GUANGMING XING
Generating NFA for Efficient Pattern Matching
(Under the direction of E. RODNEY CANFIELD)

Research in regular languages and their associated computational problems has been revitalized by the rapid development of the Internet and its applications. The construction of finite automata from regular expressions is of central importance to a variety of practical problems, including DFA construction in Unix systems, lexical scanning, Internet searching, content-based network service, and computational biology.

First, a "smart" parsing algorithm is developed which constructs a parse tree with at most $(3l - 1)$ nodes from a regular expression with l literals. Based on smart parsing, two NFA construction algorithms are presented. The first one works on the NFA from Thompson's construction, eliminating as many auxiliary states as possible while maintaining Thompson's properties. It is shown that the resulting NFA is a minimized Thompson's NFA, which means that no auxiliary states can be eliminated without violating the defining properties of Thompson NFA.

The second construction method is based on smart parsing and "divide and conquer." From a regular expression with l literals, we construct a normalized NFA with $2l$ states and $4l$ transitions in the worst case.

Finally, it is shown that the Emptiness-of-complement problem for semi-extended regular expressions is $EXPSPACE$-complete. If this problem could be solved in $EXPTIME$, it would lead to the first known equivalence between a time complexity class and a space complexity class.

INDEX WORDS: Algorithm, Regular Expression, Parse Tree, Finite Automaton, Pattern Matching, Space Complexity

Generating NFA for Efficient Pattern Matching

by

Guangming Xing

B.S., Nankai University, 1996

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Athens, Georgia

2001

GENERATING NFA FOR EFFICIENT PATTERN MATCHING

by

GUANGMING XING

Approved:

Major Professor:	E. Rodney Canfield
Committee:	Robert Robinson
	John Miller
	Thiab Taha
	Mitchell Rothstein

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
ember 2001

Acknowledgments

First of all, I wish to express my profound gratitude to my advisor, Dr. E. Rodney Canfield. Without his encouragement, support, and patient guidance, most of this would not have been possible. He has always been available and enthusiastic to help when I need it during the last four years. His insightful advice laid the foundation of my work.

I am very grateful to Dr. Robert Robinson and Dr. John Miller for their continuous advice and support throughout my thesis research and serving on my Ph.D. committee. I would also like to thank Dr. Thiab Taha and Dr. Mitchell Rothstein, my thesis committee members, for their careful reading and suggestions.

Part of this research was supported by an Internship with LocalDirector Group, Cisco Systems, Inc. I would like to thank all my colleagues for their suggestions and encouragement. In particular, I would like thank Curt Kersey, Bruce Wong, and Justin Pecequeue for their discussion and comments.

My biggest gratitude goes to my family. My wife Yan Huang gave me a lot of support and encouragement. I thank my sister Guangwei and my brothers Guangzhen and Guangju. This dissertation is dedicated to my parents, for their love and encouragement that I can always count on.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction and Literature Review

1.1 Background

Research in regular languages and their associated computational problems has been revitalized by the rapid development of the Internet and its applications. In particular, the construction of finite automata from regular expressions is of central importance to the interprocess communication [11], string pattern matching [3, 12], approximate string pattern matching [9], lexical scanning [1], content-based network service [15], regular expression compilation in VLSI layout design [21], computational biology [12], and DFA construction from regular expressions such as RegEx implemented in the UNIX operating system [16].

Using regular expressions for pattern searching is widely known and well understood. It is regarded as a precise, succinct way to specify patterns of interest. However, as a computation model, the NFA (nondeterministic finite automaton) is a more useful tool for pattern matching. Simulation of NFAs for pattern matching is a basic method used for text searching. So the construction of finite automata from regular expressions is of central importance.

1.2 Terminology

We follow the same notations as used in [1]. By an *alphabet*, we mean a finite non-empty set of symbols. In this thesis, we use Σ to denote an alphabet. If Σ is an alphabet, Σ^*

denotes the set of all finite strings of symbols in Σ. The empty string is denoted by ϵ. Any subset of Σ^* is a *language* over Σ.

Definition 1 A *regular expression* over an alphabet Σ is defined as follows [1]:

1. ϵ, ϕ and a for each $a \in \Sigma$ are regular expressions denoting the regular language $\{\epsilon\}$, the empty set and $\{a\}$ respectively;
2. If R_1, R_2 are regular expressions denoting the languages L_1, L_2, respectively, then $(R_1 + R_2)$, $(R_1 R_2)$ and (R_1^*) are regular expressions, denoting $L_1 \cup L_2$, $L_1 L_2$ and L_1^*, which we call alternation, concatenation, and star, respectively;
3. All regular expressions can be defined by the above rules.

We use $L(R)$ to denote the language denoted by a regular expression R.

We define the length of a regular expression to be the number of occurrences of characters, ϵ, ϕ and the above three operations. This is slightly different from the definition used elsewhere [1, 7] (In their definition, the concatenation operation is not counted in the length of the regular expression.) Observe that the length we use is at most double the length used in [1, 7] for every regular expression.

For each character occurrence, we call it a *literal*. The number of literals in a regular expression means the number of character occurrences. Because a character set (alternation between characters as in $a + b + c$) behaves like a single character, we call it a literal also.

Regular expressions have been used in a variety of practical applications to specify regular languages, which will be covered later in this chapter. The problem of deciding whether a given string belongs to the language denoted by a particular regular expression can be implemented efficiently using finite automata which are now defined.

Definition 2 A *nondeterministic finite automaton* (NFA for short) N is defined as a 5-tuple

$$(S, \Sigma, \delta, s_0, F),$$

where

1. S is the finite set of states of the control;
2. Σ is the alphabet from which input symbols are chosen;
3. δ is the *state transition function* which maps $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of S;
4. s_0 in S is the *initial state* of the finite control;
5. $F \subseteq S$ is the set of *final (or accepting) states*.

For all $q \in S$ and $a \in \Sigma$, $\delta(q, a)$ is the set of all states reachable from state q by any single edge labeled a. The label a can be any alphabet character in Σ or ϵ.

We can extend the transition function to a larger domain, i.e. extend δ to $\hat{\delta} : \mathcal{P}(S) \times (\Sigma \cup \{\epsilon\})^* \to \mathcal{P}(S)$.

First, for $T \subseteq S$, we define ϵ-closure(T) as the set of states reachable from some states of T by 0 or more ϵ-transitions.

Then, for $T \subseteq S$ and $w \in \Sigma^*$ we define $\hat{\delta}$ recursively as:

$$\hat{\delta}(\phi, w) = \phi$$

$$\hat{\delta}(T, \epsilon) = \epsilon\text{-closure}(T)$$

$$\hat{\delta}(q, cw) = \hat{\delta}(\hat{\delta}(\delta(q, c), \epsilon), w)$$

$$\hat{\delta}(T, w) = \cup_{q \in T} \hat{\delta}(q, w)$$

A language accepted by M, denoted by L_M, is defined as

$$L_M := \{w \in \Sigma^* \mid \hat{\delta}(\{s_0\}, w) \cap F \neq \phi\}$$

Definition 3 We call an NFA a normalized NFA (NNFA for short) if

1. It has a unique start state and a unique final state;
2. The start state does not have any in-transitions, and the final state does not have any out-transitions.

Definition 4 An NFA M is a *deterministic finite automaton* (DFA for short) if there are no ϵ-transitions and $\mid \delta(q, a) \mid \leq 1$ for all $q \in S$ and $a \in \Sigma$.

Clearly, every DFA is an NFA, and it is shown in [2], for each NFA N there is a DFA D, such that $L(D) = L(N)$. The construction of D from N will be covered in later sections.

Throughout this thesis our model of computation is a uniform cost sequential RAM which is used in [1].

In our algorithmic analysis, we use O, Θ, and Ω notations. We say $f(n) = O(g(n))$, if there are positive constants c and n_0, s.t. for all $n \geq n_0$, $f(n) \leq cg(n)$. Similarly, we say $f(n) = \Theta(g(n))$, if there are positive constants c_1, c_2, and n_0, s.t. for all $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$; $f(n) = \Omega(g(n))$, if there are positive constants c and n_0, s.t. for all $n \geq n_0$, $f(n) \geq cg(n)$.

1.3 From Regular Expression to NFA

There are two basic methods of converting a regular expression to an equivalent NFA. One is due to Thompson [6], and the other is due to McNaughton and Yamada [5].

In [6], Thompson gave a linear time and space construction to convert a regular expression to an equivalent NFA with ϵ-transitions, as illustrated in Figure 1.1. In the figure, unlabeled arcs represent $\epsilon-$transitions. We will follow this convention in the rest of this thesis unless otherwise specified.

In an NFA, we call a state *auxiliary* if it has only ϵ-transitions from other states;

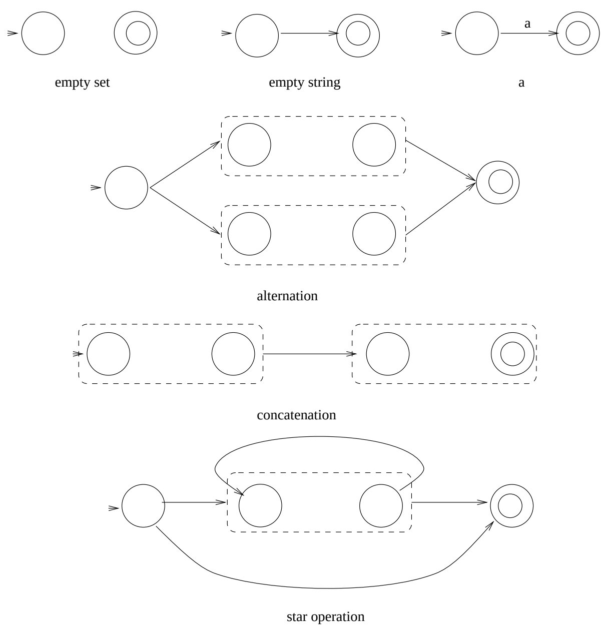

Figure 1.1: Thompson's Construction of NFA

otherwise, we call it a *transition state*. In Chapter 3, we will show that auxiliary states are candidates for deletion to make our NFA more compact. One nice property of Thompson's construction is that there are at most two transitions with the same label leaving each state in the resulting NNFA, and we call this property Thompson property.

In [1], Algorithm 1.1 was given to eliminate all the auxiliary states and ϵ-transitions.

```
**algorithm** $\epsilon$-elimination
Input: any NFA $N = (S, \Sigma, \delta, s_0, F)$
Output: NFA $N' = (S', \Sigma', \delta', s_0, F')$
such that $L(N') = L(N)$ and $N'$ has no $\epsilon$-transitions.
    $S' := \{s_0\} \cup \{t \mid t \in \delta(s, a)\ for\ some\ a \in \Sigma\}$
    **for** each $s \in S'$ and $a \in \Sigma$
        $\delta'(s, a) := \{u \mid t \in \hat{\delta}(s, \epsilon)\ and\ \delta(t, a)\ contains\ u\}$
    $F' := \{s \mid f \in \hat{\delta}(s, \epsilon) \cup F \neq \phi\}$
```

Algorithm 1.1: ϵ-elimination

We call the above procedure ϵ-elimination.

Another basic method to convert a regular expression to an NFA is based on Berry and Sethi's [10] improvement to McNaughton and Yamada's method [5]. The basic idea is to have a distinct state for each character occurrence in the regular expression and to make the transition from p to q if S_pS_q appears in some substring of the language $L(r)$, where S_p and S_q denote the characters labeling the states p and q, respectively. By this convention, each state is labeled with an alphabet symbol except the start state. To transform such a construction into the form in our definition, we can label each transition (p, q) by q's symbol.

The figure 1.2 illustrates McNaughton and Yamada's method.

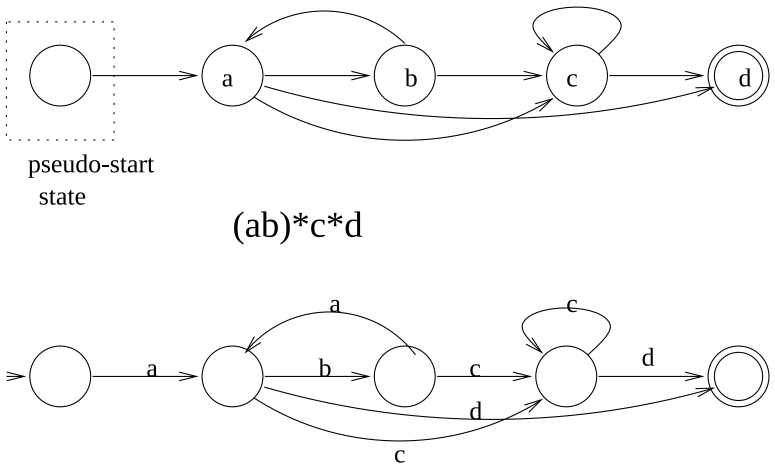

Figure 1.2: McNaughton and Yamada's Construction of NFA

Surprisingly, we can view McNaughton and Yamada's NFA as the result of ϵ-elimination from Thompson's construction. Each state in McNaughton and Yamada's NFA corresponds to a transition state in Thompson's construction.

There is a path from transition state p to transition state q in Thompson's construction if and only if there is a path with the edges spelling the string from state p to state q in McNaughton and Yamada's NFA.

1.4 Regular Expression Pattern Matching

One immediate application of converting a regular expression to an NFA is regular expression pattern matching by NFA simulation. Beginning with an NFA and a string, the following procedure determines the sequence of state sets for the given input. The NFA used here can be constructed by either Thompson's method or McNaughton and Yamada's method or any other method.

```
algorithm NFA Simulation
Input: An NFA $N$ and a string $x = a_1....a_k$ in $\Sigma^*$.
Output: The sequence of states set $C_0, C_1, ..., C_k$ such that
$C_i := \{s \mid s \in \hat{\delta}(s, a_1....a_i)\}$
  for $i := 0$ to $k$ do
    if $i = 0$ then $C_i := \{s_0\}$
    else $C_i := \cup_{s \in C_{i-1}} \delta(s, a_i)$
    mark each $t$ in $C_i$ as considered
    mark each $t$ in $S - C_i$ as unconsidered
    $QUEUE := C_i$
    while $QUEUE$ not empty do
      Find and delete $t$, the first element of $QUEUE$
      for each $u$ in $\delta(t, \epsilon)$ do
        if $u$ is unconsidered then
          mark $u$ as considered
          add $u$ to $QUEUE$ and to $C_i$
        fi
      end(for)
    end(while)
  end(for)
end(simulation)
```

Algorithm 1.2: NFA Simulation Algorithm

The algorithm runs in one step for each character in the input string. In each step of the simulation, the procedure takes a subset of the NFA states C_i, and finds a new state set

C_{i+1} reachable from C_i by taking a transition labeled with a_{i+1} and some ϵ-transitions if possible. In the next section, we will use the idea in this procedure to give an algorithm to construct a DFA from an NFA.

