in s \vee A_i^t \in s\}$ is the set of accepting states, which are states that contain A or a marked A_i^t for any marker t and subscript i.*

- *$\mathtt{C}(s) = \{S_i^t \in s\} \cup \{T_{i,j}^t \in s\} \cup \{A_i^t \in s\}$ is the set of commands of a given state s.*

We now prove that *staDFA(R)* construction terminates.

Theorem 1. *Given a regular expression R and a finite number n of markers $t = 1, \ldots, n$, staDFA(R) terminates with a finite set of states.*

Proof. By Definition 5, every state $s \in S$ of the *staDFA(R)* is a set of marked or unmarked positions and commands. States are uniquely identified by the content of their sets. Every subscript index i and j in marked positions a_i and commands S_i^t, $T_{i,j}^t$ and A_i^t are bounded from below by 0 (or 1 in the case of $T_{i,0}^t$) and bounded from above by $||R||$. This gives a finite number of possibilities for the content of the state's sets, and therefore *staDFA(R)* terminates with a finite set of states S. □

As a consequence of Theorem 1, the *MPS* memory used in Algorithm 1 *staDFA-Match* has a finite number of memory cells for markers. In the worst case, the *MPS* has a full $n \times ||R||$ matrix of memory cells in use for n markers. In practice however, there will be far fewer matrix cells in use when *staDFA-Match* executes, suggesting that a sparse representation such as a hash map should be used that is more space friendly.

The number of states of an *staDFA(R)* may be greater than the number of states of a DFA(R), given $n > 0$ markers. For $n = 0$ markers, the number of states is the same as expected.

Now that we can construct an *staDFA(R)* in a finite amount of time and space, the question is whether or not it accepts strings matching the regular expression R.

Theorem 2. *The* staDFA(R) *of a regular expression R accepts strings matching R.*

Proof. Given that *DFA*(R) defined in Definition 2 accepts strings matching R, we prove that *staDFA(R)* accepts strings matching R by showing that the states generated by the transition functions of the two DFAs are semantically equivalent.

Two states s and s' are semantically equivalent $s \equiv s'$, but not necessarily structurally identical, if states s and s' contain the same positions a_i that are either marked (as in $a_{i_j^t}$) or unmarked. Any S_i^t and $T_{i,j}^t$ commands in states s or s' are irrelevant with respect to their semantic equivalence.

Starting by proving the base case s_0, we then prove that δ of the *staDFA(R)* generates semantically equivalent states to the δ' of the *DFA*(R):

- Let $s_0 = \bigcup_{a_i \in \mathit{firstpos}(R)} \mathcal{C}_0(a_i)$ be the initial state of *staDFA(R)* by Definition 4 and let $s'_0 = \mathit{firstpos}(R)$ be the initial state of *DFA*(R) by Definition 2. Then $s_0 \equiv s'_0$, since the initial state compilation function $\mathcal{C}_0(a_i)$ returns $\{a_i\}$ for all $a_i \in \mathit{firstpos}(R)$, together with (semantically irrelevant) commands $T_{i,0}^t$.
- Given the transition function δ of the *staDFA(R)* and the transition function δ' of the *DFA*(R), for any state s we have that $\delta(s,a) \equiv \delta'(s,a)$, since $\delta'(s,a) = \bigcup_{a_i \in s} \{(a_i, b_j) \in \mathit{followpos}(R) \colon b_j\}$ and the transition compilation function $\mathcal{C}(a_i, b_j)$ in $\delta(s,a) = \bigcup_{a_i \in s, (a_i,b_j) \in \mathit{followpos}(R)} \mathcal{C}(a_i, b_j)$ returns $\{b_j\}$ or a set of marked $b_{j_i^t}$ for all $a_i \in s$ and $(a_i, b_j) \in \mathit{followpos}(R)$, together with (semantically irrelevant) commands S_i^t and $T_{i,h}^t$.

Observe that the accepting states of *staDFA(R)* are semantically equivalent to the accepting states of *DFA*(R), since both accepting states contain the accepting position A or a marked accepting position A_i^t. Since the initial states and states generated by δ of the *staDFA(R)* are semantically

equivalent to the states generated by δ' of the *DFA*(R), and the fact that the *DFA*(R) accepts strings matching R, the *staDFA(R)* accepts strings matching R. □

By Theorems 1 and 2, an *staDFA(R)* can be constructed for a given regular expression R with n markers in a finite amount of time and space that accepts strings matching R.

See for example the *staDFA(R)* constructed for $(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^{1}\!/\, \mathtt{b_4}\,\mathtt{a_5}$ with $\mathtt{b_4}$ marked $t = 1$, as depicted in Fig. 4.1.

4.2.1 Identifying Subexpression Positions

To identify the particular *sub-expression* R_1 within the *RE* R, we have to mark the *sub-expression* R_1 using the marker meta operators defined in Section 4.1.2. However, for a simple marker t on *RE* R the semantic meaning of these marker meta operators can be defined by a *marked positions set* $M^t(R)$ as follows:

$$M^t(R) = \begin{cases} M^t(R_1) \cup firstpos(R_1) & \text{if } R = {}^{t}\!/R_1 \\ M^t(R_1) \cup lastpos(R_1) & \text{if } R = R_1 \backslash\!^{t} \\ M^t(R_1) & \text{if } R = (R_1)^* \\ M^t(R_1) \cup M^t(R_2) & \text{if } R = R_1\, R_2 \\ M^t(R_1) \cup M^t(R_2) & \text{if } R = R_1 \mid R_2 \\ \varnothing & \text{otherwise} \end{cases}$$

Consider for example *RE* $R = {}^{1}\!/(\mathtt{a_1}\,\mathtt{b_2}\,\mathtt{c_3})^* \backslash\!^{2}\,\mathtt{d_4}$. Here, this *RE* R contains a *sub-expression* $R_1 = {}^{1}\!/(\mathtt{a_1}\,\mathtt{b_2}\,\mathtt{c_3})^* \backslash\!^{2}$ known as capturing group, too. Now, using *marked positions set*, we get $M^1(R_1) = firstpos(a_1\, b_2\, c_3) = \{a_1\}$ and $M^2(R_1) = lastpos(a_1\, b_2\, c_3) = \{c_3\}$. However, in some cases *sub-expression* R_1 can be ambiguous. For example, if we slightly change the above *sub-expression* $R_1 = {}^{1}\!/(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{c_3})^* \backslash\!^{2}$, then it becomes ambiguous and at this point we can have staDFA(R_1) conflicts. So, extra care has to be taken, which we will describe in detail in Section 4.3. Actually, for this special case we introduced two additional meta operators called as "leftmost-first marker" ${}^{1}\!/\!_{f} R_1$ and "leftmost-last marker" $R_1 \backslash\!_{l}^{1}$ for disambiguating markers with conflict resolution policies described in detail in Section 4.3. Therefore, by disambiguating markers with conflict resolution policies, the non-ambiguous representation of the above ambiguous sub-expression R_1 will be $R_1 =$

$^{1}\!/\!_{f}(\mathtt{a}_1 \mid \mathtt{a}_2\,\mathtt{c}_3)^*\,{}^{2}\!\backslash\!_{l}$. That is as follows:

$$M^t(R_1) = \begin{cases} \mathit{firstpos}(\mathtt{a}_1) = \mathtt{a}_1 & \text{for marker t = 1 is leftmost-first } {}^{1}\!/\!_{f} \\ \mathit{firstpos}(\mathtt{a}_2\,\mathtt{c}_3) = \mathtt{a}_2 & \text{for marker t = 1 is leftmost-first } {}^{1}\!/\!_{f} \\ \mathit{lastpos}(\mathtt{a}_1) = \mathtt{a}_1 & \text{for marker t = 2 is leftmost-last } {}^{2}\!\backslash\!_{l} \\ \mathit{lastpos}(\mathtt{a}_2\,\mathtt{c}_3) = \mathtt{c}_3 & \text{for marker t = 2 is leftmost-last } {}^{2}\!\backslash\!_{l} \end{cases}$$

However, we have already learned how to identify *sub-expression* positions effectively using marker t. So, in the next section we are going to present the complete staDFA matcher algorithm to do the efficient *sub-expression* matching.

4.2.2 staDFA Match

Algorithm 1 staDFA-*Match* is a matching algorithm. In addition to the subset construction algorithm, we added a command function $\mathcal{C}$ which adds commands S_i^t *(store)*, $T_{i,j}^t$ *(transfer)*, and A_i^t *(accept)* to states, for markers $t = 1, \ldots, n$ with bounded subscripts $0 \leq i \leq ||R||$, $0 \leq j \leq ||R||$, where $||R||$ refers to the alphabetic length of R. In Algorithm 1, line 4 to 16 traverse the complete staDFA generated by the subset construction algorithm which includes *followpos, firstpost, etc.* The staDFA *Matcher* efficiently and effectively matches strings $w_i \in w[0 : \ell - 1]$, where w_i directly refers to the number of matched sub-expression R_i within the *RE* R for the complete accepted input string w.

The expected (but not worst case) upper bound running time complexity of Algorithm 1 (i.e. *staDFA* Match) is $\mathcal{O}(|w| \cdot (n + k))$, where $|w|$ is the length of input strings, n is the constant $||R||$ refers to the alphabetic length of R, and k refers to the number of repetition operators in *RE* R.

