From Regular Expressions to DFA's Using Compressed NFA's

by

Chia-Hsiang Chang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Computer Science Department
Courant Institute of Mathematical Sciences
New York University
October, 1992

Approved: Robert A. Paige

To my parents and uncle Frank

ACKNOWLEDGMENTS

I would like to express the greatest gratitude to my advisor Professor Robert A. Paige for his encouragement and patience with me through all these years. During these long years of my study, Professor Paige guided my research direction, contributed with fresh ideas and solutions, and helped to improve the presentation of the material. I thank Professor Chee. K. Yap and Professor Jiazhen Cai for many fruitful discussions. I also thank Dr. Janming Ho for his encouragement. Finally, I thank my wife Amy for her long time support; without her, this thesis would have been impossible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The growing importance of regular languages and their associated computational problems in languages and compilers is underscored by the granting of the Turing Award to Rabin and Scott in 1976, in part, for their ground breaking logical and algorithmic work in regular languages [25]. Of special significance was their construction of the canonical minimum state DFA that had been described nonconstructively in the proof of the Myhill-Nerode Theorem[20,21]. Rabin and Scott's work, which was motivated by theoretical considerations, has gained in importance as the number of practical applications has grown. In particular, the construction of finite automata from regular expressions is of central importance to the compilation of communicating processes[4], string pattern matching[3,19], approximate string pattern matching[32], model checking[10], lexical scanning[2], and VLSI layout design[31]; unit-time incremental acceptance testing in a DFA is also a crucial step in LR_k parsing[17]; algorithms for acceptance testing and DFA construction from regular expressions are implemented in the UNIX operating system[26].

Throughout this thesis our model of computation is a uniform cost sequential RAM [1]. We report the following six results.

1. Berry and Sethi[5] use results of Brzozowski[8] to formally derive and improve McNaughton and Yamada's algorithm[18] for turning regular expressions into NFA's. NFA's produced by this algorithm have fewer states than NFA's produced by Thompson's algorithm[30], and are believed to outperform Thompson's NFA's for acceptance testing. Berry and Sethi's algorithm has two passes and can easily be implemented to run in time $\Theta(m)$ and auxiliary space $\Theta(r)$, where r is the length of the regular expression, and m is the number of edges in the NFA produced. More recently, Brüggemann-Klein[6] presents a two-pass algorithm to compute McNaughton and Yamada's NFA using the same resource bounds as Berry and Sethi. We present an algorithm that computes the same NFA in the same asymptotic time $\Theta(m)$ as Berry and Sethi, but it improves the auxiliary space to $\Theta(s)$, where s is the number of occurrences of alphabet symbols appearing in the regular expression.

2. One disadvantage of McNaughton and Yamada's NFA is that its worst

case number of edges is $m = \Theta(s^2)$. If $s_{()}$ is the number of occurrences of parentheses (right or left), then Thompson's NFA only has between $r - s_{()} + 1$ and $2r$ states and between $r - s_{()}$ and $4r - 3$ edges. We introduce a new compressed data structure, called the CNNFA, that uses only $\Theta(s)$ space to represent McNaughton and Yamada's NFA. The CNNFA can be constructed from a regular expression R in $\Theta(r)$ time and $O(s)$ auxiliary space.

3. Our main theoretical result is a proof that the CNNFA can be used to compute the set of states U one edge away from an arbitrary set of states V in McNaughton and Yamada's NFA in optimal time $O(|V| + |U|)$. The previous best worst-case time is $O(|V| \times |U|)$. This is the essential idea that explains the superior performance of the CNNFA in both acceptance testing and DFA construction.

4. For regular expression R with s alphabet symbol occurrence, the CNNFA M_R has no more than $5s/2$ states and $(10s-5)/2$ edges. CNNFA M_R has no more states or edges than Thompson's machine for R.

5. We give empirical evidence that our algorithm for NFA acceptance testing using the CNNFA outperforms competing algorithms using either Thompson's or McNaughton and Yamada's NFA. We give more dramatic empirical evidence that constructing a DFA from the CNNFA can be achieved in time one order of magnitude faster than the

classical Rabin and Scott subset construction (cf. Chapter 3 of [2]) starting from either Thompson's NFA or McNaughton and Yamada's NFA. Our benchmarks also indicate better performance using Thompson's NFA over McNaughton and Yamada's NFA for acceptance testing and subset construction. This observation runs counter to the judgment of those using McNaughton and Yamada's NFA throughout UNIX.

6. A UNIX egrep compatible software called cgrep based on the CNNFA is implemented. Our benchmark shows that cgrep is significantly faster than both UNIX egrep and GNU e?grep, which are two popular egrep implementations currently in use.

The next section presents standard terminology and background material. In Chapter 2, we reformulate McNaughton and Yamada's algorithm from an automata theoretic point of view, and describe a new algorithm to turn regular expressions into McNaughton and Yamada's NFA's. In Chapter 3 we show how to construct and optimize the CNNFA. Analysis of the CNNFA is presented in Chapter 4. Chapter 5 discusses experimental results showing how the CNNFA compares with other NFA's in solving acceptance testing and DFA construction. The benchmark result for cgrep and competing softwares is also in Chapter 5. Chapter 6 mentions more optimization tactics. In Chapter 7, we summarize our results and present future research directions.

1.1 Terminology

With few exceptions the following basic definitions and terminology can be found in [2,14]. By an *alphabet* we mean a finite nonempty set of symbols. If Σ is an alphabet, then Σ^* denotes the set of all finite strings of symbols in Σ. The empty string is denoted by λ. If x and y are two strings, then xy denotes the concatenation of x and y. Any subset of Σ^* is a *language* over Σ.

Definition 1.1 Let L, L_1, L_2 be languages over Σ. The following expressions can be used to define new languages.

- $\emptyset$ denotes the empty set
- $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$ denotes *product*
- $L^0 = \{\lambda\}$ if $L \neq \emptyset$; $\emptyset^0 = \emptyset$
- $L^{i+1} = LL^i$, where $i \geq 0$
- $L^* = \cup_{i=0}^{\infty} L^i$
- $L^T = \{x : ax \in L | a \in \Sigma\}$ denotes the *tail* of L

In later discussions we will make use of the identities below, which follow directly from the preceding definition.

$$L\{\lambda\} = \{\lambda\} L = L \tag{1.1}$$

$$L\,\emptyset = \emptyset\,L = \emptyset \tag{1.2}$$

$$(L_1 \cup L_2)^T = L_1^T \cup L_2^T \tag{1.3}$$

$$(L_1 L_2)^T = L_1^T L_2 \; if \; \lambda \notin L_1; \; otherwise, \; (L_1 L_2)^T = L_1^T L_2 \cup L_2^T \tag{1.4}$$

$$(L^*)^T = L^T L^* \tag{1.5}$$

Kleene [16] characterized a subclass of languages called *regular languages* in terms of *regular expressions.*

Definition 1.2 The regular expressions over alphabet Σ and the languages they denote are defined inductively as follows.

- $\emptyset$ is a regular expression that denotes the empty set
- λ is a regular expression that denotes set $\{\lambda\}$
- a is a regular expression that denotes $\{a\}$, where $a \in \Sigma$

If J and K are regular expressions that represent languages L_J and L_K, then the following are also regular expressions:

- $J|K$ (alternation) represents $L_J \cup L_K$
- JK (product) represents $L_J L_K$
- J^* (star) represents $\cup_{i=0}^{\infty} L_J^i$

By convention star has higher precedence than product, which has higher precedence than alternation. Both product and alternation are left associative. Parentheses are used to override precedence. Two regular expressions are *equal* if they are, syntactically, the same; two regular expressions are *equivalent* if they denote the same language. The *length* of regular expression R is the number of symbol occurrences in R including λ, alphabet symbols, parentheses, star, and alternation operator. Since product operators are implicit, product operators are not counted in calculating the length of regular expressions. Regular expression $((ab|cd^{**}))^*$ is of length 12. Without loss of generality, we will assume throughout this thesis that regular expressions have no occurrences of $\emptyset$.

Regular expressions have been used in a variety of practical applications to specify regular languages in a perspicuous way. 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 defined below.

Definition 1.3 A *nondeterministic finite automata* (abbr. NFA) M is a 5-tuple $(\Sigma, Q, I, F, \delta)$, where Σ is an alphabet, Q is a set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta \subseteq Q \times (\Sigma \times Q)$ is a state transition map. It is useful to view NFA M as a labeled directed graph with states as vertices and an edge labeled a connecting state q to state p for every pair $[q,[a,p]]$ belonging to δ. For all $q \in Q$ and $a \in \Sigma$ we use the notation $\delta(q, a)$ to denote the set $\{p : [q, [a, p]] \in \delta\}$ of all states reachable from state q by a single edge labeled a.

It is helpful to extend the notation for transition map δ in the following way. If $q \in Q$, $V \subseteq Q$, $a \in \Sigma$, $x \in \Sigma^*$, and $B \subseteq \Sigma^*$, then we define

$$\delta(q, \lambda) = \{q\},$$

$$\delta(q, ax) = \delta(\delta(q, a), x),$$

$$\delta(V, x) = \cup_{q \in V} \delta(q, x),$$

$$\delta(V, B) = \cup_{b \in B} \delta(V, b).$$

The language accepted by M, denoted by L_M, is defined by the rule, $x \in L_M$ if and only if $\delta(I, x) \cap F \neq \emptyset$. In other words, $L_M = \{x \in \Sigma^* | \delta(I, x) \cap F \neq \emptyset\}$. NFA M is a *deterministic finite automata* (abbr. DFA) if transition map δ has no more than one edge with the same label leading out from each state, and if I contains exactly one state. Two NFA's are *equivalent* if they accept the same language.

1.2 Background and Related Work

Kleene also characterized the regular languages in terms of languages accepted by DFA's. Rabin and Scott [25] showed that NFA's also characterize the regular languages, and their work led to algorithms to decide whether an arbitrary string is accepted by an NFA.

Let n be the number of NFA states, m be the number of edges, and k be the alphabet size. For an NFA represented by an adjacency matrix of size n^2 for each alphabet symbol, acceptance testing takes $O(n|x|)$ bit vector operations and $O(n)$ auxiliary space. Alternatively, for an NFA implemented by an adjacency list of size m with a perfect hash table [11] storing the alphabet symbols at each state, this test takes time proportional to $m|x|$ in the worst case. For DFA's the same data structure leads to a better time bound of $\theta(|x|)$. However, there are NFA's for which the smallest equivalent DFA (unique up to isomorphism of state labels as shown by Myhill [20] and Nerode [21]) has an exponentially greater number of states. Thus, the choice between using an NFA or DFA is a space/time tradeoff.

1.2.1 From Regular Expressions to NFA's

There are two main approaches for turning regular expressions into equivalent NFA's. One is due to Thompson [30], and the other one is due to McNaughton and Yamada[18].

Definition 1.4 A *λ-NFA* M is a 5-tuple $(\Sigma, Q, I, F, \delta)$. λ-NFA M is an NFA except that state transition map $\delta \subseteq Q \times ((\Sigma \cup \{\lambda\}) \times Q)$. Let V be a set of λ-NFA states. The *λ-closure* of V, denoted by $\lambda-\text{closure}(V)$, is the smallest set of states V' such that $V \subseteq V'$ and $V' = V' \cup \{y : x \in V', [x, [\lambda, y]] \in \delta\}$. If $q \in Q$, $V \subseteq Q$, $a \in \Sigma$, $x \in \Sigma^*$, and $B \subseteq \Sigma^*$, then we define

$$\delta(q, \lambda) = \lambda-\text{closure}(\{q\}),$$
$$\delta(q, a) = \{p : q' \in \lambda-\text{closure}(\{q\}), [q', [a, p]] \in \delta\},$$

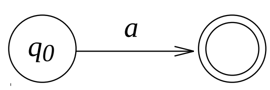
$$\delta(q, ax) = \delta(\delta(q, a), x), \text{ for } x \neq \lambda,$$

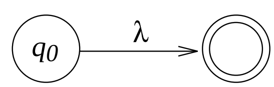
$$\delta(V, x) = \cup_{q \in V} \delta(q, x),$$

$$\delta(V, B) = \cup_{b \in B} \delta(V, b).$$

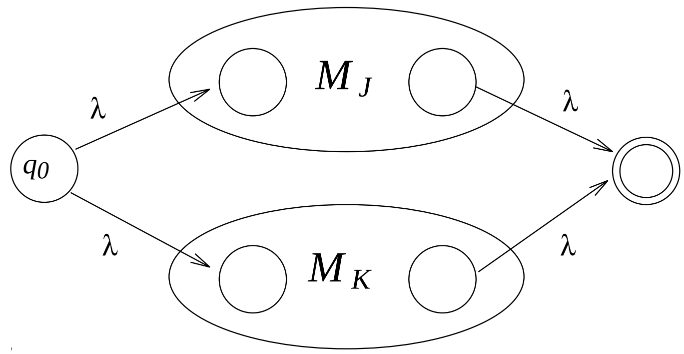
A string $x \in L_M$ if and only if $\lambda-$closure$(\delta(I, x)) \cap F \neq \emptyset$.

Thompson's construction is a simple, bottom-up method that processes the regular expression and construct λ-NFA's as it is parsed. For regular expression R, the rules for constructing Thompson's NFA M_R that accepts L_R are as follows: There are exactly one initial and one final state in Thompson's NFA M_R. If the regular expression is λ or an alphabet symbol, say a, then Thompson's algorithm constructs an equivalent NFA as follows, where state labeled q_0 is the initial state, and double circled state the final state. The number of states in M_a state_count$(M_a) = 2$, and NFA M_a has edge_count$(M_a) = 1$ edge. For Thompson's NFA M_λ, we have state_count$(M_\lambda) = 2$ and edge_count$(M_\lambda) = 1$.

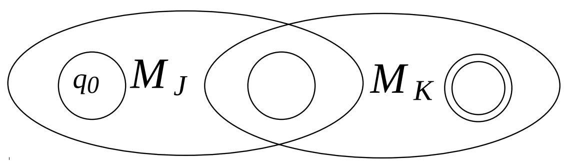




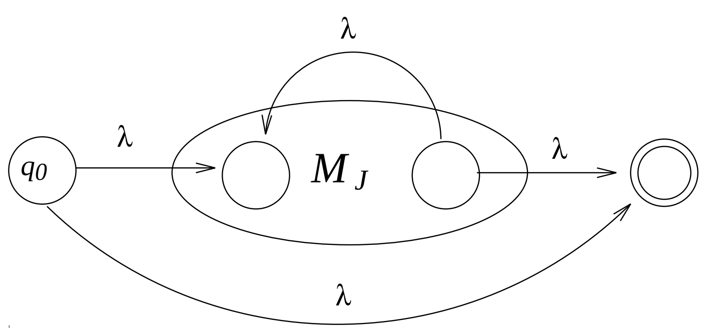
If regular expression $R = J|K$, then an equivalent NFA M_R can be constructed from M_J and M_K. Thompson's NFA $M_{J|K}$ has state_count(M_J) + state_count(M_K) + 2 states and edge_count(M_J) + edge_count(M_K) + 4 edges.



If regular expression $R = JK$, then NFA M_R denoting L_R is constructed as follows. Thompson's NFA M_{JK} has state_count(M_J) + state_count(M_K) - 1 states and edge_count(M_J) + edge_count(M_K) edges.



If regular expression $R = J^*$, then NFA M_R is constructed from M_J by adding two states and four edges to M_J. Thompson's NFA M_{J^*} has exactly state_count(M_J) + 2 states and edge_count(M_J) + 4 edges.



Let $s_\Sigma, s_\lambda, s_|, s_., s_*$ and $s_{()}$ be the number of occurrences of alphabet symbols, λ, alteration operator, product operation, star operator, and paren-

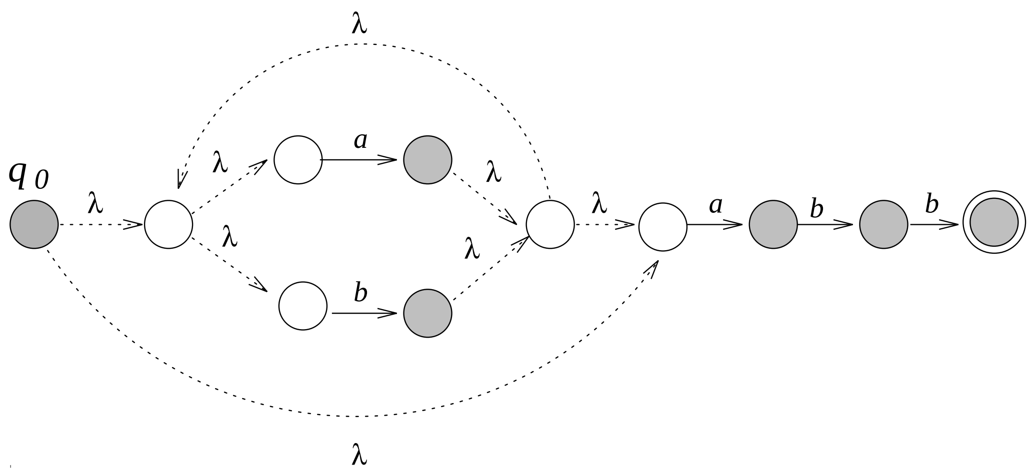


Figure 1.1: Thompson's NFA equivalent to $(a|b)^*abbb$

theses in regular expression R. The length of R is $r = s_\Sigma + s_\lambda + s_| + s_* + s_{()}$. Moreover, the number of states and edges of Thompson's NFA M_R are bounded by

$$r - s_{()} + 1 \leq \text{state\_count}(M_R) = 2(s_\Sigma + s_\lambda + s_| + s_*) - s_. \leq 2r,$$
$$r - s_{()} \leq \text{edge\_count}(M_R) = s_\Sigma + s_\lambda + 4(s_| + s_*) \leq 4r - 3.$$

The indegree and the outdegree of any state in Thompson's NFA M_R is no greater than 2. If a state is the tail of an edge labeled by an alphabet symbol in M_R, then the indegree of this state is one. The time and space for Thompson's algorithm is linear in r. The NFA equivalent to $(a|b)^*abb$ constructed by Thompson's algorithm is shown in Fig. 1.1.

Another approach is based on Berry and Sethi's [5] improvement to McNaughton and Yamada [18]. In addition to an initial state, McNaughton

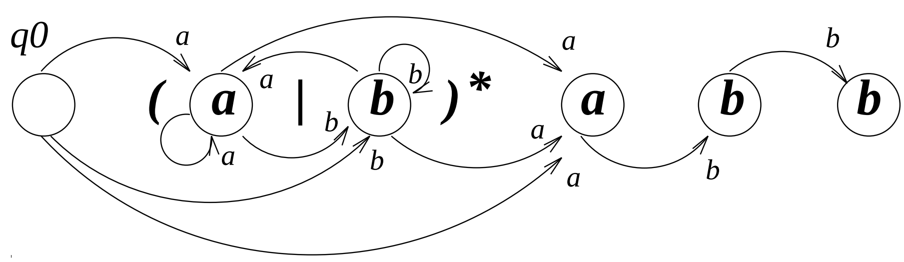


Figure 1.2: McNaughton and Yamada's NFA equivalent to $(a|b)^*abbb$

and Yamada's NFA has a distinct state for every alphabet symbol occurrence in the regular expression. All the edges in McNaughton and Yamada's machine are labeled by alphabet symbols; all the incoming edges of each state are labeled by the same symbol.

We shall discuss McNaughton and Yamada's NFA in the next Chapter. However, it suffices to say at this point that McNaughton and Yamada's machines can also be viewed as NFA's transformed from Thompson's NFA's. The initial state and states that are tails of edges labeled by alphabet symbols in Thompson's NFA are called *transition states.* These states correspond to states in McNaughton and Yamada's NFA (e.g. shaded nodes in Fig. 1.1). Let q_1, q_2 be states in a McNaughton and Yamada's NFA. There is a path from transition state q_1 to transition state q_2 in a Thompson's Machine spelling a if and only if there is an edge labeled a from q_1 to q_2 in McNaughton and Yamada's corresponding machine. McNaughton and Yamada's corresponding NFA for $(a|b)^*abb$ is shown in Fig. 1.2.

Berry and Sethi construct an NFA in which the number of states n is precisely one plus the number s of occurrences of alphabet symbols appearing in the regular expression. In general, s can be arbitrarily smaller than r. However, the number of edges in McNaughton and Yamada's NFA is $m = \Omega(s^2)$ in the worst case, which can be one order of magnitude larger than the bound for Thompson's NFA. Berry and Sethi's construction scans the regular expression twice, and, with only a little effort, both passes can be made to run in linear time and auxiliary space with respect to r plus the size of the NFA. More recently, Brüggemann-Klein[6] presents a two-pass algorithm to compute McNaughton and Yamada's NFA using the same resource bounds as Berry and Sethi.

For the bit matrix representation, McNaughton and Yamada's NFA can be used to solve acceptance testing using $O(s|x|)$ bit vector operations, which is superior to the time bound for Thompson's NFA. With the adjacency list representation the worst case number of edges $m = \Omega(s^2)$ leads to a worst case time bound $\Theta(m|x|)$ which is one order of magnitude worse than the time bound for Thompson's machine. However, the fact that McNaughton and Yamada's NFA is a DFA when all of the alphabet symbols are distinct may explain, in part, why it is observed to outperform Thompson's NFA for a large subclass of the instances. Berry and Sethi's construction scans the regular expression twice, and, with only a little effort, both passes can be made to run in linear time and auxiliary space with

$\sigma := \emptyset$
workset := $\{\lambda$-closure$(\{q_0\})\}$
while $\exists V \in$ workset do
    workset := workset - $\{V\}$
    for each symbol $a \in \Sigma$ and set of states $B = \{x \in \delta(V, \Sigma) | A(x) = a\}$,
            where $B \neq \emptyset$ do
        $B := \lambda$-closure(B)
        $\sigma(V, a) := B$
        if B does not belong to the domain of σ or to workset then
            workset := workset $\cup \{B\}$
        end if
    end for
end while

Figure 1.3: Rabin and Scott's subset construction

respect to r plus the size of the NFA (for either adjacency list or matrix implementations).

1.2.2 From NFA's to DFA's

There is one main approach for turning NFA's (constructed by either of the two methods above) into DFA's. This is by Rabin and Scott's subset construction[25]. A high level specification of Rabin and Scott's classical subset construction for producing a DFA σ from an NFA δ is given in Fig. 1.3. The λ-closure is a means to produce smaller DFA's. The λ-closure step is often omitted when we construct DFA's from McNaughton and Yamada's NFA's because McNaughton and Yamada's NFA does not have λ-edges. There is a heuristic, similar to λ-closure, being proposed (see pp. 141 in [2]) for McNaughton and Yamada's NFA to reduce the size of DFA's.

1.2.3 An NFA/DFA hybrid machine

Recently, Meyer [19] gave an $O(|x||R|/\log|x|)$ space and time acceptance testing algorithm for regular expression R and string x. He makes use of node listing and the "four Russians" trick to devise an $O(\log|x|)$-fold speed up algorithm. Basically, his machine is a hybrid of NFA and DFA. More precisely, he divides a Thompson's NFA into $O(|R|/\log|x|)$ modules, and each module is replaced by a DFA. Using a bit-vector model of complexity, his algorithm runs in $O(|x||R|/\log|x|)$ time and space. In practice, his algorithm, which is based on bit-vector operation, is fast for small regular expressions [19].

Chapter 2

McNaughton and Yamada's NFA

In this Chapter we give a one-pass algorithm to construct McNaughton and Yamada's NFA M from a regular expression in linear time with respect to the size of M. We present a space efficient parsing algorithm for regular expressions so that our construction algorithm uses only $O(s)$ auxiliary space, where s is the number of alphabet symbol occurrences in the regular expression. Recall that s can be arbitrarily smaller than the length of regular expression. The previous best algorithms [5,6] are linear time, but they use two passes and $O(|R|)$ auxiliary space.

2.1 A Space Efficient Parsing Algorithm for Regular Expressions

For any regular expression R, there are equivalent regular expressions that are arbitrarily longer than R. A regular expression is equivalent to the empty string λ if and only if it contains no alphabet symbol occurrences. For convenience, we call a regular expression a *λ-expression* if it is equivalent to the empty string λ. We can concatenate a regular expression with a λ-expression and preserve its meaning. If $\lambda \in L_R$, then $R|\lambda \equiv R|\lambda \equiv R$. We can enclose a regular expression by a pair of parentheses without changing the language it denotes. For regular expression R, all the popular parsing algorithms (for example, LR or LL parsers) take $O(|R|)$ time and space (both output and auxiliary space). We give an algorithm to parse R in $O(|R|)$ time and $O(s + \log |R|)$ auxiliary space, where s is the number of alphabet symbol occurrences in R. Using a conventional assumption that each memory word has $O(\log |R|)$ bits, our algorithm uses $O(s)$ space.

Lemma 2.1 Let R be a regular expression with s occurrences of alphabet symbols. There is an $O(s)$ long regular expression R' equivalent to R. [7]

Proof: Suppose we augment a regular expression R by adding a $*$ on top of R, enclosing R by a pair of parentheses, concatenating R with a λ-expression, adding a λ-expression as an alternative for R, or applying an arbitrary number of operations previously stated. An augmented regular expression R' of R denotes one of the languages of L_R, $L_{(R)|\lambda}$ or $L_{(R)^*}$.

By an inductive argument, we show that for any regular expression R with $s > 0$ alphabet symbol occurrences, there is a regular expression R' of length at most $14(s-1)+5$ equivalent to R. For any regular expression R containing one alphabet symbol occurrence a, R denotes one of $L_a, L_{a|\lambda}$ or L_{a^*}. Let R be JK or $J|K$, where J and K are regular expressions containing s_J and s_K, $s_J, s_K > 0$, alphabet symbol occurrences respectively. By the induction hypothesis, there exist J' and K' which are equivalent to J and K respectively, and the length of J' and K' is at most $14(s_J - 1) + 5$ and $14(s_K - 1) + 5$ respectively. Therefore, either $(J')|(K')$ or $(J')(K')$ is equivalent to R, and it is at most $14(s-1)+1$ long. Consider that regular expression R contains $s > 1$ occurrences of alphabet symbols, and R_1 is the smallest subexpression of R containing s alphabet symbol occurrences. There must be a $14(s-1)+1$ long regular expression R'_1 equivalent to R_1. Therefore, there is a $14(s-1)+5$ long regular expression equivalent to R.
□

Lemma 2.1, by a non-constructive approach, shows that there exists an $O(s)$ long equivalent regular expression R' of regular expression R with s alphabet symbol occurrences. If we can transform R into R' in $O(|R|)$ time using $O(s)$ auxiliary space, then we can "parse" R using the same time and space bounds.

We can always transform R into an $O(s)$ long equivalent regular expression by applying the following simplification rules [7]:

$$\lambda^* = \lambda \tag{2.1}$$

$$R\lambda = \lambda R = R \tag{2.2}$$

$$R|\lambda = \lambda|R = R, \ where\ \lambda \in L_R \tag{2.3}$$

$$R^{**} = (R^*)^* = R^* \tag{2.4}$$

$$((R)) = (R) \tag{2.5}$$

However, in order to apply these simplification rules, we need to parse R first. There is work to be done to achieve our goal.

All the popular parsing algorithms cannot parse regular expressions with s alphabet symbol occurrences using only $O(s)$ auxiliary space. In illustration, let us consider the following examples. There could be an arbitrary number of "*" on top of regular expression, but we can incorporate rule (2.4) in a parser to solve this problem. We can also incorporate rule (2.2) and (2.3) in a parser to eliminate most of the redundant λ's so that a parser hopefully uses no more than $O(s)$ auxiliary space. However, to efficiently handle parentheses is a problem hard to overcome.

We can enclose a regular expression by an arbitrary number of pairs of parentheses without changing its meaning. Consider regular expression $R = (\cdots(a)\cdots)$ which has k pairs of parentheses. An LR parser keeps up to k left parentheses in the stack before R is completely parsed. One quick heuristic to reduce auxiliary space is that instead of keeping every left parenthesis in the stack, we keep track only the number of consecutive left parentheses. However, this heuristic does not work. Regular expressions R and $(\lambda|(R))$ denote different languages if $\lambda \notin L_R$. The action to be taken after scanning k consecutive left parentheses is different from the one after scanning a sequence "$(\cdots(\lambda|\cdots(\lambda|\cdots$", which also contains k left parentheses. Therefore, we cannot use an $O(\log|R|)$-bit counter to encode this sequence. Moreover, we cannot perform a reduction immediately even when we know it is possible because there could be an arbitrary number of λ-expressions spread in a regular expression with s alphabet symbol occurrences. An eager reduction strategy for λ-expressions would result in an $O(|R|)$ auxiliary space bounded parsing algorithm. We shall make use of special properties of regular expressions to develop a variant of operator precedence parsing. Our algorithm constructs an $O(s)$ long equivalent expression of R in $O(|R|)$ time and using only $O(s)$ auxiliary space.

We first consider the following simpler cases.

λ-expressions: Let us consider an expression R containing no alphabet symbol occurrences first. Since a regular expression containing no alphabet symbol occurrences is equivalent to λ, we can transform R to λ if R is a valid regular expression. We use a counter to decide whether or not the parentheses in R is balanced. In $O(|R|)$ time and using $O(\log|R|)$-bit auxiliary space, we can decide the validity of expression R.

conjunct reduction: Now we consider an expression R of the form

$(J_1)(J_2)\cdots(J_k)$. Let $K_1, K_2, \ldots, K_t$ be all the top-level subexpressions (i.e. each $K_i = (J_l)$, for some l) which contain alphabet symbol occurrences. Assume that $K_1, K_2, \cdots$ and K_t are valid, been parsed, and they are represented as atoms (pointers pointing to subexpressions). Then $R \equiv K_1K_2\cdots K_t$ if R is valid. Following the argument used in the previous paragraph, we can transform R (including validity checking) into $K_1K_2\cdots K_t$ in $O(l+t)$ time and $O(\log|R|)$-bit auxiliary space, where l is the sum of the length of all the J_i's, $1 \leq i \leq k$, which do not contain alphabet symbol occurrences.

disjunct reduction: Consider the case that expression R is of the form $(J_1)|\cdots|(J_k)$. Let $K_1, K_2, \ldots, K_t$ be all the top-level subexpressions which contain alphabet symbol occurrences. As before, we assume that $K_1, K_2, \ldots K_t$ are valid, been parsed, and they are represented as atoms. Suppose that R is a valid regular expression. If all the J_i's, $1 \leq i \leq k$, are not λ-expressions, or if there is a K_i, $1 \leq i \leq t$, such that K_i accepts the empty string λ, then we transform R into an equivalent regular expression $K_1|K_2|\cdots|K_t$; otherwise, we transform R into $\lambda|K_1|K_2|\cdots|K_t$. This transformation (including validity checking) can be done in $O(l+t)$ time and using $O(\log|R|)$-bit auxiliary space, where l has the same definition as before.