The time needed for this algorithm is bounded by the following theorem given in [1].

Theorem 1 *Suppose M is an NFA that has no more than e transitions leaving any vertex with the same label, and suppose M has m states. Then the simulation algorithm takes $O(emn)$ steps on an input string of length n.*

At its most general, the problem of matching a regular expression in a string may be characterized as follows: Given a string x and a regular expression r, locate all substrings of x that match r.

Based on the above definition, the universe is the set of all substrings of x. Unfortunately, the cardinality of this set is quadratic in the length of x. Moreover, searching this universe may yield overlapping and nested results. For this reason, implementations attempting a general search will restrict the search to find and report only a linear subset of the possible solutions. These arbitrary linearizing restrictions, which often alter the semantics of the search, appear to be simple, but are difficult to formalize, difficult to use precisely, and may be difficult to implement efficiently in some circumstances.

Two problems that any regular expression matching algorithm must address are:

- Where the beginning of the matching is, and if there are several, which one should be reported;
- Starting from the state of the last match, which state should we follow for the next one?

The most common restriction is the "leftmost longest match" rule, which is stated as a POSIX standard: "The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.” However, in [14], Clark argued that the longest matching does not make sense for highly formated text. Also, this approach seems impossible for applications involving essentially infinite strings (for example, the data-stream transported over the Internet).

```
algorithm shortest_match
Input: an NFA and a string $A = a_1a_2...a_n$
Output: The starting and ending position pairs of shortest matching
    for $j = 1$ to $|Q|$ do
        $P_j := 0$
    for $i = 1$ to $n$ do
        $P_1 := i$
        for $j = 1$ to $|Q|$ do
            $P'_j := 0$
        for $j = 1$ to $|Q|$ do
            for $q \in \delta(j, a_i)$ do
                $P'_q := \max(P'_q, P_j)$
        for $j = 1$ to $|Q|$ do
            for $q \in \delta(j, \epsilon)$ do
                $P'_q := \max(P'_q, P_j)$
        $u := 0$
        for $j = 1$ to $|Q|$ do
            if $j \in F$ then $u := \max(u, P'_j)$
        if $u > 0$ then
            $output(u, i)$
            for $j = 1$ to $|Q|$ do
                if $P'_j \leq u$ then $P'_j := 0$
        fi
        for $j = 1$ to $|Q|$ do
            $P_j := P'_j$
end
```

Algorithm 1.3: Shortest Matching Algorithm

To answer the second question, we propose two possible solutions:

1. Begin the search again after the first character of the match or

2. Begin the search again after the last character of the match.

The first solution is preferred when we allow overlapping in the match, while the second one reduces overlapping. The first choice may result in a large number of nested solutions. The second is the one usually taken, but it creates a bias for reporting the leftmost of overlapping matches.

Clark introduced the shortest matching solution defined in [14]. As argued in [14], shortest matching is useful when we search in structured data formatted with tags like SGML data and Internet data-stream. The time for such matching is $O(Qdn)$, where Q is the number of the states in the NFA, d is the degree bound for any state and symbol, and n is the length of the string.

The modified version that allows ϵ-transitions is presented in in Algorithm 1.3.

Based on Algorithm 1.3, we have theorem 2.

Theorem 2 *The time needed for the above algorithm is $O(n(Q + T))$, where n is the length of the input string, Q is the number of states and T is the number of transitions.*

1.5 From NFA to DFA

The classical method to construct a DFA from an NFA is Rabin and Scott's subset construction as given in [8].

Algorithm 1.4 is a general procedure, which can be applied to an NFA with ϵ-transitions.

From Algorithm 1.4, we know that each state in the DFA constructed by the Rabin-Scott method is a state vector (a subset of the NFA's states), and by the simulation algorithm, we know NFA simulation is a "time-consuming" version of the DFA working on an input string.

```
algorithm subset_construction
Input: an NFA $N$
Output: a DFA $D$ such that $L(D) = L(N)$
    $\chi := \phi$
    $workset := \{\epsilon\text{-}closure(\{q_0\})\}$
    while $\exists V \in workset$ do
        $workset := workset - \{V\}$
        for each symbol $a \in \Sigma$ and set of states
            $B := \delta(V, a)$, where $B \neq \Phi$
            $B := \epsilon\text{-}closure(B)$
            $\chi(V, a) := B$
            if $B$ does not belong to the domain of $\chi$
                or to workset then
                $workset := workset \cup \{B\}$
            fi
        end for
    end while
end
```

Algorithm 1.4: Rabin-Scott Subset Construction

Based on the above algorithm, we have the following theorem about the size of the DFA constructed from an NFA without optimization.

Theorem 3 *For each regular expression, the DFA constructed using Rabin-Scott's subset construction method from either the NFA by Thompson construction and or the NFA by McNaughton and Yamada's construction has the same number of states.*

Proof Suppose there are l literals in the regular expression. From McNaughton and Yamada's construction, we know the number of states equals the number of literals and there is a natural correspondence ϕ_1 between each state and each literal. From Thompson's construction, we know the number of transition states equals the number of literals in the regular expression, and there is also a "natural" correspondence ϕ_2 between each transition state and each literal.

So there is a natural correspondence $\phi_1\phi_2$ between the states in McNaughton and Yamada's NFA and transition states in Thompson's NFA. Let ϕ be such a mapping from the states in the McNaughton and Yamada's NFA to the transition states in Thompson's NFA.

For a Thompson's NFA, we define a projection function Π, which takes a state set and returns the corresponding transition state set.

For any state set in a Thompson's NFA Q, we have

$$\epsilon\text{-}closure(Q) = \epsilon\text{-}closure(\Pi(Q))$$

and

$$\phi(\Pi\epsilon\text{-}closure(Q)) = \phi(\Pi\epsilon\text{-}closure(\Pi(Q)))$$

So the states constructed in the DFA by Thompson's NFA have a one-to-one correspondence with the states constructed in the DFA using McNaughton and Yamada's NFA. So the number of states in the DFA is equal whether we use Thompson's NFA or McNaughton and Yamada's NFA when we have $\epsilon\text{-}closure$ in the Rabin-Scott subset construction. $\square$

From the above theorem, we know that having ϵ-transition in an NFA will not increase the number of states to convert an NFA to a DFA. But one disadvantage to having auxiliary states in an NFA is that there are more states in the state vector. This situation can be improved when we consider the projection function for each state vector.

1.6 Literature Review and Motivation

As mentioned in the previous sections, we know there are two basic methods available to convert a regular expression into an NFA.

Based on these two methods, improvement algorithms were reported as follows. Berry and Sethi formally derived and improved McNaughton and Yamada's algorithm [5] for converting regular expressions into NFA's in [10]. More recently, Brüggemann-Klein [20] presents a two-pass algorithm to compute McNaughton and Yamada's NFA using the same resource bounds as Berry and Sethi. Chang presented an algorithm in [7] that computes the same NFA in the same asymptotic time $O(n)$ as Berry and Sethi, but it improves the auxiliary space to $O(l)$, where n is the length of the regular expression and l is the number of literals. In Chang's construction, the result has $5l/2$ states and $5l$ transitions in [7]. It is a version of McNaughton and Yamada's construction which he called CNNFA. He proved that there are no more transitions than in Thompson's construction without optimization, and the CNNFA is more efficient than the Thompson's NFA for string matching.

McNaughton and Yamada's NFA is more efficient than Thompson's NFA in some special cases. For example, if each alphabet symbol appears at most once in a regular expression, then McNaughton and Yamada's NFA is in fact a DFA. However, in general, as argued in [1], working with an NFA without eliminating all ϵ-transitions is more efficient than working with an NFA after ϵ-reduction, although the new NFA has fewer states, because one disadvantage of McNaughton and Yamada's NFA is that its worst case number of edges is $m = s^2$ (and this can be achieved by adding stars to the regular expression). If $s_{()}$ is the number of occurrences of parentheses (right or left), then Thompson's NFA has only between $r - s_{()} + 1$ and $2r$ states and between $r - s_{()}$ and $4r - 3$ edges.

For regular expression pattern matching, by theorem 1, we know that the following factors affect the performance of NFA simulation:

1. number of states;

2. number of transitions for certain characters;

3. degree bound for each state and character.

It is good if we can make the number of states, the number of transitions for certain characters, and degree bound for each state and character as small as possible at the same time. This is the primary goal of this thesis.

1.7 Organization of the Thesis

In this thesis, we report the following results:

1. We give a parsing algorithm that for any regular expression r with l literals, we have a parse tree with l leaf nodes, $l-1$ concatenation and alternation nodes, and at most l star nodes.

2. We give an algorithm which takes a "smart" parse tree as input, constructs a Thompson's NFA and deletes all possible auxiliary states, so that no auxiliary states can be deleted without violation the Thompson property.

3. We give another algorithm that can construct an NFA with at most $2l$ states and $4l$ transitions. This improves the best known algorithm by Chang in [7].

4. We proved that the emptiness-of-complement problem for semi-extended regular expression is $EXPSPACE$-complete, and posed the question whether $EXPSPACE = EXPTIME$?

The remainder of this thesis will be organized as follows: Chapter 2 presents the "smart parsing" algorithm. Chapter 3 presents an algorithm that eliminates redundant states in a TNFA so that no states can be further deleted. Chapter 4 gives another simple construction algorithm that improves on the best previously known algorithm in [7] by Chang. Chapter 5 is devoted to implementation details and performance testing results.

Chapter 6 will prove that "Emptiness-of-complement" for semi-extended regular expression is $EXPSPACE$-complete. Although it is about regular expressions, this result is not closely related with the previous chapters, so this chapter has a separate introduction and references. Conclusions are given in Chapter 7.

Chapter 2

Smart Parsing

In this chapter, we show that from a regular expression with l literals, we can construct a parse tree with l leaf nodes (corresponding to the l literals in the regular expression), $(l-1)$ alternation and concatenation nodes and at most l stars. The parse tree constructed here will be used in Chapters 3 and 4.

When we talk about converting a regular expression to an NFA, we need to know the complexity of the regular expression. There are several reasonable criteria that can be used in characterizing for the complexity of a regular expression:

1. the written length, which includes the literals, parentheses, and operations;
2. the number of literals and operations;
3. the weight functions: $wt(r) := wt(t)$, where t is the corresponding parse tree of the regular expression r by "some" parsing algorithm.

As a regular expression may contain an arbitrary number of parentheses to make it more understandable, it is useful to translate a regular expression to a parse tree.

The other motivation for using a parse tree is that it is easy to optimize. Consider the following example. For an arbitrary regular expression, it may not be in the simplest form: take $r_1 = ((a|b)^*(c|d)^*)^*$ as an example, it is easy to verify that $L(r_1) = L((a|b|c|d)^*)$, which is in a simpler form. Another example is $r_1 = (((a^*b)^*c)^*d)^*$, a simpler form would be $L(r_1) = L((a|b|c|d)^*)$. For parsing techniques, please refer to [2] for more details.

We list the properties of regular expressions that are useful to reduce the size of the parse tree for regular expressions.

$$(A^*)^* = A^* \tag{2.1}$$

$$A|A = A \tag{2.2}$$

$$\epsilon A = A \tag{2.3}$$

$$(\epsilon|A)^* = A^* \tag{2.4}$$

$$(A|B)^* = (A^*|B^*)^* \tag{2.5}$$

$$(A^*B^*)^* = (A|B)^* \tag{2.6}$$

$$((A)) = (A) \tag{2.7}$$

Figure 2.1 illustrates the difference between the smart parsing algorithm and the usual “parsing” algorithm based on rule 2.6.

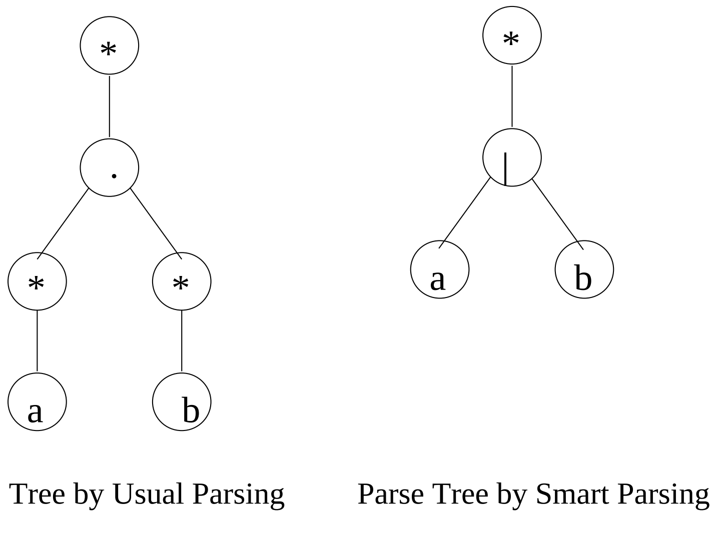

Figure 2.1: Parse Tree Comparison between Smart and Usual Parsing

2.1 Algorithm Description

Algorithm 2.1 is the recursive version of the “usual” parsing technique for regular expressions.

```
algorithm $parse\_re(s, end)$
Input: A regular expression conforming to our syntax, and
an deliminator for the end of the r.e
Output: Parse tree for this r.e.
    Stack stk;
    while $(!end(s))$
        switch ($get\_token(s)$) {
            case ‘)’, END:
                return $Top(cat(stk))$;
            case ‘(’:
                $Push(stk, parse\_re(s, ‘)’))$;
            case ‘*’:
                $Push(stk, wrap(Top(stk), ‘*’))$;
            case ‘$|$’:
                $rl := cat(stk)$;
                $rr := parse\_re(s, end)$;
                $Push(stk, mk\_alt(rl, rr))$;
            case LIT:
                $Push(stk, mk\_leaf(s))$;
            default:
                $/*error....*/$
        end(Switch)
    end(While)
end
```

Algorithm 2.1: “Usual” Parsing Algorithm

The helper functions used in parsing are described as follows:

- $Top(stk)$: Return the top of Stack stk;
- $wrap(re, ‘*’)$: Make a node labeled with ‘$*$’ with child re;

- $Push(stk, re)$: Push re to Stack stk;
- $cat(stk)$: Concatenate all the regular expressions on Stack stk.

From a regular expression, we have four kinds of nodes in the parse tree: leaf (literal) nodes which correspond to characters in the regular expression, stars, alternations and concatenations which correspond to the three operators allowed in the regular expression.

We call a node (corresponding to a sub-expression) R of a parse tree *nullable* if it is

1. a star node in the parse tree,
2. an alternation of a node with an empty string,
3. a concatenation of which both children are nullable or
4. It is an alternation of which at least one of the children is nullable.