4.2.3 Subexpression Matching with an staDFA

Now, at this point we know how to identify subexpression positions, and can describe *staDFA* matcher algorithm. Let's look at a real example and see how our proposed *staDFA* matcher efficiently and effectively matches the sub-expression positions and tells its extent on the accepted input string w. Consider, $R = (\mathtt{a}_1 \mid \mathtt{a}_2\,\mathtt{b}_3)^*\,{}^{1}\!/\,\mathtt{b}_4\,\mathtt{a}_5$ where, sub-expression $R_1 = (\mathtt{a}_1 \mid \mathtt{a}_2\,\mathtt{b}_3)^*$ and its extent is at $\mathtt{b}_4\,\mathtt{a}_5$ in the *RE* R. Now, for the simple accepted input strings such as "`aba`", and "`abba`" we will demonstrate, how an *staDFA* matcher matches the sub-expression R_1 and tells its extent to $\mathtt{b}_4\,\mathtt{a}_5$ in the *RE* R. Fig. 4.1 shows the *staDFA* representation of the above *RE* R.

Table 4.1 shows the step by step processing of input strings symbol. At the same time, showing the *MPS* operations associated with states of *staDFA* matcher. For example, on accepted input

Algorithm 1: staDFA *Match*

input : staDFA $\langle S, \Sigma, \delta, s_0, F, \mathtt{C}\rangle$, $||R||$ markers, and string $w \in \Sigma^*$

output: $\ell \neq -1$ when sub-string $w_i \in w[0 : \ell - 1]$ matches *sub-expression* with marker positions $M[1 : ||R||]$, Otherwise $\ell = -1$ means w_i does not match the *sub-expression*

```
1  ℓ ← −1;
2  k ← 0;
3  s ← s_0;
4  while s ≠ ∅ do
5      forall S_i^t ∈ C do m_i^t ← k;
6      forall T_{i,j}^t ∈ C do m_i^t ← m_j^t;
7      if s ∈ F then
8          ℓ ← k;
9          M[1 : ||R||] ← −1;
10         forall A_i^t ∈ C do M[t] ← m_i^t;
11     if k ≥ |w| then
12         s ← ∅;
13     else
14         a ← w[k];
15         s ← δ(s, a);
16         k ← k + 1;
```

strings "abba", the *staDFA* matcher returns the value for marker $\mathtt{t_1} = 2$. This implies that it correctly identified the sub-expression R_1 positions and it's extent to $\mathtt{b_4\, a_5}$ on accepted input strings "abba" starting from 0.

4.3 Ambiguity and staDFA Conflicts

For example, the regular expression $R = (\mathtt{a_1} \mid \mathtt{a_2\, b_3})^*\, \mathtt{b_4}$ is ambiguous. Because, it contains the ambiguous *sub-expression* $R_1 = (\mathtt{a_1} \mid \mathtt{a_2\, b_3})^*$, which also contains the repetition quantifier. For example, if we have an accepted input string like "ababab" then for identifying this capturing group *sub-expression* R_1 positions in repetitions using only one marker $\mathtt{t} = 1$ will produce incorrect result. This is because the presence of ambiguity in *RE* and in repetitions this capturing group *sub-expression* R_1 positions may match with the first or last substrings, which results in *staDFA* conflicts on the overall accepted input string "ababab". Hence, the presence of this kind of ambiguities in

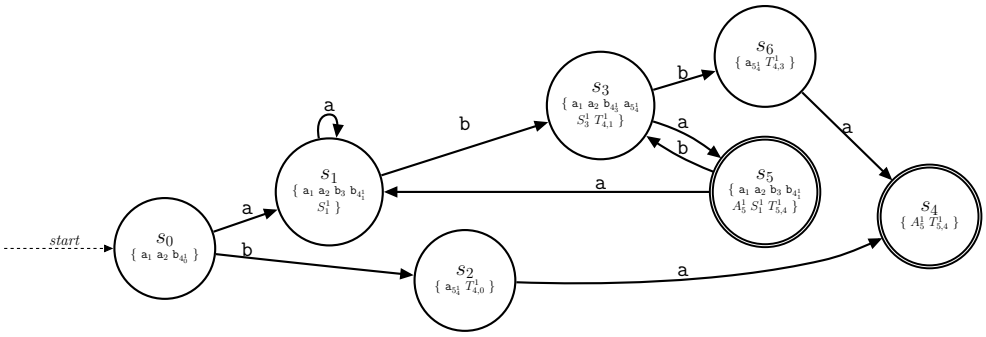


Figure 4.1: An *staDFA* for *RE* $(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^1\!/\,\mathtt{b_4}\,\mathtt{a_5}$

Table 4.1: Steps of *sub-expression* $R_1 = (\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*$ matching and its extent to position $\mathtt{b_4}$ marked using marker $t = 1$ with the staDFA matcher shown in Fig. 4.1 on input strings "aba" and "abba" for *RE* $R = (\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^1\!/\,\mathtt{b_4}\,\mathtt{a_5}$

k	$w[k]$	*State* s	*Commands* C	*MPS Updates*
0	a	s_0	$\varnothing$	
1	b	s_1	$\{S_1^1\}$	$m_1^1 \leftarrow k$ $(=1)$
2	a	s_3	$\{S_3^1, T_{4,1}^1\}$	$m_3^1 \leftarrow k$ $(=2)$ $m_4^1 \leftarrow m_1^1$ $(=1)$
3	-	s_5	$\{S_1^1, T_{5,4}^1, A_5^1\}$	$m_1^1 \leftarrow w[k] = \emptyset$ $(=-1)$ $m_5^1 \leftarrow m_4^1$ $(=1)$ accept $M[1] \leftarrow m_5^1 (=1)$

k	$w[k]$	*State* s	*Commands* C	*MPS Updates*
0	a	s_0	$\varnothing$	
1	b	s_1	$\{S_1^1\}$	$m_1^1 \leftarrow k$ $(=1)$
2	b	s_3	$\{S_3^1, T_{4,1}^1\}$	$m_3^1 \leftarrow k$ $(=2)$ $m_4^1 \leftarrow m_1^1$ $(=1)$
3	a	s_6	$\{T_{4,3}^1\}$	$m_4^1 \leftarrow m_3^1$ $(=2)$
4	-	s_4	$\{T_{5,4}^1, A_5^1\}$	$m_5^1 \leftarrow m_4^1$ $(=2)$ accept $M[1] \leftarrow m_5^1 (=2)$

RE R, could be leading us to have the following staDFA conflicts on the state $s \in S$ of an *staDFA* matcher:

Definition 6. *Ambiguities in* staDFA

- *state* $s \in S$ *has a* store-transfer conflict, *when both* $S_i^t \in$ C *and* $T_{i,j}^t \in$ C *for the same address*

location i tries to save values at the same time on the same MPS *memory cell represented by the marker t.*

- *state $s \in S$ has a* transfer-transfer conflict*, when both $T^t_{i,j} \in \mathtt{C}$ and $T^t_{i,h} \in \mathtt{C}$ such that $j \neq h$, tries to transfer values in the same* MPS *memory cell represented by the marker t for the same address location i.*

- *state $s \in F$, has an* accept-accept *conflict, when both $A^t_i \in \mathtt{C}$ and $A^t_j \in \mathtt{C}$ such that $i \neq j$, tries to set the final value on the* MPS *memory cell for the same marker t.*

To avoid *staDFA* conflicts in each state $s \in S$, we must have at most one S^t_i(store), one $T^t_{i,j}$ (transfer) and one A^t_i (accept) commands associated with the same marker t and index i. Hence, by imposing following *staDFA* conflict resolution policy, we standardized our proposed *staDFA* Methods:

Definition 7. *Removing Disambiguations in* staDFA

- *resolving store-transfer conflicts, among S^t_i and $T^t_{i,j}$ for marker t with identical subscripts i by keeping the transfer $T^t_{i,j}$ while removing the store S^t_i,*

- *resolving transfer-transfer conflicts, among $T^t_{i,j}$ and $T^t_{i,h}$ for markers t with identical subscripts i by keeping the transfer $T^t_{i,j}$ while removing all other transfers $T^t_{i,h}$ such that $h > j$, and*

- *resolving accept-accept conflicts, among A^t_i and A^t_h for marker t by keeping the accept A^t_i while removing all other accepts A^t_h such that $h > i$.*

Now, after adopting above conflicts resolution policy we will have the conflict-free *staDFA* Match which can effectively and efficiently match the *sub-expression* positions and its extent. More specifically, the *sub-expression*, which is not only limited to ambiguousness, but can also have repetition quantifiers, backreferences, capturing groups, and etc. Note that the above conflict resolution will result the longest left-most first match. If we look for longest right-most match then we just simply need to flip the above choices. For example $R = {}^{1}\!/_{f}\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^{2}\!\backslash\!\!\backslash_{l}\,\mathtt{b_4}$, where *sub-expression* $R_1 = {}^{1}\!/_{f}\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^{2}\!\backslash\!\!\backslash_{l}$ is marked by using markers t_1 and t_2, respectively.