We shall describe our algorithm as follows. We assume that the input expression is stored in an array or a doubly linked list. Because of $O(s)$

auxiliary space restriction, we cannot afford to store all the parsing ingredients in the auxiliary space (as what LR_k or recursive-descendent parser does). In particular, we need a special mechanism to store "(", ")", λ and their related operators. Conceptually, we divide expression R into up to s *basic blocks* and $s+1$ *non-basic blocks.* A basic block is a valid subexpression of R that contains at least one alphabet symbol occurrence. Non-basic blocks are regions of expression R separated out by basic blocks. Each of basic and non-basic blocks has two pointers pointing to the leftmost and rightmost symbols of the subexpression it represents. Every basic block B_i with k alphabet symbol occurrences is attached to an $O(k)$ long equivalent regular expression r_i. In addition, two flags *null* and *star* are attached to every basic block. The flag *null* in a basic block is true if and only if this basic block accepts the empty string. We use flag *star* to indicate whether or not there is a "*" on top of the equivalent regular expression attached to this block. Basic blocks and non-basic blocks are doubly linked, and named $b_1, B_1, b_2, B_2, \ldots, b_{s+1}$ according to their left/right appearances in R. The essential idea of our algorithm is to perform reductions around basic blocks as many time as possible, and only around basic blocks, to avoid harmful reductions. so as not to blow up the $O(s)$ auxiliary space bound constraint. Initially, there is no basic or non-basic block being built. Without loss of generality, we assume that R contains at least one alphabet symbol occurrence, R is universally parenthesized, and block pointers are updated when

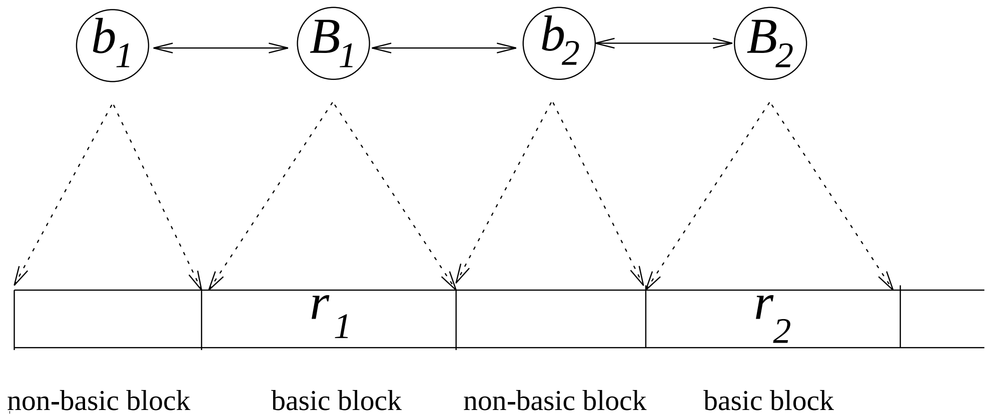


Figure 2.1: Block Structure Organization

it is obvious.

step 1: In this step, we construct a new basic and a new non-basic block whenever there is nothing obvious to do. Initially, we scan the first (leftmost) symbol of R. We scan R rightward until an alphabet symbol or a right parenthesis, of which the matching left parenthesis is to the left of the starting scanning position, is encountered. If such a right parenthesis is found, then we perform step 5. If an alphabet symbol, say a, is encountered, then we construct a non-basic block b to cover the region recently scanned (excluding a), and create a new basic block B to cover a. The equivalent regular expression attached to B is a. Both $null$ and $star$ flag of B have value false. Then, we perform step 2. Consider the case that we reach the end of R. If there is only one basic block covering the entire R, then the transformation is completed; otherwise, R is not a valid regular expression.

step 2: The goal of this step is to perform reductions around a basic

left	right	action
don't care	$*$	if the *star* flag of B is false, then the equivalent regular expression of B become $(r)^*$; otherwise, we do nothing. The *star* and *null* flags of B become true.
.	.	perform step 3
.	$\vert$	perform step 3
.	)	perform step 3
$\vert$	$\vert$	perform step 4
$\vert$	)	perform step 4
(	)	do nothing
$\vert$	.	perform step 1
all other cases		perform step 1

Table 2.1: Reduction Rules used in step 2

block. Let B be the basic block currently examined, and r be the equivalent regular expression attached to B. We examine the left and right operators around B, and perform reductions as many as possible according to rules given in Table 2.1.

step 3: We scan R leftward from the rightmost position to the left of B (we treat basic blocks as atomic elements) in order to find a maximal conjunct (i.e. of the form $J_1 J_2 \cdots J_K B$). If a right parenthesis is encountered, we move our scanning position (and ignore the contents) to its matching left parenthesis which must be in the same non-basic block. On the way scanning leftward, if a "|" operator or a left parenthesis is encountered, then we find a maximal conjunct which covers B (and up to B). We then create an new basic block B' to cover this conjunct. According to the transformation previously described, we construct an equivalent regular expression for B', delete/modify those blocks which have non-empty overlapping with B', maintain block structure, and update star and null flag in B'. This step takes $O(l + k)$ time and $O(\log |R|)$-bit auxiliary space, where l is the number of symbols scanned in non-basic blocks, and k is the number of basic blocks covered by B'.

step 4: Similar to step 3, we scan R leftward from the rightmost position to the left of B (treat basic blocks as atomic elements as before) in order to find a maximal sequence of conjuncts connected by "|". Similar to step 3, we simply scan R leftward until a left parenthesis which its matching right parenthesis is to the right of B, is encounter. Similar to step 3, this step takes $O(l + k)$ time and $O(\log |R|)$-bit auxiliary space, where k and l is the same as we define in step 3.

step 5: In this step,we reduce a subexpression enclosed by a pair of parentheses. The method used in this step is the same as step 4. The time and space complexities are the same as step 4.

For any regular expression R with s alphabet symbol occurrences, our method transforms R into an $O(s)$ long equivalent expression R' in $O(|R|)$ time and $O(s)$ auxiliary space. Our method uses $O(s)$ auxiliary space because at most $2s + 1$ basic and non-basic blocks are built. Since we scan each symbol in R at most twice, our method takes $O(|R|)$ time. Because the transformations used in step 2, 3, 4 and 5 are correct, R' is equivalent to R. Similar to Lemma 2.1, the length of R' is $O(s)$ bounded.

Without loss of generality, we assume, throughout this thesis, all the regular expressions with $s > 0$ occurrences of alphabet symbols are of length $O(s)$.

2.2 McNaughton and Yamada's NFA Reformulation

It is convenient to reformulate McNaughton and Yamada's transformation from regular expressions to NFA's[18] in the following way.

Definition 2.2 A *normal NFA* (abbr. NNFA) is an NFA in which all edges leading into the same state have the same label. Thus, it is convenient to label states instead of edges, and we represent an NNFA M as a 6-tuple $(\Sigma, Q, \delta, I, F, A)$, where Σ is an alphabet, Q is a set of states, $\delta \subseteq Q \times Q$ is a set of (unlabeled) edges, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $A : Q \rightarrow \Sigma$ maps states $x \in Q$ into labels $A(x)$ belonging to alphabet Σ. The language L_M accepted by NNFA M is the set of strings $x \in \Sigma^*$ formed from concatenating labels on all but the first state of a path from a state in I to a state in F. A *McNaughton/Yamada NNFA* (abbr. MYNNFA) is an NNFA with one initial state of zero in-degree (see Fig. 2.2).

We sometimes omit Σ in NNFA specifications when it is obvious. It is useful (and completely harmless) to sometimes allow the label map A to be undefined on states with zero in-degree. For example, we will not define A on the initial state of an MYNNFA.

Definition 2.3 The *tail* of an MYNNFA $M = (\Sigma, Q, \delta, I = \{q_0\}, F, A)$ is an NNFA $M^T = (\Sigma^T, Q^T, \delta^T, I^T, F^T, A^T)$, where $\Sigma^T = \Sigma$, $Q^T = Q - \{q_0\}$,

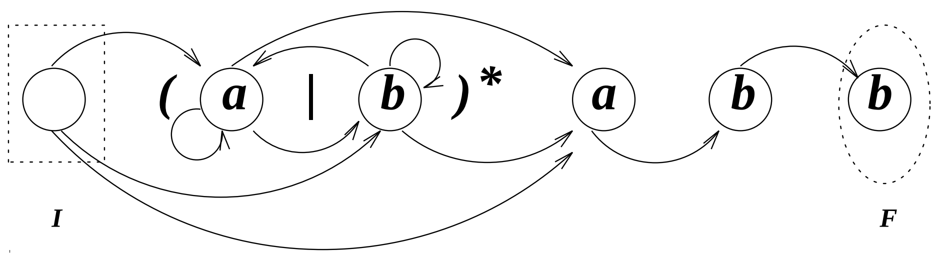


Figure 2.2: An MYNNFA equivalent to regular expression $(a|b)^*abb$

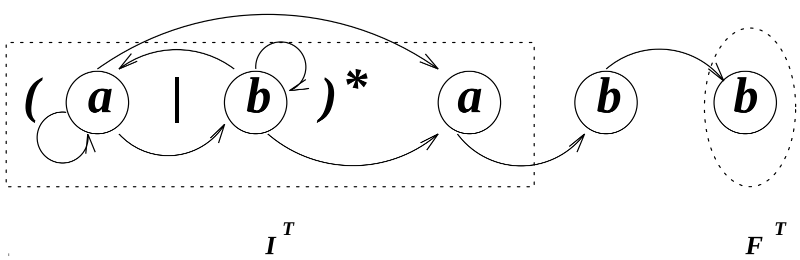


Figure 2.3: The tail of an MYNNFA equivalent to regular expression $(a|b)^*abb$

$\delta^T = \{[x,y] \in \delta | x \neq q_0\}$, $I^T = \{y : [q_0, y] \in \delta\}$, $F^T = F - \{q_0\}$, and $A^T = A$. Fig. 2.3 shows the tail of the MYNNFA given in Fig. 2.2.

If MYNNFA M accepts language L_M, then the identity $L_M^T = (L_M)^T$ holds; that is, the language accepted by tail machine M^T is the same as the tail of the language accepted by M. However, given the tail of some MYNNFA M, we cannot compute an MYNNFA equivalent to M without also knowing whether $\lambda \in L_M$. Let $null_M = \{\lambda\}$ if $\lambda \in L_M$; otherwise, let $null_M = \emptyset$. Now we can reconstruct an MYNNFA $M = (\Sigma, Q, \delta, I, F, A)$

from its tail $M^T = (\Sigma^T, Q^T, \delta^T, I^T, F^T, A^T)$ and from $null_M$ using the equations

$$\Sigma = \Sigma^T,\ Q = Q^T \cup \{q_0\},\ \delta = \delta^T \cup \{[q_0, y] : y \in I^T\},\ I = \{q_0\},$$
$$F = F^T \cup \{q_0\}null_M,\ A = A^T, \tag{2.6}$$

where q_0 is a new state. Note that since $null_M$ is either $\{\lambda\}$ or $\emptyset$, and by language operations defined previously, the expression $\{q_0\}null_M$ is $\{q_0\}$ if $null_M = \{\lambda\}$; otherwise, $\{q_0\}null_M = \emptyset$.

It is a desirable and obvious fact (which follows immediately from the definition of an MYNNFA) that when A is one-to-one, then no state can have more than one edge leading to states with the same label. Hence, such an MYNNFA is a DFA. More generally, an MYNNFA is a DFA if and only if the binary relation $\{[x, y] \in \delta | A(y) = a\}$ is single-valued for every alphabet symbol $a \in \Sigma$.

McNaughton and Yamada's algorithm take an input regular expression R, and computes an MYNNFA M that accepts L_R. Their algorithm can be implemented within a left-to-right parse of R without actually producing a parse tree. To explain how the construction is done, we use the notational convention that M_R denotes an MYNNFA equivalent to regular expression R. Each time a subexpression J of R is reduced during parsing, $null_J$ and M_J^T are computed, where M_J is an MYNNFA equivalent to J. The last step computes an MYNNFA M_R from M_R^T and $null_R$ by equations (2.6).

Theorem 2.4 (McNaughton and Yamada) Given any regular expression R with s occurrences of alphabet symbols from Σ, an MYNNFA M_R with $s+1$ states can be constructed.

Proof: The proof uses structural induction to show that for any regular expression R, we can always compute $null_R$ and M_R^T for some MYNNFA M_R. Then equations (2.6) can be used to obtain M_R. We assume a fixed alphabet Σ. There are two base cases, which are easily verified.

$$
\begin{aligned}
M_\lambda^T &= (Q_\lambda^T = \emptyset, \delta_\lambda^T = \emptyset,\ I_\lambda^T = \emptyset,\ F_\lambda^T = \emptyset,\ A_\lambda^T = \emptyset),\ null_\lambda = \{\lambda\} & (2.7)\\
M_a^T &= (Q_a^T = \{q\},\ \delta_a^T = \emptyset,\ I_a^T = \{q\},\ F_a^T = \{q\},\ A_a^T = \{[q,a]\}),\\
&\quad null_a = \emptyset,\ where\ a \in \Sigma, and\ q\ is\ a\ distinct\ state & (2.8)
\end{aligned}
$$

To use induction, we assume that J and K are two arbitrary regular expressions equivalent respectively to MYNNFA's M_J and M_K with $M_J^T = (Q_J^T, I_J^T, F_J^T, \delta_J^T, A_J^T)$ and $M_K^T = (Q_K^T, I_K^T, F_K^T, \delta_K^T, A_K^T)$, where Q_J^T and Q_K^T are disjoint. Then we can use (1.3), (1.4), and (1.5) to verify that

$$
\begin{aligned}
M_{J|K}^T &= (Q_{J|K}^T = Q_J^T \cup Q_K^T,\ \delta_{J|K}^T = \delta_J^T \cup \delta_K^T,\ I_{J|K}^T = I_J^T \cup I_K^T,\\
&\quad F_{J|K}^T = F_J^T \cup F_K^T,\ A_{J|K}^T = A_J^T \cup A_K^T),\\
&\quad null_{J|K} = null_J \cup null_K & (2.9)\\
M_{JK}^T &= (Q_{JK}^T = Q_J^T \cup Q_K^T,\ \delta_{JK}^T = \delta_J^T \cup \delta_K^T \cup F_J^T I_K^T,\\
&\quad I_{JK}^T = I_J^T \cup null_J I_K^T,\ F_{JK}^T = F_K^T \cup null_K F_J^T,
\end{aligned}
$$

$$A^T_{JK} = A^T_J \cup A^T_K),\ null_{JK} = null_J null_K \tag{2.10}$$

$$\begin{aligned} M^T_{J*} &= (Q^T_{J*} = Q^T_J,\ \delta^T_{J*} = \delta^T_J \cup F^T_J I^T_J,\ I^T_{J*} = I^T_J,\ F^T_{J*} = F^T_J, \\ &\quad A^T_{J*} = A^T_J),\ null_{J*} = \{\lambda\} \end{aligned} \tag{2.11}$$

The preceding formulas are illustrated in Fig. 2.4.

Disjointness of the unions used to form the set of states for the cases $J|K$ and JK proves the assertion about the number of states. The validity of the disjointness assumption follows from the fact that new states can only be obtained from rule (2.8), and each new state is distinct. We can convert M^T_R into M_R using equations (2.6). □

The proof of Theorem 2.4 leads to McNaughton and Yamada's algorithm. The construction of label function A shows that when all of the occurrences of alphabet symbols appearing in the regular expression are distinct, then A is one-to-one. In this case, a DFA would be produced.

Analysis determines that this algorithm falls short of optimal performance, because the operation $\delta^T_J \cup F^T_J I^T_J$ within formula (2.11) for M^T_{J*} is not disjoint; all other unions are disjoint and can be implemented in unit time. In particular, this overlapping union makes McNaughton and Yamada's algorithm use time $O(s^3)$ to transform regular expression $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ into an MYNNFA with $s+1$ states and s^2 edges.

This redundancy is made explicit in two examples. After two applica-

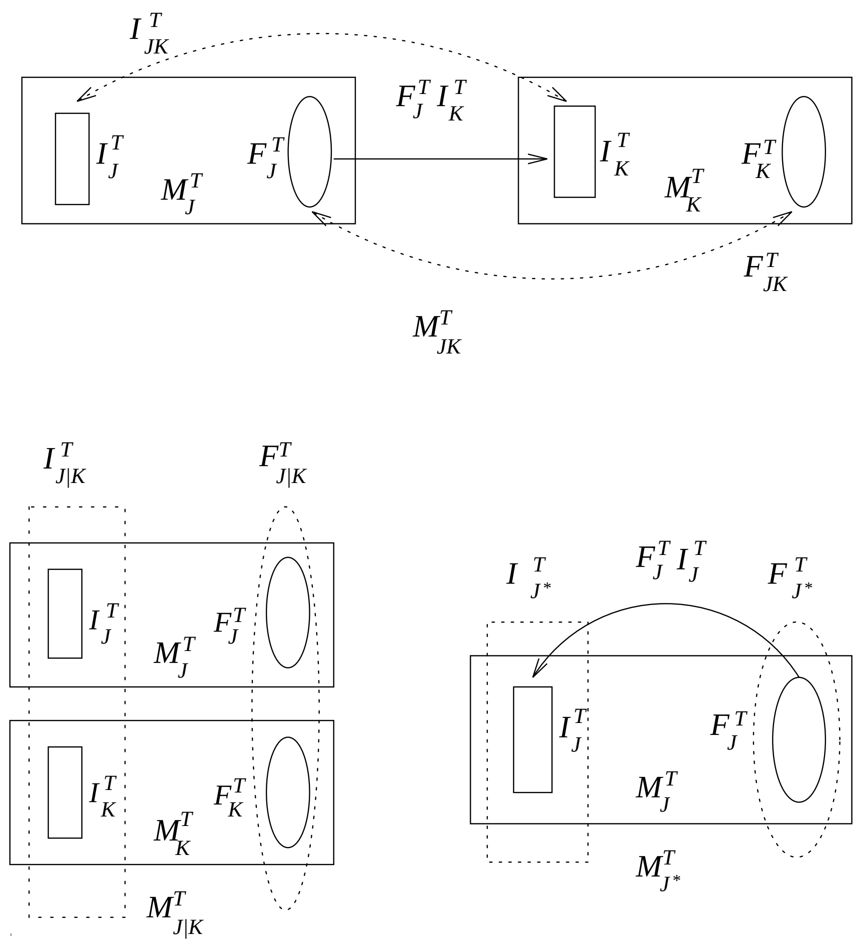


Figure 2.4: Tail machine construction.

tions of rule (2.11), we obtain the expansion

$$\begin{aligned}\delta^T_{J**} &= \delta^T_{J*} \cup F^T_{J*} I^T_{J*} \\ &= \delta^T_J \cup F^T_J I^T_J \cup F^T_J I^T_J,\end{aligned}$$

in which that product $F^T_J I^T_J$ is redundant. If $null_J = null_K = \{\lambda\}$, then application of rules (2.11) and (2.10) gives us the expansion

$$\begin{aligned}\delta^T_{(JK)*} &= \delta^T_{JK} \cup F^T_{JK} I^T_{JK} \\ &= \delta^T_J \cup \delta^T_K \cup F^T_J I^T_K \cup (F^T_J \cup F^T_K)(I^T_J \cup I^T_K),\end{aligned}$$

in which product $F^T_J I^T_K$ is redundant.

2.3 Faster NFA Construction

By recognizing the overlapping union $\delta^T_J \cup F^T_J I^T_J$ within formula (2.11) for M^T_{J*} as the source of inefficiency, we can maintain invariant $nred_J = F^T_J I^T_J - \delta^T_J$ in order to replace the overlapping union by the equivalent disjoint union $\delta^T_J \cup nred_J$. In order to maintain $nred_R$ as a component of the *tail* NNFA computation given above, we can use the following recursive definition, obtained by simplifying expression $F^T_R I^T_R - \delta^T_R$ and using the rules from the proof of Theorem 2.4.

$$nred_\lambda = \emptyset \tag{2.12}$$

$$nred_a = F^T_a I^T_a, where\ a \in \Sigma \tag{2.13}$$

$$nred_{J|K} = nred_J \cup nred_K \cup F^T_J I^T_K \cup F^T_K I^T_J \tag{2.14}$$

$$nred_{JK} = F_K^T I_J^T \cup null_K nred_J \cup null_J nred_K \tag{2.15}$$

$$nred_{J*} = \emptyset \tag{2.16}$$

Rules (2.12), (2.13) and (2.16) are trivial. Rule (2.14) follows from applying distributive laws to simplify formula

$$nred_{J|K} = (F_J^T \cup F_K^T)(I_J^T \cup I_K^T) - (\delta_J^T \cup \delta_K^T)$$

Rule (2.15) is obtained by applying distributed laws to simplify formula,

$$nred_{JK} = (F_K^T \cup null_K F_J^T)(I_J^T \cup null_J I_K^T) - (\delta_J^T \cup \delta_K^T \cup F_J^T I_K^T)$$

The preceding idea embodies a general method of symbolic finite differencing for deriving efficient functional programs. This method has been mechanized and used extensively by Douglas Smith within his program transformation system called KIDS (see for example [28]).

Each union operation in the preceding rules is disjoint and, hence, $O(1)$ time implementable. However our solution creates a new problem. Potentially costly edges resulting from product operations occurring in rules (2.14) and (2.15) may be useless, because they are never incorporated into δ. These edge may be useless for two reasons – (1) if the regular expression is star free, and (2) if the edges are eliminated by rule (2.15).

To overcome this problem we will use lazy evaluation to compute products only when they actually contribute edges to the NNFA. Thus, instead of maintaining an union $nred_R$ of products, we will maintain a set

$lazynred_R$ of pairs of sets. Consequently, the overlapping union $\delta_J^T \cup F_J^T I_J^T$ within formula (2.11) for M_{J*}^T can be replaced by

$$\delta_J^T \cup (\cup_{[A,B] \in lazynred_J} AB) \tag{2.17}$$

However, this solution creates another problem: the sets forming F^T and I^T, which are computed by the rules to construct the *tail* of an NNFA, must be persistent in the following sense. Let the sets in the sequence forming F^T (respectively I^T) be called *F-sets* (respectively *I-sets*). Each *F-set* (respectively *I-set*) could be stored as a first (respectively second) component of a pair belonging to *lazynred*. Given any such pair, we need to iterate through the *I-set* G stored in the second component of the pair in $O(|G|)$ time.

The sequence of *F-sets* (respectively *I-sets*) are formed by two operations: 1. create a new singleton set; and 2. form a new set by taking the disjoint union of two previous sets in the sequence. Clearly, each of these sequences can be stored as a binary forest in which each subtree in the forest represents a set in the sequence, where the elements of the set are stored in the frontier (i.e. leaves). By construction each internal node in the forest has two children.

We call the forest storing the *F-sets* (respectively *I-sets*) the *F-forest* (respectively *I-forest*). For each node n belonging to the *F-forest* (respectively *I-forest*), let $Fset(n)$ (respectively $Iset(n)$) denote the *F-set* (respectively

I-set) represented by n.

Each node in the *F-forest* and *I-forest* except the roots stores a parent pointer. Each node n in the *I-forest* also stores a pointer to the leftmost leaf of the subtree rooted in n and a pointer to the rightmost leaf of the subtree rooted n. The frontier nodes (i.e. leaves) of the *I-forest* are linked.

This data structure preserves the unit-time disjoint union for *F-sets* and *I-sets*, and supports linear time iteration through the frontier of any node in the *I-forest*. Since all the *F-sets* and *I-sets* are subsets of the NNFA states Q, the *F-forest* and *I-forest* each is stored in $O(|Q|)$ space.

Theorem 2.5 For any regular expression R we can compute $lazynred_R$ in time $O(r)$ and auxiliary space $O(s)$, where r is the size of regular expression R, and s is the number of occurrences of alphabet symbols appearing in R.

Proof: If G and H are two sets, let $pair(G, H) = \{[G, H]\}$ if both G and H are nonempty; otherwise, let $pair(G, H) = \emptyset$. The proof makes use of the following recursive definition of $lazynred_R$ obtained from the recursive definition of $nred_R$.

$$lazynred_\lambda = \emptyset \tag{2.18}$$

$$lazynred_a = pair(F_a^T, I_a^T), where\ a \in \Sigma \tag{2.19}$$

$$\begin{aligned} lazynred_{J|K} = {} & lazynred_J \cup lazynred_K \cup pair(F_J^T, I_K^T) \cup \\ & pair(F_K^T, I_J^T) \end{aligned} \tag{2.20}$$

$$\begin{aligned} lazynred_{JK} &= pair(F_K^T, I_J^T) \cup null_K lazynred_J \cup \\ &\quad null_J lazynred_K && (2.21) \\ lazynred_{J*} &= \emptyset && (2.22) \end{aligned}$$

Operation $pair(G, H)$ takes unit time and space. Each union operation occurring in the rules above is disjoint and, hence, implementable in unit time. Rule (2.19) contributes unit time and space for each alphabet symbol occurring in R, or $O(s)$ time and space overall. Rule (2.20) contributes unit time for each alternation operator appearing in R or $O(r)$ time overall. It contributes two units of space for each alternation operator both of whose alternands contain at least one alphabet symbol. Hence, the overall space contributed by this rule is less than $2s$. By a similar argument, Rule (2.21) contributes $O(r)$ time and less than s space overall. The other two rules contribute no more than $O(r)$ time overall. Hence, the time and space needed to compute $lazynred_R$ are $O(r)$ and $O(s)$ respectively. $\square$

By Theorems 2.4 and 2.5, and by the fact that $nred_R$ can be computed from $lazynred_R$ in $O(|nred_R|)$ time using formula (2.17), we have our first theoretical result.

Theorem 2.6 For any regular expression R we can compute an equivalent MYNNFA with $s + 1$ states in time $O(r + m)$ and auxiliary space $O(s)$, where r is the size of regular expression R, m is the number of edges in the MYNNFA, and s is the number of occurrences of alphabet symbols appearing in R.

Chapter 3

The Compressed NNFA

In this Chapter we present the compressed NNFA, the CNNFA, for short. The CNNFA is an $O(s)$ space compressed representation of the $O(s^2)$ space McNaughton/Yamada NFA. Given any subset V of NFA states, the CNNFA can be used to compute the set U of states one transition away from the states in V in optimal time $O(|V|+|U|)$. Using McNaughton and Yamada's NFA, it takes $O(|V| \times |U|)$ time in the worst case.

3.1 Improving Space for McNaughton and Yamada's NFA

Theorem 2.6 leads to a new algorithm that computes the adjacency form of the MYNNFA M_R in a left-to-right shift/reduce parse of the regular expression R. Although this improves upon the algorithm of Berry and Sethi, McNaughton and Yamada's NFA has certain theoretical disadvantages over Thompson's simpler NFA. Recall that for regular expression $(((((a_1^*|a_2)^*|a_3)^*...a_k)^*$ the number of edges in McNaughton and Yamada's NFA is the square of the number of edges in Thompson's NFA.

Nevertheless, we can modify the algorithm just given so that in $O(r)$ time, it produces an $O(s)$ space CNNFA that encodes McNaughton and Yamada's NFA, and that supports acceptance testing in $O(s|x|)$ time. In the same way that $nred_R$ was represented more compactly as $lazynred_R$, we can represent δ_R, which is an union of cartesian products, as a set $lazy\delta_R$ of pairs of set-valued arguments of these products. If M_R is the CNNFA equivalent to regular expression R, then the rules for M_R^T are given just below:

$$lazy\delta_\lambda^T = \emptyset \tag{3.1}$$

$$lazy\delta_a = \emptyset \tag{3.2}$$

$$lazy\delta_{J|K}^T = lazy\delta_J^T \cup lazy\delta_K^T \tag{3.3}$$

$$lazy\delta_{JK}^T = pair(F_J^T, I_K^T) \cup lazy\delta_J^T \cup lazy\delta_K^T \tag{3.4}$$

$$lazy\delta_{J*}^T = lazy\delta_J^T \cup lazynred_J \tag{3.5}$$

After the preceding rules are processed we can obtain a representation for M_R by introducing a new state q_0 and by adding the pair $[q_0, I_R^T]$ to $lazy\delta_R^T$ in accordance with equation (2.6).

We now show how to use $lazy\delta_R$ to simulate δ_R. If V is a subset of the

MYNNFA states Q, then we can compute the collection of states $\delta(V, a)$ for all of the alphabet symbols $a \in \Sigma$ as follows. First we compute

$$finddomain(V) = \{X : [X, Y] \in lazy\delta | V \cap X \neq \emptyset\}$$

which is used to find the set of next states

$$next\_states(V) = \{Y : [X, Y] \in lazy\delta | X \in finddomain(V)\}$$

Finally, for each alphabet symbol $a \in \Sigma$, we see that

$$\delta(V, a) = \{q : Y \in next\_states(V), q \in Y | A(q) = a\}$$

In order to explain how $lazy\delta$ is implemented, we will use some additional terminology. The CNNFA consists of an *F-forest*, an *I-forest*, and a collection of *crossing* edges to represents pairs in $lazy\delta$. The *F-forest* and *I-forest* share the same set of leaves. Forest nodes are connected by tree edges. Each crossing edge, originating from an *F-set* node to an *I-set* node, represents a pair in $lazy\delta$. For each *F-set* G represented by node n in the *F-forest*, n stores a pointer to a list of edges originating from n. Furthermore, in addition to forest leaves, the *F-forest* and *I-forest* are compressed to store nodes representing sets that appear as the first or second components of a pair $[X, Y] \in lazy\delta$. This can be achieved on-line as the *F-forest* and *I-forest* are constructed by a kind of path compression that affects the preprocessing time and space by no more than a small constant factor.

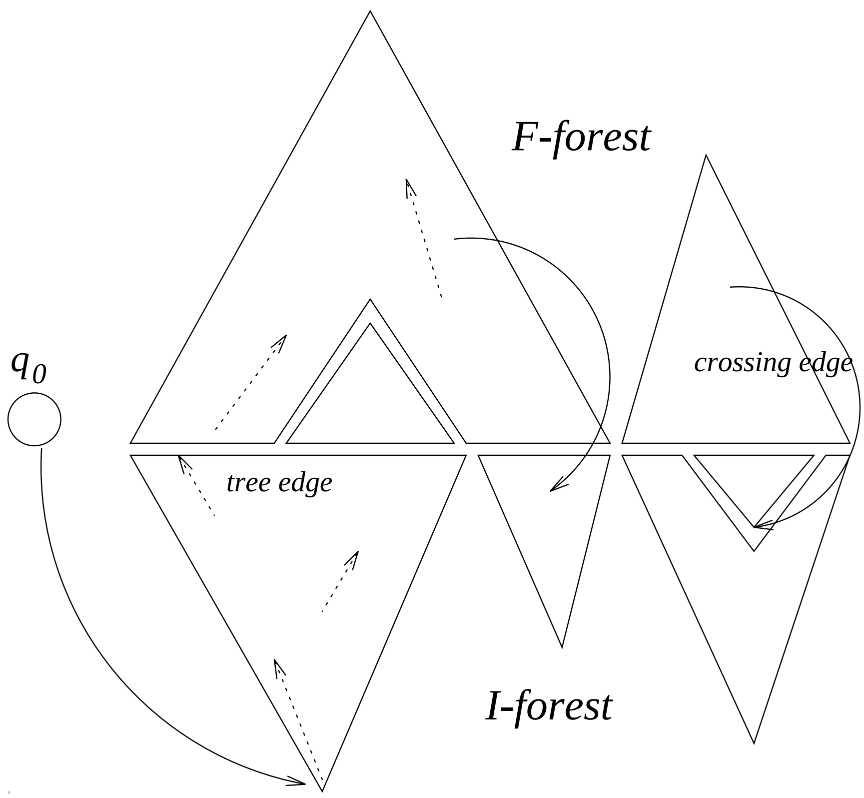


Figure 3.1: The CNNFA organization.