A regular expression is nullable, if the root of its parse tree is nullable. It is easy to see that R is nullable iff $\epsilon \in L(R)$. We call a node *non-nullable* if it is not nullable. We use the following data structure to represent a node in the parse tree:

```
Data Structure Node {
    Op : can be a Star, Lit, Alternation, Concatenation
    union {
        Lchild, Rchild : when Op is Alternation or Concatenation
        Child : when Op is Star
        lit : when Op is Lit
    }
}
```

For the rest of this thesis, we are assuming that no ϵ occurs in a regular expression. To handle this in a real world program, one could add a flag for each node in the parse tree, say $withepsilon$, to specify whether this node contains an ϵ or not. Whenever we make an alternation between a node r and ϵ, we check to see if $\epsilon \in L(r)$; if yes, do nothing, if no, we set $withepsilon$ to 1.

Whenever a star node is created, the following procedure $denull(root)$ will be invoked. The argument root identifies the node over which the star will be created.

```
algorithm $denull(root)$
Input: A Parse Tree
Output: A Parse Tree without Star over nullable node
    if not $nullable(root)$ then
        return
    else if $root.Op =$‘*’ then
        $root = root.Child$
        return
    else if $root.Op =$‘.’ then
        $root.Op =$‘$|$’
        $denull(root.Lchild)$
        $denull(root.Rchild)$
        return
    else if $root.Op =$ ‘$|$’ then
        $denull(root.Lchild)$
        $denull(root.Rchild)$
        return
    fi
end
```

Algorithm 2.2: *denull* Procedure

For the construction of a star node, we use the following procedure:

```
algorithm $mk\_star(child)$
Input: A Parse Tree
Output: A Parse Tree rooted with ‘*’ whose child is the input
    if $nullable(child)$ then
        $denull(child)$
    fi
    $root = new\_node(‘*’)$
    $root.Child = child$
end
```

Algorithm 2.3: Improved Star Operation

For the construction of other types of nodes, we use the same procedure described in [2].

Because the algorithm described above behaves smarter than the "usual" parsing algorithm, we call it the "smart parsing" algorithm.

2.2 Analysis of the Smart Parsing Algorithm

In this section, we first give a proof for the correctness of the algorithm presented above; then we give the analysis for the node-number complexity and time and space complexity of the smart parsing algorithm.

2.2.1 Correctness of the Algorithm

Theorem 4 *Smart parsing produces a parse tree which is equivalent to the one produced by the usual parsing algorithm. Moreover, the new parse tree contains no star nodes with a nullable child.*

Proof To see that the first assertion is correct, we show $r^* = \hat{r}^*$, where $\hat{r} = denull(r)$.

We have the following cases:

1. If r is not nullable, then $r = \hat{r}$ and $r^* = \hat{r}^*$;

2. If $r = r' \cup \{\epsilon\}$, then $\hat{r} = r'$ and $r^* = \hat{r}^*$;

3. If $r = r'^*$, then $\hat{r} = denull(r) = r'$(as we do not have star whose child is nullable), $r^* = \hat{r}^*$;

4. If $r = r_1 r_2$, where r_1 and r_2 are nullable, $r^* = (r_1 r_2)^* = (r_1|r_2)^* = (\hat{r_1}|\hat{r_2})^*$;

5. If $r = r_1|r_2$, where at least one of r_1 or r_2 is nullable, then we can apply a $denull$ procedure to r_1 and r_2;

The second assertion of the theorem is true because the result of $denull(r)$ is always non-nullable by induction.

This completes the proof. $\square$

2.2.2 Parsing Tree Node Complexity

We have the following theorem that bounds the number of nodes in the parse tree.

Theorem 5 *For a regular expression with l literals, we can construct a parse tree with internal nodes labeled with concatenation, alternation and star, and each leaf node labeled with a literal. There are exactly l leaf nodes, $l-1$ alternations and concatenations and at most l stars.*

Proof Let us analyze the generation of the parse tree of a regular expression. There are three kinds of nodes in a parse tree:

1. Leaf nodes;
2. Internal nodes with out-degree 1 (star);
3. Internal nodes with out-degree 2 (concatenation and alternation).

In the construction, each leaf node corresponds to one alphabet character (or it will be merged with other nodes). So there are exactly l leaves.

By a standard property of trees, we know

$$\sum deg_{in}(v) = \sum deg_{out}(v),$$

so the number of leaves is 1 more than the binary nodes (alternation and concatenation).

Because we have at most l leaf nodes,

$$l = \#(alternation) + \#(concatenation) + 1$$

The key new property of our parse tree is that `Each star does not have a nullable child`. By using this property, we can show that a tree with l leaves having l stars must be nullable.

The base case obviously holds, for a regular expression with 1 literal a, we know it has at most 1 star, in the form a^*.

Induction step: Suppose for any regular expression with l literals, if it has l stars, it must be nullable.

For any regular expression with $l + 1$ literals, if the root is a star, we know the child of the root can not be a star as we do not have a star over a nullable node. If the child is marked with alternation, the two subtrees have l_1 and l_2 literals respectively; we know that neither T_1 nor T_2 can have l_1 or l_2 stars, or the new node is nullable. Similarly, we have the same

So, we have an equivalent parse tree having at most $3l - 1$ nodes. □

Based on the above theorem, when we write down a regular expression from a parse tree, we need to add at most $(l - 1)$ pairs of parenthesis to make the regular expression unambiguous, and we have the following lemma about the property of a regular expression:

Lemma 1 *For each regular expression with l literals, there is an equivalent one with length $5l$.*

This is a better result than the $14(l - 1) + 5$ result in Chang's thesis [7].

2.2.3 Time and Space Analysis

The following theorem bounds the time needed for this algorithm:

Theorem 6 *The construction of the parse tree can be done in linear time w.r.t the size of the regular expression.*

Proof During the construction, each node will be marked as nullable or non-nullable, but once a node is marked as non-nullable, it will not be visited anymore (except one test when it is a root and we try to put it as a child for a star) during $denull$ from the algorithm. So the time for $denull$ is bounded by the number of nodes generated from the regular expression, which is bounded by the length of the regular expression.

And the time needed for other operations is to scan the regular expression from left to right and construct the parse tree using procedure $atomic\_nfa$, mk_alt, mk_cat and mk_star, so the overall time for the construction is linear w.r.t the size of the regular expression.

□

For space complexity, Chang gave an algorithm in [7] based on operator grammar [2] and showed the extra space needed is bounded by the l, which is the number of literals in the regular expression.

2.3 Parsing Regular Expressions with More Operators

We know that the alternation, concatenation, and star are powerful enough to denote any regular language. But sometimes, it is useful to have more operators like option operator $?$, which is postfix and denotes that a regular expression occurs zero or one time; and plus operator $+$ which is also postfix and denotes a regular expression occurs one (at least one) or more times.

To include these features in, we only need to add the following

$$nullable(A?) = true$$

$$nullable(A+) = nullable(A)$$

2.4 More Optimizing Techniques and Further Research

For an l-literal regular expression, we know there are l Kleene stars in the worst case (take $a^*b^*c^*...$ as an example), so our upper bound is in fact a lower bound. When we apply another equivalent formula 2.5, we can get an even more compact parse tree.

Based on the parsing algorithm given above, we pose the following question: Can we get a reduced regular expression with some parameter in polynomial time? To formalize this problem, for each regular expression, we have a weight function $wt(r)$, the regular expression r' with the minimum weight that is equivalent with r, the goal is to find r_a in polynomial time such that $L(r) = L(r_a) = L(r')$ and $wt(r_a) \leq \alpha wt(r')$, What we hope is that α is a constant.

Chapter 3

Minimized TNFA

In this chapter, we give a construction for minimized NFA with Thompson's property (TNFA for short). To our knowledge, this is the first algorithm that can minimize a TNFA in polynomial time, and in fact, all the time needed for this algorithm is linear w.r.t the length of the regular expression.

3.1 Algorithm

Algorithm 3.1 constructs an TNFA from the parse tree obtained by smart parsing in Chapter 2.

The $atomic\_nfa$, mk_alt, mk_cat and mk_star works as illustrated in Figure 1.1.

Recall in Chapter 1, we classified the states of the TNFA into two classes: a state is *auxiliary* if it has only ϵ-transitions from other states; otherwise, we call it a *transition state*. Auxiliary states are the candidates for deletion in this thesis. When we delete an auxiliary state, we merge all of its out-transitions to those states having ϵ-transitions.

We call a state or a state set *deletable* if it can be deleted without violating the defining property of TNFA.

From [13], we know NFA (without ϵ-transition) minimization in general is $PSPACE$-hard. But we will show that we can get a minimized TNFA such that no *auxiliary state* can be deleted without violation of the defining property of TNFA. And the construction

```
algorithm $tnfa\_from\_tree(root)$
Input: a parse tree $root$ by smart parsing
Output: an NFA with Thompson's property
    switch $type(root)$
        case LIT:
        return $atomic\_nfa(root)$
        case '|':
        $l = tnfa\_from\_tree(Lchild(root))$
        $r = tnfa\_from\_tree(Rchild(root))$
        return $mk\_alt(l, r)$
        case '.':
        $l = tnfa\_from\_tree(Lchild(root))$
        $r = tnfa\_from\_tree(Rchild(root))$
        return $mk\_cat(l, r)$
        case '$*$':
        $l = tnfa\_from\_tree(Child(root))$
        return $mk\_star(l)$
    end
end
```

Algorithm 3.1: Thompson's Construction from a Parse Tree

has the following properties:

- The size of the TNFA is linear w.r.t the number of literals in the regular expression by smart parsing algorithm;
- The construction runs in linear time w.r.t the size of the regular expression;
- The number of states is minimized in a sense that no *auxiliary state* can be deleted without violating the defining property of TNFA.

To proceed, let's analyze how transition states and auxiliary states get generated. From Figure 1.1, we know there are at most l transition states in TNFA for a regular expression with l literals, which are the final states when we generate the TNFA for a character. The auxiliary states are generated due to the adding of the starting states and final states. When a new start state is added, start states of sub-TNFAs become auxiliary; when a new final state is added, it is auxiliary.

When we try to delete some auxiliary states, we need to make sure the resulting NFA will not violate the property of TNFA. Intuitively, we need to examine all subsets of the auxiliary states, which will make the overall run time exponential. In the following, we will show that the sequence of examining the auxiliary states can be localized. This may be regarded as the most significant contribution in this chapter.

We call a set of auxiliary states *deletable* if the deletion of these auxiliary states results in a TNFA. And we call it *undeletable* if it is not deletable. We may have some undeletable states in a deletable set, as when we talk about the set, we only care about the resulted NFA is a TNFA. But we do not have to guarantee that it is a TNFA after deletion of a single auxiliary state.

For ease of analysis, we use the following notation. For any two auxiliary states p, q, we define

$$p \succ q \quad \textit{iff} \quad \text{there is an } \epsilon\text{-transition from } p \text{ to } q$$

and

$$p \succ^{+} q \quad \textit{iff} \quad \text{there is an } \epsilon\text{-transition path that only touches auxiliary states from } p \text{ to } q$$

We define a relation $brother(x, y)$ over auxiliary states if there exists another state z such that $z \succ x$ and $z \succ y$. And we use $B(x)$ to denote x's brother if its brother exists and $F(x)$ to denote those states z such that $z \succ x$.

Property 1: There is no auxiliary state cycle in the TNFA constructed by $tnfa\_from\_tree$.

Referring to Figure 1, the first time an ϵ-cycle can appear in the TNFA is during the star construction. However, even then, an ϵ-cycle is created only if the child node already contains an ϵ-path running from the start to the final. However, since we always assume that the child of a star is not nullable, this will not happen.

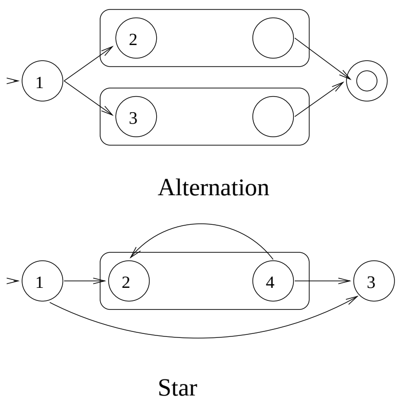

Figure 3.1: Illustration of Brother Relation

From this property, we know $\succ$ is a partial order, and by topological sorting described below, we can get the sequence of the auxiliary states that we can examine to see if we can further remove auxiliary states from the TNFA.

1. For each auxiliary state p, put it to queue if there is no q such that $p \succ q$;

2. Remove one element p from the queue, and for those states r that have ϵ-transition in to p, delete p from r's auxiliary-state-transition-in list, put r in the queue if p is the only one in the list before deletion;

3. Repeat step (2) until the queue is empty.

Property 2: For any $p \succ q$, there is no auxiliary state s s.t $s \succ p$ and $s \succ q$.

Proof The auxiliary states that contain two out ϵ-transitions in the TNFA constructed are:

i. the start state of an alternation,

ii. the start state of a star,

iii. the final state of an NFA to which the last star operation is applied.

For each state with two ϵ-transitions, the two auxiliary states are either in two sub-NFAs that do not have any transition in between (the first case) or are separated by one or more transition states (the second and third cases). $\square$

Using the above two properties of the TNFA, we have the following lemma:

Lemma 2 *Suppose Q is a deletable set. Then there exists a state q such that we can either delete q first, and then delete $Q - \{q\}$, or delete $\{q, B(q)\}$ first, and then delete $Q - \{q, B(q)\}$.*

Proof The contrapositive of the above theorem is: if we can neither delete q first, and then delete $Q - \{q\}$, nor delete $\{q, B(q)\}$ first, and then delete $Q - \{q, B(q)\}$, then Q is not a deletable set.

If we can neither delete q nor delete $\{q, B(q)\}$, there must be a state $p \in F(q)$ that have more transitions to violate the degree bound property of TNFA. From Property 2, we know $p \neq B(q)$.

If $p \in Q$, when delete Q, all p's out-transitions will be merged to its ancestor's, and cause some states have more transitions to violate the degree bound of TNFA. This makes Q not deletable.

If $p \notin Q$, we know delete q or $\{q, B(q)\}$ will have no effect on the deletability on the other states, making Q undeletable. $\square$

Property 3: Each auxiliary state has at most one brother state. This follows from the same proof as for property 2.