Fig. 4.2 shows the staDFA representation of the above *RE* R and Table 4.2 shows the step by step processing of input strings symbol. At the same time, Table 4.2 shows the *MPS* operations associated with states of the staDFA matcher. For example, on the accepted input string "`ababab`",

Table 4.2: Steps of *sub-expression* $R_1 = \frac{1}{f}\,(\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*\,\frac{2}{\backslash}$ matching by using markers t_1 and t_2 and its extent to position $\mathtt{b_4}$ with the staDFA matcher shown in Fig. 4.2 on input strings "ababab" for *RE* $R = \frac{1}{f}\,(\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*\,\frac{2}{\backslash}\,\mathtt{b_4}$

k	$w[k]$	*State* s	*Commands* C	*MPS Updates*
0	a	s_0	$\varnothing$	
1	b	s_1	$\{S^2_2, T^1_{1,0}, T^1_{2,0}, T^2_{1,0}\}$	$m^2_2 \leftarrow k \quad (=1)$ $m^1_1 \leftarrow m^1_0 \quad (=0)$ $m^1_2 \leftarrow m^1_0 \quad (=0)$ $m^2_1 \leftarrow m^2_0 \quad (=0)$
2	a	s_4	$\{T^1_{3,2}, T^1_{4,1}, T^2_{3,2}, T^2_{4,1}, A^1_4, A^2_4\}$	$m^1_3 \leftarrow m^1_2 \quad (=0)$ $m^1_4 \leftarrow m^1_1 \quad (=0)$ $m^2_3 \leftarrow m^2_2 \quad (=1)$ $m^2_4 \leftarrow m^2_1 \quad (=0)$ accept $M[1] \leftarrow m^1_4 (=0)$ accept $M[2] \leftarrow m^2_4 (=0)$
3	b	s_5	$\{T^1_{1,3}, T^1_{2,3}, T^2_{1,3}, T^2_{2,3}\}$	$m^1_1 \leftarrow m^1_3 \quad (=0)$ $m^1_2 \leftarrow m^1_3 \quad (=0)$ $m^2_1 \leftarrow m^2_3 \quad (=1)$ $m^2_2 \leftarrow m^2_3 \quad (=1)$
4	a	s_4	$\{T^1_{3,2}, T^1_{4,1}, T^2_{3,2}, T^2_{4,1}, A^1_4, A^2_4\}$	$m^1_3 \leftarrow m^1_2 \quad (=0)$ $m^1_4 \leftarrow m^1_1 \quad (=0)$ $m^2_3 \leftarrow m^2_2 \quad (=1)$ $m^2_4 \leftarrow m^2_1 \quad (=1)$ accept $M[1] \leftarrow m^1_4 (=0)$ accept $M[1] \leftarrow m^2_4 (=1)$
5	b	s_5	$\{T^1_{1,3}, T^1_{2,3}, T^2_{1,3}, T^2_{2,3}\}$	$m^1_1 \leftarrow m^1_3 \quad (=0)$ $m^1_2 \leftarrow m^1_3 \quad (=0)$ $m^2_1 \leftarrow m^2_3 \quad (=1)$ $m^2_2 \leftarrow m^2_3 \quad (=1)$
6	-	s_4	$\{T^1_{3,2}, T^1_{4,1}, T^2_{3,2}, T^2_{4,1}, A^1_4, A^2_4\}$	$m^1_3 \leftarrow m^1_2 \quad (=0)$ $m^1_4 \leftarrow m^1_1 \quad (=0)$ $m^2_3 \leftarrow m^2_2 \quad (=1)$ $m^2_4 \leftarrow m^2_1 \quad (=1)$ accept $M[1] \leftarrow m^1_4 (=0)$ accept $M[1] \leftarrow m^2_4 (=1)$

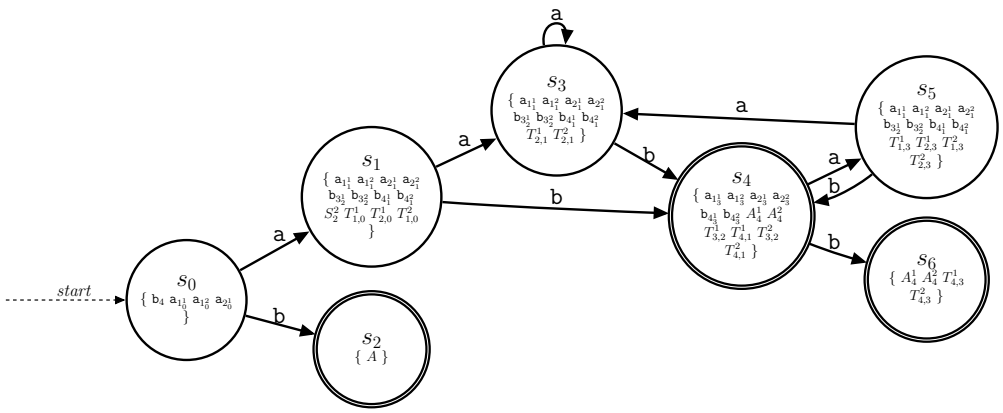


Figure 4.2: The leftmost-first *staDFA* for *RE* $R = \not{1}_f \left(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3}\right)^* \not{l}^2_{\backslash} \mathtt{b_4}$

the staDFA matcher returns the value for marker $\mathtt{t_1} = \mathtt{0}$ and marker $\mathtt{t_2} = \mathtt{1}$ which on *MPS* represented as $M[1:0\,,\,2:1]$. This implies that it correctly identified the sub-expression R_1 positions and it's extent to $\mathtt{b_4}$ on accepted input strings "ababab" starting from 0.

However, one more important thing is that in case of repetitions of *sub-expression*, it can match the first or last sub-strings $w_i \in w[0:\ell-1]$. For example, if we examine Table 4.2, we see that *sub-expression* $R_1 = \not{1}_f \left(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3}\right)^* \not{l}^2_{\backslash}$ in repetitions(i.e. capturing groups) matches the first sub-strings $w_i \in w[0:\ell-1]$ instead of last. Therefore, if we are interested in finding the last matches of *sub-expression* positions in repetitions then, we have to make a minor change to our proposed staDFA matcher Algorithm 1. The change is simple which is changing the lines 5 and 6 of Algorithm 1 to:

---

5 **forall** $\bar{S}_i^t \in \bar{\mathtt{C}}$ **do** $m_i^t \leftarrow k-1$;
6 **forall** $S_i^t \in \mathtt{C}$ **do** $m_i^t \leftarrow k$;
7 **forall** $T_{i,j}^t \in \mathtt{C}$ **do** $m_i^t \leftarrow m_j^t$;

Actually, to get the last match we added a new compilation command function represented by $\bar{\mathcal{C}}$ which executes a $\bar{S}_i^t$ command called a *delayed store* defined by:

- $\bar{S}_i^t$ executes $m_i^t \leftarrow k-1$, storing the previous position $k-1$ of the input string into the *MPS* memory cell m_i^t of marker t at address location i.

To facilitate this delayed store, the transition compilation function $\mathcal{C}$ is changed and renamed to $\bar{\mathcal{C}}$ as follows:

$$\bar{\mathcal{C}}(a_i, b_j) = \begin{cases} \bar{\mathcal{S}}(a_i) \cup \bar{\mathcal{T}}(b_j) \cup \{b_{j_i^t}, T_{i,h}^t\} & \text{if } a_i = a_{i_h^t} \text{ for some } t \text{ and } h \\ \bar{\mathcal{T}}(b_j) & \text{if } \bar{\mathcal{T}}(b_j) \neq \varnothing \\ \{b_j\} & \text{otherwise,} \end{cases}$$

where the source position a_i is compiled with

$$\bar{\mathcal{S}}(a_i) = \{t' \in T, a_i \in M^{t'}(R) \colon \bar{S}_i^{t'}\},$$

and where the target position b_j is compiled with

$$\bar{\mathcal{T}}(b_j) = \{t' \in T, b_j \in M^{t'}(R) \colon b_{j_i^{t'}}\},$$

where $T = \{1, \ldots, n\}$ is the set of n markers.

Note that it also implies that a *delayed store* $\bar{S}_i^t$ command must be executed before the *transfer* $T_{i,j}^t$ command in Algorithm 1 *staDFA* as stores are delayed, which can be easily done by resolving store-transfer conflicts defined in Definition 6. That is, in case of store-transfer conflicts we need to prioritize a store command over a transfer command. Now, by having these simple change on *staDFA* Match, we can get the last occurrence of *RE* positions matching the sub-strings on the accepted input strings. Again, let's consider the previous ambiguous *RE* $R = {}^{1}\!/_{f}\,(\mathtt{a_1} \mid \mathtt{a_2\,b_3})^* \,{}^{2}\!\backslash\!\!\backslash\, \mathtt{b_4}$, where sub-expression $R_1 = {}^{1}\!/_{f}\,(\mathtt{a_1} \mid \mathtt{a_2\,b_3})^* \,{}^{2}\!\backslash\!\!\backslash$ is marked by using markers t_1 and t_2, respectively.