Fig. 3.1 shows how the CNNFA is organized; another convenient view is shown in Fig 3.2 in which *F-forest* and *I-forest* are separated. Fig. 3.3 illustrates a CNNFA equivalent to regular expression $(a|b)^*abb$.

Theorem 3.1 For any regular expression R, its equivalent CNNFA, consisting of *F-forest*, *I-forest* and $lazy\delta$, takes up $O(s)$ space and can be computed in time $O(r)$ and auxiliary space $O(s)$.

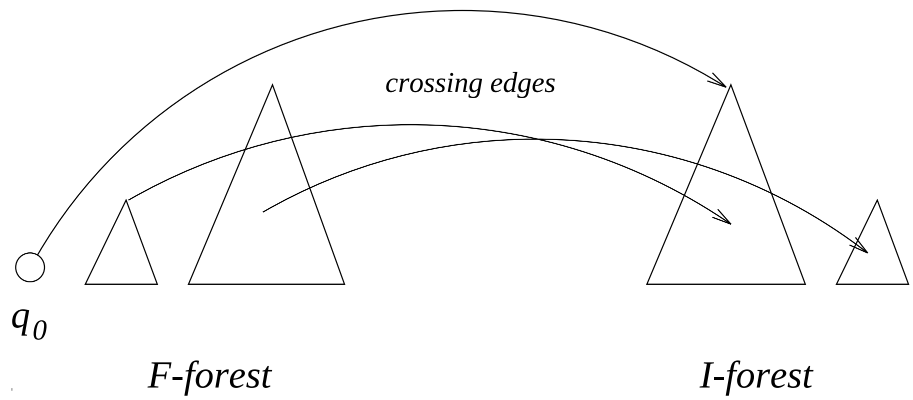


Figure 3.2: Another View of the CNNFA organization.

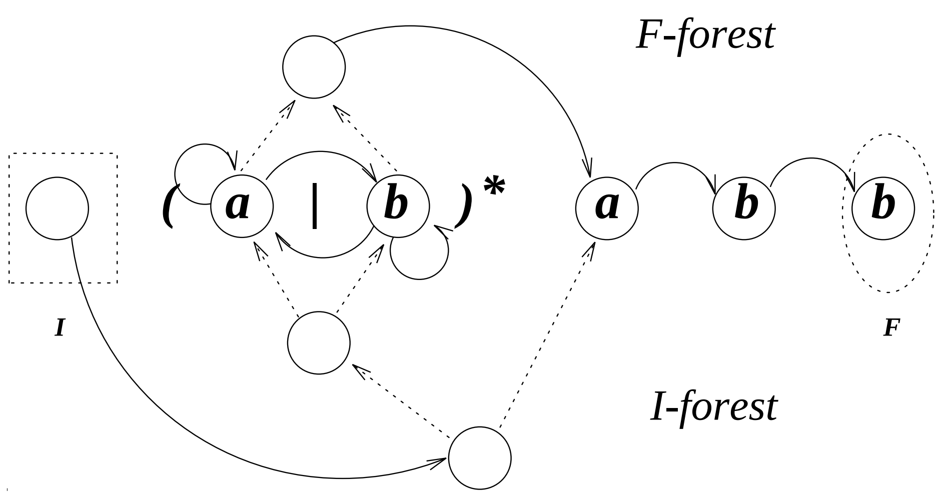


Figure 3.3: A CNNFA equivalent to regular expression $(a|b)^*abb$

Proof: Since each internal node in the *F-forest* and *I-forest* have at least two children, and since their leaves are distinct occurrences of alphabet symbols, they take up $O(s)$ space. Each of the unions in the rules to compute $lazy\delta^T$ is disjoint, and hence takes unit time. By the same argument used to analyze the overall space contributed by Rule (2.21) in the proof of Theorem 2.5, we see that Rule (3.4) contributes $O(s)$ space and $O(r)$ time overall to $lazy\delta^T_R$. By Rule (2.22), Theorem 2.5, and a simple application of structural induction, we also see that the space contributed by Rule (3.5) (which results from adding $lazynred$ to $lazy\delta^T$) overall is $O(s)$. It takes unit time and space to construct $lazy\delta_R$ from $lazy\delta^T_R$ and $null_R$. The overall time bound for each rule is easily seen to be $O(r)$. $\square$

The CNNFA also supports an efficient evaluation of the three preceding queries in order to simulate transition map δ. The best previous worst case time bound for giving a subset V of states and computing the collection of sets $\delta(V, a)$ for all of the alphabet symbols $a \in \Sigma$ is $O(|V| \times |\delta(V, \Sigma)|)$ using an adjacency list implementation of McNaughton and Yamada's NFA, or $\Theta(r)$ using Thompson's NFA.

In Theorem 3.3 we improve this bound, and obtain, essentially, optimal asymptotic time without exceeding $O(s)$ space. This is our main theoretical result. It explains the apparent superior performance of acceptance testing using the CNNFA over Thompson's. It explains more convincingly why constructing a DFA starting from the CNNFA is at least one order of magnitude faster than when we start from either Thompson's or McNaughton and Yamada's NFA. These empirical results are presented in section 5.1.

Before proving the theorem, we will first prove the following technical lemma.

Lemma 3.2 Let V be a set of states in the CNNFA built from regular expression R, and let $lazy\delta_V = \{[X,Y] : [X,Y] \in lazy\delta | X \cap V \neq \emptyset\}$. Then $|lazy\delta_V| = O(|V| + |\delta(V,\Sigma)|)$.

Proof: The result follows from proving that $O(|V| + |\delta(V,\Sigma)|)$ is a bound for each of the subsets of $lazy\delta_V$ contributed by rules (2.19), (2.20), (2.21), and (3.4) respectively. The bound holds for subsets contributed by rules (2.19), (2.20), and (3.4), because they form one-to-one maps.

The proof for the subset contributed by (2.21) is split into two cases. For convenience, let V_J denote the set of states in V such that their corresponding symbol occurrences appear in regular expression J, where J is a subexpression of R. First, consider the set B of pairs $[F_K^T, I_J^T] \in lazy\delta_V$ for subexpressions JK, where $V_J = \emptyset$. We claim that these edges form an one-to-many map, which implies the bound. Suppose this were not the case. Then we would have a subexpression JK, and a subexpression J_1J_2 of J such that $I_J^T = I_{J_1}^T$ and pairs $[F_K^T, I_J^T]$ and $[F_{J_2}^T, I_{J_1}^T]$ belonging to B. However, since J contains no occurrences of an alphabet symbol in V, then J_2 does not either. Hence, the pair $[F_{J_2}^T, I_{J_1}^T]$ cannot belong to B. Hence,

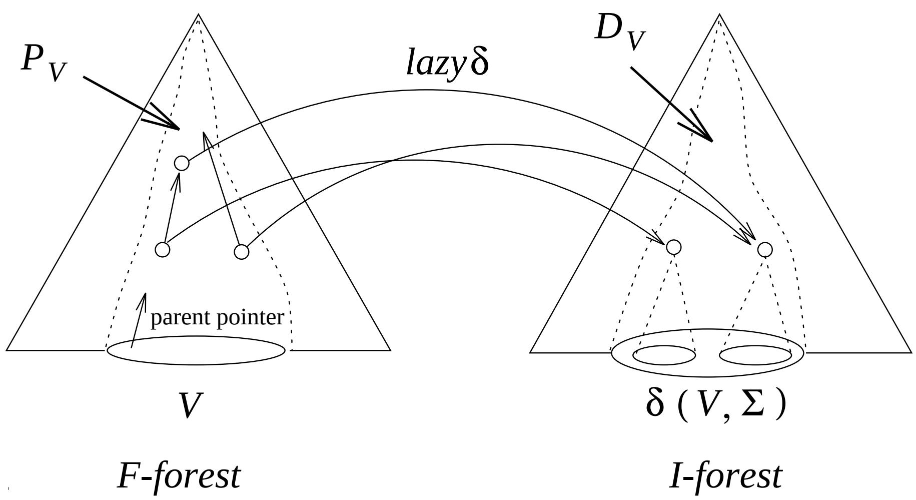


Figure 3.4: To compute $\delta(V, a)$ in a CNNFA

the claim holds.

Next, consider the set C of pairs $[F_K^T, I_J^T] \in lazy\delta_V$ for subexpressions JK, where $V_J \neq \emptyset$. Proceeding from inner-most to outer-most subexpression JK, we charge each pair $[F_K^T, I_J^T] \in C$ to an uncharged state in V_J. A simple structural induction would show that V_J contains at least one uncharged state. Let $J_1 J_2$ be an inner-most subexpression in R such that V_{J_1} is nonempty, and $[F_{J_2}^T, I_{J_1}^T] \in lazy\delta_V$. Then both V_{J_1} and V_{J_2} contains at least one uncharged state. After an uncharged state in V_{J_1} is charged, $V_{J_1 J_2}$ still contains an uncharged state from V_{J_2}. The inductive step is similar. The result follows. $\Box$

Theorem 3.3 Given any subset V of CNNFA states, we can compute all of the sets $\delta(V, a)$ for every alphabet symbols $a \in \Sigma$ in time $O(|V| + |\delta(V, \Sigma)|)$.

Proof: The sets belonging to $finddomain(V)$ are represented by all the nodes P_V along the paths from the states belonging to V to the roots of the *F-forest*. These nodes P_V can be found in $O(|V| + |P_V|)$ time by a marked traversal of parent pointers in the forest. Observe that $|P_V|$ can be much larger than $|V|$.

Computing $next\_states(V)$ involves two steps. First, for each node $n \in P_V$, we traverse a nonempty list of nodes in the *I-forest* representing $\{Y : [Fset(n), Y] \in lazy\delta\}$. This step takes time linear in the sum of the lengths of these lists. (Observe that this number can be much larger than $|P_V|$.) Second, if D_V is the set of all nodes in the *I-forest* belonging to these lists, then $next\_states(V) = \{Iset(n) : n \in D_V\}$. We can compute the set $next\_states(V)$ in $O(|\{[Fset(n), Y] : n \in P_V, [Fset(n), Y] \in lazy\delta\}|) = O(|V| + |\delta(V, \Sigma)|)$ time by Lemma 3.2.

Calculating $\delta(V, \Sigma)$ involves computing the union of the sets belonging to $next\_states(V)$. This is achieved in $O(|\delta(V, \Sigma)|)$ time using the left and right descendant pointers stored in each node belonging to D_V, traversing the unmarked leaves in the frontier, and marking leaves as they are traversed. Multiset discrimination [9] can be used to separate out all of the sets $\{q \in \delta(V, \Sigma) | A(q) = a\}$ for each $a \in \Sigma$ in time $O(|\delta(V, \Sigma)|)$. See Fig. 3.4 for an illustration of $\delta(V, a)$ computation. $\square$

Consider an NFA constructed from the following regular expression:

$$(\ \overbrace{\lambda|(\lambda|(\cdots(\lambda|a)^*)^*)\cdots)^*}^{k\ *'s}\ )^n$$

In order to follow transitions labeled "a", we have to examine $\Theta(n^2)$ edges and $\Theta(n)$ states in $O(n^2)$ time for McNaughton and Yamada's NFA, $\Theta(kn)$ states and edges in $\Theta(kn)$ time for Thompson's machine, and $\Theta(n)$ states and edges in $\Theta(n)$ time for the CNNFA.

3.2 Optimizing the CNNFA

In this section, we introduce two simple transformations, *packing* and *path compression* that can greatly improve the CNNFA representation. For simplicity, we sometimes use F_J and I_J to denote the *F-set* and *I-set* of the tail machine M_J^T respectively.

Packing is defined in terms of a simpler transformation called *promotion.* If $lazy\delta$ contains both $[F_1, I]$ and $[F_2, I]$, and if there exists an *F-set* $F = F_1 \cup F_2$, then the *F-set promotion* transformation replaces $[F_1, I]$ and $[F_2, I]$ within $lazy\delta$ by a single pair $[F, I]$ (see Fig. 3.5). Similarly, if $lazy\delta$ contains both $[F, I_1]$ and $[F, I_2]$, and if there exists an *I-set* $I = I_1 \cup I_2$, then the *I-set promotion* transformation replaces $[F, I_1]$ and $[F, I_2]$ within $lazy\delta$ by a single pair $[F, I]$ (see Fig. 3.6). The packing transformation involves a bottom-up traversal of the *F-forest* in which *I-set* and *F-set* promotion are performed. If *F-set* node v is a leaf, then by applying the *I-set* promotion

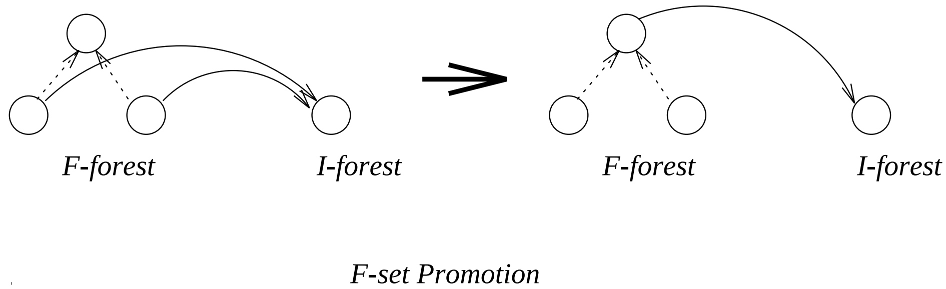


Figure 3.5: *F-set* promotion

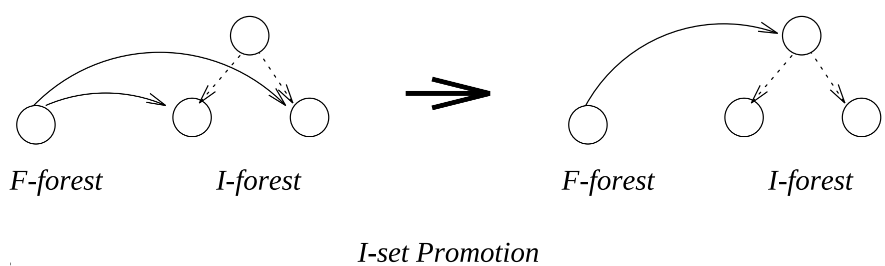


Figure 3.6: *I-set* promotion

to edges attached to v. If v is an internal *F-set* node whose left child v_l and right child v_r are both processed, then v is processed by first applying *F-set* promotion to edges attached to v_l and v_r, and then applying *I-set* promotion to edges attached to v.

In a single linear time pass, we can perform the packing transformation. Before we discuss our packing algorithm, we need some additional terminology. Every *I-set* is a collection of alphabet symbol occurrences. We say that *I-set* I_1 is *less than I-set* I_2 if the rightmost symbol occurrence in I_1 is to the left of the rightmost symbol occurrence in I_2. Pair $[F, I_1]$ is *less than* pair $[F, I_2]$ if $I_1 < I_2$. Without loss of generality, we assume that initially, edge list attached to every *F-set* node is in descending order. Moreover, we also maintain this ordering during packing.

Consider an *F-set* node v with left child v_l and right child v_r. Let J_l and J_r be the outermost subexpressions of R whose *I-sets* are denoted by v_l and v_r respectively, and J be the innermost subexpression whose *F-set* is denoted by v. Notice that J must be either $J_l J_r$ or $J_l | J_r$. We shall show in the next Chapter that common tails of edges attached to v_l and v_r must denote I_{J_l}, J_{J_r} or $I_{J_l} \cup I_{J_r}$. Those edges with common tails are heads or tails of the edge lists attached to v_l and v_r. Therefore, to apply *F-set* promotion to edges attached to v_l and v_r we need to examine heads and tails of the edge lists attached to v_l and v_r only.

Let v be an internal *F-set* node denoting F_J, for some subexpression J,

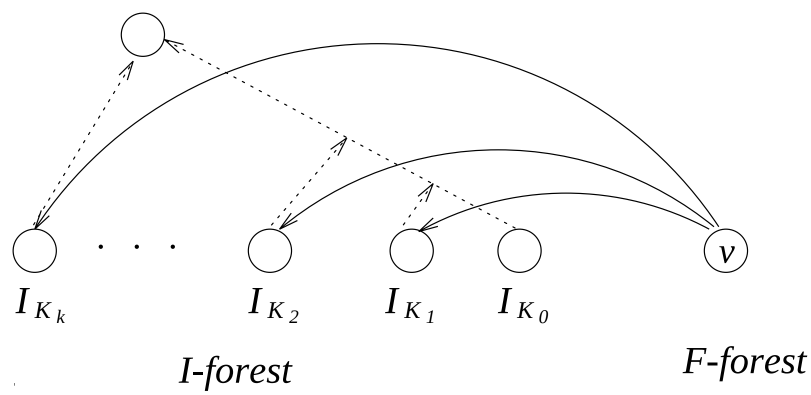


Figure 3.7: *I-set* promotion applied to edges attached to internal *F-set* node v

and $[F_J, I_{K_1}]$, $[F_J, I_{K_2}]$, $\cdots$, $[F_J, I_{K_k}]$ be the original list of edges attached to v, where $K_1, \cdots, K_k$ are the outermost subexpressions in R whose *I-sets* are $I_{K_1}, \cdots, I_{K_k}$ respectively. Assume that J' is the innermost subexpression such that $F_{J'} = F_J$. Consider that case that K_1 is not right to J'. By the CNNFA construction rules, there must exist some superexpression J_1 of J', where $F_{J_1} = F_{J'}$, such that without considering useless λ-expressions, (1) $J_{i+1} = K_i|J_i$ or K_iJ_i, $1 \leq i < k$, is a subexpression of R, and (2) $null_{K_i} = \{\lambda\}$, $1 \leq i \leq k$. Fig 3.7 illustrates the edge structure of $[F_J, I_{K_1}]$, $\cdots$, $[F_J, I_{K_k}]$. Form Figure 3.7, there are edges to be packed if and only if $[F_J, I_{K_0}]$ is constructed by a previous *F-set* promotion. Hence, *I-set* promotion applied to edges attached to v can be done in $O(k)$ time. As to other cases, their packing algorithms are the same. Since each crossing edge is examined by packing transformation at most twice, the packing

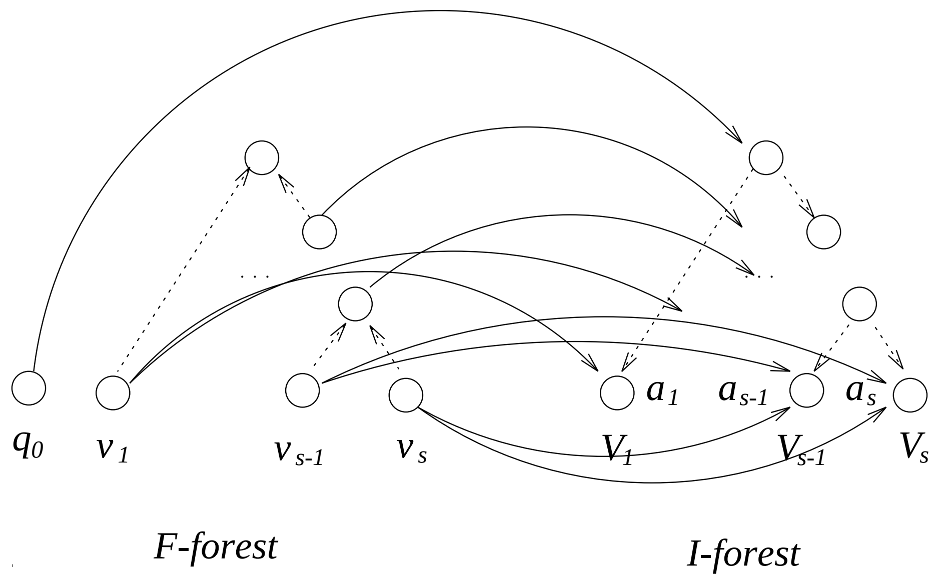


Figure 3.8: A CNNFA equivalent to $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ without optimization.

transformation can be done in $O(s)$ time.

The packing transformation can greatly simplify the representation of $lazy\delta$. In the case of regular expression

$$((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$$

packing can simplify $lazy\delta$ from $3s-1$ pairs into two pairs. The original CNNFA equivalent to $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ is shown in Fig. 3.8. The CNNFA resulting from packing transformation is illustrated in Fig. 3.9.

When applying the packing transformation, we can also carry out the

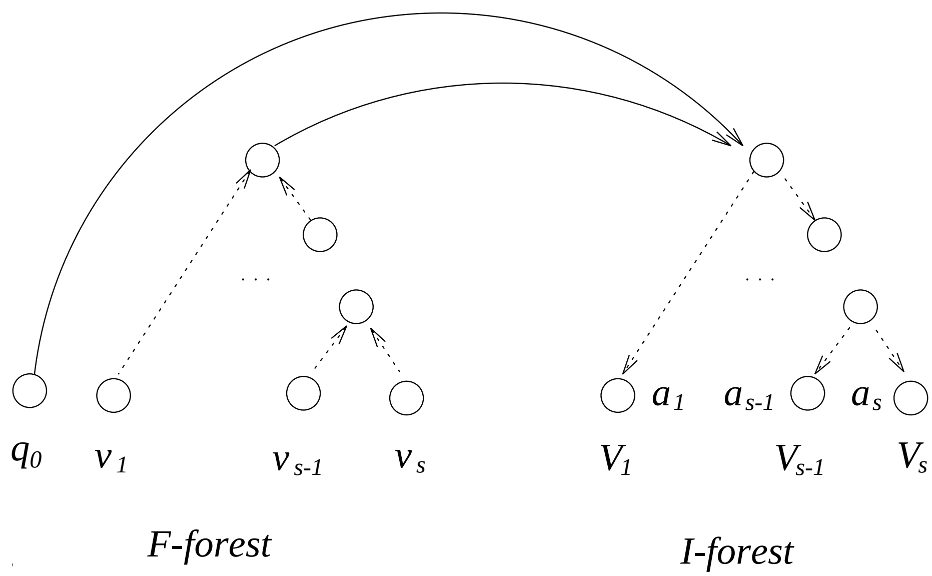


Figure 3.9: A CNNFA equivalent to $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ resulting from Packing.

same kind of path compression described in the previous section, so that the *F-forest* and *I-forest* only contain nodes in the domain (respectively range) of $lazy\delta$. Whereas previously the forest leaves (corresponding to NFA states) were unaffected by compression, the packing transformation can remove leaves in the *F-forest* and *I-forest* from the domain and (respectively) range of $lazy\delta$. Moreover, those *F-set* and *I-set* nodes that denote the same set can be merged into a single node. When path compression eliminates leaves, we need to turn the symbol assignment map A into a multi-valued mapping defined on *I-forest* leaves; that is, whenever *I-forest* leaves $q_1, \cdots, q_k$ are replaced by leaf q, we take the following steps;

- remove the old leaves $q_1, \cdots, q_k$ from the domain of A;
- assign the set of symbols $\{y : x \in \{q_1, \cdots, q_k\}, [x, y] \in A\}$ to A at q.

However, we can also have alphabet symbol occurrences only in those sets denoted by internal *I-forest* nodes. For each such symbol occurrence a, we add a *linking* edge labeled a originating from the lowest *I-forest* node v_1 to the *F-forest* leaf v_2, where a is in both sets denoted by v_1 and v_2 (see Fig. 3.10 for an example). Consider the expression

$$((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$$

again. Path compression will turn the data structure into the one depicted in Fig. 3.11. The original CNNFA has $4s - 1$ states, but after path compression, it has only two states. In using our compressed representation to

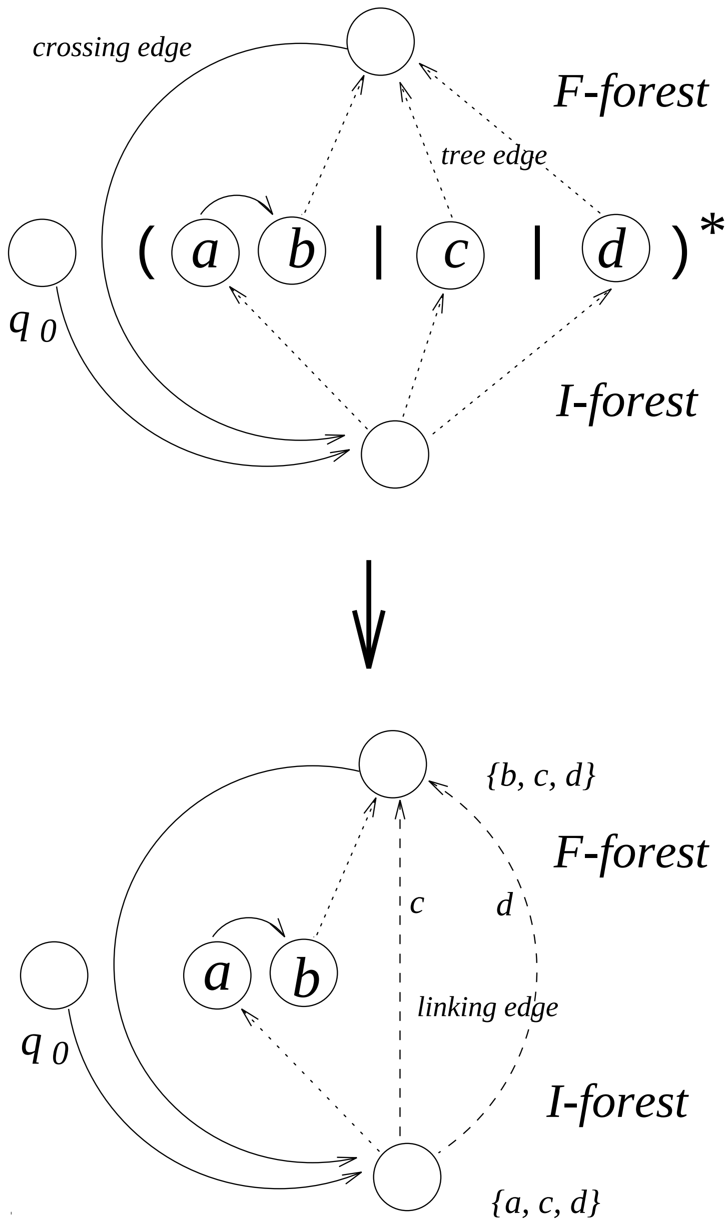


Figure 3.10: Linking edges in the CNNFA for $(ab|c|d)^*$

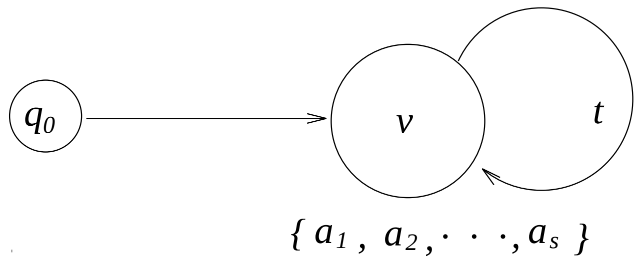


Figure 3.11: A CNNFA equivalent to $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ resulting from packing and path compression.

simulate an NFA, the transition edge t can be taken only when the current transition symbol belongs to $\{a_1, a_2, \cdots a_s\}$ which labels node v.

The reduced number of NFA states, resulting from packing and path compression, partly explains the superior performance of the CNNFA in both acceptance testing and DFA construction. Fig. 3.12 illustrates a CN-NFA resulting from applying packing and path compression to the CNNFA in Fig. 3.3. It contains 6 states and 6 edges in contrast to the 9 states and 14 edges found in the MYNNFA of Fig. 3.3. Hereafter, we assume that the CNNFA is optimized.

Those forest nodes touched by crossing edges are CNNFA states. The following Theorem shows the space complexity of the CNNFA.

Theorem 3.4 Let R be a regular expression containing s occurrences of alphabet symbols. If there are c crossing edges (excluding $[q_0, I_R]$) and f forest nodes touched by crossing edges (excluding q_0) in CNNFA M_R, then

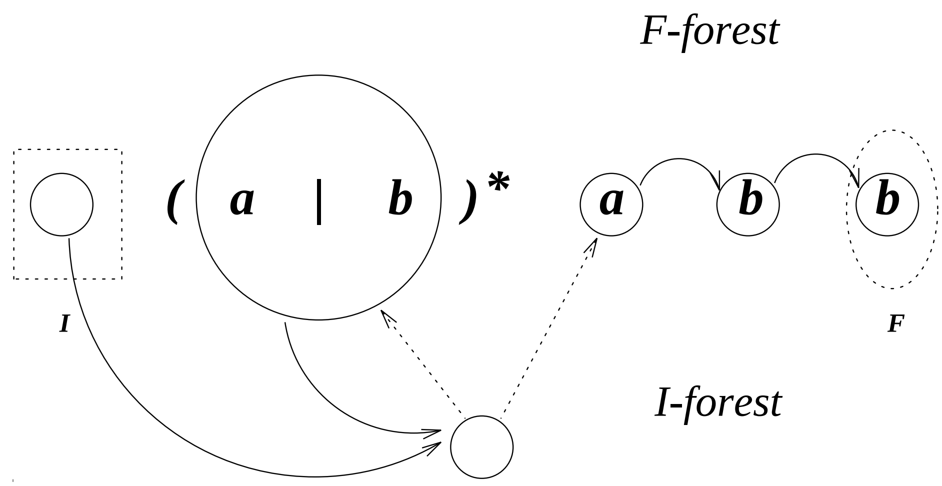


Figure 3.12: A CNNFA equivalent to $(a|b)^*abb$

M_R has exactly $f+1$ states and $f+c+s-1$ edges at most.