We need to take a look at a state's brother. But from Property 3, we know for each state, there is at most 1 brother, and we have Algorithm 3.2 based on the above topological sorting to reduce ϵ-transitions. In the following pseudo-code, for an auxiliary state p, we use $isReady(p)$ to represent "All of p's children are either transition states or visited".

```
algorithm Reduce
Input: a TNFA
Output: a TNFA without deletable auxiliary state set
    for each auxiliary state $p$
        mark $p$ as unvisited
        put it to queue if there is no auxiliary
        state $q$ such that $p \succ q$ and mark
        $p$ as ready
    repeat the following until the queue is empty:
        take one state $p$ out of the queue
        case $p$ marked as deleted
            break
        case $p$ is deletable
            mark it as deleted
            merge its transitions to $F(p)$'s transitions
            add its $F(p)$ to queue if $isReady(F(p))$
            mark $F(p)$ as ready
            break
        case $p$ is not deletable and
            $B(p)$ is ready
            if $\{p, B(p)\}$ is deletable
                mark $p$ and $B(p)$ as deleted
                merge their transitions to $F(p)$'s transition
                else
                    mark $p$ as undeletable
                fi
            add its $F(p)$ to queue if $isReady(F(p))$
            mark $F(p)$ as ready
            break
        case $p$ is not deletable and
            $B(p)$ is not ready
            break
    end(repeat)
end
```

Algorithm 3.2: Procedure to Reduce ϵ-transitions

We have the following theorem about the property of the Algorithm 3.2:

Theorem 7 *The TNFA produced by procedure $Reduce$ does not have a non-empty deletable auxiliary state subset.*

Before proving the theorem, we need the following lemma:

Lemma 3 *For any two auxiliary states p and q, if $p \succ^+ q$ and q is not deletable, then $\{p, q\}$ is not deletable.*

Proof Suppose $\{p, q\}$ is deletable. Then for any state $s \in F(p) \cup F(q)/\{p\}$ will not violate the Thompson's property (there are no more than two transitions with the same label), but there exists an auxiliary state $r \in F(q)$.

If $r = p$, then p can not have any c-violations for each $c \in \Sigma \cup \{\epsilon\}$, or these violations will be propagated to those states in $F(p)$. This will make $\{p, q\}$ not deletable.

If $r \neq p$, from property 2, there is no ϵ-transition from r to p, so the deletion of p will not affect the transitions of r, and $\{p, q\}$ is not deletable. □

Now, we are ready to give the proof for the theorem:

Proof Based on the description, we know there is no state p such that $\{p\}$ is deletable, as we have examined each auxiliary state one at a time.

Suppose there is a state set $\{s_1, s_2, ..., s_k\}$ that is deletable. Because there is no auxiliary state cycle, we can partition this deletable set into paths. And each state set in a path can not make the other states in other paths deletable. So we only need to consider an auxiliary state set $\{s'_1, s'_2, ..., s'_{k'}\}$, such that $s'_1 \succ^+ s'_2, ..., s'_{k'-1} \succ^+ s'_{k'}$, using lemma 3 $k - 1$ times, we know $\{s'_1, s'_2, ..., s'_{k'}\}$ is not deletable.

So the non-empty deletable set does not exist. □

After giving the description of the work done on each auxiliary state, we finish this section.

For each auxiliary state, we must decide if this state can be deleted. To make this decision, we check to see if merging the state's out-transition with its ancestors' transitions causes one of the ancestors to violate the Thompson's property. In a similar way, we can check to see if a state pair $\{p, B(p)\}$ is deletable.

The following is a summary of our algorithm:

1. Construct a parse tree from the regular expression by our smart parsing algorithm presented above;

2. Construct a TNFA (without auxiliary state cycle) from the parse tree;

3. While topologically sorting the auxiliary states, examine each auxiliary state to see if it can be deleted; if yes, delete it and merge its transitions with its ancestors'.

3.2 Time and Space Analysis of the Algorithm

In this section, we give the analysis of the time needed for this algorithm.

From the above algorithm, the time needed is to traverse the parse tree constructed by our smart parsing algorithm presented in previous chapter. For each node in the parse tree, we need constant time to do $atomic\_nfa$, mk_star, mk_alt and mk_cat, so the overall time needed to construct an NFA without auxiliary state cycle is linear w.r.t the size of the parse tree.

From Thompson's construction, we know there are at most $6l$ states and $12l$ transitions in the worst case.

Theorem 8 *The overall time for the deletion of the auxiliary states is bounded by the number of the states in the TNFA.*

Proof Notice the number of ϵ-transitions is bounded by the number of the nodes in the parse tree, and the overall number of states in the list will never increase. Once the state got deleted, we do not have to keep the list and its deletion doesn't affect other auxiliary states' list. So the overall time needed is $O(\#\text{of } \epsilon\text{-transitions} + \# \text{ of auxiliary states}) = O(l)$. □

The extra space needed during the construction is the parse tree and the lists for each auxiliary state that keeps which states have ϵ-transition in. Because the bound of the number of ϵ-transitions, this is also in $O(l)$.

Chapter 4

Simple NNFA Construction

In this chapter, we show: For a parse tree with l leaf nodes, we can construct an NFA with $2l$ states and $4l$ transitions in the worst case. Combined with the parsing algorithm presented in Chapter 2, for a regular expression with l literals, we can construct an NFA with $2l$ states and $4l$ transitions in the worst case. And our algorithm runs in linear time w.r.t the length of the regular expression. This improves the construction algorithm given by Chang in [7], which constructs an NFA with $5l/2$ states and $(10l - 5)/2$ transitions in the worst case. The method we present is much simpler and easier to implement, as we use only naive ideas: divide and conquer.

4.1 Construction Algorithm

Just like all the divide and conquer algorithms, our algorithm consists of two stages: First, the construction method for regular expressions that are small; second, combining the NFAs constructed from sub-expressions.

For 1-literal and 2-literal regular expressions, we construct NFAs as illustrated in Figure 4.1.

Following the convention in previous chapters, we use unlabeled edge to denote an ϵ-transition in a diagram for an NFA. Recall that an NNFA is an NFA in which there is no transition to the start state and there is no transition from the final state. With this property, we know the NNFA is suitable for recursive construction. And this is coincident with the recursive property of regular expressions.

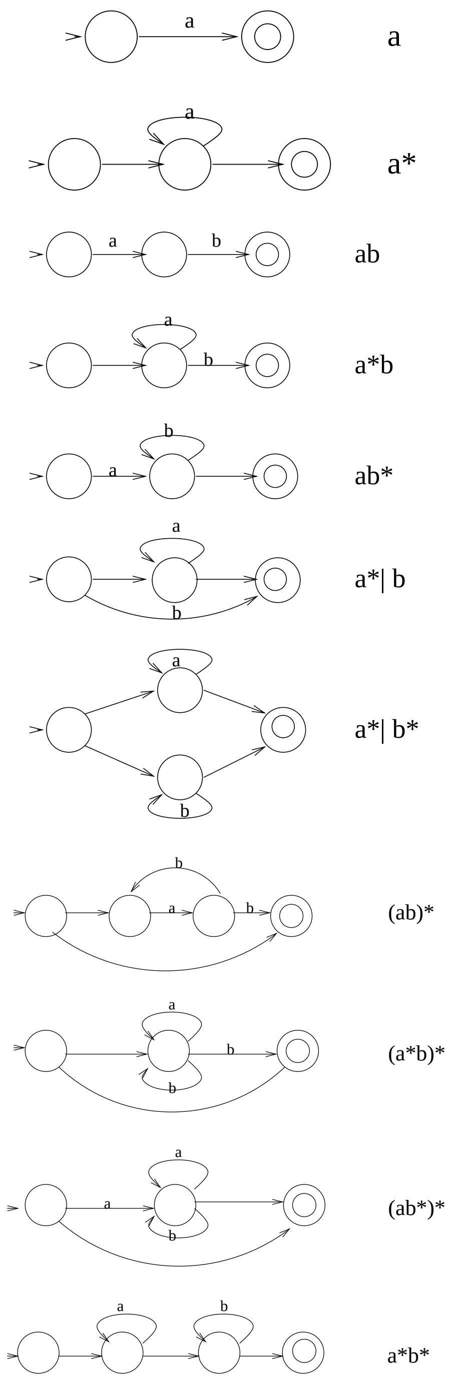

Figure 4.1: Construction of NFA from 1,2-literal regular expression

From Chapter 3, we know there are auxiliary states and redundant transitions that can be removed from the NFA. So when we combine two sub-NFAs together, we may need to delete some states and transitions as described in Chapter 3. This will make the algorithm complicated. To handle this, we propose the following lazy schema for the construction.

As the candidate states for deletion must be either the start or the final state of an NNFA, we use special labels: "$-$" and "$\square$" for the start and final state of the NNFA. One advantage of this is when we make an alternation, we still have a single start state and final state, so these two special states are not labeled the same as other states until they become inner states which, a condition makes them not special anymore.

The combining procedure is very similar with naive Thompson's construction, but it does special construction for alternation and concatenation illustrated by Figure 4.2, 4.3, 4.4.

4.2 Analysis

For an NNFA n, we S_n to denote the number of states and T_n to denote the number of transitions. We define $\chi(n_1, n_2)$ to be 1 if and only if there is only one transition to the final state of n_1 and all the transitions from the start state of n_2 is labelled with ϵ, or there is only one transition from the start state of n_2 and all the transitions from the final state of n_1 is labelled with ϵ.

Going through the NNFAs in Figure 4.1, it is easy to verify:

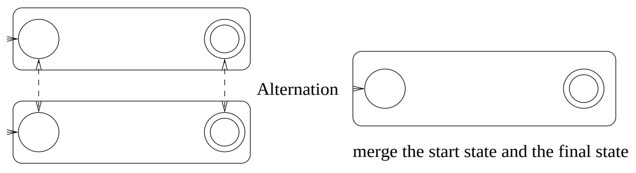

Figure 4.2: Illustration of Alternation

Fact 1 *For each regular expression r with 1 literal, there is an NNFA n s.t $S_n = 2, T_n = 1$ if r is not nullable, and $S_n \leq 3, T_n \leq 3$ if it is.*

Fact 2 *For each regular expression r with 2 literals, there is an NNFA n s.t $S_n = 3$, $T_n = 3$ if r is not nullable, and $S_n \leq 4, T_n \leq 6$ if it is.*

Based on the construction algorithm, we have the following recursive relation:

$$S_{n_1 n_2} = S_{n_1} + S_{n_2} - 1 - \chi(n_1, n_2) \tag{4.1}$$

$$S_{n_1 | n_2} = S_{n_1} + S_{n_2} - 2 \tag{4.2}$$

$$S_{n_1^*} = S_{n_1} + 2 \tag{4.3}$$

$$T_{n_1 n_2} = T_{n_1} + T_{n_2} \tag{4.4}$$

$$T_{n_1 | n_2} = T_{n_1} + T_{n_2} \tag{4.5}$$

$$T_{n_1^*} = T_{n_1} + 4 \tag{4.6}$$

And we know for any regular expression r and any character c, we have $\chi(r^*, c) = 1$ and $\chi(c, r^*) = 1$.

The main theorem in this chapter is:

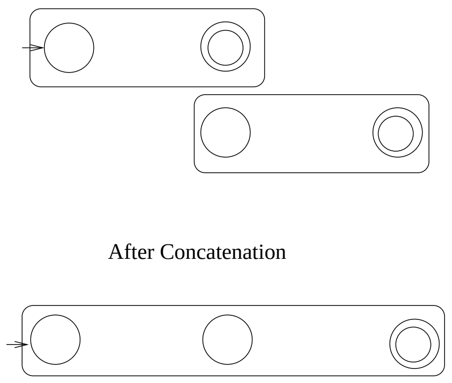

Figure 4.3: Illustration of Concatenation

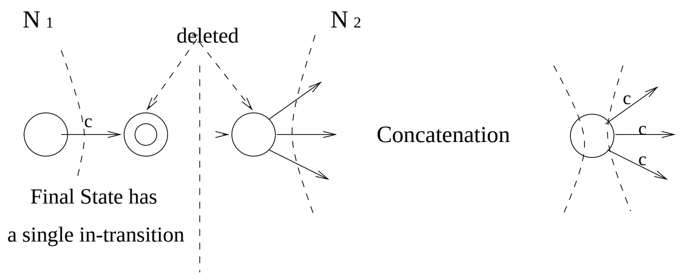

Figure 4.4: Illustration of Concatenation when $\chi(n_1, n_2) = 1$

Theorem 9 *For each regular expression with $l \geq 3$ literals, the resulting NNFA from the above construction has*

1. *$2l - 2$ states and $4l - 6$ transitions if it is not nullable;*

2. *$2l - 1$ states and $4l - 3$ transitions if it is nullable but the root is not a star;*

3. *$2l$ states and $4l - 2$ transitions if the root is a star.*

Proof Based on Smart Parsing presented in Chapter 2, we know that we never have a star over a star. Thus, the first assertion implies the third assertion based on the star in Thompson's construction as illustrated in Figure 1.1.

So we need to consider only the regular expression whose parse tree is labeled with alternation or concatenation at the root. We prove this by structure induction. For each regular expression whose parse tree having l_1 literals in its left child and l_2 literals in its right child, we have the following cases:

1. $l_1 + l_2 = 3$, which is handled in Case 1;

2. $l_1 = l_2 = 2$, which is handled in Case 2;

3. one of l_1 and l_2 is 1 and the other is ≥ 3, which is handled in Case 3;

4. one of l_1 and l_2 is 2 and the other is ≥ 3, which is handled in Case 4;

5. $l_1, l_2 \geq 3$, which is handled in Case 5.

For each case, based on the label of the root and whether it is nullable, we have the following 4 sub-cases:

i. The root is labeled with alternation and is nullable;

ii. The root is labeled with alternation and is non-nullable;

iii. The root is labeled with concatenation and is nullable;

iv. The root is labeled with concatenation and non-nullable.

We give a detailed proof for Case 1 and 5, and we leave the details for the other cases to readers.

Case 1 *$l_1 = 1$ and $l_2 = 2$ or $l_1 = 2$ and $l_2 = 1$*

Because alternation is symmetric, if the root of the parse tree is alternation and n is non-nullable, then both n_1 and n_2 are non-nullable. So $S_{n_1|n_2} \leq 2 + 3 - 2 = 3$ and $T_{n_1|n_2} \leq 1 + 3 = 4$.

If n is nullable, the worst case is that both n_1 and n_2 are nullable; it is easy to verify that $S_{n_1|n_2} \leq 3 + 4 - 2 = 5$ and $T_{n_1|n_2} \leq 3 + 6 = 9$.

To represent the above proof, list the bounds of the number of states and transitions as follows:

$$S_{n_1|n_2} = \begin{cases} 3 < 2L - 3 & \text{non-nullable} \\ \leq 5 = 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1|n_2} = \begin{cases} 6 < 4L - 6 & \text{non-nullable} \\ \leq 9 = 4L - 3 & \text{nullable} \end{cases}$$

Similarly, we have the following argument for the concatenation case: If the root of the parse tree is concatenation and n is non-nullable, then at least either n_1 or n_2 is non-nullable. So $S_{n_1n_2} \leq \min(3+3-1-1, 2+4-1-1) = 4$ and $T_{n_1n_2} \leq \min(1+6, 3+3) = 7 < 4L-5$.