Fig. 4.3 shows the staDFA representation of the above *RE* R which matches the last occurrence of *sub-expression* R_1 positions matching the sub-strings $w_i \in w[0 : \ell - 1]$ on the accepted input strings. Also, Table 4.3 shows the step by step processing of input strings symbol. At the same time, Table 4.3 shows the *MPS* operations associated with states of staDFA matcher. For example, on the accepted input string "`ababab`", the staDFA matcher returns the value for marker $\mathtt{t_1} = 4$ and marker $\mathtt{t_2} = 4$ which on *MPS* is represented as $M[1:4\,,\,2:4]$. This implies that it correctly identified the last occurrence of *sub-expression* R_1 positions and it's extent to $\mathtt{b_4}$ on accepted input strings "`ababab`" starting from 0. Therefore, we can say that by having these user-defined/flexible compilation command functions such as $\mathcal{C}, and\ \bar{\mathcal{C}}$ along with these *STA* conflicts resolution policy on our proposed *staDFA* Matcher algorithm, we can achieve any type of matching policy (i.e. first, last, etc.).

Table 4.3: Steps of the last occurrence of *sub-expression* $R_1 = \not{/}_f\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*\,\not{\backslash}^2$ matching by using markers t_1 and t_2 and its extent to position $\mathtt{b_4}$ with the staDFA matcher shown in Fig. 4.3 on input strings "ababab" for *RE* $R = \not{/}_f\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*\,\not{\backslash}^2\,\mathtt{b_4}$

k	$w[k]$	*State* s	*Commands* C	*MPS Updates*
0	a	s_0	$\varnothing$	
1	b	s_1	$\{S_1^1, S_2^1, S_1^2\}$	$m_1^1 \leftarrow k-1 \quad (=0)$ $m_2^1 \leftarrow k-1 \quad (=0)$ $m_1^2 \leftarrow k-1 \quad (=0)$
2	a	s_4	$\{S_3^2, T_{3,2}^1, T_{4,1}^1, T_{4,1}^2, A_4^1, A_4^2\}$	$m_3^2 \leftarrow k-1 \quad (=1)$ $m_3^1 \leftarrow m_2^1 \quad (=0)$ $m_4^1 \leftarrow m_1^1 \quad (=0)$ $m_4^2 \leftarrow m_1^2 \quad (=0)$ accept $M[1] \leftarrow m_4^1 (=0)$ accept $M[2] \leftarrow m_4^2 (=0)$
3	b	s_5	$\{S_1^1, S_2^1, S_1^2, T_{2,3}^2\}$	$m_1^1 \leftarrow k-1 \quad (=2)$ $m_2^1 \leftarrow k-1 \quad (=2)$ $m_1^2 \leftarrow k-1 \quad (=2)$ $m_2^2 \leftarrow m_3^2 \quad (=1)$
4	a	s_4	$\{S_3^2, T_{3,2}^1, T_{4,1}^1, T_{4,1}^2, A_4^1, A_4^2\}$	$m_3^2 \leftarrow k-1 \quad (=3)$ $m_3^1 \leftarrow m_2^1 \quad (=2)$ $m_4^1 \leftarrow m_1^1 \quad (=2)$ $m_4^2 \leftarrow m_1^2 \quad (=2)$ accept $M[1] \leftarrow m_4^1 (=2)$ accept $M[2] \leftarrow m_4^2 (=2)$
5	b	s_5	$\{S_1^1, S_2^1, S_1^2, T_{2,3}^2\}$	$m_1^1 \leftarrow k-1 \quad (=4)$ $m_2^1 \leftarrow k-1 \quad (=4)$ $m_1^2 \leftarrow k-1 \quad (=4)$ $m_2^2 \leftarrow m_3^2 \quad (=3)$
6	a	s_4	$\{S_3^2, T_{3,2}^1, T_{4,1}^1, T_{4,1}^2, A_4^1, A_4^2\}$	$m_3^2 \leftarrow k-1 \quad (=5)$ $m_3^1 \leftarrow m_2^1 \quad (=4)$ $m_4^1 \leftarrow m_1^1 \quad (=4)$ $m_4^2 \leftarrow m_1^2 \quad (=4)$ accept $M[1] \leftarrow m_4^1 (=4)$ accept $M[2] \leftarrow m_4^2 (=4)$

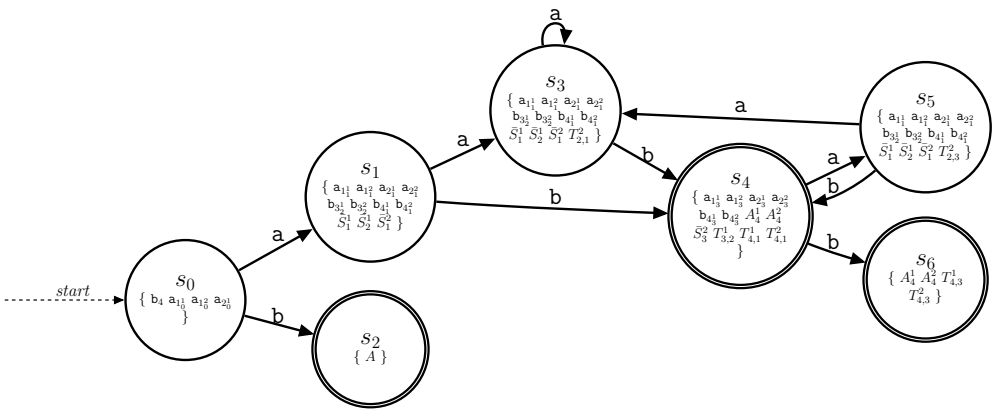


Figure 4.3: The leftmost-last *staDFA* for *RE* $R = {}^{1}\!/\!_{f}\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^{*}\,{}^{2}\!\backslash\!_{l}\,\mathtt{b_4}$

4.4 Tagged Regular Expressions and staDFA

In this section, I will show how a tagged *RE* (i.e. *TNFA-TDFA*) [15] is integrated in *staDFA*. Since tags t_x used in [15] refers to matches of ε, hence it can easily be incorporated in our proposed staDFA*s*. Because, the staDFA Matcher accepts *RE* R which can contain ε as defined in Section 2.1. Thus, allowing these tags t_x in the followpos, firstpos, and lastpos set of *RE* R. For example, tagged edge transitions (a_i, b_j) in the followpos set refers to mark positions b_j in staDFA states. Therefore, integrating the tagged *RE* (i.e. *TNFA-TDFA*) proposed in [15] on staDFA*s* is straightforward.

To compile the staDFA states, the modified transition compilation function $\hat{\mathcal{C}}$ uses a modified function $\hat{\mathcal{T}}(a_i, b_j)$ to recognize tags as follows:

$$\hat{\mathcal{C}}(a_i, b_j) = \begin{cases} \hat{\mathcal{T}}(a_i, b_j) \cup \{b_{j_i^x}, T^x_{i,h}\} & \text{if } a_i = a_{i_h^x} \text{ for some tag } t_x \text{ and some } h \\ \hat{\mathcal{T}}(a_i, b_j) & \text{if } \hat{\mathcal{T}}(a_i, b_j) \neq \varnothing \\ \{b_j\} & \text{otherwise} \end{cases}$$

where edge (a_i, b_j) is compiled with

$$\hat{\mathcal{T}}(a_i, b_j) = \bigcup_{(a_i,t_x)\in followpos(R),(t_x,b_j)\in followpos(R)} \{b_{j_i^x}, S^x_i\}$$

The modified initial state compilation function $\hat{\mathcal{C}}_0$ uses a modified function $\hat{\mathcal{I}}(a_i)$ as follows:

$$\hat{\mathcal{C}}_0(a_i) = \begin{cases} \hat{\mathcal{I}}(a_i) & \text{if } \hat{\mathcal{I}}(a_i) \neq \varnothing \\ \{a_i\} & \text{otherwise} \end{cases}$$

where

$$\hat{\mathcal{I}}(a_i) = \{t_x \in \mathit{firstpos}(R), (t_x, a_i) \in \mathit{followpos}(R) \colon a_{i_0^x}\}$$

In other words, tagged edges (a_i, b_j) in the $\mathit{followpos}(R)$ set are used to mark positions b_j in staDFA states. Otherwise, tags are simply ignored. Conflicts in staDFA states are subsequently resolved for leftmost-first matching as described in Section 4.3. However, leftmost-last matching poses a problem with tags, because the delayed store commands required cannot be associated with states based on the tagging of transitions alone.

After conflict resolution we obtain an staDFA without tagged transitions but with marked states. Because, staDFA construction is easily adaptable to tagged regular expressions, staDFAs subsume *TDFAs*. However, one thing we have to keep in mind, that is in the case of leftmost-last matching using the tagging of transitions alone is not sufficient. In other words, extra care has to be taken on delayed store $\bar{S}_i^t$ command.

While staDFA matcher has several advantages over *TDFA*, *TDFA* has one advantage over staDFA Matcher. This particular case is, tags t_x can be placed on *RE* which itself even contains a ε. For example, *RE* $R = (t_1 \mid a_1)\, b_2$ where *sub-expression* $R_1 = (t_1 \mid a_1)$ itself contains a ε position marked by marker t_1. In this case, for an accepted input string such as `b` Marker Positions Store (*MPS*) returns M[1] = 0 means matches t_1 but if we have the accepted input string like `ab`, then *MPS* returns M[1] = -1 means that did not match the t_1.