Proof: It is clear that including the initial state, there are $f+1$ states. Suppose there are k forest nodes touched by crossing edges served as both *I-forest* and *F-forest* leaves. Then there are no more than $f+k-2$ tree edges. Notice that k is smaller or equal to s. We have at most $s-k$ linking edges. By summing up the number of tree, crossing, linking edges and $[q_0, I_R]$, we have at most $(f+k-2)+c+(s-k)+1 = f+c+s-1$ edges in M_R. $\square$

Chapter 4

Analysis of the CNNFA

In this Chapter we analyze the time and space complexities of the CNNFA more precisely than in the previous chapter. Section 4.1 counts crossing edges. Section 4.2 shows that there are $5s/2$ CNNFA states in the CNNFA constructed from a regular expression with s alphabet symbol occurrences. In section 4.3 we compare the CNNFA with Thompson's NFA.

4.1 Crossing Edges

In this section, we present several tight bounds regarding crossing edges. Previously we showed that given any subset V of the unoptimized CNNFA states, to compute the set $U = \delta(V, \Sigma)$ at most $O(|V| + |U|)$ crossing edges are visited. We shall show that $\min(3|U|/2, |V| + |U| - 1)$ is a better bound on the number of crossing edges. We use $cnnfa_R$ to denote the CNNFA for regular expression R, and use $cnnfa\delta_R$ to denote the set of crossing edges in $cnnfa_R$. We shall also show that CNNFA $cnnfa_R$ has no more than $(3s+1)/2$ crossing edges as a corollary, where R contains s alphabet symbol occurrences. Throughout this and next section, the forest structure in the CNNFA is a collection of branching binary trees (i.e. we do not perform path compression transformation), and we assume that no regular expressions contain λ subexpressions. Our argument can be easily extended to the general case.

4.1.1 The Relaxed CNNFA

Without loss of generality, we will analyze a relaxed form of the CNNFA that is larger and slower than the CNNFA. It can be converted into the CNNFA easily. There are two kinds of crossing edges in the relaxed CNNFA: one stems from products, and the other from star operators. We first present properties of the CNNFA.

Lemma 4.1 Let R be a regular expression. If $\lambda \notin null_R$, then there exists a pair $[a,b]$, $[a,b] \in F_R \times I_R$, such that $[a,b] \notin \delta_R$.

Proof: Trivial. $\Box$

Corollary 4.2 Let R be a regular expression. If $\lambda \notin null_R$, then there is no crossing edge $[F_R, I_R]$ in both $lazy\delta_R$ and $cnnfa\delta_R$.

Proof: Directly from Lemma 4.1. $\Box$

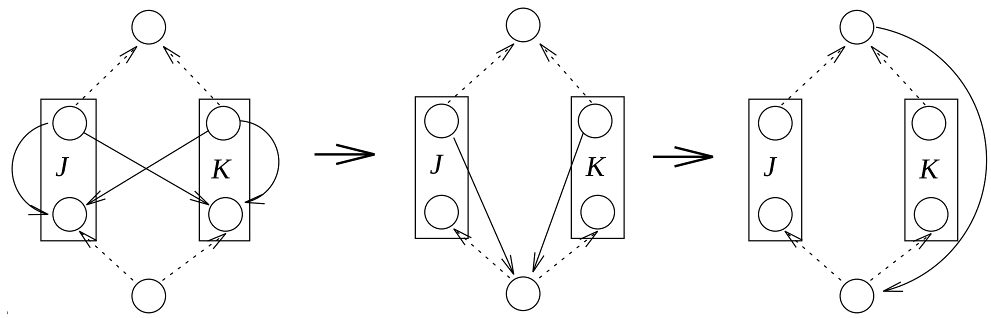


Figure 4.1: To derive $[F_{(J|K)^*}, I_{(J|K)^*}]$ in the CNNFA for $(J|K)^*$.

Lemma 4.3 Let R be a regular expression. Then, $[F_R, I_R] \in cnnfa\delta_{R^*}$.

Proof: It is clear for the base case that $[F_a, I_a] \in cnnfa\delta_{a^*}$, where a is an alphabet symbol. If $R = J^*$, then, by the induction hypothesis, $[F_J, I_J] \in cnnfa\delta_{J^*}$. Therefore, $[F_R, I_R] \in cnnfa\delta_{R^*}$.

Consider the case $R = J|K$. We have $lazy\delta_{R^*} = lazy\delta_{J^*} \cup lazy\delta_{K^*} \cup \{[F_J, I_K], [F_K, I_J]\}$. By the induction hypothesis, $[F_J, I_J] \in cnnfa\delta_{J^*}$, and $[F_K, I_K] \in cnnfa\delta_{K^*}$. By further applications of *F-set* and *I-set* promotion, we pack $[F_J, I_J]$, $[F_J, I_K]$, $[F_K, I_J]$ and $[F_K, I_K]$ into $[F_R, I_R]$. Fig. 4.1 shows how it is accomplished. We use a rectangle labeled J to denote CNNFA $cnnfa_J$. The upper circle in $cnnfa_J$ is the *F-set* node denoting F_J, and the lower circle is the *I-set* node denoting I_J.

The proof for $R = JK$ is split into four cases. If both $null_J$ and $null_K$ are $\{\lambda\}$, then $lazy\delta_{R^*} = lazy\delta_{J^*} \cup lazy\delta_{K^*} \cup \{[F_J, I_K], [F_K, I_J]\}$. Like the $R = J|K$ case, we have $[F_R, I_R] \in cnnfa\delta_{R^*}$.

Consider the case that both $null_J$ and $null_K$ are empty. Then, $I_{R^*} = I_J$, $F_{R^*} = F_K$ and $lazy\delta_{R^*} = lazy\delta_J \cup lazy\delta_K \cup \{[F_J, I_K], [F_K, I_J]\}$. Since both $null_J$ and $null_K$ are empty, $[F_K, I_J]$ cannot be packed with any edge in $cnnfa\delta_J \cup cnnfa\delta_K \cup \{[F_J, I_K]\}$. Therefore, $[F_R, I_R] = [F_K, I_J] \in cnnfa\delta_{R^*}$.

Next consider the case that $null_J = \{\lambda\}$ and $null_K = \emptyset$. Then, $I_{R^*} = I_J \cup I_K$, $F_{R^*} = F_K$ and $lazy\delta_{R^*} = lazy\delta_J \cup lazy\delta_{K^*} \cup \{[F_J, I_K], [F_K, I_J]\}$. By the induction hypothesis, $[F_K, I_K] \in cnnfa\delta_{K^*}$. An application of *I-set* promotion packs $[F_K, I_K]$ and $[F_K, I_J]$ into $[F_R, I_R]$.

Finally consider the case that $null_J = \emptyset$ and $null_K = \{\lambda\}$. Then, $I_{R^*} = I_J$, $F_{R^*} = F_J \cup F_K$ and $lazy\delta_{R^*} = lazy\delta_{J^*} \cup lazy\delta_K \cup \{[F_J, I_K], [F_K, I_J]\}$. Though, by the induction hypothesis, $[F_J, I_J] \in cnnfa\delta_{J^*}$, $[F_J, I_J]$ cannot be packed with $[F_J, I_K]$. Furthermore $[F_K, I_J]$ cannot be packed with any edges in $cnnfa\delta_K$ even when $[F_K, I_K] \in cnnfa\delta_K$. An application of *F-set* promotion packs $[F_J, I_J]$ and $[F_K, I_J]$ into $[F_R, I_R]$. $\square$

Before continuing our discussion, we need some additional terminology. Let R be a regular expression, and J and K be subexpressions of R. If $R = K^*$, then R is a *star-top* regular expression. A crossing edge $[F, I]$ is a *product* edge with respect to J and K if $F = F_J$ and $I = I_K$, and if JK is a subexpression of R. We sometimes simply use $[F_J, I_K]$ to denote the product edge with respect to J and K. A crossing edge $[F, I]$ is a *star* edge with respect to J if $F = F_J$ and $I = I_J$. A star edge $[F, I]$ with respect to J is *real* if J is star-top. We sometimes use $[F_{J^*}, I_{J^*}]$ to denote the star edge with respect to J^*. For convenience, we call a real star edge $[F_{J^*}, I_{J^*}]$ a *left star* edge if J^*K or $J^*|K$ is a subexpression; otherwise, we regard the real star edge $[F_{J^*}, I_{J^*}]$ as a *right star* edge. Crossing edge $[I, F]$ is a *lazy* edge if $[I, F]$ is kept in the set *lazynred* first, and if it is added to $lazy\delta_R$ by rule 2.17. Two crossing edges $[F_1, I_1]$ and $[F_2, I_2]$ are *disjoint* if $F_1 \times I_1 \cap F_2 \times I_2 = \emptyset$. By definition, all the edges in $lazy\delta_R$ and $cnnfa\delta_R$ are mutually disjoint.

Lemma 4.4 For any regular expression R, except real star edges, all the lazy edges in $lazy\delta_R$ are not in $cnnfa\delta_R$.

Proof: Suppose lazy edge $[F_J, I_K] \in lazy\delta_R$ is not a real star edge, but $[F_J, I_K] \in cnnfa\delta_R$. Since $[F_J, I_K] \in lazy\delta_R$, there must exist a regular expression Q^* such that Q^* is the innermost subexpression in R containing both J and K. Moreover, by the CNNFA construction rules, $F_K \subseteq F_{Q^*}$ and $I_J \subseteq I_{Q^*}$. By Lemma 4.3, $[F_{Q^*}, I_{Q^*}] \in cnnfa\delta_{Q^*}$. Since all the edges in $cnnfa\delta_{Q^*}$ are mutually disjoint, $[F_J, I_K] \notin cnnfa\delta_{Q^*}$. Hence, $[F_J, I_K] \notin cnnfa\delta_R$, a contradiction. $\square$

We shall relax our packing transformation to facilitate our argument: namely, promotions are applied only if they introduce new star edges. We call this relaxed transformation the *relaxed packing* transformation. We use $cnnfa'_R$ to denote the relaxed CNNFA resulted from applying relaxed packing transformations to $lazy\delta_R$, and use $cnnfa'\delta_R$ to denote the set of crossing edges in $cnnfa'_R$.

Lemma 4.5 For any regular expression R, $cnnfa'\delta_R$ contains only product and real star edges.

Proof: Originally, $lazy\delta_R$ contained product and lazy edges. The relaxed packing transformation, by Lemma 4.4, eliminates all the lazy edges that are not real star edges. In addition, relaxed packing transformations only introduce real star edges. □

It is interesting to note that the relaxed CNNFA is closely related to the star normal form notation for regular expressions [6] used by Brüggemann-Klein to develop a linear time McNaughton/Yamada NFA construction algorithm.

The crossing edge set $cnnfa'\delta$ is easy to analyze, and is crucial to our argument. However, before further discussion, we need to study more properties of crossing edges in the relaxed CNNFA.

Lemma 4.6

1. Every *F-set* node has at most one outgoing product edge and/or one outgoing star edge, and every *I-set* node has at most one incoming product edge and/or one incoming star edge,
2. Let *I-set* node n have left child n_1 and right child n_2. *I-set* node n, n_1 and n_2 denote *I-set* I, I_1 and I_2 respectively. Assume that J_1

and J_2 are the outermost subexpressions of R such that $I_{J_1} = I_1$ and $I_{J_2} = I_1$. Then, there is a subexpression K which is either $J_1 J_2$ or $J_1|J_2$ (depends on R) such that $I_K = I$. Furthermore, if $null_{J_2} = \{\lambda\}$, then the outermost subexpression K_1 of R, in which $I_{K_1} = I$, is $K^{**\cdots*}$, where there is zero or more $*$ on top of K.

3. For any regular expression R, there is no product edge originating from the *F-set* node denoting F_R and there is also no product edge reaching the *I-set* node denoting I_R.

Proof: (1) is trivial.

As to (2), either $J_1 J_2$ or $J_1|J_2$ must be a subexpression of R, and $null_{J_1} = \{\lambda\}$. Therefore, $I_K = I$. If $null_{J_2} = \{\lambda\}$, then $null_K = \{\lambda\}$. Thus, the outermost subexpression whose *I-set* is I_K must be of the form $K^{*\cdots*}$.

(3) can be easily proved by an inductive argument. □

A pair of crossing edges can be packed if and only if they share the same origin or tail, and their tails or origins are siblings. Since every *F-set* (respectively *I-set*) node has at most one outgoing (respectively incoming) product edge, no pairs of product edges can be packed. Similarly, every *F-set* and *I-set* node is touched by at most one star edge. Therefore, no pairs of star edges can be packed. By edge e *touching* node v, we mean that node v is either the origin or the tail of edge e. We shall see that a pair of crossing edges can be packed only if one of them is a product edge, and if

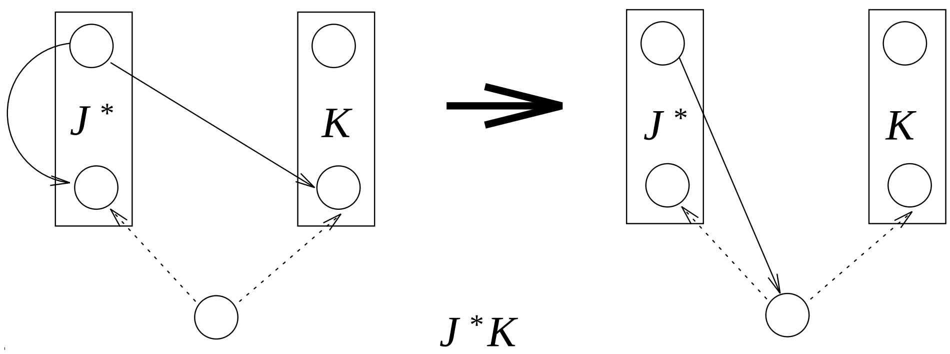


Figure 4.2: Edge Packing in $cnnfa'_{J^*K}$

the other is a star edge. There are two basic patterns of edge packing in the relaxed CNNFA.

- two edges share the same origin: Let star edge $[F_{J^*}, I_{J^*}]$ and product edge $[F_{J_1} = F_{J^*}, I_K]$, where J_1K is a subexpression, share the same origin, and let the *I-set* nodes denoting I_{J^*} and I_K be siblings. These two edges can be packed into $[F_{J^*}, I_{J^*} \cup I_K]$. By (3) in Lemma 4.6, J_1 is not a proper subexpression of J. However, J_1 is not a proper super expression of J^* since $null_{J^*} = \{\lambda\}$. Therefore, $J^* = J_1$, i.e. J^*K is a subexpression of R (see Fig. 4.2).

- two edges share the same tail: Let star edge $[F_{K^*}, I_{K^*}]$ and product edge $[F_J, I_{K_1} = I_{K^*}]$, where JK_1 is a subexpression, share the same tail, and let the *F-set* nodes denoting F_{K^*} and F_J be siblings. These two edges can be packed into $[F_J \cup F_{K^*}, I_{K^*}]$. By (3) in Lemma 4.6,

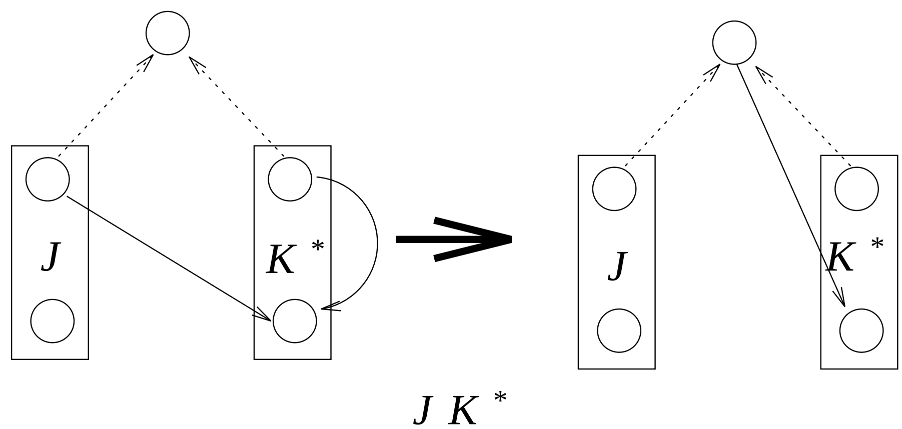


Figure 4.3: Edge Packing in $cnnfa'_{JK^*}$

K_1 is not a proper subexpression of K. However, K_1 is not a proper super expression of K^* since $null_{K^*} = \{\lambda\}$. Therefore, $K^* = K_1$, i.e. JK^* is a subexpression of R (see Fig. 4.3).

A crossing edge that results from packing a star and a product edge is a *singly promoted* edge. We shall see that except for the two patterns previously stated, there is no other case in which promotion can be applied.

Lemma 4.7 No singly promoted edge can be involved in an application of *F-set* or *I-set* promotion.

Proof: We prove this lemma by a case analysis. A singly promoted edge cannot be packed with a star edge. Consider a singly promoted edge $[F_{J^*}, I_{J^*K}]$, where J^*K is a subexpression. The only star edge originating from the node denoting F_{J^*} is $[F_{J^*}, I_{J^*}]$. There is no star edge reaching the node denoting I_{J^*K} while it originates from an *F-set* node that is the sibling of the node denoting F_{J^*}. Therefore, no star edge can be packed with $[F_{J^*}, I_{J^*K}]$. Similarly, no star edge can be packed with a singly promoted edge $[F_{JK^*}, I_{K^*}]$.

A singly promoted edge cannot be packed with a product edge. Consider a singly promoted edge $[F_{J^*}, I_{J^*K}]$, where J^*K is a subexpression. There is no product edge other than $[F_{J^*}, I_K]$ originating from the node denoting F_{J^*}. There is no product edge reaching the node denoting I_{J^*K} while it originates from an *F-set* node that is the sibling of the node denoting F_{J^*}. Therefore, no product edge can be packed with $[F_{J^*}, I_{J^*K}]$. Similarly, no product edge can be packed with a singly promoted edge $[F_{JK^*}, I_{K^*}]$.

Two singly promoted edges cannot be packed. Consider the case in which $[F_{J^*}, I_{J^*K}]$, is a singly promoted edge, where J^*K is a subexpression. The only singly promoted edge, if there is any, other than $[F_{J^*}, I_{J^*K}]$ originating from the *F-set* node denoting F_{J^*} is $[F_{J_1J_2^*} = F_{J^*}, I_{J_2^*}]$, where $J_1J_2^*$ is a subexpression. Regular expression J^* is not a subexpression of $J_1J_2^*$ because $F_{J^*} = F_{J_1J_2^*}$. Moreover, if $J_1J_2^*$ is a subexpression of J^*, then the node denoting $I_{J_2^*}$ is not the sibling of the node denoting I_{J^*K}. Therefore, singly promoted edges $[F_{J^*}, I_{J^*K}]$ and $[F_{J_1J_2^*}, I_{J_2^*}]$ cannot be packed. The only singly promoted edge, if there is any, other than $[F_{J^*}, I_{J^*K}]$ reaching *I-set* node denoting I_{J^*K} is $[F_{K_1K_2^*}, I_{K_2^*} = I_{J^*K}]$, where $K_1K_2^*$ is a subexpression. For a similar reason, $[F_{J^*}, I_{J^*K}]$ cannot be packed with $[F_{K_1K_2^*}, I_{K_2^*}]$.

The proof for the singly promoted edge $[F_{JK^*}, I_{K^*}]$ case is similar. $\Box$

From preceding discussion, we can easily convert a relaxed CNNFA $cnnfa'_R$ into CNNFA $cnnfa_R$ only by packing pairs of product and star edges of $cnnfa'_R$ that share the same origin or tail. Moreover, it suggests a more efficient CNNFA construction algorithm. The packing transformation presented in the previous chapter is slightly inefficient. Nevertheless, it suggests a formal derivation of the CNNFA. Since singly promoted edges cannot be promoted again, we call them *promoted* edges, for short.

4.1.2 Counting Crossing Edges

Definition 4.8 Let T_v be a branching binary tree rooted at node v. All the right children in T_v and root v are *charged* nodes. If we only charge credit to charged nodes in T_v, then T_v is a *charged tree.* A k-leaf charged tree has exactly k charged nodes. By definition, tails of product edges are charged nodes.

Theorem 4.9 Let V be a set of CNNFA states such that $V = \delta(\{q_0\}, x)$, for some string $x \in \Sigma^*$. To compute the set $U = \delta(V, \Sigma)$ at most $|V|+|U|-1$ crossing edges are visited.

Proof: For each crossing edges visited, we charge one unit of credit to a charged nodes. Only charged nodes are charge. Charged nodes are at most doubly charged. We shall show that there are at most $|U|$ charged nodes being charged, and among them at most $|V|-1$ nodes are doubly charged. Therefore, there are at most $|V|+|U|-1$ crossing edges visited.

We use the following charging scheme.

1. For each product edge visited, we charge one unit of credit to its tail.

2. If the tail of a visited star edge is a charged node, then we charge one unit of credit to its tail; otherwise, we charged one unit of credit to the sibling of its tail (which is a charged node).

3. For each promoted edge visited, we give one unit of credit to the tail of the product edge which this promoted edge is promoted from.

4. If $[q_0, I_R]$ is visited, then we give one unit of credit to the node denoting I_R.

Under this charging scheme, only charged nodes can be charged, all the charged node are at most singly charged except those *I-set* nodes denoting I_K, where J^*K^* or $J^*|K^*$ is a subexpression. All the *I-set* nodes are at most doubly charged (see Fig. 4.4).

An *I-set* I is *maximal* with respect to U if $I \subseteq U$, and if there is no *I-set* I' such that $I \subset I' \subseteq U$. An *I-set* node is *maximal* with respect to U if it denotes a maximal *I-set*. If a maximal *I-set* node v is not a charged node, then we move the credit which is charged to the sibling of v to v. After credit being moved, only the descendents of maximal *I-set* nodes can

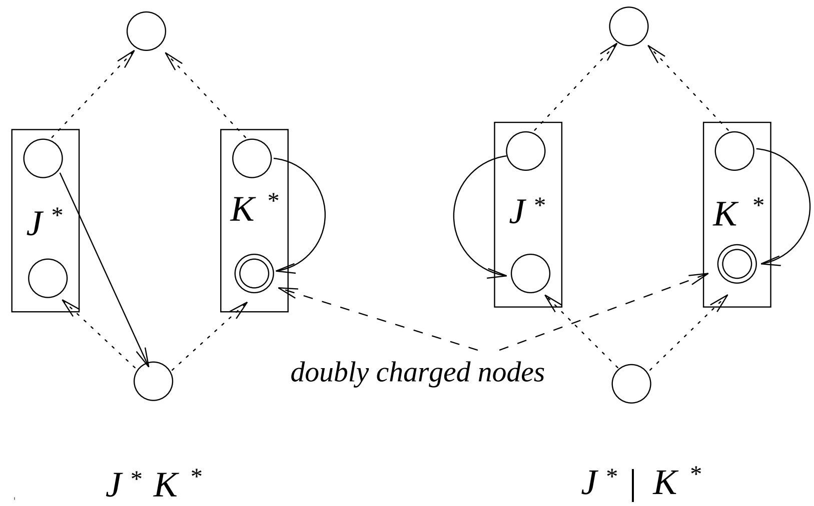


Figure 4.4: Patterns of doubly charged node

be charged, and subtrees rooted at maximal *I-set* nodes are charged trees. Therefore, there are at most $|U|$ charged nodes being charged.

Every maximal *I-set* node v is at most singly charged. Consider the case that v is a root in the *I-forest*. If v is not the node denoting I_R, then v is at most singly charged because there is at most one crossing edge from which v can be charged. If v is the node denoting I_R, then v can be doubly charged only when R is star-top (one is due to star edge $[F_R, I_R]$, the other is due to $[q_0, I_R]$). However, because the start state q_0 has no incoming edges, these two edges cannot be visited simultaneously during acceptance testing. Therefore, v is at most singly charged.

Suppose a maximal *I-set* node v is doubly charged, where v is not a root in the *I-forest*. But it is not the case because if v is doubly charged, then all the leaves of the subtree rooted at the sibling of v are also in U (see Fig. 4.4). Therefore, v is not a maximal *I-set* node. A contradiction.

By a bottom-up structural induction on the structure of regular expressions, we can show that there are at most $|V| - 1$ doubly charged nodes. Recalling that an *I-set* node v is doubly charged, then v denotes some *I-set* I_K, where J^*K^* or $J^*|K^*$ is a subexpression, and both J^* and K^* contain at least one alphabet symbol occurrence in V. Let each alphabet symbol occurrence in V carry one unit of credit. If J^*K^* or $J^*|K^*$ is a subexpression of R, and if both J^* and K^* contain at least one alphabet symbol occurrence in V, we then pay one unit of credit to the node denoting I_K from the credit that J^*K^* or $J^*|K^*$ processes. The inductive argument that any subexpression of R, which contains alphabet symbol occurrences in V, processes at least one unit of credit is easy to see. Hence, there are at most $|V| - 1$ doubly charged nodes. □

The CNNFA for $((\cdots((a_1|\lambda)(a_2|\lambda))\cdots)(a_s|\lambda))$ is shown in Fig. 4.5. If V is $\{a_1\}$, then $U = \{a_2 \cdots a_s\}$, and there are $s-1$ crossing edges visited.

We shall show in the next theorem that $3|U|/2$ is also a bound of the number of crossing edges visited. Before we proceed, we need to prove the following technical lemma.

Lemma 4.10 Let K_n^* be a subexpression of R, and *I-set* node v_n denote $I_{K_n^*}$. There is no doubly charged *I-set* node in the right path from the right

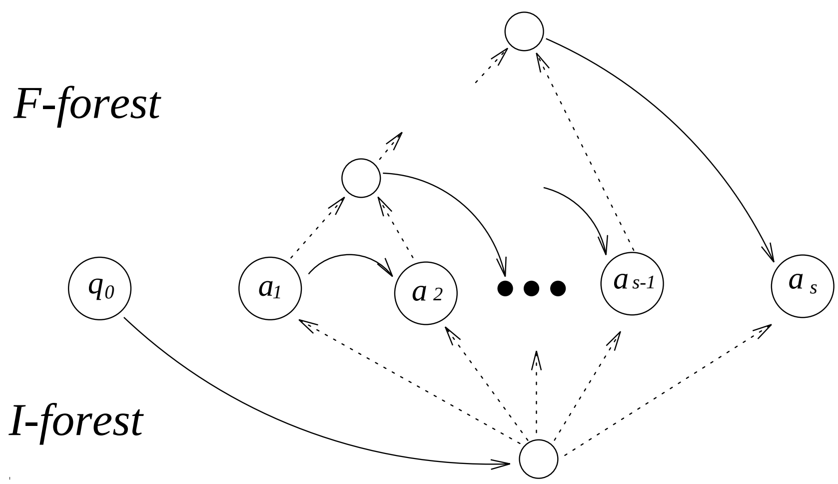


Figure 4.5: The CNNFA for $((\cdots((a_1|\lambda)(a_2|\lambda))\cdots)(a_s|\lambda))$

child of v_n to a leaf in the *I-forest*.

Proof: Suppose an *I-set* node v_1 in the right path from the right child of v_n is doubly charged (see Fig. 4.6). Since v_1 is doubly charged, the outermost subexpression, whose *I-set* is denoted by v_1, must be K_1^*, for some regular expression K_1^*, and $[F_{K_1^*}, I_{K_1^*}] \in cnnfa\delta_R$. Let J_i, $1 \le i < n$, be the outermost subexpression whose *I-set* is denoted by the sibling of v_i. Because each node denoting I_{J_i}, $1 \le i < n$, has a right sibling, $null_{J_i} = \{\lambda\}$. Then, by (2) of Lemma 4.6, the outermost subexpression K_i, $1 < i \le n$, whose *I-set* is denoted by v_i, is of the form $(J_{i-1}K_{i-1})^{**\cdots*}$ or $(J_{i-1}|K_{i-1})^{**\cdots*}$. Therefore, $F_{K_1^*} \subset F_{K_n^*}$ and $I_{K_1^*} \subset I_{K_n^*}$. However, by Lemma 4.3, $[F_{K^*}, I_{K^*}] \in cnnfa\delta_{K_n^*}$. Then, by disjointness of $cnnfa\delta$, $[F_{K_1^*}, I_{K_1^*}] \notin cnnfa\delta_{K_n^*}$. A contradiction. $\square$

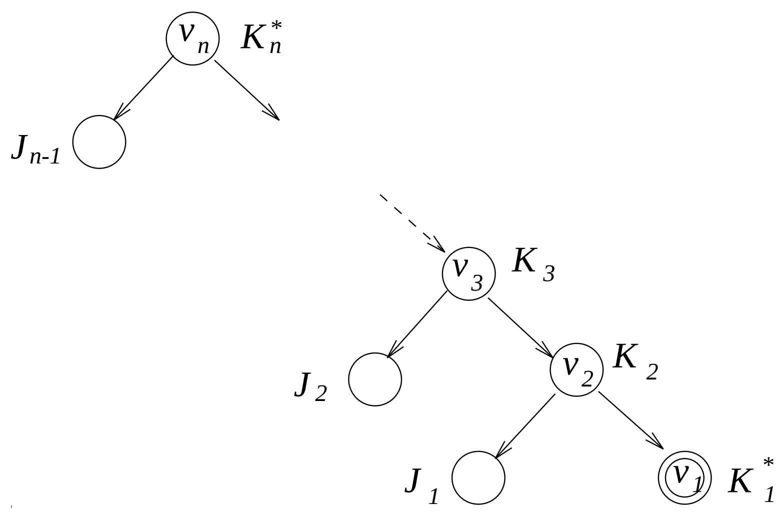


Figure 4.6: No doubly charged node in the right path

Theorem 4.11 Let V be a set of CNNFA states such that $V = \delta(\{q_0\}, x)$, for some string $x \in \Sigma^*$. To compute the set $U = \delta(V, \Sigma)$ at most $3|U|/2$ crossing edges are visited.

Proof: Using the same charging scheme as Theorem 4.9, we summarize the following charging properties.

- Only charged nodes can be charged, and they are at most doubly charged.
- Maximal *I-set* nodes with respect to U is at most singly charged.
- By Lemma 4.10, if *I-set* node v is doubly charged, then all the nodes in the right paths from both the right child and the left sibling of v

to a leaf are at most singly charged.

A charged tree is *feasible* if it satisfies the charging constraint listed above. A feasible charged tree is *unsaturated* if there is no doubly charged node in the right path from the root to a leaf. We count the number of visited edges during $\delta(V, \Sigma)$ computation by calculating how much credit can be charged to feasible charged trees.

By an inductive argument, we shall show that a k-leaf feasible charged tree is charged at most $3k/2$ unit of credit, and a k-leaf unsaturated feasible charged tree is charged at most $(3k-1)/2$ unit of credit. If an unsaturated feasible tree T is a single nodes, then T is charged at most one unit of credit. Let T rooted at v be a k-leaf unsaturated feasible charged tree with k_1 and k_2 leaves in the left and right subtrees respectively. The left child of v is not charged. If we move the credit, which is charged to v, to the left child of v, then the left subtree of T is a k_1-leaf feasible charged trees, and the right subtree of T is a k_2-leaf unsaturated feasible charged trees. By the induction hypothesis, the left and right subtrees are charged at most $3k_1/2$ and $(3k_2 - 1)/2$ unit of credit respectively. Therefore, T is charged at most $(3k-1)/2$ unit of credit.