If the root of the parse tree is concatenation and n is nullable, then both of n_1 and n_2 are non-nullable. So $S_{n_1n_2} \leq 3+4-1-1 = 5$ and $T_{n_1n_2} \leq 3+6 = 9 \leq 4L-3$.

$$S_{n_1n_2} = \begin{cases} \leq 4 = 2L-2 & \text{non-nullable} \\ \leq 5 = 2L-1 & \text{nullable} \end{cases}$$

$$T_{n_1n_2} = \begin{cases} 7 \leq 4L-5 & \text{non-nullable} \\ \leq 9 = 4L-3 & \text{nullable} \end{cases}$$

Case 2 $l_1 = l_2 = 2$

In this case $L = l_1 + l_2 = 4$, and we have:

$$S_{n_1|n_2} = \begin{cases} \leq 4 < 2L-2 & \text{non-nullable} \\ \leq 6 < 2L-1 & \text{nullable} \end{cases}$$

$$T_{n_1|n_2} = \begin{cases} \leq 9 < 4L-6 & \text{non-nullable} \\ \leq 12 < 4L-3 & \text{nullable} \end{cases}$$

$$S_{n_1n_2} = \begin{cases} \leq 6 = 2L-2 & \text{non-nullable} \\ \leq 7 = 2L-1 & \text{nullable} \end{cases}$$

$$T_{n_1n_2} = \begin{cases} \leq 9 < 4L-6 & \text{non-nullable} \\ \leq 12 < 4L-3 & \text{nullable} \end{cases}$$

Case 3 $l_1 = 1$ *and* $l_2 \geq 3$

In this case $L = l_1 + l_2 = L_2 + 1$, and we have:

$$S_{n_1|n_2} = \begin{cases} \leq 2l_2 - 2 < 2L - 2 & \text{non-nullable} \\ \leq \max(2l_2, 2l_2 - 2 + 1, 2l_2 + 1) \leq 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1|n_2} = \begin{cases} \leq 4l_2 - 6 + 1 < 4L - 6 & \text{non-nullable} \\ \leq \max(4l_2 - 2 + 1, 4l_2 - 6 + 3, 4l_2 - 2 + 3) \leq 4L - 3 & \text{nullable} \end{cases}$$

$$S_{n_1 n_2} = \begin{cases} \leq \max(2l_2 - 2 + 1, 2l_2, 2l_2 - 2 + 1) \leq 2L - 2 & \text{non-nullable} \\ \leq 2l_2 + 1 = 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1 n_2} = \begin{cases} \leq \max(4l_2 - 6 + 1, 4l_2 - 2, 4l_2 - 6 + 2) < 4L - 6 & \text{non-nullable} \\ \leq 4l_2 - 2 + 2 < 4L - 3 & \text{nullable} \end{cases}$$

Case 4 $l_1 = 2$ *and* $l_2 \geq 3$

In this case, $L = l_1 + l_2 = l_2 + 2$, and we have:

$$S_{n_1|n_2} = \begin{cases} \leq 2l_2 - 2 + 1 < 2L - 2 & \text{non-nullable} \\ \leq 2l_2 + 2 < 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1|n_2} = \begin{cases} \leq 4l_2 - 6 + 3 < 4L - 6 & \text{non-nullable} \\ \leq 4l_2 - 2 + 6 < 4L - 3 & \text{nullable} \end{cases}$$

$$S_{n_1 n_2} = \begin{cases} \leq \max(2l_2 + 3 - 1, 2l_2 - 2 + 4 - 1) \leq 2L - 2 & \text{non-nullable} \\ \leq 2l_2 + 4 - 1 = 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1 n_2} = \begin{cases} \leq 4l_2 + 1 < 4L - 6 & \text{non-nullable} \\ \leq 4l_2 - 2 + 6 < 4L - 3 & \text{nullable} \end{cases}$$

Case 5 $l_1, l_2 \geq 3$

Based on the recursive relation, it is easy to verify that the following is true.

$$S_{n_1|n_2} = \begin{cases} \leq 2(l_1 + l_2) - 6 < 2L - 2 & \text{non-nullable} \\ \leq 2(l_1 + l_2) - 2 = 2L - 2 & \text{nullable} \end{cases}$$

$$T_{n_1|n_2} = \begin{cases} \leq 4(l_1 + l_2) - 12 < 4L - 6 & \text{non-nullable} \\ \leq 4(l_1 + l_2) - 4 < 4L - 3 & \text{nullable} \end{cases}$$

$$S_{n_1 n_2} = \begin{cases} \leq 2(l_1 + l_2) - 3 < 2L - 2 & \text{non-nullable} \\ \leq 2(l_1 + l_2) - 1 = 2L - 1 & \text{nullable} \end{cases}$$

$$T_{n_1 n_2} = \begin{cases} \leq 4(l_1 + l_2) - 12 < 4L - 6 & \text{non-nullable} \\ \leq 4(l_1 + l_2) - 4 < 4L - 3 & \text{nullable} \end{cases}$$

This completes the proof. □

4.3 Time and Space Analysis

The following two theorems will bound the time needed for this algorithm:

Theorem 10 *The construction of the parse tree can be done in linear time w.r.t the size of the regular expression.*

Proof During the construction, each node will be marked as nullable or non-nullable, but once a node is marked as non-nullable, it will not be visited anymore (except one test when it is a root and we try to put it as a child for a star).

And the time needed for other operations is to scan the regular expression from left to right, so the overall time for the construction is linear w.r.t the size of the regular expression. □

Theorem 11 *The overall time for the NFA construction from a parse tree is bounded by the number of nodes in the parse tree.*

Proof At each step, we need to do one or more of the following:

1. Add new states;
2. Add new transitions;
3. Change ϵ-transitions to character-transitions.

The time spent on changing the ϵ-transitions to character-transitions is bounded by the number of ϵ-transitions ever created in the NFA. Because once an ϵ-transition is changed to a character-transition, it can not be changed anymore. So the overall time needed is at most double the time for creating new transitions plus the time on creating new states. From the theorem stating that the number of states is bounded by $2l$ and the number of transitions is bounded by $4l$, we know the number of states and transitions is bounded linearly by the number of literals in the regular expression. □

4.4 More Optimization Techniques

In this section, we will give techniques that can be used to further the optimization of the construction.

The first method we will introduce is a contraction method that is based on a simple idea. Whenever we need to add ϵ-transitions from a state set to another state set, we will add a new state if and only if the following function evaluates to be true: $R(r_1, r_2)$ if $F(r_1)S(r_2) > F(r_1) + S(r_2)$, where $F(r)$ specifies the number of final states in the NFA for r and $S(r)$ specifies the number of start states in the NFA for r. Using this technique, we never have fewer transitions when we have more states.

The second optimizing technique is: instead of pre-computing the optimal NFA for regular expression up to 3 literals, we can pre-compute the optimal NFA for regular expression with up to l, say 4 literals.

4.5 Conclusion

We presented a very simple algorithm to construct NNFA with fewer states and transitions which improves Chang's work in [7]. No elaborate data structures are required in this algorithm, and it is very easy to implement.

Chapter 5

Implementation and Experimental Results

We implemented both construction algorithms as part of a Unix text search tool: `xgrep`, which does extended regular expression pattern matching. In this chapter, we first show the heuristics used in our implementation and discuss the representation of an NFA, which affects the performance. Then we show the comparison between Simple NFA presented in Chapter 4 (SNFA for short), minimized Thompson's NFA (mTNFA for short), original Thompson's NFA (TNFA), and McNaughton and Yamada's NFA (MYNFA for short) with regard to the number of states, the number of transitions, NFA simulation time, and the time needed for pattern matching using "lazy construction."

5.1 Some Heuristics Used in Implementation

The first heuristic we took is treating unions over characters as a literal. Take $(a|b|c|d)$ as an example, we need only one node to represent it. Thompson's original construction will divide it into a, b, c, d, and then combine the NFAs by adding more states, but this can be done by using two states $(0, 1)$ and four transitions $(0, a, 1)$, $(0, b, 1)$, $(0, c, 1)$, and $(0, d, 1)$.

The second heuristic we took is trying to reduce the number of stars by smart parsing algorithm presented in Chapter 2.

The third heuristic we took is treating NEWLINE and EOF the same as ordinary symbol: transitions for NEWLINE and EOF in each state are constructed. This saves the work needed to determine whether the current symbol is NEWLINE or EOF.

5.2 Representation of an NFA

Each NFA consists of a set of states and transitions. Because each reachable state must appear among the transitions at least once, we can represent an NFA by a list of transitions, the start state, and the final state. Based on how the triple (s, c, t) is stored, we have the following three representations:

1. State-Indexed Representation: Those transitions that have the same starting state are stored in the same list. The advantage of this approach is that if there are not many transitions from a state, the storage needed is small, and it is efficient for pattern matching.

2. Alphabet-Indexed Representation: For each $a \in \Sigma \cup \{\epsilon\}$, we have a list of transitions whose label is a. The advantage of this representation is that it is efficient for storage and simulation if most of the characters in the regular expression are unique.

3. State-and-Alphabet-Indexed Representation: This is a multiple-indexed representation. It is indexed by state first, and then indexed by alphabet. The advantage of this representation is that there may be a lot of empty entries for each state and symbol.

5.3 NFA Profile Comparison

In this section, we compare each construction with regard to the number of states and the number of transitions.

Table 5.1: Number of States Comparison

RegEx	MYNFA	TNFA	mTNFA	SNFA
$(\mathtt{a}_1\mathtt{a}_2...\mathtt{a}_\mathtt{n})^*$	$n+1$	$2n+2$	$n+2$	$n+3$
$(\mathtt{a}_1\vert\mathtt{a}_2\vert...\mathtt{a}_\mathtt{n})^\mathtt{m}$	$m+1$	$m(2n-2)$	$m+1$	$m+2$
$((\mathtt{a}_1\vert\epsilon)(\mathtt{a}_2\vert\epsilon)...(\mathtt{a}_\mathtt{n}\vert\epsilon))^*$	$n+1$	$6n+2$	$n+2$	$n+3$
$(\mathtt{a}\vert\mathtt{b})^*\mathtt{a}(\mathtt{a}\vert\mathtt{b})^\mathtt{n}$	$n+3$	$6n+12$	$2n+5$	$n+5$

Table 5.2: Number of Transitions Comparison

RegEx	MYNFA	TNFA	mTNFA	SNFA
$(\mathtt{a_1a_2...a_n})^*$	n	$2n+3$	$n+2$	$n+4$
$(\mathtt{a_1}\|\mathtt{a_2}\|...\mathtt{a_n})^{\mathtt{m}}$	nm	$(5n-1)m-1$	$(m+1)n-1$	nm
$((\mathtt{a_1}\|\epsilon)(\mathtt{a_2}\|\epsilon)...(\mathtt{a_n}\|\epsilon))^{\mathtt{m}}$	nm	$(7n-1)m-1$	$n+2$	$2nm$
$(\mathtt{a}\|\mathtt{b})^*\mathtt{a}(\mathtt{a}\|\mathtt{b})^{\mathtt{n}}$	$2n+3$	$7n+12$	$2n+5$	$2n+6$

5.4 NFA Simulation for Pattern Matching

Comparisons of NFA simulation time between SNFA, mTNFA, TNFA and, and MYNFA are listed in Tables 5.3 and 5.4, where R_1 = `script LANGUAGE="JavaScript" TYPE="text/javascript"`,
R_2 = `(s*c*r*i*p*t*)* LANGUAGE = "JavaScript" T*Y*P*E*=*" t*e*x*t/*j*a*v*a*s*c*r*i*p*t*"`. We use “mm:ss” to represent the time for each simulation. For example, for SNFA simulation on R_1, it takes 1 minute and 16.7 seconds, which is represented as “1:16.7”. The computer we used is a Sun Sparc 5, and the text file used for testing is a $6.7M$ text file.

Table 5.3: NFA Simulation Time Comparison 1

RegEx	SNFA	m TNFA	TNFA	MYNFA
`LANG`	19.1	24.5	30.0	18.7
`LANGUAGE`	25.0	30.2	40.4	24.7
`JavaScript`	26.5	33.2	46.1	25.4
`LAN*G`	21.1	23.7	32.7	19.9
`(LANG)*LANG`	44.9	29.7	1:02.6	42.9
R_1	1:16.7	1:31.4	2:46.5	1:14.7

5.5 Pattern Matching Tools

Using a DFA for pattern matching is more efficient than using an NFA, but constructing a DFA is time consuming and can take as much as exponential time in the size of the NFA

Table 5.4: NFA Simulation Time Comparison 2

RegEx	SNFA	mTNFA	TNFA	MYNFA
`LL*AA*NN*GG*`	28.2	31.8	53.3	24.47
`LL*AA*NN*GG*UU*AA*GG*EE*`	41.6	44.2	1:27.2	37.2
`(JJ*aa*vv*aa*)*JavaScript`	49.9	45.1	1:42.8	48.8
`(LANG)*(LL*AA*N*G*)*UAGE`	56.5	37.5	1:49.8	2:53.4
R_2	2:00.2	1:34.9	5:29.0	30:23.4

we are working from. In [1], Aho reported a highly efficient heuristic: “lazy construction.” When we need to do regular expression matching, we construct an NFA, but we do not construct the DFA explicitly; instead, we construct it as needed: a transition is not stored in the transition table until it is used once. Based on this idea, we have implemented the TNFA construction as part of a text search tool `xgrep`. It is our understanding that “lazy construction” was also used in `grep` and `egrep`. From our experiment, the tool runs a little faster than `egrep`, `grep` for general regular expression pattern. We think the following makes the difference:

1. Our NFA construction is a good choice to work from for DFA construction;
2. Lazy construction is good;
3. The heuristics used in our implementation are good;
4. `grep` and `egrep` are more general and can handle more.

We can not say that this is the result of smart construction of the TNFA, as the lazy construction feature will make the program run like a DFA. But for NFA simulation, it is much faster than the original TNFA, and memory consumption is reduced to half. For large and complicated patterns which can not be handled by `agrep` [9], we think this is a good tool for regular expression matching.

For the experiments, we ran grep tools over P1 to P14 against testing files $t[1]$ to $t[10]$ respectively, where $t[1]$ is a file of size 1.7K, and for each file $t[i+1]$, $size(t[i+1]) = 2size(t[i])$. The computer we used is a Sun Sparc 5, and the time unit is a millisecond.

The patterns we used are listed in Table 5.5.