4.5 Minimizing an staDFA

The deterministic finite automata (*DFA*) minimization is an important aspect, as it ensures the minimal storage requirements while keeping the overall pattern matching the same. Minimization of an *staDFA* is performed in two steps:

In the first step, we remove the unnecessary commands operations in the compilation functions $\mathcal{C}$ and $\bar{\mathcal{C}}$ associated with each state $s \in S$ of an *staDFA*. More specifically, during the construction of an *staDFA* process at each state $s \in S$, we can have commands like idempotent transfers $T_{i,i}^t$, useless accept marked positions A_i^t, and etc. Thus, by removing these worthless commands generated by the compilation functions such as $\mathcal{C}$ and $\bar{\mathcal{C}}$ associated with each state $s \in S$ of an *staDFA*, we could have used less memory which refers to having less space complexity, too. However, we have to be very careful about removing these useless commands. Because removing of essential commands

could lead us to have significant changes on *staDFA* conflict resolution policies, which in turn could result in mixed leftmost/rightmost behaviors.

In the second step, I have chosen a well-known *DFA* minimization algorithm known as Hopcroft's minimization proposed in [14]. This minimization algorithm is perfect for the *staDFA* Matcher whereas is imperfect/inapplicable to the *TNFA-TDFA* in [15]. Because, command operations are associated with states $s \in S$ of an *staDFA* Match which means used markers t_i are *state-based*. Whereas the used tags t_x on *TNFA-TDFA* in [15] are *edge-based*. Therefore, applying of Hopcroft's minimization algorithm in *staDFA* Matcher is easy and simple while keeping the overall pattern matching strategy same. Nevertheless, we have to be very careful about applying Hopcroft's minimization algorithm in staDFA Matcher. Because, two states in staDFA will be considered as equivalent if and only if they are both non-accepting states and have the same set of commands. So, we should not merge states that have the different set of commands. For this, we need to make a minor modification to the Hopcroft's minimization algorithm.

Thus, the modified version of the Hopcroft's minimization algorithm has two parts. In the first part, we are removing all the unreachable states $\varnothing$ from the staDFA. Secondly, we applied the Hopcroft's minimization algorithm while keeping the fact intact which is states are equivalent if and only if they are both non-accepting states and have the same set of commands. For example, the *staDFA* representation shown in Fig. 4.1 for *RE* $R = (\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^* \,{}^1\!/\,\mathtt{b_4}\,\mathtt{a_5}$ is the minimal *staDFA* and it has seven states. Note that, it is not the minimal *DFA*. If we apply the unmodified Hopcroft's minimization algorithm, then we could have six states instead of seven states for the above *RE* R. But, the principle objective of *staDFA* Matcher defined in Algorithm 1 will fail, which is matching the *sub-expression* positions and telling its extent on the accepted input strings.

4.6 Applications

In this section, I will show some real world applications of *sub-expression* matching using staDFA. This mainly includes *trailing contexts*, *lookaheads*, *capturing groups*, and *back-reference*. The detailed explanation of each category is given below:

4.6.1 Trailing Contexts

Trailing Contexts refers to matching a *sub-expression iff* it is followed by other particular *sub-expression* which refers to end of the overall *RE*. For an example, $R = R_1/R_2$ known as trailing

context because, here R_1 is followed by R_2 which refers to end of the overall $RE\ R$. More specifically, let's we have a $RE\ R = (\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*\,\mathtt{b_4\,a_5}$ where $R_1 = (\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*$ and $R_2 = \mathtt{b_4\,a_5}$. Now, according to the concept of trailing context *sub-expression* R_1 will be matched as long as it's followed by R_2. However, Lex/Flex fail to implement the trailing contexts correctly [4, 15].

However, using *staDFA* it is simple to implement the trailing contexts correctly. For an example, if we mark the above $RE\ R$ using a right marker $^t\!/$, then it will be as follows: $R = (\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*\,{}^1\!/\,\mathtt{b_4\,a_5}$ where, matching the *sub-expression* $R_1 = (\mathtt{a_1} \mid \mathtt{a_2\,b_3})^*$ positions followed by *sub-expression* $R_2 = \mathtt{b_4\,a_5}$ on the accepted input strings is pretty simple and straightforward. Fig. 4.1 shows the staDFA representation of the above $RE\ R$. Also, Table 4.1 shows the steps of *sub-expression* R_1 matching positions and it's extent to *sub-expression* R_2 on the accepted input strings. So, it clearly shows that, we don't have any limitation to implement the trailing contexts correctly with an staDFA.

4.6.2 Lookaheads

Lookaheads are interesting. Because, the concept is more generic than trailing contexts. Lookaheads can be used anywhere in regular expressions, whereas trailing context can only be used to refer to end of the overall regular expression. For an example, the lookaheads in $RE\ R = \ldots, (\mathtt{?=}R_1)\,R_2, \ldots$ does not refer to end of the overall regular expression R. However, this lookaheads can also be implemented accurately using staDFA. Because, in staDFA Matcher we can place marker t_i at any positions in the $RE\ R$ which directly resembles having the lookaheads at any positions in *RE*.

Let's consider a simple $RE\ (\mathtt{?=}R_1)\,R_2$ with lookaheads. Using *staDFA* it's very simple to implement this lookahead. First, we need to mark the lookaheads in the *RE* as follows: $(\mathtt{?=}\,{}^t\!/R_1)\,R_2$. Second, applying the staDFA Matcher algorithm in the *RE*. At this point, we have to keep in mind that, when matching *sub-expression* R_1 ends and we have a transition to a_i to the state with $a_i \in \mathit{firstpos}(R_2)$, we must execute the backup command say B_i^t which does this $k \leftarrow m_i^t$. That is setting up the cursor k back to the start positions of *sub-expression* R_1 before continuing the macthing of *sub-expression* R_2.

Therefore, we can say that, as staDFA allows flexibility of executing user defined compilation functions such as $\mathcal{C}$, $\bar{\mathcal{C}}$ which contains $S_i^t(store)$, $\bar{S}_i^t(delayedstore)$, $T_{i,j}^t(transfer)$, $B_i^t(backup)$, $A_i^t(accept)$, and etc. Thus, we can have any kind of desired *sub-expression* positions matching and

its extent in the *RE*. Hence, lookaheads is just one kind of example which can be even implemented so easily and accurately.

4.6.3 Capturing Groups

Regular expression captured by parentheses is know as capturing groups. These are very often used in pattern matching. In capturing groups, the captured *RE* can have all kinds of quantifier such as greedy, non-greedy operators. In addition to that, this captured *RE* can be ambiguous. Also, in repetition a captured *RE* can match the first or last sub-strings $w_i \in w[0 : \ell - 1]$.

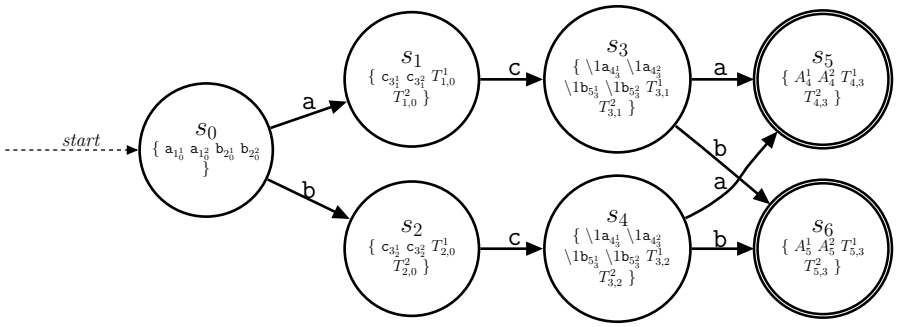
Therefore, special care has to be taken on capturing groups in *RE R* using the staDFA Matcher. Suppose, we have a *RE R* $= (\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*\,\mathtt{b_4}$ where, it has a capturing group $R_1 = (\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*$. Note, this R_1 is ambiguous, and in repetition it can match the first or last sub-strings $w_i \in w[0 : \ell - 1]$. That's why to unambiguously match the capturing groups, we aleady defined the meta operators in Section 4.1.2. Hence, the exact representation of the above *RE R* will be as follows: $R = {}^{1}\!/\!_{f}\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*\,{}_{l}\!\backslash^{2}\,\mathtt{b_4}$, where *sub-expression* $R_1 = {}^{1}\!/\!_{f}\,(\mathtt{a_1} \mid \mathtt{a_2}\,\mathtt{b_3})^*\,{}_{l}\!\backslash^{2}$ is marked by using markers t_1 and t_2, respectively.

Fig. 4.2 shows the *staDFA* representation of the above *RE R*. Also, Table 4.3 shows the step by step processing of input strings symbol, and at the same time, shows the *MPS* operations associated with states of the *staDFA* match. Finally, we see that our proposed staDFA unambiguously finds the starting position $M[t_1]$ and ending position $M[t_2]$ of *sub-expression* R_1 in an accepted input strings.

4.6.4 Back-References

Nowadays, back-references is an interesting and promising research topic for the computer science theorists. In the past, to support back-references we had to depend on backtracking which in turns lead us to choose *NFA* representation. Thus, we may have exponential time complexity in the worst-case scenario and an *NP-complete* problem [9]. However, we have made an important contribution to this research topic. More simply, our proposed staDFA Matching Algorithm support most forms of Back-References.