A k-leaf feasible charged tree is charged at most $3k/2$ unit of credit. A single node feasible charged tree is charged at most one unit of credit. Let T rooted at v be a k-leaf feasible charged tree with k_1 and k_2 leaves in the left and right subtrees respectively. Consider the case that the right child of v is at most singly charged. If we move the credit, which is charged to v, to the left child of v, then the left and right subtree of T are k_1-leaf and k_2-leaf feasible charged trees respectively. By the induction hypothesis, the left and right subtrees are charged at most $3k_1/2$ and $3k_2/2$ unit of credit respectively. Therefore, T is charged at most $3k/2$ unit of credit.

Consider the case that the right child of v is doubly charged. If we move the credit, which is charged to v, to the left child of v, and if we deduce one unit of credit from the right child of v, then both left and right subtrees of T are unsaturated charged trees. By the induction hypothesis, T is at most charged $3k/2$ unit of credit.

Because each k-leaf subtree rooted at a maximal *I-set* node is charged at most $3k/2$ unit of credit, there are at most $3|U|/2$ crossing edges visited. $\square$

The CNNFA for regular expression $(a_1^* b_1^*) \cdots (a_s^* b_s^*)$ is shown in Fig. 4.7. If V is $\{a_1, b_1, \cdots, a_s, b_s\}$, then $U = \delta(V, \Sigma)$ consists of $2s$ states, and in computing $\delta(V, \Sigma)$ there are exactly $3s - 1$ crossing edges visited.

Corollary 4.12 For any regular expression R with s alphabet symbol occurrences, $|cnnfa\delta_R| \leq (3s + 1)/2$.

Proof: If V is the set of all CNNFA states, then all the crossing edges are visited in computing $\delta(V, \Sigma)$. All the trees in the *I-forest* are feasible charged trees except the one rooted at *I-set* node v_R denoting I_R. The

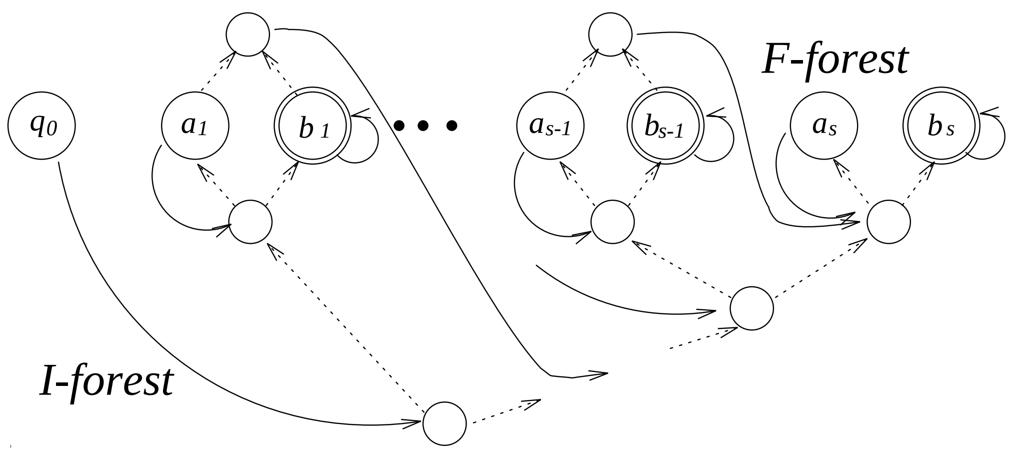


Figure 4.7: The CNNFA for $(a_1^* b_1^*) \cdots (a_s^* b_s^*)$.

reason that the tree rooted at v_R is not a feasible charged tree is that v_R could be doubly charged. *I-set* node v_R is doubly charged only if R is star-top (one is due to star edge $[F_R, I_R]$, and the other is due to $[q_0, I_R]$). However, if R is star-top, and if we deduce one unit of credit charged at v_R, then, by Lemma 4.10, the charged tree rooted at v_R is an unsaturated charged tree; otherwise, v_R is singly charged and the subtree rooted at v_R is a feasible charged tree. Hence by an argument similar to Theorem 4.11, there are at most $(3s + 1)/2$ crossing edges in $cnnfa_R$. $\square$

The CNNFA for $(a_1^* b_1^*) \cdots (a_s^* b_s^*)$ has $2s$ alphabet symbol occurrences, and it has exactly $3s$ crossing edges (see Fig. 4.7). Our crossing edge complexity presented in Corollary 4.12 is not optimal, but it is, however, very tight.

4.2 CNNFA state complexity

In this section we present a bound regarding the number of CNNFA states. Every CNNFA state must be a forest node which is touched by crossing edges. The number of CNNFA states is no greater than the number of the forest nodes. For a regular expression R with s alphabet symbol occurrences, there are at most $3s$ forest nodes (including the start state q_0). However, not all the forest nodes are touched by crossing edges. Moreover, forests shrink if subexpressions of R do not accept the empty string. The CNNFA for $a_1 \cdots a_s$ has exactly $s+1$ forest nodes. We shall show in the next theorem that excluding the start state, there are at most $(5s-2)/2$ forest nodes touched by crossing edges. For convenience, we call those forest nodes touched by crossing edges *state nodes*.

Theorem 4.13 For any regular expression R with s alphabet symbol occurrences, except the start state q_0, there are at most $(5s-2)/2$ state nodes in CNNFA $cnnfa_R$.

Proof: We shall prove our theorem by building forests of CNNFA $cnnfa_R$ first, adding edges in relaxed CNNFA $cnnfa'_R$ to forests gradually, performing necessary promotion transformation to convert $cnnfa'_R$ into $cnnfa_R$, and showing that the number of state nodes (excluding the start state q_0) is bounded by $(5s-2)/2$.

Step 1: adding product edges: Because there are at most $s-1$ product edges, there are at most $2s-2$ state nodes.

Step 2: adding left star edges: Consider each left start edge $[F_{J^*}, I_{J^*}]$ in $cnnfa'\delta_R$. If $J^*|K$ is a subexpression of R, then $[F_{J^*}, I_{J^*}] \in cnnfa\delta_R$; and we regard $[F_{J^*}, I_{J^*}]$ as if it were a "product" edge with respect J^* and K. If J^*K is a subexpression of R, we then pack $[F_{J^*}, I_{J^*}]$ and $[F_{J^*}, I_K]$ into $[F_{J^*}, I_{J^*K}]$ (see Fig 4.2), and regard promoted edge $[F_{J^*}, I_{J^*K}]$ as if it were a "product" edge with respect J^* and K. After we add all the left star edges, there are at most $s-1$ "product" edges. Therefore, there are at most $2s-2$ state nodes.

Step 3: adding right star edges: Consider each right star edge $[F_{K^*}, I_{K^*}]$ in $cnnfa'\delta_R$. If $J|K^*$ is a subexpression of R, and if J is not star-top, then we treat right star edge $[F_{K^*}, I_{K^*}] \in cnnfa\delta_R$ as if it were a "product" edge with respect to J and K^*. If JK^* is a subexpression of R, and if J is not star-top, then we can pack $[F_{K^*}, I_{K^*}]$ and $[F_J, I_{K^*}]$ into $[F_{JK^*}, I_{K^*}]$ (see Fig 4.3), and treat promoted edge $[F_{JK^*}, I_{JK^*}]$ as if it were a "product" edge with respect J and K^*. Now there are still at most $s-1$ "product" edges. Therefore, there are at most $2s-2$ state nodes.

The case that $J^*|K^*$, J^*K^* or $R = K^*$ is a subexpression is more complicated. The right star edge $[F_{K^*}, I_{K^*}]$ is in $cnnfa\delta_R$, and it seems that adding this right star edge to forests increases the number of state nodes by 2. But if regular expression K^* contains only one alphabet symbol occurrence, then both F_{K^*} and I_{K^*} are denoted by the same forest node.

Therefore, at most one forest node is newly touched by star edge $[F_{K^*}, I_{K^*}]$. If K^* contains more than one alphabet symbol occurrence, then we shall see that adding $[F_{K^*}, I_{K^*}]$ to forests only increases the number of state nodes by one because there always exists a node which is touched by two distinct crossing edges.

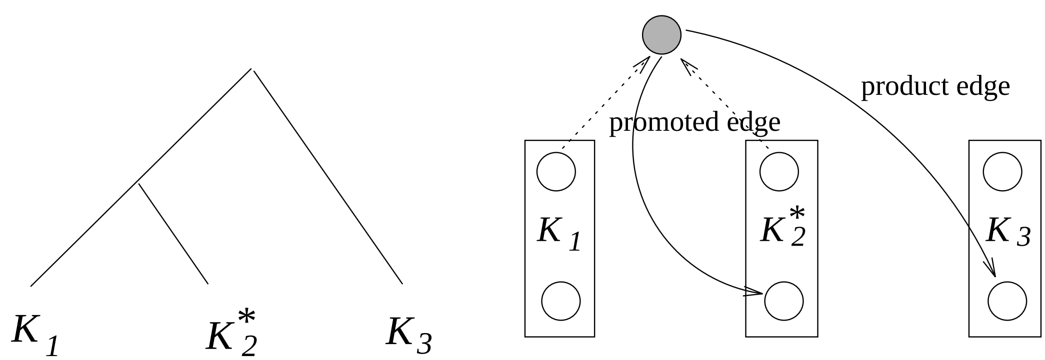
Consider the case that K^* contains more than one alphabet symbol occurrence. Then, after eliminating stars at the top of K^*, and without loss of generality after renaming, we assume that it is K. Regular expression K is either K_1K_2 or $K_1|K_2$, for some regular expressions K_1 and K_2. If $null_K = \{\lambda\}$, then $F_K = F_{K_1} \cup F_{K_2}$ and $I_K = I_{K_1} \cup I_{K_2}$. By disjointness of *cnnfa*δ, there is no "product" edge in *cnnfa*δ_R with respect to K_1 and K_2. As a result, we can regard star edge $[F_{K^*}, I_{K^*}]$ as if it were a "product" edge with respect to K_1 and K_2.

If K contains more than one alphabet symbol occurrence, and if $null_K \neq \{\lambda\}$, then there must be a path from the root to a leaf in the parse tree of K such that all the subexpressions in this path do not accept the empty string. If subexpression $K_1|K_2$ is in the path, for some regular expressions K_1 and K_2, then there is no "product" edge with respect to K_1 and K_2 because neither K_1 nor K_2 can be star-top (otherwise, $null_{K_1|K_2} = \{\lambda\}$). Therefore, we regard star edge $[F_{K^*}, I_{K^*}]$ as if it were a "product" edge with respect to K_1 and K_2.

Consider the case that each subexpression in the path discussed is either of the form K_1K_2 or an alphabet symbol occurrence. There is at least one *F-set* or *I-set* node denoting the *F-set* or *I-set* of a subexpression in this path touched by two crossing edges. Adding right star edge $[F_{K^*}, I_{K^*}]$ to forests increases the number of state nodes by one. Before we continue our argument, we need to develop the following technical lemma.

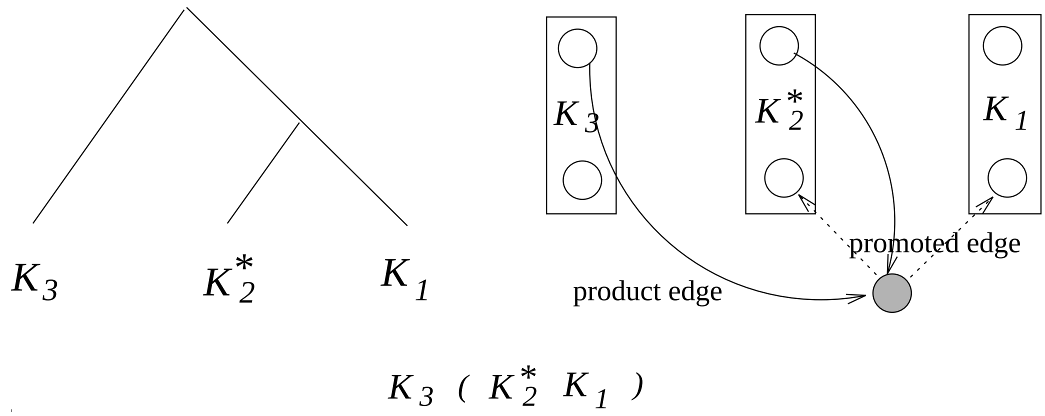
Lemma 4.14 If there is a subexpression in the path discussed of the form $(K_1K_2^*)K_3$ or $K_3(K_2^*K_1)$, and if K_3 is not star-top, then there is a forest node touched by two crossing edges.

Proof: Note that subexpression K_1 cannot be star-top since $null_K$ is empty. Form the figure below, the node denoting $F_{K_1K_2^*}$ is touched by two crossing edges.



$(\ K_1\ \ K_2^*\ )\ K_3$

The *I-set* node denoting $I_{K_2^*K_1}$ is also touched by two crossing edges.



□

Consider a parse tree of K shown by Fig. 4.8. The discussed path goes through the left child of K first. Since $[F_{K^*}, I_{K^*}] \in cnnfa\delta_R$, both nodes denoting F_K and I_K are touched crossing edges. If the right subexpression of K $K_{1,1}$ is star-top, and if K is $J_{1,1}K_{1,1}$, for some regular expression $J_{1,1}$, then promoted edge $[F_K, I_{K_{1,1}}]$ is also in $cnnfa\delta_R$. Therefore, *F-set* node denoting F_K is touched by two crossing edge. If $K_{1,1}$ is not star-top, and if $K_{1,i}$ is star-top, for some i, $1 < i \leq n_1$, then there exists some j, $1 < j < n_1$, such that K_{j-1} is not star-top, but K_j is. By Lemma 4.14, an *F-set* node is touched by two crossing edges. If all $K_{1,i}$'s are not star-top, $1 \leq i \leq n_1$, then the *F-set* node denoting F_{K_2} has an outgoing edge pointing to the *I-set* node denoting $I_{K_{1,n_1}}$. Since $J_{1,1}, \cdots, J_{1,n_1-1}$ and K_2 do not accept the empty string, $I_K = I_{K_2}$. Hence, the right star edge $[F_{K^*}, I_{K^*}]$ touches the node denoting I_{K_2}. If K_2 is an alphabet symbol, then the *F-set* node denoting F_{K_2} and the *I-set* node denoting I_{K_2} are identical, and it is touched by two

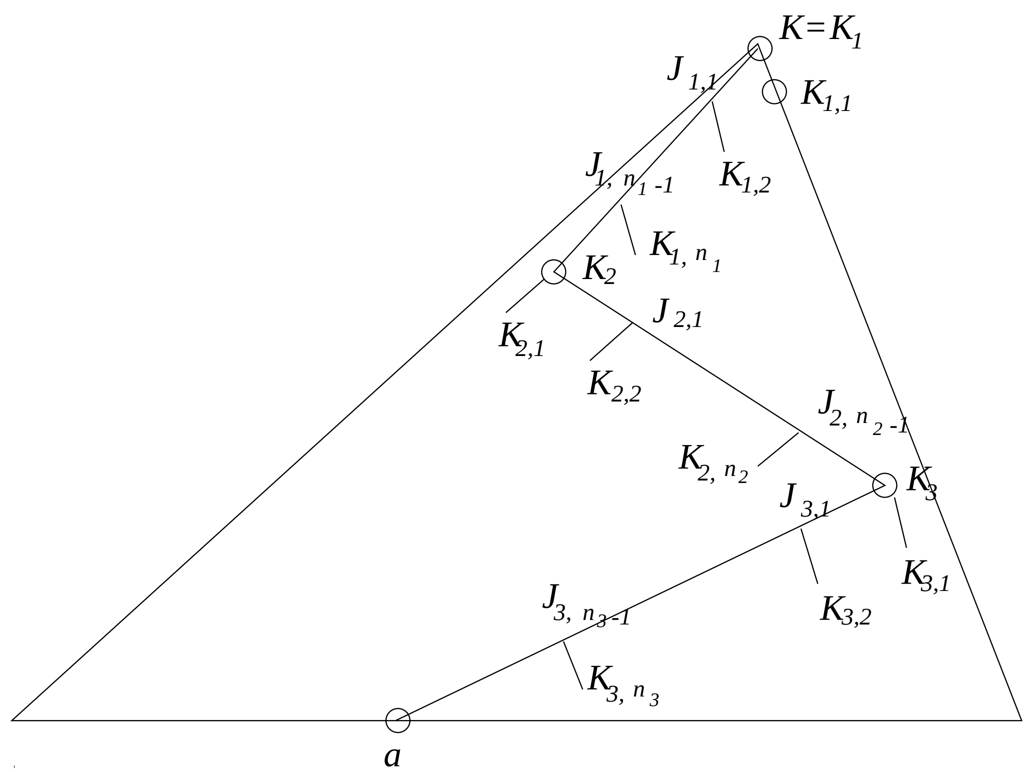


Figure 4.8: Non-null Path

distinct crossing edges. If K_2 is not an alphabet symbol, then we repeat the previous argument from K_2 alone the path down to a leaf, and we can eventually find a forest node touched by two crossing edges. The argument for the other case which the discussed path goes through the right child of K is similar.

From the discussion above, CNNFA $cnnfa_R$ has $s-1$ "product" edges, and these edges touch at most $2s-2$ forest nodes. Though each right star edge which cannot be regarded as a "product" edge can touch two new nodes, but adding each of such an edge to forests increases the number of state nodes only by one. Note that we find a node, which is touched by two crossing edges, denoting an *I-set* or *F-set* of a subexpression in the path, and this path does not go through a star-top subexpression. Every such node can never be found twice.

Similar to Theorem 4.11, there are at most $(s+1)/2$ right star edges which can not be regarded as "product" edges. Therefore, excluding the start state q_0 but counting the *I-set* node denoting I_R, there are at most $(5s-2)/2$ state nodes. □

Corollary 4.15 For any regular expression R with s alphabet symbol occurrences, there are at most $5s/2$ CNNFA states and $(10s-5)/2$ edges.

Proof: By previous Theorem, including the starting state, there are $5s/2$ CNNFA states in CNNFA $cnnfa_R$. By Theorem 4.12 and Theorem 3.4,

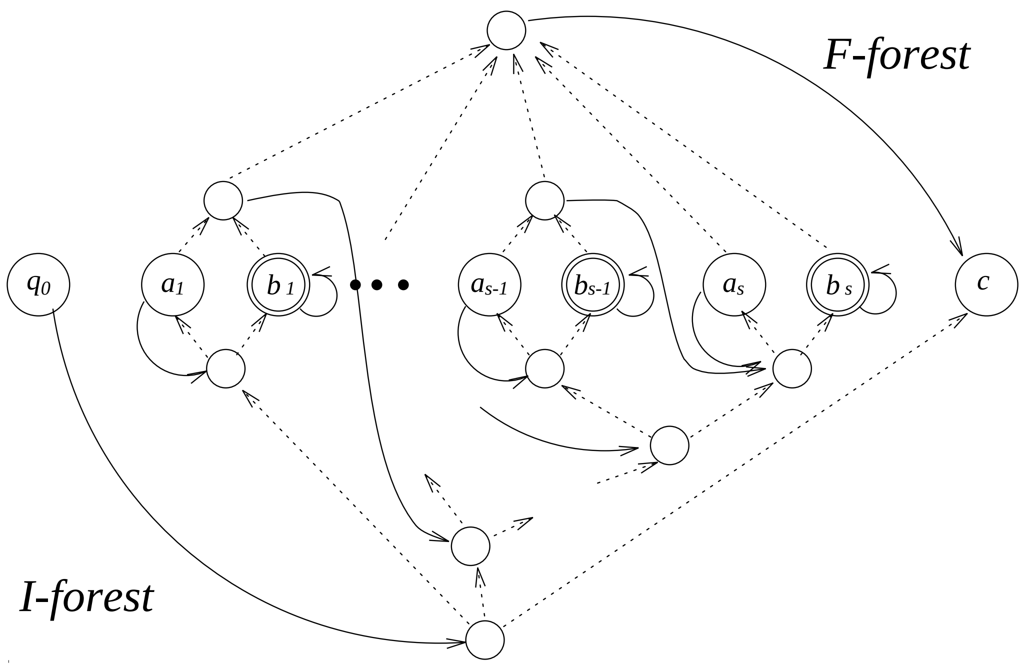


Figure 4.9: The CNNFA for $((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$.

there are at most $(3s + 1)/2 - 1 + (5s - 2)/2 + s - 1 = (10s - 5)/2$ edges. □

The CNNFA for $((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$ is shown in Fig. 4.9. Regular expression $((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$ contains $2s + 1$ occurrences of alphabet symbol, and the CNNFA for $((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$ is consisted of $5s + 1$ states and $10s - 1$ edges. This example shows that our theorem gives a very sharp upper bound for the size of the CNNFA.

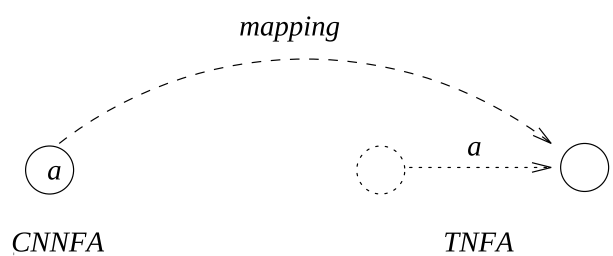
4.3 CNNFA vs. Thompson's NFA

In this section we compare the CNNFA with Thompson's NFA. Let R be a regular expression of length r with s alphabet symbol occurrences. CNNFA $cnnfa_R$ has at most $5s/2$ states and $(10s - 5)/2$ edges. Thompson's NFA $tnfa_R$ has between $r - s_{()} + 1$ and $2r$ states and between $r - s_{()}$ and $4r - 3$ edges, where $s_{()}$ is the number of occurrences of parentheses in R. Though s can be arbitrary smaller than r, s is equal to r in the best case. Nevertheless, we shall show that $cnnfa_R$ is has no more states and edges than $tnfa_R$. Given any subset V of NFA states, the $\delta(V, \Sigma)$ computation (similar to computing the λ-closure of V in Thompson's NFA) is a fundamental operation in both acceptance testing and subset construction. The $\delta(V, \Sigma)$ computation takes $O(|V| + |\delta(V, \Sigma)|)$ time in $cnnfa_R$, and $O(r)$ time in $tnfa_R$. We shall show that to compute $\delta(V, \Sigma)$ in $cnnfa_R$ fewer nodes and edges are visited. In another words, the CNNFA is smaller and faster than Thompson's NFA.

The comparison is done by constructing a one-to-many map that maps states and edges in $cnnfa_R$ to states and edges in $tnfa_R$. The CNNFA is thus smaller than Thompson's NFA. Let V_V and E_V be the set of states and edges visited during the $\delta(V, \Sigma)$ computation in $cnnfa_R$. We show that the images of V_V and E_V in $tnfa_R$ are also visited during the $\delta(V, \Sigma)$ computation. We consider a larger and slower version of the CNNFA in this section: the path compression transformation does not eliminate forest leaves. We shall briefly discuss our mapping in the rest of this section.

A state in Thompson's NFA is an *important* state if it has an outgoing edge labeled by an alphabet symbol. The tail of an edge labeled by an alphabet symbol is a *transition* state. State q' is λ-*reachable* from state q if there is a λ-path from q to q', and edge $[q', q'']$ is λ-*reachable* from state q if q' is λ-reachable from q.

There is a one-to-one correspondence between leaves in $cnnfa_R$ and transition states in $tnfa_R$. For each alphabet symbol occurrence a in R, the forest node denoting $\{a\}$ is mapped to the final state in $tnfa_a$. For convenience, for each alphabet symbol occurrence a, we use v_a to denote the forest node denoting $\{a\}$, and use $mapping(v_a)$ to denote the final state of $tnfa_a$.



Set $\{mapping(v_a) : a \in I_R\}$ is the set of transition states which are reachable from the start state by paths spelled alphabet symbols (strings of length 1) in $tnfa_R$, and set $\{mapping(v_a) : a \in F_R\}$ is the set V of transition states such that the final state of $tnfa_R$ is λ-reachable from every state in V. For simplicity, we also use F_R and I_R to denote sets of transition states $\{mapping(v_a) : a \in F_R\}$ and $\{mapping(v_a) : a \in I_R\}$ respectively.

The central idea of our mapping is as follows. If an *F-set* (respectively *I-set*) node denoting F_J (respectively I_J) is visited during the $\delta(V, \Sigma)$ computation in CNNFA $cnnfa_R$, then the final state (respectively start state) of $tnfa_J$ is also visited in $tnfa_R$. Therefore, for each internal *F-set* node v denoting F_J in $cnnfa_R$, we choose a state v' in $tnfa_R$ (perhaps outside of $tnfa_J$) as the image of v such that v' is λ-reachable from every states in F_J (v' is not necessary to be λ-reachable from the final state of $tnfa_J$). As to internal *I-set* node v denoting I_J in $cnnfa_R$, we choose a state v' in $tnfa_J$ as the image of v, where v' is reachable from the start state of $tnfa_J$ by a path spelling a string of length one.

For each tree or crossing edge e originating from *F-set* node v denoting F_J, we choose a set E of edges in $tnfa_R$ (perhaps outside of $tnfa_J$) such that for each state v' in F_J there is at least one edge e' in E which is λ-reachable from v'. Consider a tree edge $e = [v, v']$ in the *I-forest*, where *I-set* node v denotes I_J. We choose an edge e' in $tnfa_J$ as the image of e such that e' is λ-reachable from the start state of $tnfa_J$. In the way that we construct a mapping, if our mapping is one-to-many, then the CNNFA is faster than Thompson's NFA.

In the following, we present an inductive algorithm to map a tail machine $cnnfa_R^T$ to a Thompson's NFA $tnfa_R$. To extend the domain of our mapping to $cnnfa_R$ is straightforward. We assume that regular expressions are not λ-expressions. We shall use the word "unassigned" to describe states and edges in Thompson's NFA which are not in the range of our mapping. We use figures to describe our mapping with a convention that dotted circles and lines are unassigned states and edges respectively. In each inductive step of our algorithm, the following invariants hold.

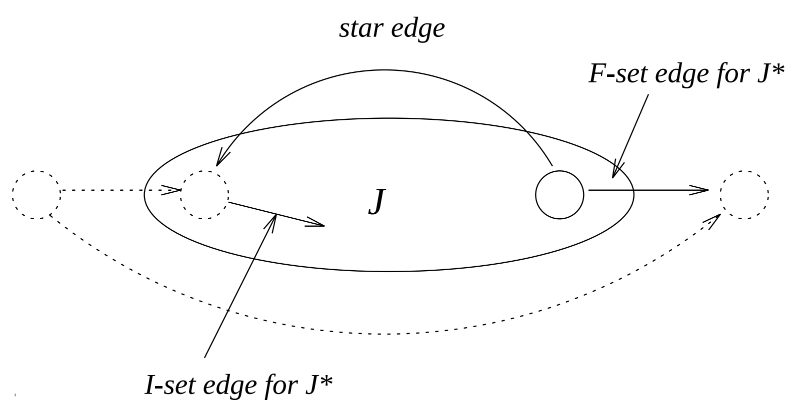
Invariant 1: If $null_R$ is empty, then all the states and edges in $cnnfa_R^T$ are mapped. The start state q_0 of $tnfa_R$ and one edge which is λ-reachable from q_0 are unassigned.

Invariant 2: If R is a star-top regular expression, then all the states and edges in $cnnfa_R^T$ are mapped. In addition, tree edges connecting nodes denoting F_R and I_R are pre-mapped. By tree edge e *connecting F-set* node v, we mean that v is the origin of e, and tree edge *connects I-set* node v if e reaches v. Beside of the start state and the final state, one state and two edges both of which are λ-reachable from the start state in $tnfa_R$ are unassigned.

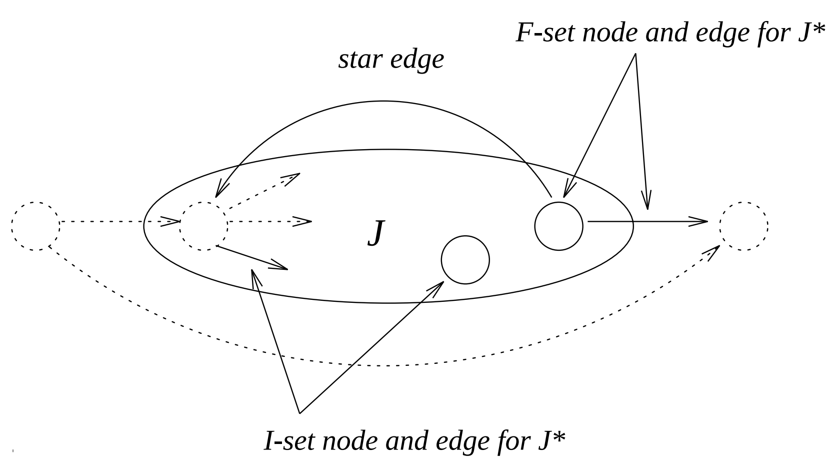
Invariant 3: If R is not star-top, but if $null_R = \{\lambda\}$, then all the states and edges in $cnnfa_R^T$ except nodes v_1 and v_2 denoting F_R and I_R respectively, are mapped. Except the start state and the final state, one state and three edges both of which are λ-reachable from the start state are unassigned.

The $R = J^*$ Case

Consider the case where $null_J$ is empty. By Invariant 1, we need to map the star edge with respect to J^* and tree edges connecting nodes denoting F_J and I_J to some proper unassigned edges in $tnfa_{J*}$ only. The following figure shows how the mapping is arranged.

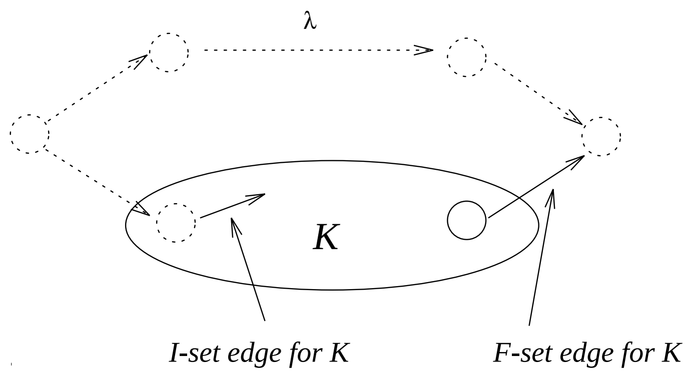


Consider the case $null_J = \{\lambda\}$. Without loss of generality, we assume that J is not star-top. In addition to the assignment for the star edge, we need to map forest nodes denoting F_J and I_J, and tree edges connecting them. The assignment figure below shows we have at least three states and four edges unassigned.