Table 5.5: RegEx Patterns

P1	GEOR
P2	GEORGIA
P3	GEORGIA S
P4	(GEORGIA)|(FLORIDA)
P5	(GEORGIA S)|(FLORIDA S)
P6	(G|F)(E|L)(O)(R)(G|I)(A|D) S
P7	((GE)|(FL))((OR))((GI)|(ID))((A)|(A))
P8	(GEOR*GIA)|(FLOR*IDA)
P9	(GE*ORGI*A)|(F*LORID*A)
P10	(G*EO*RGI*A)|(F*LO*RID*A)
P11	((GE*)|(FL))((OR*))((GI*)|(ID))((A *)|(A))S
P12	((GE)*|(FL)*)((OR))((GI)|(ID))((A)*|(A))*S
P13	GEORGIA IS A GREAT STATE, so is florida
P14	GEORGIA IS A GREAT STATE, *s*o* *i*s* *f*l*o*r*i*d*a*

Tables 5.6, 5.7, 5.8 list the run times of `xgrep`, `grep`, and `egrep` over different patterns and files respectively.

To see the differences between these tools more clearly, we show the run time comparison between `xgrep`, `grep`, and `egrep` running P1, P7, and P14 over the files of different sizes.

There is another text searching tool named `agrep` developed by Wu in [9], which favors searching for small strings. When the pattern is small and is a plain string, it is about 4 times faster than our tool. But for a normal regular expression, it is just like `grep` or `egrep`. The other disadvantage of `agrep` is that it can not process arbitrary strings and regular expressions.

Table 5.6: xgrep Run Time

File	P1	P2	P3	P4	P5	P6	P7
1	3103	2987	3065	3115	3216	3241	3271
2	3053	3070	3026	3147	3199	3140	3135
4	3168	3131	3154	3229	3267	3261	3250
8	3582	3552	3562	3669	3713	3728	3673
16	4325	4276	4317	4382	4417	4429	4412
32	6108	5865	5927	6001	6091	6051	5991
64	9201	9096	9117	9296	9248	9266	9229
128	15440	15440	15628	16070	15667	15825	15602
256	28325	28443	28524	28651	28334	28467	28269
512	59506	57896	59187	58369	58627	57757	59216
File	P8	P9	P10	P11	P12	P13	P14
1	3230	3307	3228	3214	3197	3474	3751
2	3130	3203	3218	3216	3184	3440	3696
4	3275	3274	3307	3341	3306	3595	3831
8	3733	3734	3748	3748	3708	4002	4220
16	4424	4435	4470	4476	4534	4755	4938
32	5994	6130	6101	6093	6067	6335	6576
64	9250	9299	9219	9337	9362	9658	9689
128	15551	15786	15743	15660	15807	15987	16036
256	28565	28697	28392	28472	28495	28950	28810
512	58633	59718	59469	56660	56844	57321	58881

Observation: If a pattern is a short plain string, `agrep` will be the fastest one, based on shift-or operation. As `xgrep`, `grep`, and `egrep` are all based on lazy construction, the differences are small as DFA construction time is short for small patterns. In this case, the NFA simulation time is a good criterion for evaluating an NFA.

Table 5.7: egrep Run Time

File	P1	P2	P3	P4	P5	P6	P7
1	3042	3015	2989	3092	3005	2967	3052
2	3380	3293	3344	3260	3373	3351	3344
4	3687	3619	3673	3731	3661	3572	3584
8	4273	4169	4138	4133	4187	4148	4191
16	5544	5466	5577	5481	5586	5546	5513
32	8054	7937	7931	7992	7948	8028	7898
64	13531	13452	13448	13397	13434	13399	13331
128	24098	23792	23954	23902	23907	23715	24088
256	45878	45976	46110	45974	46401	45717	46178
512	88535	87797	87722	87650	88127	87556	87981
File	P8	P9	P10	P11	P12	P13	P14
1	3015	3028	3017	2994	3060	3038	3043
2	3289	3297	3306	3352	3340	3404	3410
4	3511	3654	3634	3792	3617	3606	3772
8	4128	4203	4164	4201	4181	4232	4203
16	5514	5538	5491	5502	5445	5564	5536
32	7917	7937	7934	7962	7985	7964	8062
64	13453	13426	13455	13611	13520	13541	13468
128	23778	23967	23972	23816	24009	23985	23936
256	45322	45778	45697	45900	46078	45525	45962
512	87883	87594	87629	88111	88070	87493	87904

Table 5.8: grep Run Time

File	P1	P2	P3	P4	P5	P6	P7
1	3145	2845	2808	2786	2799	2820	2802
2	3347	3012	2933	2893	2912	2912	2903
4	3806	3397	3224	3192	3116	3119	3102
8	4417	3900	3646	3441	3450	3489	3452
16	5531	4997	4551	4220	4203	4199	4245
32	7765	6854	6357	5703	5673	5706	5700
64	13659	11853	10038	8619	8705	8704	8761
128	23701	18993	17841	14904	14975	15030	15204
256	46763	37584	33983	28092	28182	28072	28397
512	93132	74642	69265	56745	55343	56062	56309
File	P8	P9	P10	P11	P12	P13	P14
1	2928	2957	2987	2806	2935	2843	3186
2	3116	3190	3203	2904	3094	2943	3633
4	3518	3634	3731	3101	3521	3184	4547
8	4297	4540	4758	3494	4293	3640	6477
16	5924	6376	6715	4218	5824	4635	9986
32	9162	9972	10785	5784	8924	6457	17221
64	15573	17273	18865	8798	15112	10335	31829
128	28691	32625	35252	15177	27933	18301	61625
256	56124	63048	68607	28312	54799	34340	120994
512	112484	126935	138899	56668	107748	68840	242884

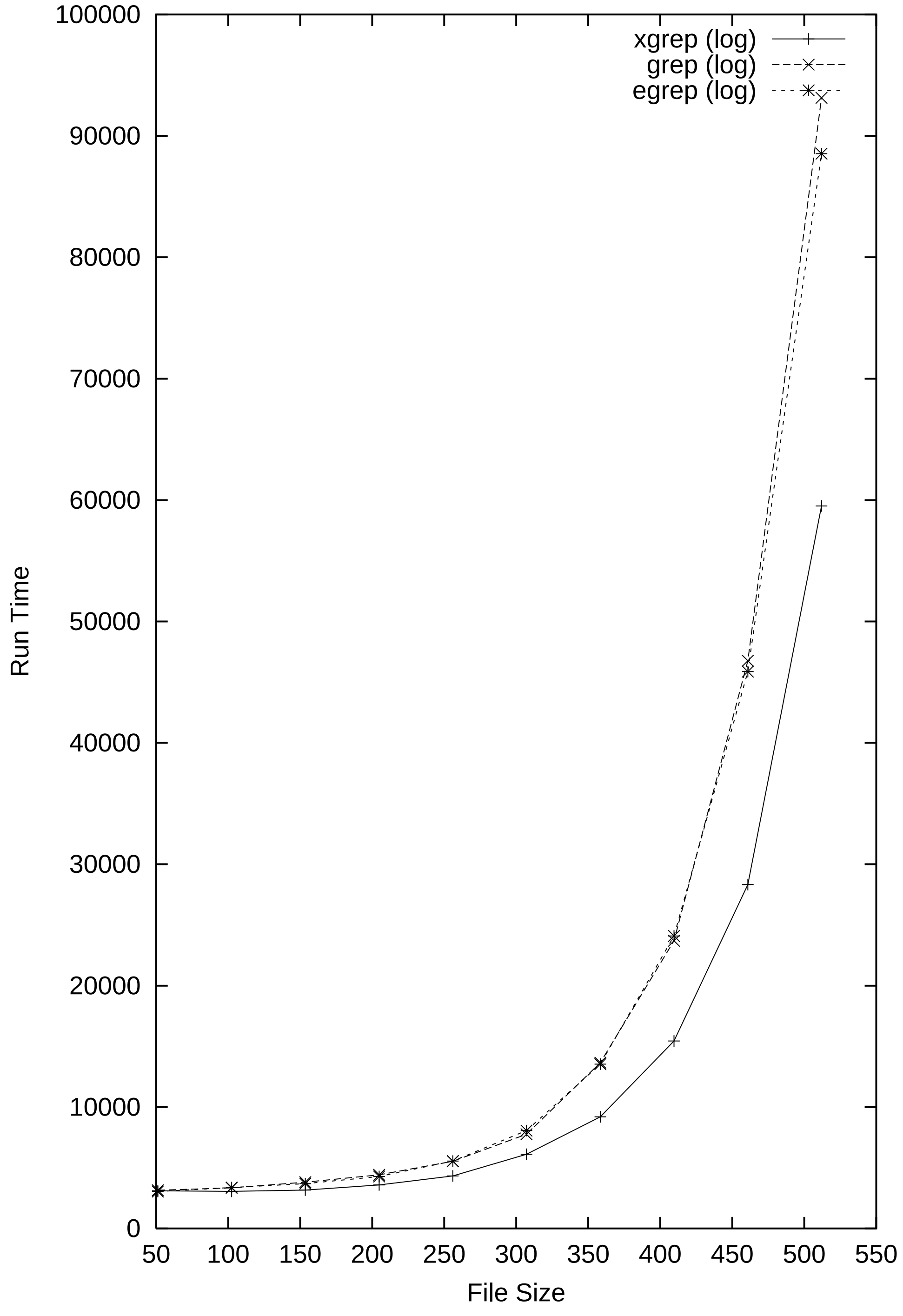

Figure 5.1: Run-time Comparison (P1)

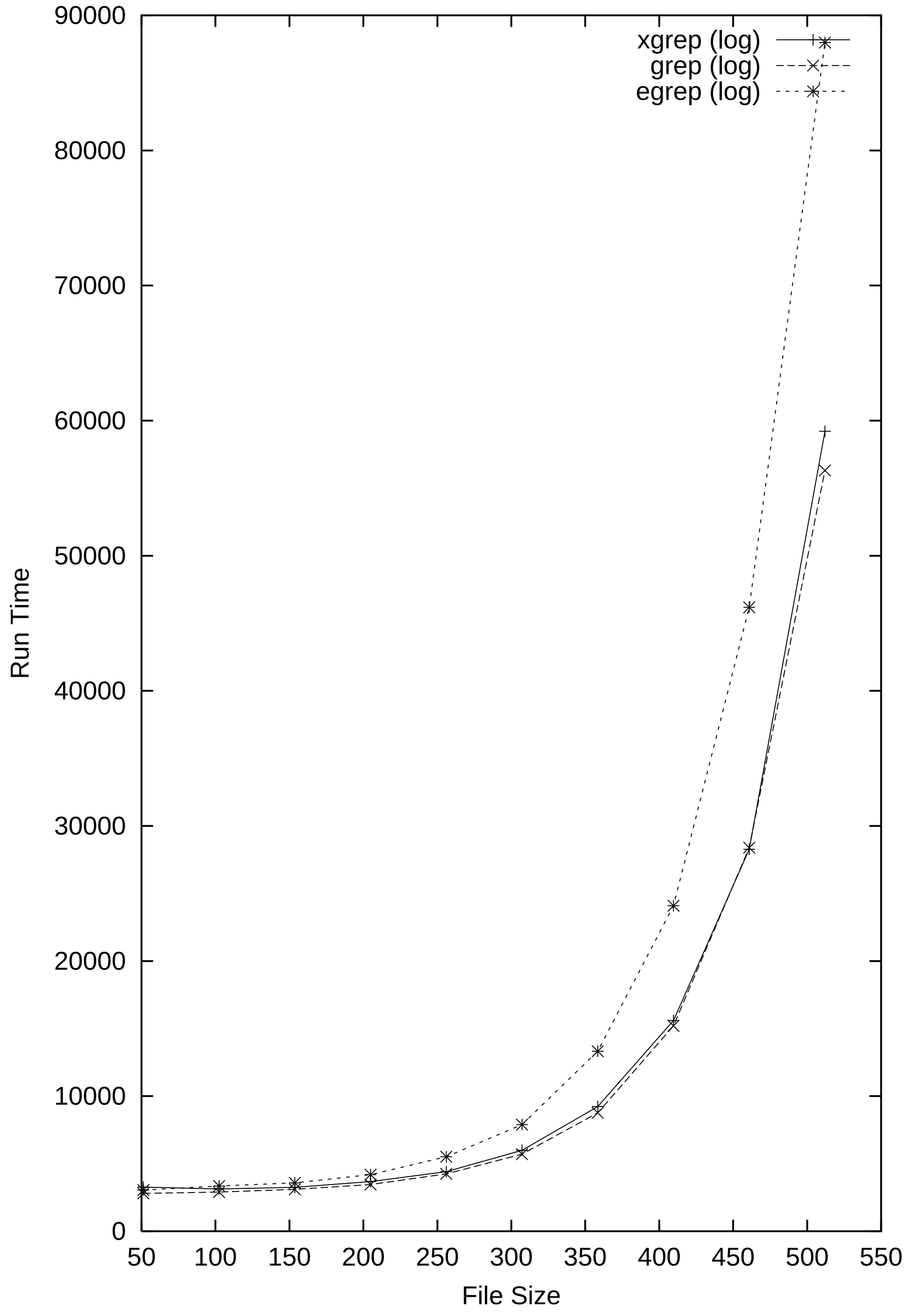

Figure 5.2: Run-time Comparison (P7)