Back-References means that a repetition of the already captured groups in the later positions of the same *RE R*. These captured groups are formed by parentheses. Therefore, a back-reference $\backslash n$ in a *RE R* means that, reappearing of the n-th numbered capturing group in the later positions



State s	*Commands* C
s_0	$\varnothing$
s_1	$\{T^1_{1,0}, T^2_{1,0}\}$
s_2	$\{T^1_{2,0}, T^2_{2,0}\}$

State s	*Commands* C
s_3	$\{T^1_{3,1}, T^2_{3,1}\}$
s_4	$\{T^1_{3,2}, T^2_{3,2}\}$

State s	*Commands* C
s_5	$\{T^1_{4,3}, T^2_{4,3}, A^1_4, A^2_4\}$
s_6	$\{T^1_{5,3}, T^2_{5,3}, A^1_5, A^2_5\}$

Figure 4.4: An staDFA state diagram of $R = (\mathtt{a_1} \mid \mathtt{b_2})\,\mathtt{c_3} \setminus 1$ and commands C for states s.

of the same *RE* R. For example, if we have a *RE* $R = (\mathtt{a_1} \mid \mathtt{b_2})\,\mathtt{c_3} \setminus 1$, then this $\setminus 1$ refers to the 1^{st} capturing group sub-expression $R_1 = (\mathtt{a_1} \mid \mathtt{b_2})$ of *RE* R.

However, in order to support back-references in *staDFA*, we have to formulate the *RE* R which contains the back-reference. Let's consider the above *RE* $R = (\mathtt{a_1} \mid \mathtt{b_2})\,\mathtt{c_3} \setminus 1$. Now, in the first step, we need to mark the *sub-expression* of capture group n as follows: $^{2n-1}\!/$ and $\backslash^{2n}$. That is, in case of above *RE* R, it will be like $^{1}\!/(\mathtt{a_1} \mid \mathtt{b_2})\backslash^{2}$. However, to support the above *RE* R using the *staDFA* Match the complete modified version will be as follows: $^{1}\!/(\mathtt{a_1} \mid \mathtt{b_2})\backslash^{2}\,\mathtt{c_3}\,\underline{(\setminus 1\mathtt{a}_4 \mid \setminus 1\mathtt{b}_5)}$ with the replacement of $\setminus 1$ underlined. The resulting staDFA for the above *RE* R is shown in Fig. 4.4. Now, to match the input strings say "aca" based on the above *RE* R containing back reference $\setminus 1$, the *staDFA* Match will continue normal execution until it reaches to the states either s_3 or s_4. At this point, we already matched the $n = 1^{st}$ capturing group marked by $^{2*1-1}\!/$ and $\backslash^{2*1}$ as follows: $^{1}\!/(\mathtt{a_1} \mid \mathtt{b_2})\backslash^{2}$. That is, $w[m^1_3 : m^2_3]$(i.e. in this case $w[0:0]$) contains the matched portion of the captured group $^{1}\!/(\mathtt{a_1} \mid \mathtt{b_2})\backslash^{2}$. Now, to handle the back-reference $\setminus 1$ from either states s_3 or s_4, where these states contain $\setminus 1\mathtt{a}_{4^1_3}$, $\setminus 1\mathtt{a}_{4^2_3}$, $\setminus 1\mathtt{b}_{5^1_3}$, and $\setminus 1\mathtt{b}_{5^2_3}$, we need to add a constraint on the *staDFA* Match Algorithm. That's match the $w[0:0]$ against next part of the input strings which is $w[2:2]$, if it matches, then continue normal execution as usual specified in Algorithm 1. Otherwise, set the state $s = \varnothing$ to a dead state to exit the loop from *staDFA* Matcher in Algorithm 1.

Therefore, we see that to support back-reference $\backslash n$ on *staDFA* Match, it adds a simple constraint which can be easily verified. Also, another advantage of staDFA Matcher is that, for a *RE* R if $\backslash n$ does not participate on matching input strings w, but current state $s \in S$ contains other non-$\backslash n$-marked positions, then staDFA Matcher will still continue normal execution until it reaches to the accepting states. For an example, *RE* $R = (\mathtt{a}_1 \mid \mathtt{b}_2)\,\mathtt{c}_3\,(\backslash 1 \mid \mathtt{d}_4)$ matches the input strings say "`acd`" without matching the back-reference $\backslash 1$.

CHAPTER 5

PERFORMANCE EVALUATION

My advisor Prof. Van Engelen implemented a staDFA prototype in SWI-Prolog available for download from `www.cs.fsu.edu/~engelen/stadfa.zip`. This implementation was used to verify the algorithms and to generate the figures in this thesis. In this chapter, I will present the performance evaluation of staDFA using RE/flex [34] and compare the performance to some other mostly used *RE* matcher such as RE2 [9], PCRE2 [13], Lex [20], Boost.Regex [23] and Flex [26].

5.1 Experimental Setup

I compared the performance of a C++ prototype implementation of staDFA in RE/flex [34] to the *DFA*-based scanners Flex [26], RE/flex [34] and *NFA*-based C/C++ regex libraries PCRE2 [13], RE2 [9], and Boost.Regex [23]. We optimized PCRE2 and RE2 by pre-compiling the *RE* patterns before matching, such that the performance measurements do not include the construction and deletion times of patterns. I picked the best performance of > 10 runs with each run executing 500 iterations to average the running time. The test programs were compiled with clang 8.0.0 with -O2 and run on a 2.9 GHz Intel Core i7, 16 GB 2133 MHz LPDDR3 machine.

5.2 Performance Evaluation of staDFA

Fig. 5.1 shows the performance of staDFA used as a scanner (a.k.a. lexical analyzer) implemented as a prototype in RE/flex using patterns `b` and `a b*(?=b a*)`, where the latter pattern uses a lookahead (Flex trailing context `a b*/b a*`) to tokenize input strings, compared to the performance of Flex, RE/flex, PCRE2, RE2 and Boost.Regex. RE2 does not support lookaheads. To include RE2, I used `(a b*) b a* | b` and advanced the string location using the end of the captured group by invoking `RE::Match()`. To emulate the correct lookahead in Flex and RE/flex, I used `yyless(n)` to limit the size of the resulting match using the specific value of `n` for each input string. The results in Fig. 5.1 show that staDFA matching with a lookahead is about as fast as Flex and RE/flex. Note that the *MPS* adds negligible overhead, by comparing the performance of staDFA in RE/flex

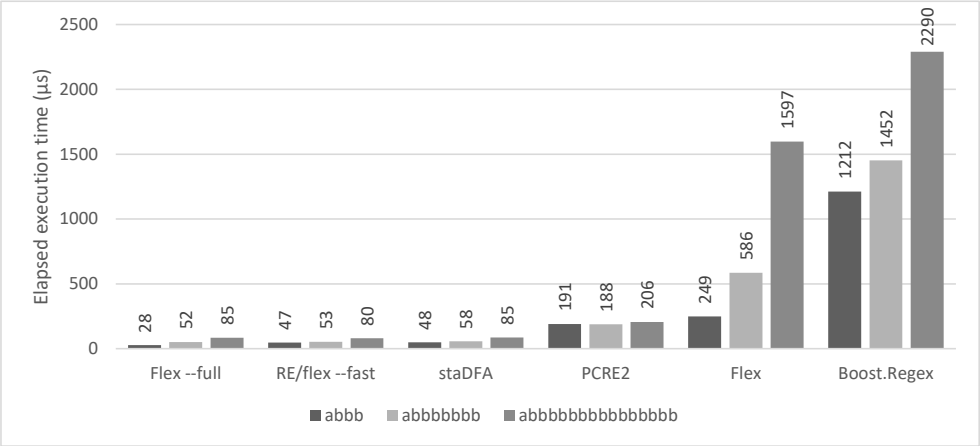


Figure 5.1: Performance of staDFA matching, compared to Flex, RE/flex, PCRE2, RE2 and Boost.Regex for tokenizing strings containing 1,000 copies of `abbb`, `abbbbbbb` and `abbbbbbbbbbbbbbb` tokenized into 2,000 tokens using two patterns `b` and `a b`*(?=`b a`*) with a lookahead (Flex trailing context `a b`*/`b a`*). Elapsed execution time is shown in micro seconds.

to the bare-bones RE/flex *DFA* matcher. Fig. 5.2 shows the performance of staDFA compared to RE2, PCRE2, and Boost.Regex for a regular expression (`a b`)* `c` with group capture (`ab`) on input strings (`ab`)^n`c` where, $n = 10, 100, \ldots, 1000000$. Boost.Regex fails with a stack error for $n > 10000$. RE2 is designed to perform very well with a regex such as (`a b`)* `c`, but performs poorly (10 times slower or worse) for regex forms with alternations, such as (`a b`)* `c` | `a`, when the other libraries and staDFA perform about the same for this regex. Comparing the best performance of RE2 and PCRE2 with staDFA, the results of Fig. 5.2 show that staDFA has the best performance.