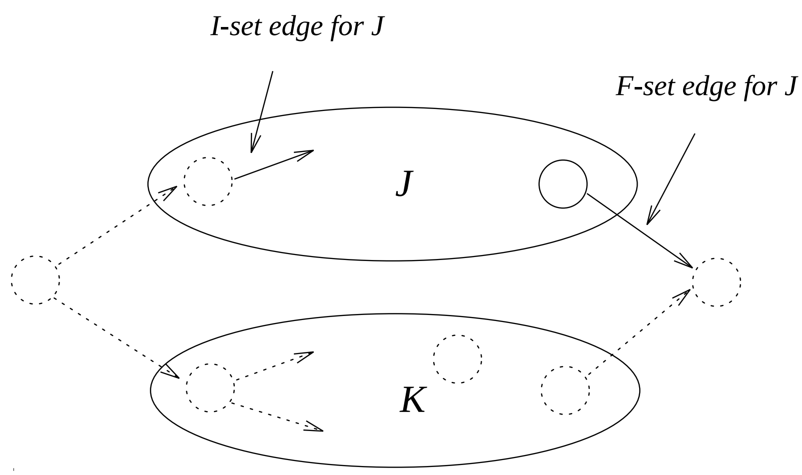


The $R = J|K$ Case

We discuss two cases only. If $J \equiv \lambda$, and if $K \not\equiv \lambda$, then from the figure below, five states and four edges are unassigned.

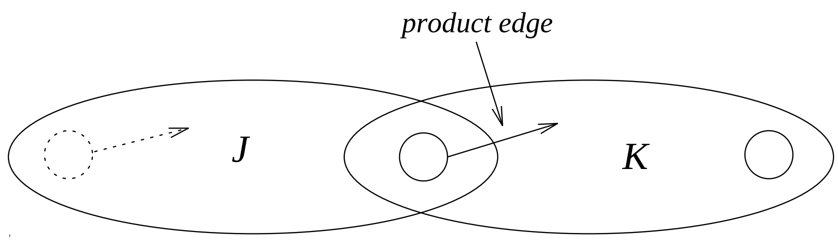


Consider the case in which both J and K are not equivalent to the empty string λ, $null_J$ is empty, and $null_K = \{\lambda\}$. We need to map the tree edges connecting to nodes denoting F_J and I_J only. By Invariant 2 and 3, $tnfa_K$ has at least two states and three edges unassigned when we map $cnnfa_K^T$ to $tnfa_K$. The assignment shown below indicates that we have six states and five edges unassigned.

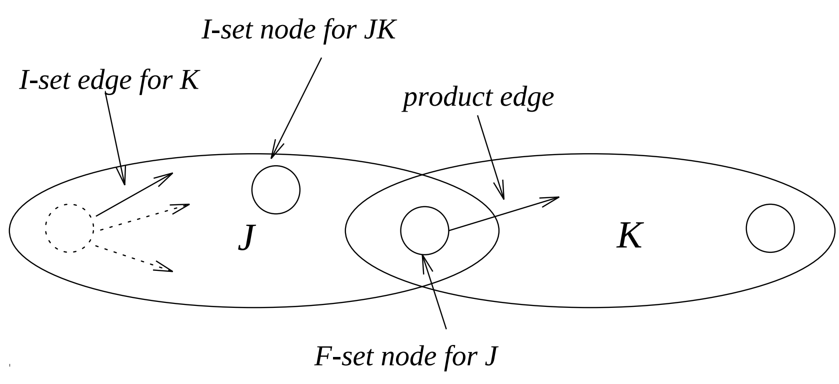


The $R = JK$ case

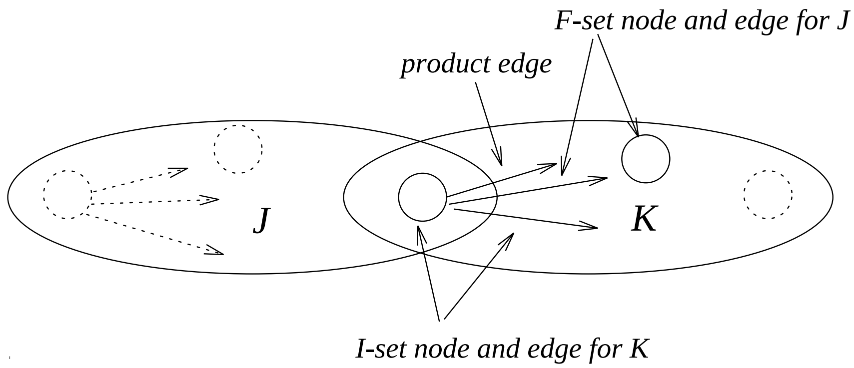
Since $\lambda R \equiv R\lambda \equiv R$, for any regular expression R, we assume neither J nor K is a λ-expression. The mapping for the case in which both $null_J$ and $null_K$ are empty is simple.



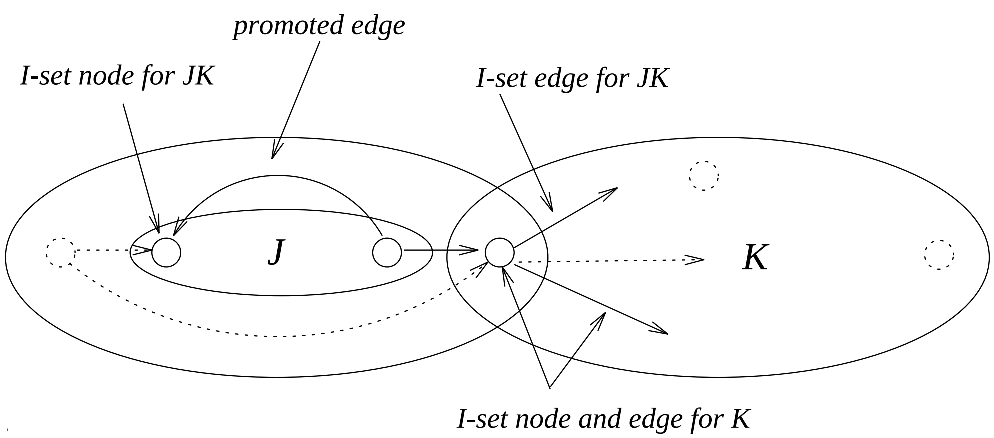
The following figure shows how the mapping is assigned for the case in which $null_K$ is empty, and J is not star-top, but $null_J = \{\lambda\}$. We have $F_R = F_K$, and $I_R = I_J \cup I_K$. We need to map the product edge $[F_J, I_K]$, the *F-set* node denoting F_J, *I-set* node v denoting I_R, and the tree edge originating from v to the *I-set* node denoting I_K only. The mapping for the case where $null_J$ is empty, and K is not star-top, but $null_K = \{\lambda\}$ is similar.



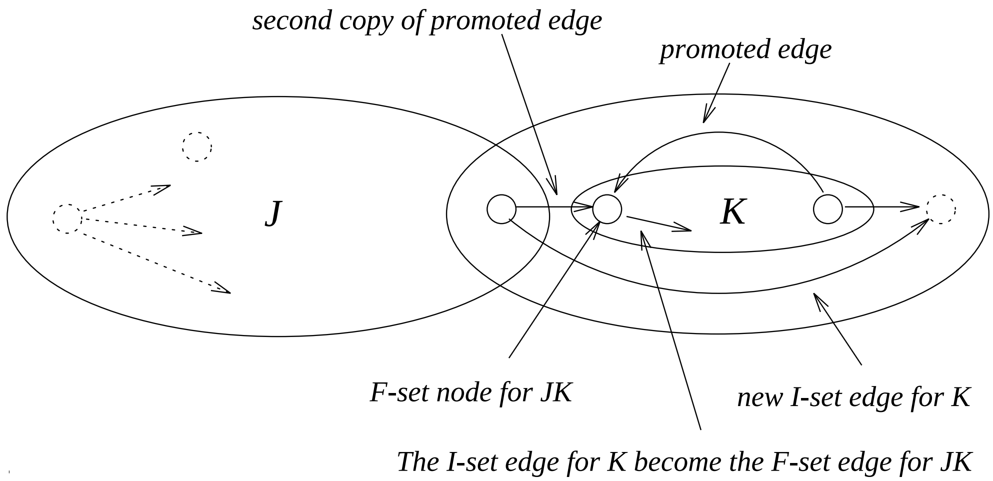
The following figure shows the mapping for the case where both J and K are not star-top, but they accept the empty string. We have three unassigned states and three unassigned edges.



The mapping for the case in which J is star-top, and K is not star-top, but $null_K = \{\lambda\}$ is shown below. The edge that is previously assigned as the image of star edge $[F_J, I_J]$ becomes the image of promoted edge $[F_J, I_{JK}]$. We have three states and three edges unassigned.



Consider the case such that K is star-top, and J is not star-top, but $null_J = \{\lambda\}$. We have two edges in $tnfa_R$ serving as images of promoted edge $[F_{JK}, I_K]$. We assign the mapping for *F-set* node denoting F_{JK} and the tree edge originating from it. The edge previously assigned as the image of tree edge of the node denoting I_K becomes the image of the tree edge of the *F-set* node denoting F_{JK}. From the figure below, we have three unassigned states and three unassigned edges.



For those cases not being considered, their mapping are similar to mappings that we have discussed. We leave them to readers as simple exercises.

Since our map is one-to-many, the CNNFA has fewer states and edges than Thompson's NFA. For the same reason, computing $\delta(V, \Sigma)$ for any subset V of NFA states in the CNNFA visits fewer states and edges than the equivalent computation in Thompson's NFA.

Chapter 5

Performance Benchmark

5.1 CNNFA Benchmark

Experiments[1] to benchmark the performance of the CNNFA have been carried out for a range of regular expression patterns against a number of machines including Thompson's NFA, an optimized form of Thompson's NFA, and McNaughton and Yamada's NFA[18]. We build Thompson's NFA according to the construction rules described in Chapter 1. Thompson's NFA usually contains redundant states and λ-edges. However, to our knowledge there is no obvious/efficient algorithm to optimize Thompson's NFA without blowing up the linear space constraint. We therefore devise

[1]The programming language test pattern used in this thesis is ((a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z) (a | b | c | d e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z)* | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0) (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0)* | [|] | (|) | while | for | struct | if | do)

pattern	TNFA	TNFA opt.	MYNFA
$(abc\cdots)$	75% slower	55% slower	75% slower
$(a\|b\|\cdots)^*$	12 times faster	2 times faster	50% faster
$((a\|\lambda)(b\|\lambda)\cdots -)^*$	2 times faster	25% faster	80% slower
$((a\|\lambda)(b\|\lambda)\cdots)^*$	16 times faster	8 times faster	50% faster
$((a\|\lambda)^n -)^*$	comparable	50% slower	linearly faster
programming language	7 times faster	50% faster	80% faster

Figure 5.1: The CNNFA acceptance testing speedup ratio

some simple but effective transformations that eliminate redundant states and edges in most of the test cases.

Our acceptance testing experiments show that the CNNFA outperforms Thompson's NFA, Thompson's NFA optimized, and McNaughton and Yamada's NFA. See Fig. 5.1 for an acceptance testing benchmark summary. The benchmark summary indicates that the CNNFA is slower than all other machines for $(abc\cdots)$ and $(abc\cdots)^*$ patterns. According the comparison between the CNNFA and Thompson's NFA discussed in the last Chapter, this is an anomalous shortcoming of our current implementation, which will be eliminated in the next version.

The benchmark for subset construction is more favorable. The CNNFA outperforms the other machines not only in DFA construction time but also in constructed machine size. Subset construction is compared among the following six starting machines: the CNNFA, Thompson's NFA, Thompson's NFA optimized, Thompson's NFA using the important-state heuristic[2], Thompson's NFA using the kernel items heuristic[2], and Mc-

Naughton and Yamada's NFA. See Fig. 1.3 for a high level specification of the classical Rabin and Scott subset construction for producing a DFA σ from an NFA δ.

We implemented the subset construction specification tailored to the CNNFA and other machines. The only differences in these implementations is in the calculation of $\delta(V, \Sigma)$, where we use the efficient procedure described by Theorem 3.3, and in the λ-closure step, which is performed only by Thompson's NFA, Thompson's NFA with important-state heuristic, and Thompson's NFA optimized. The CNNFA achieves linear speedup and constructs a linearly smaller DFA in many of the test cases. See Fig. 5.2 and 5.3 for a benchmark summary.

The raw timing data is given in the Appendix A. All the tests described in this thesis are performed on a lightly loaded SUN 3/50 or SUN3/250 server. We used `getitimer()` and `setitimer()` primitives [29] to measure program execution time. It is interesting to note that the CNNFA has a better speedup ratio on SUN Sparc based computers.

5.2 Cgrep Benchmark

Recently at Columbia University's Theory Day, Aho reported a highly efficient heuristic for deciding whether a given string belongs to the language denoted by a regular expression, i.e. both string and regular expression are dynamic(cf. page 128 of [2]). This problem is needed for UNIX tools such

pattern	TNFA	TNFA ker.	TNFA imp.
$(abc\cdots)^*$	2 times faster	comparable	comparable
$(a\|b\|\cdots)^*$	quadratic speedup	linear speedup	linear speedup
$(0\|1\cdots\|9)^n$	20 times faster	9 times faster	2 times faster
$((a\|\lambda)(b\|\lambda)\cdots-)^*$	linear speedup	20% faster	linear speedup
$((a\|\lambda)(b\|\lambda)\cdots)^*$	quadratic speedup	linear speedup	linear speedup
$(a\|b)^*a(a\|b)^n$	30 % faster	comparable	20 % faster
prog. lang.	19 times faster	3 times faster	3 times faster

pattern	TNFA opt.	MYNFA	
$(abc\cdots)^*$	comparable	comparable	
$(a\|b\|\cdots)^*$	linear speedup	linear speedup	
$(0\|1\cdots\|9)^n$	20% faster	8 times faster	
$((a\|\lambda)(b\|\lambda)\cdots-)^*$	linear speedup	5% slower	
$((a\|\lambda)(b\|\lambda)\cdots)^*$	linear speedup	linear speedup	
$(a\|b)^*a(a\|b)^n$	comparable	comparable	
prog. lang.	20% faster	3 times faster	

Figure 5.2: The CNNFA subset construction speedup ratio

pattern	TNFA	TNFA ker.	TNFA imp.
$(abc\cdots)^*$	comparable	comparable	comparable
$(a\|b\|c\cdots)^*$	linearly smaller	linearly smaller	comparable
$(0\|1\|\cdots 9)^n$	200 times smaller	10 times smaller	comparable
$((a\|\lambda)(b\|\lambda)\cdots -)^*$	3 times smaller	comparable	comparable
$((a\|\lambda)(b\|\lambda)\cdots)^*$	linearly smaller	linearly smaller	comparable
$(a\|b)^*a(a\|b)^n$	4 times smaller	comparable	comparable
prog. lang.	4 times smaller	4 times smaller	comparable

pattern	TNFA opt.	MYNFA	
$(abc\cdots)^*$	comparable	comparable	
$(a\|b\|c\cdots)^*$	comparable	linearly smaller	
$(0\|1\|\cdots 9)^n$	comparable	10 times smaller	
$((a\|\lambda)(b\|\lambda)\cdots -)^*$	comparable	comparable	
$((a\|\lambda)(b\|\lambda)\cdots)^*$	comparable	linearly smaller	
$(a\|b)^*a(a\|b)^n$	comparable	comparable	
prog. lang.	comparable	4 times smaller	

Figure 5.3: DFA size improvement ratio starting from the CNNFA

as egrep. Aho's heuristic constructs an NFA first, and subsequently builds a DFA piecemeal as the input string is scanned from left to right. There are two popular egreps currently in use. One is UNIX egrep, and the other is GNU e?grep[12] from the Free Software foundation. Both UNIX egrep and GNU e?grep are based on McNaughton and Yamada's NFA, and use Aho's heuristic. Using Aho's heuristic, we implement a UNIX egrep compatible software based on the CNNFA called cgrep. Benchmarks show substantial computational improvement of cgrep against competing softwares – the UNIX egrep and the GNU e?grep.

In contrast to the current version, an old version of UNIX egrep constructs full DFA's from McNaughton and Yamada's NFA's first, and performs acceptance testing in DFA's. For clarity, we call it egrep2. To demonstrate the performance of the CNNFA, we also build a corresponding version of cgrep cgrep2, which also builds DFA's and performs acceptance testing in DFA's. Experiments have been carried out to compare the performance of cgrep, cgrep2, egrep, egrep2 and e?grep. We measured the NFA construction, DFA construction time (cgrep2 and egrep2 only), and on-line simulation time (Aho's heuristic) to benchmark this family of egrep. Not surprisingly, in NFA and DFA construction, cgrep is at least one order of magnitude faster than the other egreps. For the programming language test pattern, cgrep is 5.2 times faster than egrep, and 9 times faster than e?grep in NFA construction. Cgrep2 is 30 times faster than egrep2 in DFA construction. Cgrep is 50% faster than egrep, and 3.14 times than e?grep in on-line simulation. See Fig. 5.4 for a benchmark summary. The benchmark was performed on a SUN 3/50. The cgrep source code is listed in Appendix B. The benchmark raw timing data are found in Appendix C.

	$a_1 \cdots a_n$				$-(a_1 \vert \cdots \vert a_n)^*-$			
length	50	100	150	200	50	100	150	200
NFA egrep/cgrep	1	3	3.5	4	2.3	5.5		
NFA e?grep/cgrep	7	17	14.5	14.3	4.7	8		
DFA egrep2/cgrep2	n/a	15	15	26.5	n/a	21		
simu egrep/cgrep	n/a	10	21	12.3	n/a	4		
simu e?grep/cgrep	n/a	28	42	19.7	n/a	247		
	$-((a_1b_1)? \cdots (a_nb_n)?)-$				$-((a_1b_1)? \cdots (a_nb_n)?)^*-$			
length	50	100	150	200	50	100	150	200
NFA egrep/cgrep	3	4.6	7.6	8.8	4.8	8.8	17.4	21
NFA e?grep/cgrep	9	15.4	31.2	43.3	10.8	20.1	45.3	65.2
DFA egrep2/cgrep2	8.3	12.3	13	16.2	8	21	40	31
simu egrep/cgrep	4.1	7.2	133	205	7	20	38	33
simu e?grep/cgrep	29.6	48.4	83.8	96.2	40	161	433	459
	$-((a_1b_1) \vert \cdots \vert (a_nb_n))^*-$				$(0 \vert \cdots \vert 9)^n(0 \vert \cdots \vert 9)^*$			
length	50	100	150	200	50	100	150	200
NFA egrep/cgrep	2.33	6.5	7.5	10.9	3	4.3	6.3	7.1
NFA e?grep/cgrep	5	9.4	8.3	11	6.3	5.8	6.9	6.6
DFA egrep2/cgrep2	n/a	21	13.3	61	n/a	31	33.5	38.7
simu egrep/cgrep	6	18	18.5	29	1	8	7	12
simu e?grep/cgrep	42	163	222	461.5	37	372	529	1155

Figure 5.4: Egreps Benchmark Summary

Chapter 6

More Optimization Techniques

In this Chapter, we present more optimization techniques for the CNNFA. The tree contraction transformation is used to construct even smaller CNNFA's. Meyer proposes an NFA/DFA hybrid machine in order that membership testing for string x against regular expression R can be done in $O(|x||R|/\log|x|)$ bit-vector operations. We show how to adopt the CNNFA to Meyer's algorithm. We also show how to construct smaller DFA's from the CNNFA by using an observation used in [2] without increasing time and space complexities.

6.1 Tree Contraction

The tree contraction transformation is like the inverse of packing. It works as follows: (1) when an internal *F-forest* node n has k_1 outgoing edges and k_2 incoming edges, and if $k_1 k_2 \leq k_1 + k_2$, then we can replace node n and

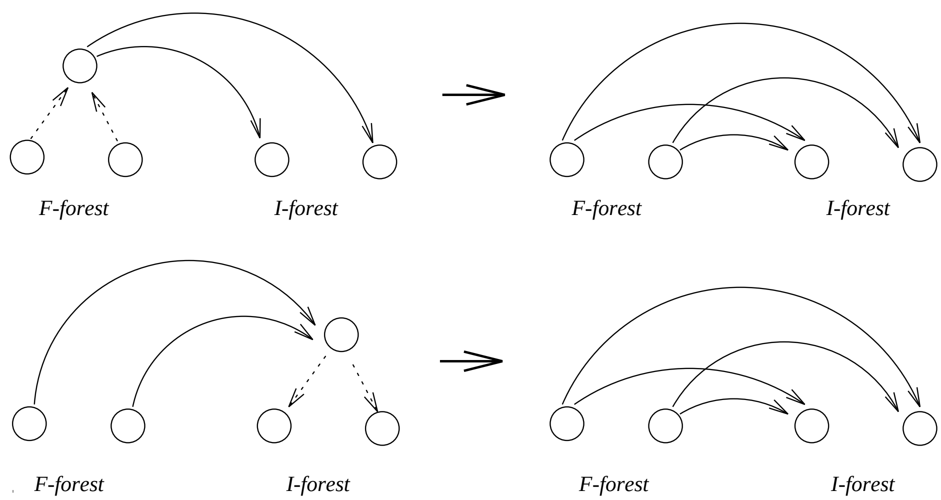


Figure 6.1: Tree contraction

the k_1+k_2 edges incident to n by k_1k_2 edges (see Fig. 6.1); and (2) when an internal *I-forest* node n has k_1 incoming edges and k_2 outgoing edges, and if $k_1k_2 \leq k_1+k_2$, then we can replace node n and the k_1+k_2 edges incident to n by k_1k_2 edges (see Fig. 6.1). After applying tree contraction to the CNNFA in Fig. 3.12, one *I-forest* node is eliminated (see Fig. 6.2). Fig. 6.2 illustrates a CNNFA improved by tree contraction for regular expression $(a|b)^*abb$. It contains 5 states and 6 edges in contrast to the 9 states and 14 edges found in the MYNNFA of Fig. 3.3.

6.2 A CNNFA/DFA Hybrid Machine

Meyer [19] shows an $O(|x||R|/\log|x|)$ space and time acceptance testing algorithm for regular expression R and string x. He uses a hybrid of NFA

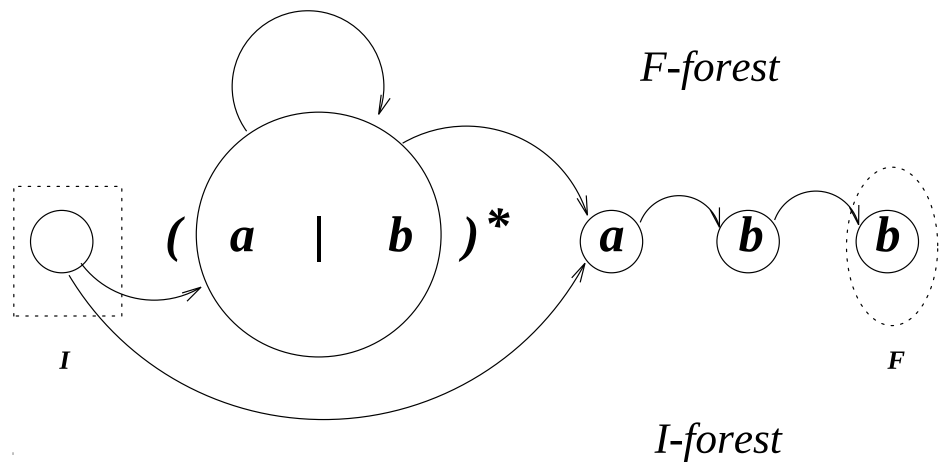


Figure 6.2: An CNNFA equivalent to regular expression $(a|b)^*abb$ improved by tree contraction

and DFA to achieve this new resource bound. More precisely, he divides a Thompson's NFA into $O(|R|/\log|x|)$ modules, and replaces each module by a DFA. In this section we show how to adopt the CNNFA to his algorithm. Starting from the CNNFA, his algorithm is simpler than starting from Thompson's NFA. Our CNNFA/DFA hybrid machine is a constant factor faster than Meyer's machine.

Given a regular expression R, Meyer first decomposes the parse tree T_R for R into modules. Similarly, we partition edges in T_R into sets, and the subgraph induced by each edge set is an $O(k)$-size subtree, for some constant k to be determined later. We call each induced subgraph a *module*. Each module is a parse tree – if a node has a child, then it has all of its children. All the modules except the one rooted at the root of T_R have between

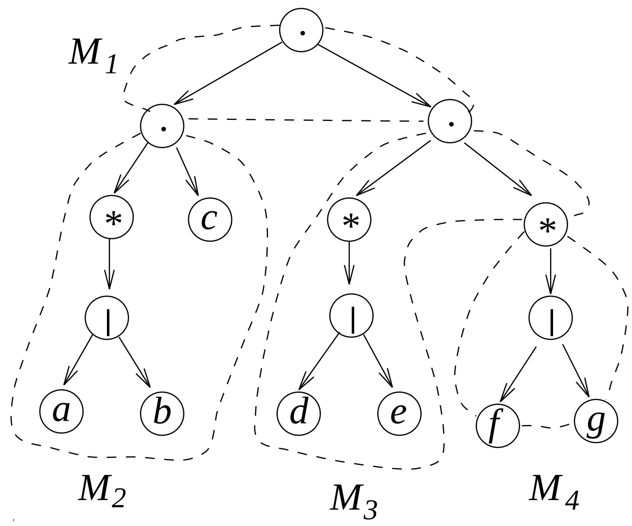


Figure 6.3: A $k = 3$ partition of T_R for $R = ((a|b)^*c)((d|e)^*(f|g)^*)$.

$\lfloor k/2 \rfloor + 1$ and k leaves. Hence, there are at most $O(|R|/k)$ modules. Each module M_i corresponds to a subexpression R_i of regular expression R. Fig 6.3 shows a partition of the parse tree for $((a|b)^*c)((de)^*(f|g)^*)$ with $k = 3$.

Let us consider a version of the relaxed CNNFA $cnnfa'_R$ which is the same as we describe in Chapter 4 except without applying path compression. According to a decomposition of T_R, we decompose $cnnfa'_R$ into modules. Each module M_i directly corresponds to a collection of *F-trees* and *I-trees*, and we call the corresponding *F-trees* and *I-trees* the *F-module* FM_i and *I-module* IM_i of M_i respectively. Fig 6.4 shows the *F-module* and *I-module* of M_3 in Fig. 6.3. We also partition crossing edges into modules.

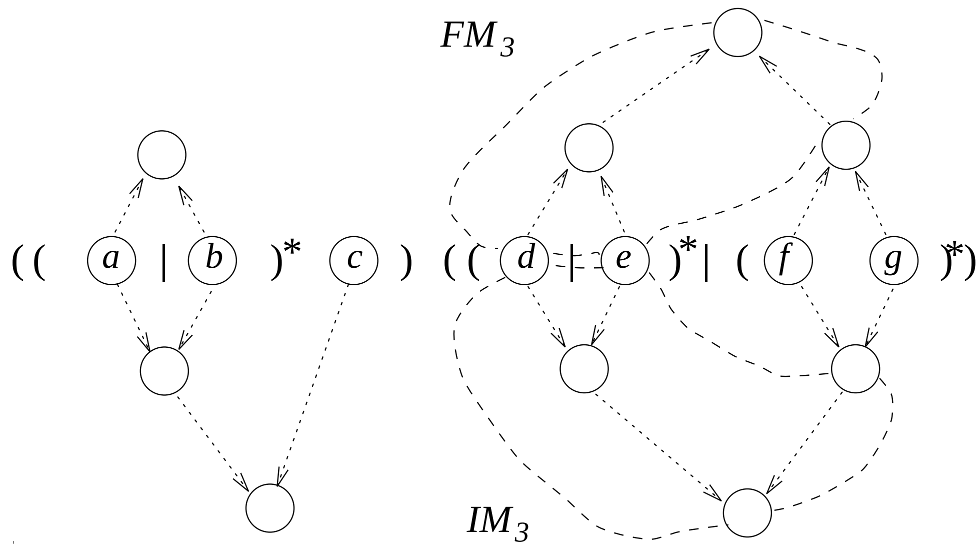


Figure 6.4: A partition of $cnnfa'\delta_R$ for $R = ((a|b)^*c)((d|e)^*(f|g)^*)$.

Each edge module EM_i contains all the edges $[F, I]$ originating from a node in FM_i to a node in IM_i.

Considering a module M_i representing subexpression R_i, let $M_{i,1}, \cdots M_{i,i_l}$ be all the modules such that their roots are leaves in M_i, and A_i denote the set of alphabet symbol occurrences in M_i. Consider each crossing edge $[F, I]$ in EM_i. Following the CNNFA construction, *I-set* I is a subset of set $A_i \cup I_{R_{i,1}} \cup \cdots I_{R_{i,i_l}}$, and moreover, if I has a non-empty intersection with $I_{i,m}$, $1 \leq m \leq i_l$, then $I_{i,m} \subseteq I$. Thereby, we can construct two functions

$$reach\_A_i : \mathcal{P}(A_i \cup \{F_{R_{i,1}} \cdots F_{R_{i,i_l}}\}) \rightarrow \mathcal{P}(A_i) \quad \text{and}$$

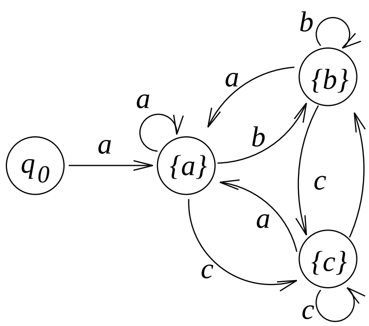
$$reach\_I_i : \mathcal{P}(A_i \cup \{F_{R_{i,1}} \cdots F_{R_{i,i_l}}\}) \rightarrow \mathcal{P}(\{I_{R_{i,1}} \cdots I_{R_{i,i_l}}\})$$

to replace edge module EM_i in an obvious way, where $\mathcal{P}(S)$ denotes the power set of set S. Using an $O(k)$-bit vector to represent each set in $\mathcal{P}(A_i \cup \{F_{R_{i,1}} \cdots F_{R_{i,i_l}}\})$, we implement each $reach\_A_i$ function as an $O(2^k|\Sigma|)$ size table of $O(k)$-bit vectors, and implement each $reach\_I_i$ function as an $O(2^k)$ size array of *I-set* node lists. It is not hard to see that we can perform the membership testing for string x against regular expression R in a CNNFA/DFA hybrid machine in $O(|x||R|/k)$ time. Choosing $k = \log|x|$, our CNNFA/DFA hybrid machine uses the same resource bounds as Meyer's but simpler and faster.