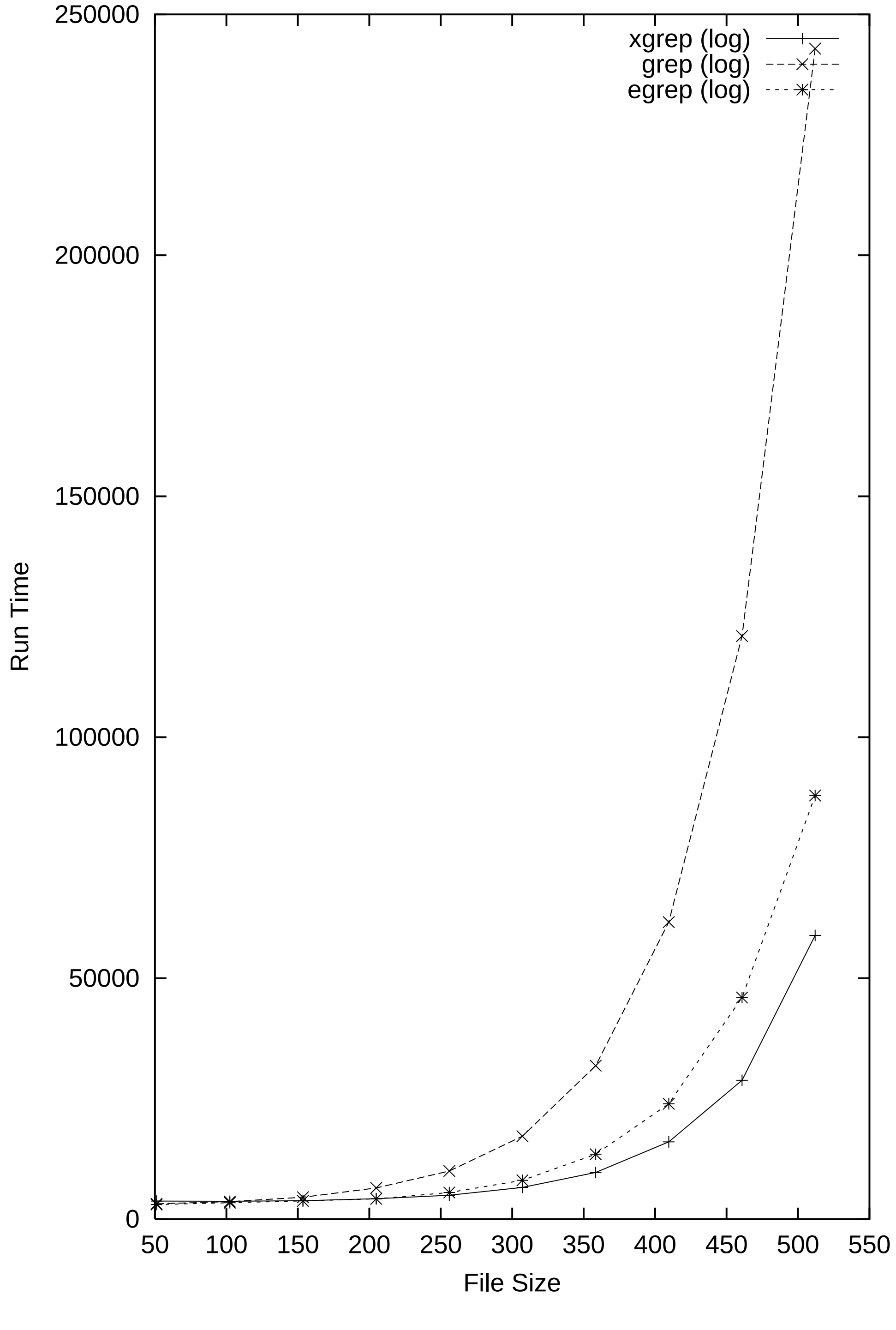

Figure 5.3: Run-time Comparison (P14)

Chapter 6

Emptiness-of-Complement for Semi-Extended Regular Expression is $EXPSPACE-$Complete

In this chapter, we show that the emptiness-of-complement problem for semi-extended regular expression is $EXPSPACE-$complete from definition[1]. A semi-extended regular expression is a generalized regular expression with intersection. Although intersection does not increase the power of expressing (the language denoted by a semi-extended regular expression is regular), it will make the expression shorter in most cases.

6.1 Notations and Definitions

A *semi-extended regular expression* over an alphabet I is defined as follows [1]:

1. ϵ, ϕ and a for each $a \in I$ are semi-extended regular expressions denoting $\{\epsilon\}$, the empty set, and $\{a\}$ respectively.

2. If R_1, R_2 are semi-extended regular expressions denoting the languages L_1, L_2 respectively, then $(R_1 + R_2)$, $(R_1 R_2)$, (R_1^*) and $(R_1 \cap R_2)$ are semi-extended regular expressions, denoting $L_1 \cup L_2$, $L_1 L_2$, L_1^*, and $L_1 \cap L_2$, respectively.

3. All semi-extended regular expressions can be generated by above rules.

[1]At the time of writing this chapter, I did not know, this is a known result presented in [6], which was told to me by Charles Rackoff at Toronto. So the purpose of this chapter is showing our proof, not claiming a new result. At the end of this chapter, I will indicate the differences between my proof and the proof by Fürer in [6].

In other words, semi-extended regular expressions are generalized regular expressions with intersection.

For convenience, we will use $a^n = \underbrace{aa...a}_{n\ times}$ and $a^+ = aa^*$ to describe the expressions more compactly. Also, we use *expression* to mean semi-extended regular expression unless otherwise noted.

We use Sipser's[5] definition of a one-tape deterministic Turing Machine as a $7-$tuple:

$$M = (Q, I, \Gamma, \delta, q_0, q_{accept}, q_{reject}).$$

All Turing Machines considered in this paper halt on all inputs, in either state q_{accept} or q_{reject}. For input string $w \in I^*$, we define $space(M, w)$ to be the number of tape squares scanned by the R/W head of M as it processes string w. For a function $f : N \to N$, as in [5], we define the space complexity class $SPACE(f(n))$ by

$$SPACE(f(n)) = \{L \subseteq I^* : \exists M\ L = L(M)\ and\ for\ all\ w \in I^*\ space(M, w) \in O(f(|w|))\}$$

Formally, we say that a language $L_1 \subseteq I_1^*$ is ***polynomial-time reducible*** to a language $L_2 \subseteq I_2^*$, written as $L_1 <_P L_2$, if there exists a polynomial-time computable function

$$f : I_1^* \to I_2^*$$

such that for all $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2.$$

We call the function f the *reduction function*, and the polynomial-time algorithm F that computes f the *reduction algorithm*.

The space complexity class $EXPSPACE$ is defined as:

$$EXPSPACE = \bigcup_{k=1}^{\infty} SPACE(2^{m^k})$$

where m is the length of the input.

A language A is said to be $EXPSPACE-$*complete* if A belongs to $EXPSPACE$ and for all languages $B \in EXPSPACE$, $B <_P A$.

6.2 Preliminary Results

In many cases, semi-extended regular expressions provide a more compact representation than regular expressions. For example, given a regular expression, we can find a string that is accepted by the regular expression by scanning it, which can be done in linear time. But for semi-extended regular expressions, this is not the case because the least length string that is accepted by a semi-extended regular expression can be of exponential size with respect to the size of the semi-extended regular expression.

The emptiness-of-complement problem for a language is to determine if the complement of the language is empty. This problem was discussed in [1, 13], and it is shown in [4] that the problem for regular expressions is $PSPACE$-complete.

Lemma 4 *[4] The emptiness-of-complement problem for NFAs is $PSPACE$-complete.*

Lemma 5 *We can construct an NFA from a semi-extended expression in $O(2^n)$ space where n is the length of the semi-extended regular expression.*

Proof It is easy to prove this by induction on the length of the semi-extended regular expression. □

From the above two lemma, we know that the emptiness-of-complement problem for semi-extended regular expressions can be solved in $EXPSPACE$. To prove it is $EXPSPACE$-complete, we only need to show that for any problem solvable in $EXPSPACE$, it can be reduced to the emptiness-of-complement problem for semi-extended regular expressions.

6.3 Main Result

We state our main theorem as:

Theorem 12 *The emptiness-of-complement problem for semi-extended regular expressions is $EXPSPACE-$hard.*

Proof Using the ideas in [5, 3], we give the following reduction.

Let $A \in EXPSPACE$ be a language that is decided by a Turing Machine M. There exists k, such that for each input w with length m, M runs in space 2^{m^k} (By definition, this should be $2^{O(m^k)}$. It is easy to convert 2^{cm^k} to the form of 2^{m^k} by choosing a larger k.) Let $n = m^k$; we know that the machine will use at most 2^n tape cells.

Let Γ' be a set of symbols disjoint from Γ with

$$|\Gamma'| = |Q| \times |\Gamma|.$$

Each symbol of Γ' is understood to correspond in a one-to-one manner with a pair $(q, A) \in Q \times \Gamma$.

Recall that a *configuration* is a string of the form $\Gamma^*\Gamma'\Gamma^*$. It shows the current tape contents, the current state, and the currently scanned symbol. One configuration *yields* another if the second can be obtained from the first by one legal move of the Turing machine. The string $w = a_0a_1 \cdots a_{m-1}$ is rejected by machine M if and only if there is a sequence of configurations

$$C_0, C_1, \ldots, C_\ell$$

such that

(i.) $C_0 = (q_0, a_0)a_1 \cdots a_{k-1}B \cdots B$

(ii.) C_i yields C_{i+1} for $0 \le i < \ell$

(iii.) C_ℓ contains the rejecting state q_{reject}.

Let $\Delta = \Gamma + \Gamma'$ and assume $0, 1 \notin \Delta$. We will define a way in which certain strings over $\Delta + 0 + 1$ can be interpreted as sequences of configurations. Then we will give a semi-extended regular expression E such that $L(E)$ equals all strings which are not rejecting computations. Thus

$$w \in L(M) \iff L(E) = (\Delta + 0 + 1)^*.$$

This will complete our proof.

To proceed, define a *block* to be a string in the language of $(0 + 1)^n\Delta$, that is, an n-bit integer in the range $0 \le i < 2^n$ followed by a symbol of Δ. A *coded potential configuration* (cpc) is a concatenation of blocks, that is, a string in the language of

$$((0 + 1)^n\Delta)^*,$$

which contains every n-bit binary number in the range $0 \le i < 2^n$ exactly once and in the right order. Thus, one way to obtain a cpc is to start with a legitimate configuration of M having length 2^n, and insert n-bit counters, in proper sequence, before each symbol. As we have defined it, however, removing all the 0's and 1's from a cpc may not necessarily leave a configuration of M, because there is nothing in our definition that says exactly one state appears. This is why we call them coded *potential* configurations.

A sequence of cpc's $\hat{C}_0, \hat{C}_1, \ldots \hat{C}_\ell$ will represent a rejecting computation of M on w provided

(i.) $\hat{C}_0$ encodes the initial configuration $(q_0, a_0)a_1 \cdots a_{k-1}B^{2^n-k}$,

(ii.) $\hat{C}_{i+1}$ encodes a legal successor configuration to the configuration encoded by $\hat{C}_i$ for all $0 \le i < l$, and

(iii.) $\hat{C}_\ell$ encodes a rejecting configuration.

We will write out a semi-extended regular expression over $\Sigma = \Delta + 0 + 1$

$$E = E_{non-block} + E_{bad-index} + E_{bad-start} + E_{bad-yield} + E_{no-reject},$$

which generates every string that is not a correct encoding of a rejecting computation.

A string fails to be a correct encoding of a rejecting computation in two ways:

1. It is not in the correct format of encoding (it is not a sequence of blocks), which will be generated by $E_{non-block}$;

2. It is a sequence of blocks, but we can't get a valid rejecting computation from the encoding. That is, the string violates at least one condition about a valid encoding of a rejecting computation. Such strings are generated by

   $$E_{bad-index} + E_{bad-start} + E_{bad-yield} + E_{no-reject}.$$

   Each expression will be explained in more detail later.

We want $L(E_{non-block})$ to be the complement of the language of

$$F_1 = ((0+1)^n\Delta)^+$$

There are three ways a string x may fail for membership in $L(F_1)$: (1) x does not start with 0 or 1 and end with a symbol in Δ; (2) x contains a consecutive pair of symbols from Δ; or (3) x contains a run of $0's$ and $1's$ whose length is different from n. The expression

$$F_2 = \epsilon + \Delta\Sigma^* + \Sigma^*(0+1) + \Sigma^*\Delta^2\Sigma^*$$

will generate errors of type (1) and (2), and the expression

$$F_3 = \Sigma^*\Delta(\epsilon + (0+1) + \cdots (0+1)^{n-1} + (0+1)^{n+1}(0+1)^*)\Delta\Sigma^*,$$

will generate errors of type (3). We have completed the first part of our proof:

$$E_{non-block} = F_2 + F_3.$$

For a sequence of *blocks*, it fails to be a cpc for rejecting computation in at least one of the following four ways:

1. The *index error*: the index in one block does not follow the previous one or the index of the first block is not 0^n or the index of the last block is not 1^n. These errors are generated by $E_{bad-index}$;

2. The initial configuration is not correct, generated by $E_{bad-start}$;

3. One configuration does not properly lead to the next configuration, generated by $E_{bad-yield}$;

4. There is no rejecting configuration, generated by $E_{no-reject}$.

Now we will turn to the construction of $E_{bad-index}$ that will generate all strings which contain an *indexing error*. That is, we want $E_{bad-index}$ to generate all strings which contain a substring of the form

$$b\Delta b'$$

where b and b' are $n-$bit binary numbers, but

$$(1) \qquad b' \neq (b+1) \textbf{ mod } 2^n,$$

or (2) the index for the first block is not 0^n or the index for the last block is not 1^n.

For the first case, suppose $b = b_n ... b_{i+2} 0 \underbrace{1...1}_{i}$. We know the correct form for b' should be $b' = b_n ... b_{i+2} 1 \underbrace{0...0}_{i}$. Now, let us further classify the errors as:

1. The error is from $(1..i)-$bit

2. The error is from $(i+1)-$bit

3. The error is from $(i+2..n)-$bit

The following three regular expressions will detect the errors for each case:

$$B_{1,i} = (0+1)^{n-i-1} 01^i \Delta (0+1)^{n-i-1} 1 (0+1)^* 1 (0+1)^*, 1 \leq i \leq n-1$$

$$B_{2,i} = (0+1)^{n-i-1} 01^i \Delta (0+1)^{n-i-1} 0 (0+1)^i, 0 \leq i \leq n-1$$

$$B^b_{3,i} = \sum_{j=0}^{n-i-2} (0+1)^j b (0+1)^{n-i-j-2} 01^i \Delta (0+1)^j \overline{b} (0+1)^*, 0 \leq i \leq (n-2), b \in \{0,1\}$$

take

$$B_i = B_{1,i} + B_{2,i} + B^0_{3,i} + B^1_{3,i}$$

For the second case, consider the regular expression

$$F_4 = (\Sigma - 0) + 0(\Sigma - 0) + 0^2(\Sigma - 0) + \cdots + 0^{n-1}(\Sigma - 0) + 0^n(0 + 1).$$

Scanning a string $x \in L(F_4)$ left to right, we encounter a symbol where we may definitely conclude that x does not begin with a prefix of the form $0^n\Delta$. Thus,

$$F_5 = \epsilon + 0 + 0^2 + \cdots + 0^n + F_4\Sigma^*$$

generates all strings which fail to be a cpc because they don't begin correctly. Similarly, with

$$F_6 = (\Sigma - 1) + (\Sigma - 1)1 + \cdots (\Sigma - 1)1^{n-1} + (0 + 1)1^n$$

we have that

$$F_7 = \Sigma^* F_6 \Delta$$

generates all strings which fail to be a cpc because they don't end correctly. Finally, let

$$E_{bad-index} = \Sigma^*(\sum_{i=0}^{n} B_i)\Sigma^* + F_5 + F_7$$

Clearly, $E_{bad-index}$ generates all strings that contain an indexing error.