Fig. 5.3 shows the *MPS* operations overhead of staDFA matching, compared to, tags overhead of *TDFA* matching for regex $R_1 = (\mathtt{b_1}^*)\,(\mathtt{b_2}^*)\,\mathtt{b_3}$ with average-case, and regex $R_2 = (\,\mathtt{b_1} \mid \mathtt{b_2\,b_3} \mid \mathtt{b_4\,b_5\,b_6} \mid \mathtt{b_7\,b_8\,b_9\,b_{10}})^*$ with worst-case. More particularly, the overhead calculation for average-case, the regex $R_1 = (\mathtt{b_1}^*)\,(\mathtt{b_2}^*)\,\mathtt{b_3}$ has been evaluated for sub regex matching $R_1{}' = (\mathtt{b_1}^*)$ for the input string "`bbbb`" using staDFA with marker $\mathtt{t_1} = \mathtt{1}$ and using *TDFA* with tag $\mathtt{t_1}$. Results

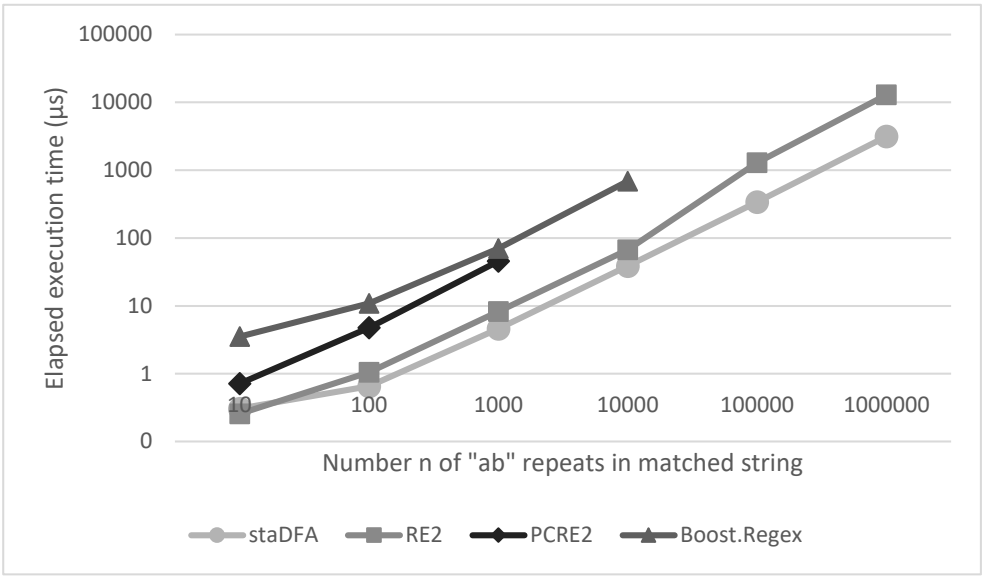


Figure 5.2: Performance of staDFA matching, compared to RE2, PCRE2 and Boost.Regex for regular expression $(\mathtt{a\,b})^*\,\mathtt{c}$ with group capture $(\mathtt{ab})$. Elapsed execution time per string match is shown in micro seconds for a range of input strings $(\mathtt{ab})^n\mathtt{c}$ where, $n = 10, 100, \ldots, 1000000$.

show that, the overall marker updates require 12 times for staDFA matching whereas tag upadates require 20 times. In worst-case scenario, for the same given input string "bbbb" the regex R_2 for *TDFA* needed to be rewriiten as $R_2 = (\,\mathtt{tag1\,b_1} \mid \mathtt{tag2\,b_2\,b_3} \mid \mathtt{tag3\,b_4\,b_5\,b_6} \mid \mathtt{tag4\,b_7\,b_8\,b_9\,b_{10}})^*$. Since, we are interested to see the overaheads during sub-matching for the input string "bbbb", hence, tag4 must give the extent of it also at the same time must give the result such that "bbb" of "bbbb" matched the extent of tag3, same as "bb" of "bbbb" matched the extent of tag2 and, finally "b" of "bbbb" matched the extent of tag1. However, if we simply depend on only using one tag $\mathtt{t_1}$ for the *TDFA* then, it will completely fail and would not give us the accurate results. Thus, we must use four tags whereas using staDFA, we only need just one marker $\mathtt{t_1} = [1, 2, 4, 7]$ since a single marker in staDFA can be used in several positions within a *RE*. On the other hand, a single tag in *TDFA* can not be used in multiple positions in a *RE*. For the worst-case scenario overall overhead introduced by *TDFA* is 28 times okay using four tags whereas staDFA requires marker updates of 19 times with using only one marker.

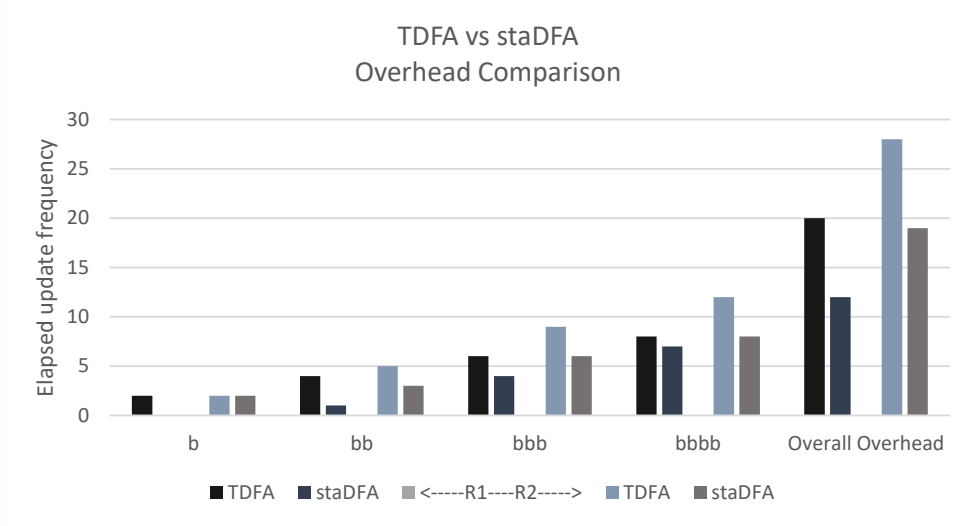


Figure 5.3: *MPS* operations overhead of staDFA matching, compared to, tags overhead of *TDFA* matching for regex $R_1 = (\mathtt{b}^*)\,(\mathtt{b}^*)\,\mathtt{b}$ with average-case and regex $R_2 = (\,\mathtt{b} \,|\, \mathtt{bb} \,|\, \mathtt{bbb} \,|\, \mathtt{bbbb})^*$ with worst-case. Elapsed update frequency per string match is shown in numerical counts for a range of input strings $(\mathtt{b})^n$ where, $n = 1, 2, \ldots, 4$.

5.3 Discussion

Future investigation will address how the *MPS* can be effectively optimized by reusing memory cells to reduce memory resource requirements. One could use graph coloring for register allocation [8] to reduce the number of memory cells required by reusing memory cells. One could alternatively construct the *static single assignment* (*SSA*) form of program code [10] by treating an staDFA as a control-flow graph with basic blocks as states s with `C` commands and use *SSA* algorithms for register allocation [12] to reuse memory cells.

CHAPTER 6

CONCLUSIONS

Throughout this thesis, I explained how *staDFA* is constructed and can be used for sub-expression matching. I described how each category of regular expressions as defined in POSIX.2 can be represented by the proposed *staDFA*. I compared *staDFA* matching to other existing regular expression matchers such as RE2 [9], PCRE2 [13], Lex [20], Boost.Regex [23], Flex [26] and RE/flex [34]. In addition, a modified Hopcroft algorithm can be used to minimize *staDFA*.

APPENDIX A

THEOREMS AND PROOFS

Theorem 1. *Given a regular expression R and a finite number n of markers $t = 1, \ldots, ||R||$,* staDFA(R) *terminates with a finite set of states.*

Proof. Each state $s \in S$ of the staDFA(R) is a set of marked or unmarked positions and commands. A state is uniquely identified by the content of its set. Every subscript index i and j in marked positions a_i and commands S_i^t, $T_{i,j}^t$ and A_i^t are bounded from below by 0 and bounded from above by $||R||$, where $||R||$ is the alphabetic width (length) of regular expression R. This gives a finite number of possibilities for the content of the state's sets, and therefore staDFA(R) terminates with a finite set of states S. □

Theorem 2. *Strings $w \in \Sigma^*$ are position-wise unambiguously accepted by an* staDFA(R) *if all states $s \in S$ are conflict-free.*

Proof. String w is position-wise unambiguously accepted by an staDFA(R) if for all $t = 1, \ldots, n$ marker position $M[t]$ identifies a unique position in the string w or is -1. By the absence of store-transfer and transfer-transfer conflicts, every m_i^t is assigned a well-defined unique value by the S_i^t and $T_{i,j}^t$ commands in `C`. Finally, by the absence of accept-accept conflicts, $M[t]$ is assigned a well-defined unique value m_i^t or -1 for all markers $t = 1, \ldots, n$. □

Theorem 3. *The mark-first operator is right distributive over alternation ${}^t\!/(R_1 \mid R_2) = {}^t\!/R_1 \mid {}^t\!/R_2$ and Kleene closure ${}^t\!/({R_1}^*) = ({}^t\!/R_1)^*$, for any marker t. The mark-last operator is left distributive over alternation $(R_1 \mid R_2)\backslash^t = R_1\backslash^t \mid R_2\backslash^t$ and Kleene closure $({R_1}^*)\backslash^t = (R_1\backslash^t)^*$, for any marker t.*