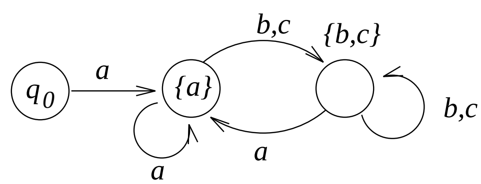
6.3 Even smaller DFA construction

The λ-closure is an important mechanism to construct a smaller DFA in Rabin and Scott's subset construction algorithm. However, the λ-closure is not applicable to both the CNNFA and McNaughton and Yamada's NFA. In lack of λ-closure, sometimes the constructed DFA is larger when starting from the CNNFA or McNaughton and Yamada's NFA. Fig. 6.5 shows DFA's which are constructed from various NFA's. DFA's constructed from both the CNNFA and McNaughton and Yamada's NFA are larger than an equivalent DFA constructed from Thompson's NFA using important state heuristic.

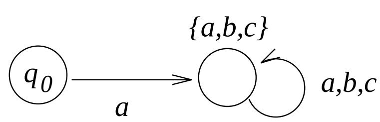
The reason that larger DFA's constructed from the CNNFA (or from



from MYNFA



from CNNFA



from TNFA using important state heuristic

Figure 6.5: DFA's equivalent to $(a(b|c)^*)^*$

McNaughton and Yamada's NFA) is that if V is the NFA state set denoted by a DFA state in a DFA from the CNNFA, then the corresponding DFA state in the DFA from Thompson's NFA using important state heuristic denotes the set $\delta(V, \Sigma)$ (by the effect of λ-closure). There are cases that DFA states d_1 and d_2 respectively denote NFA state sets V_1 and V_2 which are not equal; but if $\delta(V_1, \Sigma) = \delta(V_2, \Sigma)$, then d_1 and d_2 are equivalent DFA states.

To overcome the shortcomings, we incorporate this observation into the subset construction algorithm described in Fig. 1.3. Fig. 6.6 shows a new subset construction algorithm for the CNNFA. In our algorithm, function η is a single value map

$$\{[V, \delta(V, \Sigma)] : d \text{ is a DFA state denoting NFA state set } V\},$$

and η^{-1} is the inverse of function η. This new algorithm uses the same time and space resource bounds as the old algorithm, but it produces DFA's as small as DFA's constructed from Thompson's NFA using important state heuristic.

$\sigma := \emptyset$
workset $:= \{\{q_0\}\}$
$\eta =: \{[\{q_0\}, \delta(\{q_0\}, \Sigma)]\}$
while $\exists V \in$ workset do
    workset := workset - $\{V\}$
    for each symbol $a \in \Sigma$ and set of states $B = \{x \in \eta(V) | A(x) = a\}$,
        where $B \neq \emptyset$ do
      if B belongs to the domain of σ or to workset then
        $\sigma(V, a) := B$
      else if $(C := \delta(B, \Sigma))$ belongs to the range of η
        $\sigma(V, a) := \eta^{-1}(C)$
      else
        $\sigma(V, a) := B$
        workset := workset $\cup \{B\}$
        $\eta := \eta \cup \{[B, C]\}$
      end if
    end for
end while

Figure 6.6: Space saving subset construction for the CNNFA

Chapter 7

Conclusion

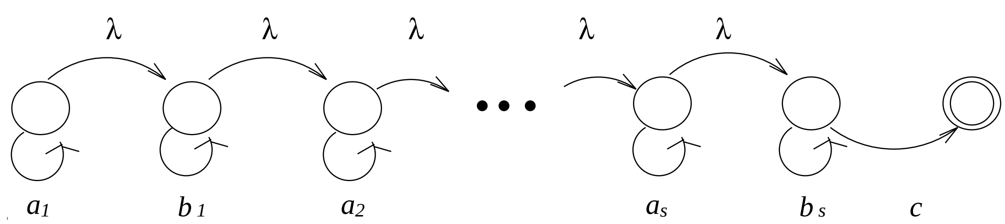
We propose the CNNFA as a better alternative for the classical Thompson's NFA and McNaughton and Yamada's NFA. Theoretical analysis and confirming empirical evidence demonstrate that our proposed CNNFA leads to a substantially more efficient way of turning regular expressions into NFA's and DFA's than other approaches in current use.

The CNNFA is one order of magnitude smaller than the McNaughton and Yamada's NFA; the CNNFA is linearly faster than McNaughton and Yamada's NFA in acceptance testing; and DFA construction starting from the CNNFA can be quadratically faster than when starting from McNaughton and Yamada's NFA. The superiority of the CNNFA is demonstrated by benchmarks showing that cgrep is dramatically faster than the McNaughton and Yamada's NFA based egreps – the UNIX egrep and the GNU e?grep.

Our benchmark result confirms theoretical analysis that the CNNFA is smaller but faster than Thompson's NFA and its variants. Thompson's NFA contains redundant states and edges. To optimize Thompson's NFA deep global analysis is often needed. In contrast, the CNNFA is efficiently constructed by a simple method, and can be regarded as an optimized Thompson's NFA. The λ-closure step is crucial in DFA construction when starting from Thompson's NFA, but it is time consuming. Most of the λ-closure operations are often omitted when starting from the CNNFA because the path compression transformation is able to eliminate NFA states.

In the following, we list a number of future research directions. Though the CNNFA is faster than popular NFA's, we believe that the performance of the CNNFA can be improved by a better implementation. Tree contraction is a technique to reduce the size of the CNNFA. However, the effectiveness of tree contraction has not been investigated.

It is worthwhile to investigate optimization techniques for the CNNFA. Thompson's NFA for regular expression $((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$ can be optimized to the machine illustrated as follows.



It consists of $2s + 1$ states and $4s$ edges. The CNNFA (see Fig. 4.9) of

$((a_1^* b_1^*) \cdots (a_s^* b_s^*))c$ has $5s + 1$ states and $10s - 1$ edges. Though we can further optimize our CNNFA through an *ad hoc* approach, it is desirable to devise more formal and general techniques to optimize the CNNFA. It would also be interesting to obtain a sharper analysis of the constant factors in comparing the CNNFA with other NFA's, particularly, Thompson's NFA optimized.

One of the merits of the CNNFA is its structure. We utilize structure properties of the CNNFA to analyze space and time complexities. In contrast to the spaghetti structure of Thompson's NFA, the CNNFA is well organized, and has no cycles. Comparing to Thompson's NFA, it is easier to adopt the CNNFA to Meyer's hybrid machine [19] (cf. Section 6.2). The CNNFA also directly corresponds to the star normal form notation for regular expressions [6]. It would be worthwhile to re-examine classical problems from the CNNFA point of view.

The choice between using NFA's or DFA's for acceptance testing of strings against regular expressions is a space/time tradeoff; however, if a DFA is the choice, then a min-state DFA is more profitable because a min-state DFA is as fast as a DFA, but it can be exponentially smaller. Consider regular expression $R = (a|b)^* a (a|b)^n (a|b)^*$. The DFA M_R constructed by subset construction has an exponential number of states; but the min-state DFA equivalent to M_R has only a linear number of states. DFA minimization algorithms currently in use are off-line [13,24]. We have to construct the whole DFA before minimization. It seems that an on-line version of min-state DFA construction algorithm uses less auxiliary space; but unfortunately, it uses exponential auxiliary space at the worst case. Consider another regular expression $R' = (a|b)^*a(a|b)^*(a|b)^n$ which is similar to R and equivalent to R. DFA $M_{R'}$ constructed by subset construction has only a linear number of states. It would be interesting to design a small collection of transformation rules for regular expressions so that DFA's constructed from transformed regular expressions would not be much larger than their equivalent min-state DFA's in most of the cases. A better DFA minimization implementation also deserves further investigation.

Appendix A

The CNNFA Benchmark Data

A.1 NFA Acceptance Testing Benchmark Timing Data

All tests in this section are performed on a SUN 3/250 server. Benchmark time is in seconds.

$(abc\cdots)$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
1000	0.14	0.34	0.58	0.76	0.18
1500	0.20	0.52	1.00	1.18	0.30
2000	0.30	0.74	1.32	1.58	0.42
2500	0.38	0.90	1.60	2.00	0.54
3000	0.44	1.12	2.00	2.44	0.66
4000	0.64	1.54	2.58	3.32	0.84
4500	0.72	1.56	2.78	3.42	0.96
5000	0.70	1.48	2.88	3.56	0.92

$(abc\cdots)^*$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
10	0.32	0.62	1.10	1.56	0.32
20	0.26	0.60	1.12	1.44	0.34
30	0.28	0.64	1.12	1.42	0.36
40	0.28	0.64	1.08	1.44	0.34
50	0.26	0.64	1.08	1.48	0.38
60	0.28	0.64	1.12	1.50	0.34
70	0.26	0.66	1.10	1.46	0.34
80	0.26	0.64	1.10	1.46	0.32
90	0.28	0.64	1.14	1.46	0.36
100	0.26	0.64	1.10	1.46	0.34

$(a\|b\|c\cdots)^*$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
10	5.46	1.36	4.20	1.76	0.82
20	10.48	2.18	7.52	2.02	1.38
30	15.70	3.04	10.86	2.18	1.86
40	21.16	3.76	14.28	2.56	2.42
50	26.22	4.60	17.28	2.84	3.00
60	31.62	5.46	22.56	3.12	3.66
70	36.62	6.20	23.94	3.26	4.36
80	42.02	7.12	27.38	3.56	5.22
90	47.94	7.92	30.44	3.90	6.00
100	52.00	8.70	35.10	4.10	6.88

$((a\|\lambda)^n-)^*$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
10	7.14	4.30	5.50	8.10	3.96
20	12.94	7.14	9.14	13.76	12.14
30	19.90	10.60	13.76	20.74	26.12
40	25.92	12.90	17.06	26.22	42.16
50	31.46	16.82	22.36	34.26	66.54
60	36.10	18.96	24.98	39.78	91.74
70	43.56	22.96	29.54	46.04	127.28
80	51.18	25.96	35.20	53.60	171.02
90	52.66	26.80	35.54	54.24	187.56
100	61.30	31.12	41.00	63.44	248.04

$((a\|\lambda)(b\|\lambda)\cdots)^*$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
10	7.08	3.92	4.26	1.94	0.86
20	13.06	7.42	7.60	2.06	1.38
30	19.92	10.96	10.78	2.30	1.92
40	26.32	14.38	14.16	2.52	2.44
50	32.32	18.00	17.34	2.84	3.00
60	37.68	21.66	20.78	3.10	3.66
70	43.82	25.12	24.06	3.24	4.34
80	51.54	28.48	27.78	3.58	5.18
90	57.80	32.08	30.80	3.88	5.90
100	64.56	35.46	33.98	4.06	6.86

$((a\|\lambda)(b\|\lambda)\cdots -)^*$					
length	TNFA	TNFA opt.	CNNFA unopt.	CNNFA	MYNFA
10	4.40	2.82	2.66	3.36	0.68
20	8.08	4.94	4.08	4.86	1.00
30	12.30	7.16	5.50	6.54	1.34
40	16.06	9.22	6.72	7.96	1.64
50	19.22	11.24	8.10	9.34	1.88
60	23.46	13.90	9.80	11.04	2.38
70	27.32	16.18	11.22	12.68	2.84
80	31.90	18.18	12.72	14.34	3.16
90	34.80	19.92	13.64	15.34	3.40
100	38.98	22.04	14.96	18.50	3.78
programming language					
	98.20	20.50	65.36	13.68	24.46

A.2 DFA Construction Benchmark Data

A.2.1 DFA construction Time

All tests in this section are performed on a SUN 3/50 server. Benchmark time is in seconds.

$(abc\cdots)^*$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
25	0.02	0.02	0.02	0.02	0.02	0.00
50	0.06	0.04	0.04	0.04	0.04	0.04
75	0.06	0.04	0.04	0.04	0.06	0.00
100	0.06	0.04	0.06	0.02	0.06	0.06
125	0.08	0.06	0.06	0.06	0.06	0.06
150	0.14	0.08	0.06	0.08	0.08	0.06
175	0.16	0.08	0.10	0.08	0.06	0.06
201	0.20	0.10	0.10	0.10	0.06	0.10
225	0.22	0.10	0.10	0.08	0.10	0.08
250	0.26	0.10	0.14	0.14	0.10	0.10

$(a\|b\|c\cdots)^*$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
25	0.68	0.14	0.02	0.02	0.02	0.12
50	4.54	0.46	0.08	0.04	0.04	0.42
75	14.72	1.10	0.16	0.08	0.04	0.96
100	34.28	1.78	0.26	0.10	0.04	1.64
125	66.00	2.80	0.40	0.14	0.02	2.28
150	113.26	4.06	0.54	0.18	0.06	3.56
175	178.52	5.52	0.76	0.26	0.06	5.00
200	265.20	7.32	1.00	0.28	0.06	6.26
225	375.88	9.18	1.22	0.38	0.10	8.26
250	514.34	11.28	1.46	0.46	0.10	9.80

$(0\|\cdots\|9)^n$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
25	1.20	0.5	0.08	0.06	0.06	0.42
50	2.40	0.98	0.22	0.12	0.12	0.88
75	3.64	1.44	0.34	0.16	0.16	1.30
100	4.88	2.00	0.42	0.24	0.20	1.80
125	6.12	2.40	0.54	0.30	0.26	2.26
150	7.36	3.00	0.64	0.36	0.30	2.74
175	8.56	3.46	0.76	0.40	0.34	3.08
200	9.74	4.10	0.84	0.48	0.40	3.58
225	11.10	4.66	0.92	0.50	0.40	3.90
250	12.44	4.94	1.02	0.60	0.52	4.40

$((a\|\lambda)(b\|\lambda)\cdots-)^*$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
25	0.22	0.12	0.16	0.14	0.12	0.08
50	1.22	0.28	0.84	0.72	0.28	0.28
75	3.54	0.66	2.66	2.04	0.66	0.64
100	8.02	1.14	5.78	4.48	1.06	1.02
125	15.16	1.72	10.78	8.08	1.68	1.60
150	25.62	2.42	18.00	13.52	2.44	2.28
175	39.88	3.36	28.02	20.84	3.30	3.12
200	59.04	4.52	41.36	30.70	4.24	4.10
225	83.58	5.52	58.26	42.52	5.40	4.94
250	113.68	7.02	79.04	57.92	5.56	5.20

$((a\|\lambda)(b\|\lambda)\cdots)^*$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
25	0.80	0.14	0.04	0.02	0.02	0.12
50	5.74	0.44	0.08	0.06	0.04	0.42
75	18.74	1.08	0.18	0.12	0.02	0.92
100	43.58	1.82	0.28	0.22	0.04	1.66
125	84.46	2.88	0.44	0.32	0.04	2.44
150	144.50	4.14	0.66	0.46	0.06	3.66
175	228.56	5.60	0.88	0.60	0.08	5.92
200	356.58	9.28	1.24	0.84	0.08	7.76
225	505.66	11.48	1.42	1.04	0.10	10.06
250	668.84	11.42	1.78	1.26	0.08	9.88

$(a\|b)^*a(a\|b)^n$						
length	TNFA	TNFA k.	TNFA i.	TNFA opt.	CNNFA	MYNFA
1	0.02	0.02	0.02	0.02	0.02	0.02
2	0.00	0.00	0.02	0.02	0.02	0.00
3	0.00	0.02	0.02	0.02	0.02	0.02
4	0.06	0.04	0.04	0.04	0.04	0.04
5	0.06	0.06	0.10	0.06	0.08	0.08
6	0.14	0.12	0.12	0.12	0.12	0.08
7	0.30	0.22	0.30	0.26	0.24	0.20
8	0.64	0.48	0.64	0.50	0.46	0.42
9	1.38	0.96	1.22	1.04	1.00	0.90
10	2.76	1.98	2.62	2.14	2.06	2.08
programming language						
	1.92	0.36	0.32	0.12	0.10	0.34

A.2.2 Constructed DFA Size

For each test pattern, we show the numbers of states and edges in the constructed DFA. Each DFA state corresponds to a set of NFA states; the weight of a DFA states is defined to be the size of its corresponding NFA states set. The weight of an edge in a DFA is the sum of the weight of its origin and tail. DFA construction time is proportional to the node and edge weight of constructed DFA's.

$(abc\cdots)^*$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	25	26	26	30	28
TNFA k.	25	26	26	26	26
TNFA i.	25	25	25	26	26
TNFA o.	25	25	25	25	25
CNNFA	25	26	26	26	26
MYNFA	25	26	26	26	26

machine	length	node no.	edge no.	node weight	edge weight
TNFA	50	51	51	55	53
TNFA k.	50	51	51	51	51
TNFA i.	50	50	50	51	51
TNFA o.	50	50	50	50	50
CNNFA	50	51	51	51	51
MYNFA	50	51	51	51	51
machine	length	node no.	edge no.	node weight	edge weight
TNFA	75	76	76	80	78
TNFA k.	75	76	76	76	76
TNFA i.	75	75	75	76	76
TNFA o.	75	75	75	75	75
CNNFA	75	76	76	76	76
MYNFA	75	76	76	76	76
machine	length	node no.	edge no.	node weight	edge weight
TNFA	100	101	101	105	103
TNFA k.	100	101	101	101	101
TNFA i.	100	100	100	101	101
TNFA o.	100	100	100	100	100
CNNFA	100	101	101	101	101
MYNFA	100	101	101	101	101
machine	length	node no.	edge no.	node weight	edge weight
TNFA	125	126	126	130	128
TNFA k.	125	126	126	126	126
TNFA i.	125	125	125	126	126
TNFA o.	125	125	125	125	125
CNNFA	125	126	126	126	126
MYNFA	125	126	126	126	126
machine	length	node no.	edge no.	node weight	edge weight
TNFA	150	151	151	155	153
TNFA k.	150	151	151	151	151
TNFA i.	150	150	150	151	151
TNFA o.	150	150	150	150	150
CNNFA	150	151	151	151	151
MYNFA	150	151	151	151	151

machine	length	node no.	edge no.	node weight	edge weight
TNFA	175	176	176	180	178
TNFA k.	175	176	176	176	176
TNFA i.	175	175	175	176	176
TNFA o.	175	175	175	175	175
CNNFA	175	176	176	176	176
MYNFA	175	176	176	176	176
machine	length	node no.	edge no.	node weight	edge weight
TNFA	200	201	201	205	203
TNFA k.	200	201	201	201	201
TNFA i.	200	200	200	201	201
TNFA o.	200	200	200	200	200
CNNFA	200	201	201	201	201
MYNFA	200	201	201	201	201
machine	length	node no.	edge no.	node weight	edge weight
TNFA	225	226	226	230	228
TNFA k.	225	226	226	226	226
TNFA i.	225	225	225	226	226
TNFA o.	225	225	225	225	225
CNNFA	225	226	226	226	226
MYNFA	225	226	226	226	226
machine	length	node no.	edge no.	node weight	edge weight
TNFA	250	251	251	255	253
TNFA k.	250	251	251	251	251
TNFA i.	250	250	250	251	251
TNFA o.	250	250	250	250	250
CNNFA	250	251	251	251	251
MYNFA	250	251	251	251	251

$(a\|b\|c\cdots)^*$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	25	26	650	1650	41574
TNFA k.	25	26	650	26	650
TNFA i.	25	1	25	26	650
TNFA o.	25	1	25	1	25
CNNFA	25	2	50	2	50
MYNFA	25	26	650	26	650

machine	length	node no.	edge no.	node weight	edge weight
TNFA	50	51	2550	6425	322524
TNFA k.	50	51	2550	51	2550
TNFA i.	50	1	50	51	2550
TNFA o.	50	1	50	1	50
CNNFA	50	2	100	2	100
MYNFA	50	51	2550	51	2550
machine	length	node no.	edge no.	node weight	edge weight
TNFA	75	76	5700	14325	1077224
TNFA k.	75	76	5700	76	5700
TNFA i.	75	1	75	76	5700
TNFA o.	75	1	75	1	75
CNNFA	75	2	150	2	150
MYNFA	75	76	5700	76	5700
machine	length	node no.	edge no.	node weight	edge weight
TNFA	100	101	10100	25350	2540049
TNFA k.	100	101	10100	101	10100
TNFA i.	100	1	100	101	10100
TNFA o.	100	1	100	1	100
CNNFA	100	2	200	2	200
MYNFA	100	101	10100	101	10100
machine	length	node no.	edge no.	node weight	edge weight
TNFA	125	126	15750	39500	4945374
TNFA k.	125	126	15750	126	15750
TNFA i.	125	1	125	126	15750
TNFA o.	125	1	125	1	125
CNNFA	125	2	250	2	250
MYNFA	125	126	15750	126	15750
machine	length	node no.	edge no.	node weight	edge weight
TNFA	150	151	22650	56775	8527574
TNFA k.	150	151	22650	151	22650
TNFA i.	150	1	150	151	22650
TNFA o.	150	1	150	1	150
CNNFA	150	2	300	2	300
MYNFA	150	151	22650	151	22650

machine	length	node no.	edge no.	node weight	edge weight
TNFA	175	176	30800	77175	13521024
TNFA k.	175	176	30800	176	30800
TNFA i.	175	1	175	176	30800
TNFA o.	175	1	175	1	175
CNNFA	175	2	350	2	350
MYNFA	175	176	30800	176	30800
machine	length	node no.	edge no.	node weight	edge weight
TNFA	200	201	40200	100700	20160099
TNFA k.	200	201	40200	201	40200
TNFA i.	200	1	200	201	40200
TNFA o.	200	1	200	1	200
CNNFA	200	2	400	2	400
MYNFA	200	201	40200	201	40200
machine	length	node no.	edge no.	node weight	edge weight
TNFA	225	226	50850	127350	28679174
TNFA k.	225	226	50850	226	50850
TNFA i.	225	1	225	226	50850
TNFA o.	225	1	225	1	225
CNNFA	225	2	450	2	450
MYNFA	225	226	50850	226	50850
machine	length	node no.	edge no.	node weight	edge weight
TNFA	250	251	62750	157125	39312624
TNFA k.	250	251	62750	251	62750
TNFA i.	250	1	250	251	62750
TNFA o.	250	1	250	1	250
CNNFA	250	2	500	2	500
MYNFA	250	251	62750	251	62750

$(0\vert\cdots\vert 9)^n$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	25	251	2410	5939	57004
TNFA k.	25	251	2410	251	2410
TNFA i.	25	26	250	251	2410
TNFA o.	25	26	250	26	250
CNNFA	25	26	250	26	250
MYNFA	25	251	2410	251	2410

machine	length	node no.	edge no.	node weight	edge weight
TNFA	50	501	4910	12039	118004
TNFA k.	50	501	4910	501	4910
TNFA i.	50	51	500	501	4910
TNFA o.	50	51	500	51	500
CNNFA	50	51	500	51	500
MYNFA	50	501	4910	501	4910
machine	length	node no.	edge no.	node weight	edge weight
TNFA	75	751	7410	18139	179004
TNFA k.	75	751	7410	751	7410
TNFA i.	75	76	750	751	7410
TNFA o.	75	76	750	76	750
CNNFA	75	76	750	76	750
MYNFA	75	751	7410	751	7410
machine	length	node no.	edge no.	node weight	edge weight
TNFA	100	1001	9910	24239	240004
TNFA k.	100	1001	9910	1001	9910
TNFA i.	100	101	1000	1001	9910
TNFA o.	100	101	1000	101	1000
CNNFA	100	101	1000	101	1000
MYNFA	100	1001	9910	1001	9910
machine	length	node no.	edge no.	node weight	edge weight
TNFA	125	1251	12410	30339	301004
TNFA k.	125	1251	12410	1251	12410
TNFA i.	125	126	1250	1251	12410
TNFA o.	125	126	1250	126	1250
CNNFA	125	126	1250	126	1250
MYNFA	125	1251	12410	1251	12410
machine	length	node no.	edge no.	node weight	edge weight
TNFA	150	1501	14910	36439	362004
TNFA k.	150	1501	14910	1501	14910
TNFA i.	150	151	1500	1501	14910
TNFA o.	150	151	1500	151	1500
CNNFA	150	151	1500	151	1500
MYNFA	150	1501	14910	1501	14910

machine	length	node no.	edge no.	node weight	edge weight
TNFA	175	1751	17410	42539	423004
TNFA k.	175	1751	17410	1751	17410
TNFA i.	175	176	1750	1751	17410
TNFA o.	175	176	1750	176	1750
CNNFA	175	176	1750	176	1750
MYNFA	175	1751	17410	1751	17410
machine	length	node no.	edge no.	node weight	edge weight
TNFA	200	2001	19910	48639	484004
TNFA k.	200	2001	19910	2001	19910
TNFA i.	200	201	2000	2001	19910
TNFA o.	200	201	2000	201	2000
CNNFA	200	201	2000	201	2000
MYNFA	200	2001	19910	2001	19910
machine	length	node no.	edge no.	node weight	edge weight
TNFA	225	2251	22410	54739	545004
TNFA k.	225	2251	22410	2251	22410
TNFA i.	225	226	2250	2251	22410
TNFA o.	225	226	2250	226	2250
CNNFA	225	226	2250	226	2250
MYNFA	225	2251	22410	2251	22410
machine	length	node no.	edge no.	node weight	edge weight
TNFA	250	2501	24910	60839	606004
TNFA k.	250	2501	24910	2501	24910
TNFA i.	250	251	2500	2501	24910
TNFA o.	250	251	2500	251	2500
CNNFA	250	251	2500	251	2500
MYNFA	250	2501	24910	2501	24910

$((a\|\lambda)(b\|\lambda)\cdots -)^*$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	25	27	351	1027	8476
TNFA k.	25	27	351	27	351
TNFA i.	25	27	351	352	2951
TNFA o.	25	27	351	352	2951
CNNFA	25	27	351	27	351
MYNFA	25	27	351	27	351

machine	length	node no.	edge no.	node weight	edge weight
TNFA	50	52	1281	3928	65076
TNFA k.	50	52	1281	53	1326
TNFA i.	50	53	1281	1330	22152
TNFA o.	50	52	1281	1328	22151
CNNFA	50	52	1281	53	1326
MYNFA	50	52	1281	53	1326
machine	length	node no.	edge no.	node weight	edge weight
TNFA	75	77	2881	8703	216676
TNFA k.	75	77	2881	78	2926
TNFA i.	75	78	2881	2930	73227
TNFA o.	75	77	2881	2928	73226
CNNFA	75	77	2881	78	2926
MYNFA	75	77	2881	78	2926
machine	length	node no.	edge no.	node weight	edge weight
TNFA	100	102	5106	15353	510151
TNFA k.	100	102	5106	103	5151
TNFA i.	100	103	5106	5155	171802
TNFA o.	100	102	5106	5153	171801
CNNFA	100	102	5106	103	5151
MYNFA	100	102	5106	103	5151
machine	length	node no.	edge no.	node weight	edge weight
TNFA	125	127	7956	23878	992376
TNFA k.	125	127	7956	128	8001
TNFA i.	125	128	7956	8005	333502
TNFA o.	125	127	7956	8003	333501
CNNFA	125	127	7956	128	8001
MYNFA	125	127	7956	128	8001
machine	length	node no.	edge no.	node weight	edge weight
TNFA	150	152	11431	34278	1710226
TNFA k.	150	152	11431	153	11476
TNFA i.	150	153	11431	11480	573952
TNFA o.	150	152	11431	11478	573951
CNNFA	150	152	11431	153	11476
MYNFA	150	152	11431	153	11476

machine	length	node no.	edge no.	node weight	edge weight
TNFA	175	177	15531	46553	2710576
TNFA k.	175	177	15531	178	15576
TNFA i.	175	178	15531	15580	908777
TNFA o.	175	177	15531	15578	908776
CNNFA	175	177	15531	178	15576
MYNFA	175	177	15531	178	15576
machine	length	node no.	edge no.	node weight	edge weight
TNFA	200	202	20256	60703	4040301
TNFA k.	200	202	20256	203	20301
TNFA i.	200	203	20256	20305	1353602
TNFA o.	200	202	20256	20303	1353601
CNNFA	200	202	20256	203	20301
MYNFA	200	202	20256	203	20301
machine	length	node no.	edge no.	node weight	edge weight
TNFA	225	227	25606	76728	5746276
TNFA k.	225	227	25606	228	25651
TNFA i.	225	228	25606	25655	1924052
TNFA o.	225	227	25606	25653	1924051
CNNFA	225	227	25606	228	25651
MYNFA	225	227	25606	228	25651
machine	length	node no.	edge no.	node weight	edge weight
TNFA	250	252	31581	94628	7875376
TNFA k.	250	252	31581	253	31626
TNFA i.	250	253	31581	31630	2635752
TNFA o.	250	252	31581	31628	2635751
CNNFA	250	252	31581	253	31626
MYNFA	250	252	31581	253	31626

$((a\|\lambda)(b\|\lambda)\cdots)^*$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	25	26	650	2028	50700
TNFA k.	25	26	650	26	650
TNFA i.	25	1	25	26	650
TNFA o.	25	1	25	25	625
CNNFA	25	2	50	2	50
MYNFA	25	26	650	26	650

machine	length	node no.	edge no.	node weight	edge weight
TNFA	50	51	2550	7803	390150
TNFA k.	50	51	2550	51	2550
TNFA i.	50	1	50	51	2550
TNFA o.	50	1	50	50	2500
CNNFA	50	2	100	2	100
MYNFA	50	51	2550	51	2550
machine	length	node no.	edge no.	node weight	edge weight
TNFA	75	76	5700	17328	1299600
TNFA k.	75	76	5700	76	5700
TNFA i.	75	1	75	76	5700
TNFA o.	75	1	75	75	5625
CNNFA	75	2	150	2	150
MYNFA	75	76	5700	76	5700
machine	length	node no.	edge no.	node weight	edge weight
TNFA	100	101	10100	30603	3060300
TNFA k.	100	101	10100	101	10100
TNFA i.	100	1	100	101	10100
TNFA o.	100	1	100	100	10000
CNNFA	100	2	200	2	200
MYNFA	100	101	10100	101	10100
machine	length	node no.	edge no.	node weight	edge weight
TNFA	125	126	15750	47628	5953500
TNFA k.	125	126	15750	126	15750
TNFA i.	125	1	125	126	15750
TNFA o.	125	1	125	125	15625
CNNFA	125	2	250	2	250
MYNFA	125	126	15750	126	15750
machine	length	node no.	edge no.	node weight	edge weight
TNFA	150	151	22650	68403	10260450
TNFA k.	150	151	22650	151	22650
TNFA i.	150	1	150	151	22650
TNFA o.	150	1	150	150	22500
CNNFA	150	2	300	2	300
MYNFA	150	151	22650	151	22650

machine	length	node no.	edge no.	node weight	edge weight
TNFA	175	176	30800	92928	16262400
TNFA k.	175	176	30800	176	30800
TNFA i.	175	1	175	176	30800
TNFA o.	175	1	175	175	30625
CNNFA	175	2	350	2	350
MYNFA	175	176	30800	176	30800
machine	length	node no.	edge no.	node weight	edge weight
TNFA	200	201	40200	121203	24240600
TNFA k.	200	201	40200	201	40200
TNFA i.	200	1	200	201	40200
TNFA o.	200	1	200	200	40000
CNNFA	200	2	400	2	400
MYNFA	200	201	40200	201	40200
machine	length	node no.	edge no.	node weight	edge weight
TNFA	225	226	50850	153228	34476300
TNFA k.	225	226	50850	226	50850
TNFA i.	225	1	225	226	50850
TNFA o.	225	1	225	225	50625
CNNFA	225	2	450	2	450
MYNFA	225	226	50850	226	50850
machine	length	node no.	edge no.	node weight	edge weight
TNFA	250	251	62750	189003	47250750
TNFA k.	250	251	62750	251	62750
TNFA i.	250	1	250	251	62750
TNFA o.	250	1	250	250	62500
CNNFA	250	2	500	2	500
MYNFA	250	251	62750	251	62750

$(a\|b)^*a(a\|b)^n$					
machine	length	node no.	edge no.	node weight	edge weight
TNFA	1	5	10	39	83
TNFA k.	1	5	10	9	19
TNFA i.	1	4	9	18	39
TNFA o.	1	4	8	8	16
CNNFA	1	5	10	9	19
MYNFA	1	5	10	9	19

machine	length	node no.	edge no.	node weight	edge weight
TNFA	2	9	18	89	183
TNFA k.	2	9	18	21	43
TNFA i.	2	8	18	44	94
TNFA o.	2	8	16	20	40
CNNFA	2	9	18	21	43
MYNFA	2	9	18	21	43
machine	length	node no.	edge no.	node weight	edge weight
TNFA	3	17	34	205	415
TNFA k.	3	17	34	49	99
TNFA i.	3	16	36	104	220
TNFA o.	3	16	32	48	96
CNNFA	3	17	34	49	99
MYNFA	3	17	34	49	99
machine	length	node no.	edge no.	node weight	edge weight
TNFA	4	33	66	469	943
TNFA k.	4	33	66	113	227
TNFA i.	4	32	72	240	504
TNFA o.	4	32	64	112	224
CNNFA	4	33	66	113	227
MYNFA	4	33	66	113	227
machine	length	node no.	edge no.	node weight	edge weight
TNFA	5	65	130	1061	2127
TNFA k.	5	65	130	257	515
TNFA i.	5	64	144	544	1136
TNFA o.	5	64	128	256	512
CNNFA	5	65	130	257	515
MYNFA	5	65	130	257	515
machine	length	node no.	edge no.	node weight	edge weight
TNFA	6	129	258	2373	4751
TNFA k.	6	129	258	577	1155
TNFA i.	6	128	288	1216	2528
TNFA o.	6	128	256	576	1152
CNNFA	6	129	258	577	1155
MYNFA	6	129	258	577	1155

machine	length	node no.	edge no.	node weight	edge weight
TNFA	7	257	514	5253	10511
TNFA k.	7	257	514	1281	2563
TNFA i.	7	256	576	2688	5568
TNFA o.	7	256	512	1280	2560
CNNFA	7	257	514	1281	2563
MYNFA	7	257	514	1281	2563
machine	length	node no.	edge no.	node weight	edge weight
TNFA	8	513	1026	11525	23055
TNFA k.	8	513	1026	2817	5635
TNFA i.	8	512	1152	5888	12160
TNFA o.	8	512	1024	2816	5632
CNNFA	8	513	1026	2817	5635
MYNFA	8	513	1026	2817	5635
machine	length	node no.	edge no.	node weight	edge weight
TNFA	9	1025	2050	25093	50191
TNFA k.	9	1025	2050	6145	12291
TNFA i.	9	1024	2304	12800	26368
TNFA o.	9	1024	2048	6144	12288
CNNFA	9	1025	2050	6145	12291
MYNFA	9	1025	2050	6145	12291
machine	length	node no.	edge no.	node weight	edge weight
TNFA	10	2049	4098	54277	108559
TNFA k.	10	2049	4098	13313	26627
TNFA i.	10	2048	4608	27648	56832
TNFA o.	10	2048	4096	13312	26624
CNNFA	10	2049	4098	13313	26627
MYNFA	10	2049	4098	13313	26627
programming language					
machine	length	node no.	edge no.	node weight	edge weight
TNFA		91	1931	5209	123739
TNFA k.		91	1931	109	1949
TNFA i.		17	415	449	10768
TNFA o.		19	451	56	946
CNNFA		24	581	42	599
MYNFA		91	1931	109	1949

Appendix B

Cgrep Source Code

We implemented cgrep based on the CNNFA proposed in this Thesis. The cgrep is fully compatible with UNIX egrep. All the command options provided by UNIX egrep are also available in cgrep. The cgrep source code is available upon request.