For $E_{bad-start}$, recall $w = a_0a_1, , , a_m$ is the input. We know that a bad start configuration must take one of three forms:

1. It does not start with (q_0, a_0), which can be generated by:

$$F_{q_0} = (0 + 1)^n(\Delta - (q_0, a_0))((0 + 1)^n\Delta)^*$$

2. It does not put the a_i for $i = 1..m - 1$ in the correct place, which can be generated by:

$$F_{a_i} = ((0 + 1)^n\Delta)^i(0 + 1)^n(\Delta - a_i)((0 + 1)^n\Delta)^*$$

3. It does not place blanks in the rest of the tape, which can be generated by:

$$F_{blank} = ((0 + 1)^n\Delta)^m\Big(((0 + 1)^n - 1^n)\Delta\Big)^*(0 + 1)^n(\Delta - \sqcup)((0 + 1)^n\Delta)^*$$

Taking

$$E_{bad-start} = F_{q_0} + \sum_{i=1}^{m-1} F_{a_i} + F_{blank},$$

generates all strings that have an invalid initial cpc.

We can have a string without rejecting state to represent no rejecting configuration as in the following:

$$E_{no-reject} = ((0+1)^n(\Delta - (\{q_{reject}\} \times \Gamma))^*$$

which is illustrated by 6.1.

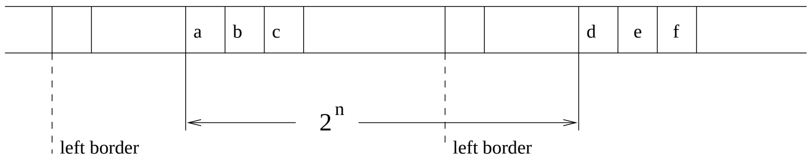

Figure 6.1: The yield relation in two adjacent configurations

Finally, we construct $E_{bad-yield}$ to generate all strings whereby one configuration does not properly lead to the next configuration. One configuration legally yields another whenever every three consecutive symbols in the first configuration correctly yield the *corresponding* three symbols in the second configuration. Two symbols are *corresponding* to each other if they are 2^n apart. Define the relation $\text{GOOD} \subseteq \Delta^3 \times \Delta^3$ to consist of those pairs (abc, def) such that if abc appears in three consecutive locations of a valid configuration, then def is an acceptable triple to appear in the corresponding window of the succeeding configuration.

If all three of a, b, and c belong to Γ, (that is, none is of the form (q,a)), then we can see that the possibilities for def are:

$$(abc, abc) \quad (abc, ab(q,c)) \quad (abc, (q,a)bc).$$

Thus, GOOD contains all $|\Gamma|^3(1+2|Q|)$ pairs of the latter form. The other pairs belonging to GOOD have one symbol of the form $(q,a) \in \Gamma'$ in the first triple. Specifically, for each left-moving transition of the machine

$$\delta(q,a) = (q', a', L)$$

GOOD contains the $3|\Gamma|^2$ pairs of the form

$$((q,a)bc, a'bc) \quad (b(q,a)c, (q',b)a'c) \quad (bc(q,a), b(q',c)a')$$

where b and c vary freely over Γ. Similarly, for each right-moving transition of the machine

$$\delta(q,a) = (q', a', R)$$

GOOD contains the $3|\Gamma|^2$ pairs

$$((q,a)bc, a'(q',b)c) \quad (b(q,a)c, ba'(q',c)) \quad (bc(q,a), bca').$$

These three classes constitute all the pairs belonging to GOOD.

Defining BAD to be the complement of GOOD within $\Delta^3 \times \Delta^3$, we have an ordered pair $(C_1, C_2) \in \Delta^{2^n} \times \Delta^{2^n}$ such that C_1 is a valid configuration fails to satisfy the *yields* relation if and only if there is a triple abc of consecutive symbols in C_1 such that the corresponding three positions in C_2 are occupied by a triple def with $(abc, def) \in$ BAD.

The next step in the construction of $E_{bad-yield}$ is to devise a way to generate all strings of the form

$$IaJbKcSIdLeMf$$

in which

1. $I, J, K, L, M \in L((0+1)^*)$;
2. $I \in L((0+1)^n)$, and we call attention to the repetition of I;
3. $(abc, def) \in BAD$;
4. S is a string in the language of $((0+1)^*\Delta)^*$ which contains one of the binary numbers 0^n or 1^n exactly once.

It is the fourth condition that assures us the error will take place in consecutive `cpc`'s, given that the string in question does not contain any of the errors already generated by $E_{non-block}$, $E_{bad-index}$, or $E_{bad-start}$.

Let ϵ denote a bit, either 0 or 1, and consider the regular expression

$$\begin{aligned} F^{\epsilon}_{same-i} \quad = \quad & (0+1)^{i-1}\epsilon(0+1)^{n-i}\Delta \\ & ((0+1)^*\epsilon(0+1)^*\Delta)^* \\ & \overline{\epsilon}^n\Delta \\ & ((0+1)^*\epsilon(0+1)^*\Delta)^* \\ & (0+1)^{i-1}\epsilon(0+1)^{n-i}\Delta \\ & ((0+1)^n\Delta)^2. \end{aligned}$$

If w is a string generated by F^{ϵ}_{same-i}, then the first n-bit number in w, and the third binary number from the end of of w, both have their i-th bit from the left ($1 \leq i \leq n$) equal to ϵ; all binary numbers in the string w between these two contain the bit ϵ, except for exactly one, which is $\overline{\epsilon}^n$. Consequently, if we define

$$\begin{aligned} G_{abc,def} \quad = \quad & (0+1)^*a(0+1)^*b(0+1)^*c \\ & ((0+1)^*)^* \\ & (0+1)^*d(0+1)^*e(0+1)^*f \end{aligned}$$

and

$$F_{same-i} \quad = \quad F^0_{same-i} \,\cup\, F^1_{same-i},$$

it follows that

$$F_{abc,def} \quad = \quad G_{abc,def} \,\cap\, \bigcap_{i=1}^{n} F_{same-i}$$

generates precisely the strings satisfying conditions (i) through (iv) above. We may thus take

$$E_{bad-yield} \quad = \quad (\Delta+0+1)^* \,(\sum_{(abc,def)\in \mathrm{BAD}} F_{abc,def})\,(\Delta+0+1)^*.$$

Obviously, each semi-extended regular expression is bounded by a polynomial, so the above construction can be carried out in polynomial time, which completes the proof.

□

Based on the above theorem and Lemmas 5.1 and 5.2, we have the following corollary:

Corollary 1 *The emptiness-of-complement problem for semi-extended regular expression is $EXPSPACE-$complete.*

6.4 Fürer's Proof

Fürer's proof is sketched in this section.

Let M be a nondeterministic $2^n - 2$ space-bounded Turing Machine. We choose another code for the sequence of subsequent ID's of M, and call this code a computation of M. For convenience, use $\&$ to specify the start of an ID and $\#$ to specify the end of an ID.

Let $ID_1, ID_2, ...ID_k$ be the sequence of subsequent instantaneous descriptions of the Turing Machine M, where ID_1 is the initial ID corresponding to the input x and ID_k is an accepting ID. We define a_{ij} to be the $j-$th symbol in $ID_i(i = 1, 2, ..., k; j = 1, 2, .., 2^n - 1)$.

Let $[j]$ be the marked binary number j of length n and $[j]^R$ be the reversed word of it. They define the word:

$$\&[0]^R\#[0]\&[1]^R a_{1,1}[1]\&[2]^R a_{1,2}[2]\&[2^n-1]^R a_{1,2^n-1}[2^n-1]$$
$$\&[0]^R\#[0]\&[1]^R a_{2,1}[1]\&[2^n-1]^R a_{k,2^n-1}[2^n-1]\&[0]^R\#[0]\&$$

There are ten cases of mistakes:

1. Two symbols with distance $n+1$ do not match;
2. The word does not start correctly;

3. The word does not end correctly;
4. A sub-word of length $\leq 2n+1$ with an a_{ij} or $\#$ in the middle is not symmetric;
5. A binary number is not correctly marked;
6. A sub-word of length $2n+1$ with $\&$ in the middle is not of the form $[j]\&[j+1]^R$ with $0 \leq i \leq 2^n - 1$ and addition mod 2^n;
7. $\#$ or $[0]$ appear other than as the pair $\#[0]$;
8. The initial ID is wrong;
9. A computation step is wrong;
10. No ID is accepting.

Comment:

From the above proof, we see that the two proofs are similar: both use a semi-extended regular expression to encode a computation. The difference is that Fürer uses “Reverse” to make detecting the successor relation easier, while we are using cpc to make the proof simpler and easier to understand.

6.5 Does $EXPTIME = EXPSPACE$?

According to exercise 11.15 in [1], deciding emptiness of complement for semi-extended regular expressions can be decided in deterministic exponential time. If this is correct, and if our proof above is correct, then it follows that $EXPTIME = EXPSPACE$. As of this time, however, we have been unable to find a solution to Exercise 11.15.

6.6 References

[1] A. Aho, J. Hopcroft and J. Ullman *The Design and Analysis of Computer Algorithms* Addisan-Wesley, Reading, Mass. (1974).

[2] T. Jiang and B. Ravikumar *A note on the space complexity of some decision problems for finite automata* Information Processing Letters 40, 25-31, (1991).

[3] D. Calvanese, G.D. Giacomo, M. Lenzerini and M.Y. Vardi *Containment of conjunctive regular path queries with inverse* Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000), 176-185, Breckenridge, Colorado, (2000).

[4] S. Yu *Regular Languages*, Handbook of Formal Languages Vol.1(G. Rozenberg and A. Salomma, eds.) Springer-Verlag, Berlin, (1997).

[5] M. Sipser *Introduction to the Theory of Computation* PWS, Boston, Mass. (1996).

[6] M. Fürer *The Complexity of the Inequivalence Problem for Regular Expressions with Intersection*, Automata, Languages and Programming, 7th Colloquium, Lecture Notes in Computer Science, Vol.85, 234-245, Springer-Verlag, 14-18 (1980).

Chapter 7

Conclusion

We propose two methods for NFA construction as better alternatives for the classical Thompson's NFA and McNaughton and Yamada's NFA. Theoretical analysis and experimental results showed that our proposed NFAs lead to a substantially more efficient way of turning regular expressions into NFA's and DFA's than do other approaches in current use.

Our benchmark result confirms the theoretical analysis that the NFA is smaller but faster than both Thompson's original construction and McNaughton and Yamada's construction. Thompson's NFA contains redundant states and edges. In [7], he said to optimize Thompson's NFA, deep global analysis is often needed. From our knowledge, our algorithm is the first one that solves the minimization problem for Thompson's NFA such that no states can be deleted without the violation of the defining property of Thompson's NFA. In contrast, the second method is efficiently constructed by a simple method, and can be regarded as an optimized Thompson's NFA.

We also proved that the emptiness-of-complement problem for semi-extended regular expression is $EXPSPACE-$complete. And we posed a question asking whether $EXPSPACE = EXPTIME$ based on an exercise in [1]. If it turns out to be true, it will be the first equivalence between time complexity class and space complexity class.

In conclusion, we propose the following questions for future research:

1. NFA minimization in general is $PSPACE$-hard. If, however, we mark some subset of states in the NFA as undeletable, can we minimize it in P(like in this thesis, we mark transition states as undeletable)?

2. For a regular expression r, let

$$Opt_{cost}(r) = \min\{\#\ cost(N) : N\ is\ an\ NFA\ and\ L(N) = L(r)\}.$$

The question is what kind of cost function will make the optimization constant approximable, that is $cost(N) \leq \alpha Opt_{cost}(r)$. For example, if we let the $cost(N) = \#\ states\ in\ N$, it is unlikely such approximation algorithm exists, because this is a $PSPACE$-hard problem.

3. Investigate the relation between the compact TNFA defined here and the CNNFA constructed by Chang in [7]. For example, is there a a succinctly described algorithm that transform one kind of NFA to another?

4. Note that that there may be several minimized TNFAs, but such NFAs need not have the smallest possible number of states, can we get the minimum, among all the minimized TNFAs in polynomial time?

5. If we allow both intersection and binary counter in semi-extended regular expression, is the emptiness-of-complement problem even harder?

Bibliography

[1] A. Aho, J. Hopocroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

[2] A. Aho and J. Ullman *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Englewood Cliffs, N.J (1972).

[3] A. Aho *Pattern Matching in Strings* Formal Language Theory(R. Book, ed.) Academic Press. (1980).

[4] S. Yu, *Regular Languages* Handbook of Formal Languages Vol.1(G. Rozenberg and A. Salomma, eds.) Springer-Verlag, Berlin, (1997).

[5] R. McNaughton and H. Yamada, *Regular Expressions and State Graphs for Automata*, Trans. IRS EC-9, 39-47, (1960).

[6] K. Thompson *Regular Expression Search Algorithm*, Communications of the ACM 11:6, 410-422, (1968).

[7] C.H. Chang *Regular Expressions to DFA's using Compressed NFA's*, Ph.D Thesis, Department of Computer Science, Courant Institute, New York, (1992).

[8] M.O. Rabin, D. Scott *Finite Automata and their decision problems* IBM J. Res 3:2, 115-125, (1959).

[9] S. Wu and U. Manber *Agrep: a fast approximate pattern matching tool* Proceedings of USENIX Winter 1992 Technical Conference, San Francisco, CA, 153-162, (1992).

[10] G. Berry and R. Sethi *From Regular expressions to Deterministic Automata* Theoretical Computer Science, 48, 117-126, (1986).

[11] G. Berry, L. Cosserat *The Esterel Synchronous Programming Language and its Mathematical Semantics* Seminar in Concurrency, S. D. Brookes, A. W. Roscoe, and G. Winskel, eds., LNCS 197, Springer-Verlag, (1985).

[12] E. Meyer, *A Four-Russians Algorithm for Regular Expression Pattern Matching*, Journal of ACM, vol.39:2, 432-448 ,(1992).

[13] T. Jiang, B. Ravikumar *A note on the space complexity of some decision problems for finite automata* Information Processing Letters 40, 25-31, (1991).

[14] C. Clark, G. Cormack *On the Use of Regular Expressions for Searching Text* ACM Trans. on Programming Languages and Systems, 19:3 413-426, (1997).

[15] G. Apostolopoulos, V. Peris, P. Pradhan, and D. Saha *L5: A self learning layer 5 switch* Technical Report RC21461, IBM, T.J. Watson Research Center, (1999).

[16] D. Ritchie, K. Thompson *The UNIX Time-Sharing System* Communications of the ACM, 17:7, 365-375, (1974).

[17] J. Myhill *Finite Automata and Representation of Events* WADC, Technical Report 57-624, (1957).

[18] A. Nerode *Linear Automaton Transformations* Proc. Amer. Math Soc., Vol9, 541-544, (1958).

[19] J. Hopcroft *An $n \log n$ Algorithm for Minimizing states in a Finite Automata* Theory of Machines and Computation, ed. Kohavi and Paz, Academic Press, New York, 189-196, (1971).

[20] A. Brüggemann-Klein *Regular Expressions into Finite Automata*, Theoretical Computer Science 120:2, 197-213, (1993).

[21] J. Ullman *Computational Aspects of VLSI*, Computer Science Press, Rockville, MD, (1984).