Proof. We have that $M^t({}^t\!/(R_1 \mid R_2)) = M^t(R_1 \mid R_2) \cup \mathit{firstpos}(R_1 \mid R_2) = M^t(R_1) \cup M^t(R_2) \cup \mathit{firstpos}(R_1) \cup \mathit{firstpos}(R_2)$ and $M^t({}^t\!/R_1 \mid {}^t\!/R_2)) = M^t({}^t\!/R_1) \cup M^t({}^t\!/R_2) = M^t(R_1) \cup M^t(R_2) \cup \mathit{firstpos}(R_1) \cup \mathit{firstpos}(R_2)$. Hence, $M^t({}^t\!/(R_1 \mid R_2)) = M^t({}^t\!/R_1 \mid {}^t\!/R_2))$. Likewise, $M^t({}^t\!/({R_1}^*)) = M^t({R_1}^*) \cup \mathit{firstpos}({R_1}^*) = M^t(R_1) \cup \mathit{firstpos}(R_1)$ and $M^t(({}^t\!/R_1)^*) = M^t({}^t\!/R_1) = M^t(R_1) \cup \mathit{firstpos}(R_1)$. Hence, ${}^t\!/({R_1}^*) = ({}^t\!/R_1)^*$. The proofs are similar for the mark-last operator. □

REFERENCES

[1] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, Reading MA, 1985.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[3] V. M. Antimirov. *Partial derivatives of regular expressions and finite automaton constructions.* Theoretical Computer Science, 155(2):291-319, 1996.

[4] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1999.

[5] M. Becchi, and P. Crowley. *Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions.* Proceedings of the 2008 ACM CoNEXT Conference. ACM, 2008.

[6] J. Brzozowski. *Derivatives of Regular Expressions.* Journal of the Association for Computing Machinery, Vol. 11, No. 4, Pages: 481-494. October 1964.

[7] M. Rabin, and D. Scott. *Finite automata and their decision problems.* IBM Journal of Research and Development, Vol. 3, Issue. 2, Pages: 114-125. April 1959.

[8] D. Callahan and B. Koblenz. *Register Allocation via Hierarchical Graph Coloring.* In ACM SIGPLAN Notices, Vol. 26, No. 6, Pages: 192-203. ACM, 1991.

[9] R. Cox. *Regular Expression Matching can be Simple and Fast*, 2007. URL: `http://swtch.com/~rsc/regexp/regexp1.html`.

[10] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.* ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 13, No. 4, Pages: 451-490, Year: 1991.

[11] Institute of Electrical and Electronics Engineers (IEEE): Standard for information technology. *Portable Operating System Interface (POSIX) Part 2 (Shell and utilities).* Section 2.8 (Regular expression notation), New York, IEEE Standard 1003.2 (1992).

[12] S. Hack, D. Grund, and G. Goos. *Register Allocation for Programs in SSA-Form.* In Proceedings of the ETAPS CC Conference, (2006): 247-262.

[13] PCRE. *Perl Compatible Regular Expressions.* URL: `http://www.pcre.org/`.

[14] J. Hopcroft. *An $\mathcal{O}(n \log n)$ Algorithm for Minimizing States in a Finite Automaton.* Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971). New York: Academic Press, Pages: 189-196, 1971.

[15] V. Laurikari. *NFAs with Tagged Transitions, Their Conversion to Deterministic Automata and Application to Regular Expressions.* IEEE Xplore Conference: String Processing and Information Retrieval (SPIRE), Pages: 181-187, February 2000.

[16] Ulya Trofimovich. *Tagged Deterministic Finite Automata with Lookahead* `http://re2c.org/2017_trofimovich_tagged_deterministic_finite_automata_with_lookahead.pdf`, 2017.

[17] S. Kumar et al. *Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia.* In Proc. of ANCS, 2007.

[18] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. *Algorithms to accelerate multiple regular expressions matching for deep packet inspection.* In Proc. of SIGCOMM '06, Pages: 339-350. ACM, 2006.

[19] D.Ficara, S.Giordano, G. Procissi, F.Vitucci, G.Antichi, A.D. Pietro. *An Improved DFA for Fast Regular Expression Matching.* ACM SIGCOMM Computer Communication Review, Volume 38, Number 5, October 2008.

[20] M. Lesk. *Lex - a Lexical Analyzer Generator.* TR-39, Bell Laboratories, Murray Hill, NJ, 1975.

[21] H. R. Lewis and C. H. Papadimitrou. *Elements of the Theory of Computation.* Prentice Hall, 1981.

[22] I. Nakata and M. Sassa. *Regular Expressions with Semantic Rules and their Application to Data Structure Directed Programs.* Advances in Software Science and Technology, 3:93-108, 1991.

[23] Boost C++ libraries. *Boost.Regex.* `http://www.boost.org/doc/libs/1_65_1/libs/regex/doc/html/index.html`

[24] Perl. *perlre.* `https://www.perl.org/`

[25] I. Nakata. *Generation of pattern-matching algorithms by extended regular expressions.* Advances in Software Science and Technology, 5:1-9, 1993.

[26] V. Paxson. *Flex - Fast Lexical Analyzer Generator.* Lawrence Berkeley Laboratory, Berkeley, California, 1995.

[27] G. A. Stephen. *String Searching Algorithms volume 3 of Lecture Notes Series on Computing.* World Scientific Publishing, 1994.

[28] R. A. Baeza-Yates and G. H. Gonnet. *Efficient text searching of regular expressions.* In Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Volume:372 of LNCS, Pages: 4662, Berlin, July 1989. Springer.

[29] R. A. Baeza-Yates and G. H. Gonnet. *Fast text searching for regular expressions or automaton searching on tries.* ACM, 43(6):915936, November, 1996.

[30] B. Watson. *Implementing and using finite automata toolkits.* Journal of Natural Language Engineering, 2(4):295302, December, 1996.

[31] B. Watson. *A new regular grammar pattern matching algorithm.* In Proceedings of the European Symposium on Algorithms, Volume: 1136 of LNCS, Pages: 364377, Berlin, September, 1996. Springer.

[32] M. Sulzmann and K.Z.M. Lu. *Regular Expression Sub-Matching using Partial Derivatives.* In Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, Pages: 79-90, ACM, 2012.

[33] L.Ilie,and S.Yu. *Follow automata.* Information and Computation, 186(1):140-162, 2003.

[34] Robert A. van Engelen. *Constructing Fast Lexical Analyzers with RE/flex* `https://www.codeproject.com/Articles/1180430/Constructing-fast-lexical-analyzers-with-RE-flex-w`, 2017.

[35] Pedro Garca, Damin Lpez, Jos Ruiz, and Gloria I. lvarez. *From regular expressions to smaller NFAs.* Theoretical Computer Science, Elsevier, 412:5802-5807, 2011.

[36] Satoshi Okui ,and Taro Suzuki. *Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions* . Language Processing Systems Laboratory, University of Aizu, Japan, June 5, 2013.

[37] Angelo Borsotti, Luca Breveglieri, Stefano Crespi Reghizzi, and Angelo Morzenti. *From Ambiguous Regular Expressions to Deterministic Parsing Automata* . International Conference on Implementation and Application of Automata(CIAA), Sweden, August, 2015.

[38] Martin Sulzmann, and Kenny Zhuo Ming Lu. *POSIX Regular Expression Parsing with Derivatives* . International Symposium on Functional and Logic Programming(FLOPS), June, 2014.

[39] Bjrn Bugge Grathwohl, Fritz Henglein, Ulrik Terp Rasmussen, Kristoffer Aalund Sholm, Sebastian Paaske Trholm, and Kenny Zhuo Ming Lu. *Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers* . ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL), St. Petersburg, FL, USA. Volume: 51, Issue: 1, Pages: 284-297, January'16.

[40] Thomas William Reps *Maximal-Munch Tokenization in Linear Time* . ACM Transactions on Programming Languages and Systems (TOPLAS), NY, USA. Volume: 20, Issue: 2, Pages: 259-273, March'98.

[41] Schwarz N, Karper A, and Nierstrasz O. *Efficiently extracting full parse trees using regular expressions with capture groups* . PeerJ PrePrints 3:e1248v1. `https://doi.org/10.7287/peerj.preprints.1248v1`, (2015).

BIOGRAPHICAL SKETCH

Mohammad Imran Chowdhury was born in Lakshmipur, Bangladesh. He graduated with a B.Sc. in Computer Science & Engineering (CSE) from Chittagong University of Engineering and Technology (CUET). Immediately after his graduation, he joined as a Lecturer in the Department of Computer Science at Premier University, Chittagong, Bangladesh. There, he worked for almost 3 years, after that he came to U.S.A. to pursue a Ph.D. degree in the Department of Computer Science at Florida State University (FSU). His research interests mainly focuses on compilers, distributed computing, program analysis & synthesis method, and Bayesian probabilistic networks. Conquest-2, Performance of optimistic peer replication, AOT: Ahead-of-Time Compilation, Real-time storage domain, Compilation Tools, Embedded Systems, Computer Networks, Internet protocols over satellite networks, Wireless networks and optical communications, Network security, BGP routing, ad-hoc networks, and power-laws of Internet topology. Artificial Intelligence, Machine Learning.