One of the most interesting implementation techniques not discussed yet is an *I-forest* traversal algorithm. Consider a branching binary tree T, and a set V of nodes. We want to find the set of leaves in T which are descendents of nodes in V. This operation is a fundamental part of the $\delta(V, a)$ operation described in Chapter 3. A naive implementation would traverse T alone tree edges. We take advantage of the fact that T is a branching binary tree, and use an array A_T to represent leaves of T. The leaves of T are stored in A_T according to their post-ordering in T. Instead of traversing T, we traverse array A_T in our implementation. A simulation shows that the array traversal algorithm is four times faster than the tree traversal algorithm.

The cgrep source consists of the following file.

1. cgrep.c – main program of cgrep.
2. parse.c – a recursive descendent parser for regular expressions to construct CNNFA's while parsing.
3. pack.c – packing transformation routines.
4. path.c – path compression routines.
5. fly.c – main routine to construct DFA from the CNNFA on the fly.
6. nfa2dfa.c – routines used by fly.c.
7. cnfa.h – the CNNFA data structures description.
8. fly.h – DFA data structures description.
9. time.h – benchmark timer routines.

B.1 Implemetation Note

B.2 cgrep program listing

B.2.1 cnfa.h

```
#ifndef NULL
```

```
#define NULL (0)
#endif

#ifndef TRUE
#define TRUE    1
#define FALSE   0
#endif

/* my character type */

#define NEWLINE_OR  TRUE

typedef unsigned char MYCHAR;
#define MAX_CHAR 128
MYCHAR *new_ch();

/* -----------------------------------------
 *  Compressed NFA
 * ----------------------------------------- */

struct edge;
struct nset;
struct set {

    struct set *left;   /* left child */
    struct set *right;  /* right child */

    struct edge *edge;  /* beginning of edge list */
    struct edge *edget; /* tail of edge list */

    MYCHAR *ch;
```

```
    int ccnt;
    char final;

    struct set *fparent, *iparent;


    /* set/nset conversion */
    char pin;       /* pin_count */
    struct set *Fclass; /* used in F-tree, pointing
              to nearest ancestor with edge */
    struct set  *Iclass,
            *Inext;

    struct set  *class,
            *rep;
    struct nset *Frep_addr;

    int     internal,
        external;


    struct nset *start,
            *stop;

    struct nset *nset;  /* corresponding nset address */
    MYCHAR  *nch;
    MYCHAR  ch_size;

};

#define FSET    1
```

```
#define ISET    2

struct edge {

    struct set *iset;   /* destination */
    struct set *fset;   /* origin */
    struct edge *next;  /* forward link */
    struct edge *back;  /* forward link */
    char    attrib;


    /* used in NSET */
    struct nset *start, *stop;
    struct edge *anc;
    char mark;
};
#define DIRECT  1
#define LAZY    2

#define INSERT 1
#define APPEND 2

typedef struct set SET;
typedef struct edge EDGE;


SET *join_sets(), *new_set();
EDGE *new_edge();

struct forest {
    SET *set;
```

```
    struct forest *next;
};


extern struct forest *Iforest, *Fforest;
extern SET  *AFset,
        *Start,
        *Final;

extern struct nset *Nstart;

/* -------------------------------------------
 *  Simplified Compressed NFA
 * ------------------------------------------- */
struct nset {
    struct nset *ext;
    struct edge *edge;
    struct nset *mnext;
    struct nset *real;
    MYCHAR *ch;

    int ext_cnt;
    int size;
    char final;
    MYCHAR ch_size;
    char mark;
};

typedef struct nset NSET;

/*
```

```
 *  Fast Dynamic memory allocation
 */

struct set_pool {
    struct set *pool;
    int cnt;
    struct set_pool *next;
} ;
struct nset_pool {
    struct nset *pool;
    int cnt;
    struct nset_pool *next;
} ;

struct edge_pool {
    struct edge *pool;
    int cnt;
    struct edge_pool *next;
} ;
struct ch_pool {
    MYCHAR *pool;
    int cnt;
    struct ch_pool *next;
} ;


extern int EGREP;
#define MAX_NFA 40960


extern int  bflag,
```

```
    cflag,
    hflag,
    iflag,
    lflag,
    nflag,
    pflag,
    sflag,
    vflag;
```

B.2.2 fly.h

```
struct dfa {
    struct dfa *trans[MAX_CHAR];
    struct dfa *next;   /* link -- used in hash or avail */
    char final;

    NSET **state;
    int cnt;
    char build;     /* is this state been built completely */
    char reserve;       /* reserved state -- for compaction */
};

typedef struct dfa DFA;

/* DFA state pool */
#define MAX_NSET_POOL   160000

#ifndef MAX_DSIZE
#define MAX_DSIZE 2*MAX_CHAR+2  /* orginal MAX_CHAR * 2 */
#endif
```

```
extern DFA  ST[MAX_DSIZE];  /* state pool --- keep at most */
extern NSET *Nsetpool[MAX_NSET_POOL];
    /* to store collection of NFA states that --- */

extern DFA  *Avail;     /* Available Dfa state pool */
extern NSET **Npp;      /* Netpool pointer */


/* open hash table */
#define MAX_HASH    101
extern DFA  *Hash[MAX_HASH];



struct mset {
    NSET  *st;
    struct mset *next;
};

/* for muiti-set discrimination */
#define MAX_MBASE 60000

extern struct mset *Mset[MAX_CHAR];
extern struct mset Mbase[MAX_MBASE];
extern struct mset *Mptr ;
extern MYCHAR Mset_dirt[MAX_CHAR];
extern int Midx ; /* cnt of entry in Mset which is non NULL */
extern DFA *Dfirst;
```

B.2.3 timer.h

```
#include <sys/time.h>
#define NULL 0
#define TIMER_DECL              \
    int ttmp;               \
    int Isec;               \
    float Esec;             \
    struct itimerval ttmp1;

#define TIMER_START             \
    {ttmp = setitimer(ITIMER_VIRTUAL, &ttmp1, NULL);}
#define TIMER_INIT              \
    ttmp1.it_interval.tv_sec = 100000L; \
    ttmp1.it_value.tv_sec = 100000L;    \
    ttmp1.it_interval.tv_usec = 0L;     \
    ttmp1.it_value.tv_usec = 0L;

#define PRINT_STIME printf(" %d msec", Isec)
#define PRINT_TIME  printf(     \
"elapsed time:  %f sec -- %d mini-sec\n",  Esec, Isec)

#define ELAPSE_TIME {   long sec, vsec; \
if( ttmp1.it_value.tv_usec  > ttmp1.it_interval.tv_usec ) { \
  sec = ttmp1.it_interval.tv_sec - ttmp1.it_value.tv_sec - 1;   \
  vsec = 1000000L +(ttmp1.it_interval.tv_usec -     \
  ttmp1.it_value.tv_usec);  \
} else {    \
  sec = ttmp1.it_interval.tv_sec - ttmp1.it_value.tv_sec ;  \
  vsec = (ttmp1.it_interval.tv_usec - ttmp1.it_value.tv_usec);  \
```

```
}; \
Esec = (float) sec + (( (float) vsec ) / 1000000. );    \
Isec = ( ((int) sec) * 1000 ) + ( ((int) vsec) / 1000 );    \
}

#define TIMER_STOP  {ttmp = getitimer(ITIMER_VIRTUAL, &ttmp1);}
```

B.2.4 cgrep.c

```
#include <stdio.h>
#include <stdlib.h>
#include "cnfa.h"
#ifndef USENFA
#include "fly.h"
extern int Uninit;
#endif

int     bflag = FALSE,  /* block # */
    cflag = FALSE,  /* count of matched lines */
    Tflag = FALSE,  /* count of matched lines */
    tflag = FALSE,  /* count of matched lines */
    hflag = FALSE,  /* file bname */
    iflag = FALSE,  /* ignore case */
    lflag = FALSE,  /* only file name are listed */
    nflag = FALSE,  /* line # */
    pflag = FALSE,
    sflag = FALSE,  /* silence */
    flag = 0,
    vflag = FALSE;  /* all but match */
```

```
int     lcnt, lnum;

#define BUF_SIZE    81920
void execute();
void read_input(), reg_compile();
MYCHAR *read_rexp();


#include "timer.h"
TIMER_DECL;

main (argc, argv)
int argc;
char *argv[];
{
    MYCHAR *pat = NULL;
    MYCHAR sstr[BUF_SIZE], *str = NULL;
    int i;
    char c;
    int errflag = 0, loopcnt = 1;
    extern char *optarg;
    extern int optind;
    int cnt, fd;
    char *fname = NULL;
    int print = 0;

    while( (c = getopt(argc, argv, "Ttbchilnsve:f:") ) != -1) {
            switch(c) {
            case 'e':
                pat = (MYCHAR *) optarg;
                break;
```

```
        case 'f':
            if( (pat = read_rexp(optarg)) == NULL) {
                fprintf(stderr,
                "can not open exp file %s\n",
                        optarg);
                exit(0);
            };
            break;

        case 'T':
            Tflag = TRUE;
        case 't':
            tflag = TRUE;
            break;
        case 'b':
            bflag = TRUE;
            break;
        case 'c':
            cflag = TRUE;
            break;
        case 'h':
            hflag = TRUE;
            break;
        case 'i':
            iflag = TRUE;
            break;
        case 'l':
            lflag = TRUE;
            break;
        case 'n':
            nflag = TRUE;
```

```
                break;
            case 's':
                sflag = TRUE;
                break;
            case 'v':
                vflag = TRUE;
                break;
            case '?':
                errflag++;
                break;
            };
            if( errflag ) {
            fprintf(stderr,
"usage:%s [-tbchilnsv] [-e e] [-f efile] exp [file]\n", argv[0]);
            exit(0);
            };
    };

    if( pat == NULL ) {
        if( optind == argc ) {
            fprintf(stderr,
"usage:%s [-tbchilnsv] [-e e] [-f efile] exp [file]\n", argv[0]);
            exit(0);
        };
        pat = (MYCHAR *) argv[optind++];
    };


if(tflag) { TIMER_INIT; TIMER_START; };
    (void) reg_compile(pat);
if(tflag) { TIMER_STOP; ELAPSE_TIME;
```

```
if( Tflag ) { PRINT_STIME; } else {
printf(">> %s NFA construction ", argv[0]); PRINT_TIME; };
};

    if( optind + 1 < argc ) {
        hflag = TRUE;
    };

    flag = 1;
    if( sflag ) {
        flag = 0;   /* print nothing */
    } else {
        if( vflag ) {
            flag = 2;
            if( cflag ) flag = 3;
        } else if( cflag ) flag = 3;
    };


#ifdef USEDFA
if(tflag) { TIMER_INIT; TIMER_START; };
    (void) csubset(pat);
if(tflag) {
TIMER_STOP; ELAPSE_TIME;
if(Tflag) { PRINT_STIME; } else {
printf(">>DFA construction "); PRINT_TIME; };
};
#endif

if(tflag) { TIMER_INIT; TIMER_START; };
```

```
#ifdef USEOL
     if( Uninit ) light_dfa_init();
    Dfirst = (DFA *) build_first();
#endif
    if( optind == argc ) {
        execute(NULL);
        lcnt = (vflag) ? (lnum - lcnt) : lcnt;
        if( flag == 3) printf("match lines:%d\n",lcnt);
    } else {
        for( ; optind < argc; optind++) {
            execute( argv[optind] );
            lcnt = (vflag) ? (lnum - lcnt) : lcnt;
            if( flag == 3) {
                if(hflag) printf("%s:",argv[optind]);
                printf("%d\n",lcnt);
            };
        };
    };
if(tflag) {
TIMER_STOP; ELAPSE_TIME;
if( Tflag ) { PRINT_STIME; putchar('\n');} else {
printf("\n>>simulation "); PRINT_TIME; };
};

    exit( (lcnt) ? 1 : 0 );

}

#define BLK_SIZE 4096
#define BLKSIZE 512
MYCHAR exec_buf[BLK_SIZE+2];
```

```
#define PRINT_HEADER {       \
if(hflag && fname ) printf("%s:",fname);    \
if(bflag) printf("%d:",bnum / BLKSIZE );    \
if(nflag) printf("%d:",lnum);       \
}

void execute( fname )
char *fname;
{

    FILE *f;
    int bnum;
    int cnt, cnt2, fd;
    MYCHAR *ptrend, cres, *ptr, *ptr1;

    lcnt = 0;

    if( fname == NULL ) {
        f = stdin;
    } else {
        if(  (f = fopen(fname, "r")) == NULL ) {
            fprintf("can not open %s\n", fname);
            return;
        };
    };

    fd = fileno(f);

    bnum = 1;
    lnum = lcnt = 0;
```

```
/* read first block */
cnt = read(fd, exec_buf + 1, BLK_SIZE);
bnum = cnt;
ptr = exec_buf;
ptrend = ptr + 1 + cnt;
*ptr = '\n';
*ptrend = '\0';


while( 1 ) {

    ptr1 = (MYCHAR *) memchr(ptr+1, '\n', cnt=(ptrend-ptr) -1);
    if( ptr1 == NULL ) {
        (void) memcpy( exec_buf+1,ptr+1, cnt);
        cnt2 = read( fd, exec_buf + cnt + 1, BLK_SIZE - cnt);
        bnum += cnt2;
        if( cnt + cnt2 <= 0 ) {
            if( f != stdin) fclose(f);
            return;
        };
        ptrend = exec_buf + cnt +cnt2 + 1;
        *ptrend = '\0';
        ptr1 = (MYCHAR *) memchr(exec_buf+cnt+1,'\n', cnt2);
        ptr = exec_buf;
        if( ptr1 == NULL ) ptr1 = ptrend;
    };

    cres = *++ptr1;
    *ptr1 = '\0';

    lnum++;
```

```
#ifdef USENFA
        if( csimu(ptr) ) {
#endif
#ifdef USEOL
        if( Uninit ) light_dfa_init();
        if( ol_simu(ptr) ) {
#endif
#ifdef USEDFA

        if( dsimu( ptr) ) {
#endif
        /* --- if start from here --- */

            lcnt++;
            if(flag == 1) {
                PRINT_HEADER;
                printf("%s", ptr+1);
            } else if( flag == 4) {
                printf("%s\n", fname);
                return;
            };
        } else {
             if( flag == 2 ) {
                /* mismatch this line */
                PRINT_HEADER;
                printf("%s", ptr+1);
            } else if( flag == 5 ) {
                printf("%s\n", fname);
                return;
            };
```

```
        } ;


        *ptr1-- = cres;

#ifdef DEBUG
printf("advance %d character %x %x\n", ptr1 - ptr, ptr, ptr1);
#endif
        ptr = ptr1;
    }; /* while */


}
```

B.2.5 parse.c

```
#include <stdio.h>
#include <stdlib.h>
#include "cnfa.h"

SET *join_sets();
EDGE *new_edge();

/* ---------------------------------------------------
 *         Parse Routine
 * --------------------------------------------------- */
void R(), E(), F(), T();
/*
 *  reg_parse() --
 *  an recursive-descendant parser for regular expressions,
```

```
 *  it builds CNFA while parsing.
 */

void reg_parse(reg)
MYCHAR *reg;
{
    EDGE *edge, *lazyh, *lazyt;
    SET  *Fset, *Iset;
    int Null;

    Iforest = Fforest = NULL;
    Null = FALSE;

    range_init();
    (void) R( &reg, &Fset, &Iset, &lazyh, &lazyt, &Null);

    if( *reg ) {
        error("parsing error -- reg_parse");
    };


    if( Fset ) {
#ifdef P_DEBUG
printf("link %x to Fforest\n", Fset);
#endif
        (void) mark_final(Fset);
        (void) append_fset(Fset, INSERT);
    };


    /* build start state */
```

```
    Start  = new_set();
    edge = new_edge(Start, Iset, DIRECT);
    Iset->pin = TRUE;
    edge->next = edge->back = NULL;
    Start->edge = edge;
    Start->edget = edge;
    Start->final = (Null) ? TRUE : FALSE;
    if( Iset ) (void) append_iset(Iset, APPEND);

    append_iset(Start, APPEND);
    append_fset(Start, INSERT);

}
void R( reg, fset, iset, lazyh, lazyt, null)
MYCHAR **reg;
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{

    SET *fset1, *iset1;
    EDGE *l1h, *l1t;
    int null1;


    (void) E( reg, fset, iset, lazyh, lazyt, null);

#ifdef NEWLINE_OR
    if( (**reg == '|' ) || (**reg == '\n') ) {
#else
    if( (**reg == '|' )  ) {
```

```
#endif
        (*reg)++;
        (void) R( reg, &fset1, &iset1, &l1h, &l1t, &null1);

        /* keep ordering -- */
        (void) add_edge(lazyh, lazyt, *fset, iset1);
        (void) add_edge(lazyh, lazyt, fset1, *iset);
        (void) cat_edges(lazyh, lazyt, &l1h, &l1t);

        *fset = join_sets(FSET, *fset, fset1);
        *iset = join_sets(ISET, *iset, iset1);
        *null |= null1;
    };
}
begin_of_E( ch )
MYCHAR ch;
{

#ifdef NEWLINE_OR
    return( ch && (! index("\n?*+|)", ch)) );
#else
    return( ch && (! index("?*+|)", ch)) );
#endif
}

void E( reg, fset, iset, lazyh, lazyt, null)
MYCHAR **reg;
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{
```

```
SET *fset1, *iset1;
EDGE *l1h, *l1t;
int null1;

(void) F( reg, fset, iset, lazyh, lazyt, null);

/* if current MYCHARacter starts an <E> */
if( begin_of_E( **reg) ) {

    register SET *set;

    /* now we have a concatenation */
    (void) E( reg, &fset1, &iset1, &l1h, &l1t, &null1);

    /* add real edge */
    if( (set = *fset) && iset1 ) {
        EDGE *edge, *edge2;

        /* add it at the end of list */
        edge = new_edge( set, iset1, DIRECT );

        edge->back = edge2 = set->edget;
        edge->next = NULL;
        if( set->edge ) {
            edge2->next = edge;
            set->edget = edge;
        } else {
            set->edge = set->edget = edge;
        };
```

```
        };


        /* concate lazy edges */
        if( ! *null ) l1h = l1t = NULL;
        if( ! null1 ) *lazyh = *lazyt = NULL;

        add_edge( lazyh, lazyt, fset1, *iset);
        cat_edges( lazyh, lazyt, &l1h, &l1t);


        /* Iset and Fset */
        if( *null ) *iset = join_sets( ISET, *iset, iset1);
        else {
            if( iset1 ) append_iset( iset1 , APPEND);
        };

        if( null1 ) *fset = join_sets( FSET, *fset, fset1);
        else {
            if( *fset ) append_fset( *fset, APPEND );
            *fset = fset1;
        };

        /* null */
        *null = *null && null1;

    };
}

void F( reg, fset, iset, lazyh, lazyt, null)
MYCHAR **reg;
```

```
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{


    int back = FALSE;


    (void) T( reg, fset, iset, lazyh, lazyt, null);

    while( **reg && index( "*?+", **reg ) ) {
        switch( *(*reg)++ ) {
            case '?': back = FALSE; *null = TRUE; break;
            case '*': back = TRUE; *null = TRUE; break;
            case '+': back = TRUE; break;
        };
    };

    if( back ) {
        EDGE *te, *edge, *e1;
        SET *fs;

        /* clear lazy edges */
        for(edge=*lazyh; edge != NULL; edge = e1) {
            fs = edge->fset;
            e1 = edge->next;


            /* now lazy edge [fs, is] become real */
            if( (te = fs->edget) != NULL ) {
```

```
                if(te->attrib == DIRECT) te = te->back;
                if( te == NULL ) {
                    /* last edge must be DIRECT
                        and only one edge */
                    fs->edge = edge;
                    edge->next = fs->edget;
                    (fs->edget)->back = edge;
                    edge->back = NULL;
                } else {
                    /* insert edge after te */
                    edge->next = te->next;
                    edge->back = te;
                    te->next = edge;
                    if( te == fs->edget ) fs->edget = edge;
                };
            } else {
                /* empty edge list */
                edge->next = edge->back = NULL;
                fs->edge = fs->edget = edge;
            };
        };

        *lazyh = *lazyt = NULL;
    };

}

void T( reg, fset, iset, lazyh, lazyt, null)
MYCHAR **reg;
SET **fset, **iset;
EDGE **lazyh, **lazyt;
```

```
int *null;
{

    MYCHAR ch;

#ifdef P_DEBUG
printf("T -- reg %x char %c %d\n", *reg, **reg, **reg);
#endif
    switch( ch = *(*reg)++ ) {
        case '(':
            (void) R( reg, fset, iset, lazyh, lazyt, null);
            if( *(*reg)++ != ')' ) {
                error("unmatch (" );
            };
            return;
        case '[':
            genrange(reg, fset, iset, lazyh, lazyt, null);
            return;
        case '.' :
            gendot(fset, iset, lazyh, lazyt, null);
            return;
        case '^':
        case '$': (void) genleaf(  '\n',
                fset, iset, lazyh, lazyt, null);
            return;
        case '\\' :
            switch( ch = *(*reg)++ ) {
                case 'e':
                    (void) geneps(
                    fset, iset, lazyh, lazyt, null);
                    return;
```

```
                    /* white space */
                case 's': (void) genleaf( ' ',
                    fset, iset, lazyh, lazyt, null);
                    return;
                case 't': (void) genleaf( '\t',
                    fset, iset, lazyh, lazyt, null);
                    return;
                case '0': (void) genleaf( '\0',
                    fset, iset, lazyh, lazyt, null);
                    return;
                case 'n': (void) genleaf(  '\n',
                    fset, iset, lazyh, lazyt, null);
                    return;
                default: (void) genleaf(  ch,
                    fset, iset, lazyh, lazyt, null);
                    return;
            };
        default: genleaf( ch, fset, iset, lazyh, lazyt, null);
            return;
    };
}

/*
 *  Build an unoptimized CNFA
 *
 *  Start state: Start
 *  F-forest: Fforest
 *  I-forest: Iforest
 */
SET *Start;
void reg_parse();
```

```
reg_compile(reg)
MYCHAR *reg;
{
    char Null;
    EDGE *edge;

    mem_init();

    Null = FALSE;
    reg_parse(reg, &Null);  /* build unoptimized CNFA */


    (void) pack();
    (void) path_compression();
}
```

B.2.6 fly.c

```
#include "nfa2dfa.c"
MYCHAR *ol_simu( str )
MYCHAR *str;
{
    register DFA *st, *st2;
    DFA *build_first();
    register MYCHAR ch, *rtn;

    /* take care of boundary condition */
    /*
    rtn = (Dfirst->final) ? (str) : (NULL) ;
    if( ! *str ) return( rtn );
```

```
    */

    if( Dfirst->final ) return(str);
    if( ! *str ) return(NULL);

    /* if( Uninit ) light_dfa_init();   */

    /* return initial state */
    /* st = build_first(); */
    st = Dfirst;
    if( ! st->build ) build_next(st);

    while(  ch = *str++) {
        /* in first state */
nextrun:
        if(  !(st2 = st->trans[ch] )) continue;
        if( st2->final ) return(str);
        st = st2;

        while( ch = *str++) {
next2:
            if(  ! (st2 = st->trans[ch] )) {
                if( st->build)  {
                    st = Dfirst;
                    goto nextrun;
                } else {
                    build_next(st);
                    goto next2;
                };
            };
            if( st2->final ) return(str);
```

```
            st = st2;
        };
        return(NULL);

    };
    return(NULL);


}
void build_follow(dptr)
DFA *dptr;
{
    struct mset *mptr, *m1, *m2;
    NSET *ns1, *ns, **nsptr, **nsptr2;
    MYCHAR keep[MAX_CHAR];
    int hashval;
    char final;
    int keep_idx, i;
    DFA *nstate;
    MYCHAR ch;

    if( dptr->build ) return;
    keep_idx = 0;
    for(i=0; i < Midx; i++) {
        ch = Mset_dirt[i];
        final = 0;
        hashval = 0;
        nsptr = Npp;
        m1 = NULL;
        for( mptr = Mset[ch]; mptr; mptr = m2) {
            m2 = mptr->next;
```

```
        ns = mptr->st;
        ns = ns->real;
        mptr->st = ns;
        if( ! ns->mark ) {

            *nsptr++ = ns;
            ns->mark = TRUE;
            hashval = (hashval + ((int) ns >> 3)) & 037777;
            final = final || ns->final;
            mptr->next = m1;
            m1 = mptr;
        };
    };
    Mset[ch] = m1;

    hashval = hashval % MAX_HASH;
    if( nsptr > Nsetpool + MAX_NSET_POOL )
        error("not enough nstate pool\n");
    if( nstate = find_state( hashval, nsptr - Npp) ) {
        UNMARK_NSET;
        dptr->trans[ch] = nstate;
        Mset[ch] = NULL;
    } else {
        UNMARK_NSET;
        if( nstate = new_dstate() ) {
            /* build a new state */
            nstate->state = Npp;
            nstate->final = final;
            nstate->cnt = nsptr - Npp;
            nstate->next = Hash[hashval];
            Hash[hashval] = nstate;
```

```
                dptr->trans[ch] = nstate;
                Mset[ch] = NULL;
                Npp = nsptr;
            } else {
                keep[keep_idx++] = ch;
            };
        };

    };

    if( keep_idx ) compact(dptr);
    for(i=0; i < keep_idx; i++ ) {
        ch = keep[i];
        final = 0;
        hashval = 0;
        nsptr = Npp;
        for( mptr = Mset[ch]; mptr; mptr = mptr->next) {
            *nsptr++ = ns = mptr->st;
            ns->mark = TRUE;
            hashval = (hashval + ((int) ns >> 3)) & 037777;
            final = final || ns->final;
        };

        hashval = hashval % MAX_HASH;
        if( nsptr > Nsetpool + MAX_NSET_POOL )
            error("not enough nstate pool\n");
        if( nstate = find_state( hashval, nsptr - Npp) ) {
            UNMARK_NSET;
            dptr->trans[ch] = nstate;
        } else {
             nstate = new_dstate();
```

```
            UNMARK_NSET;
            /* build a new state */
            nstate->state = Npp;
            nstate->final = final;
            nstate->next = Hash[hashval];
            nstate->cnt = nsptr-Npp;
            Npp = nsptr;
            Hash[hashval] = nstate;
            dptr->trans[ch] = nstate;
        };
        Mset[ch] = NULL;

    }

    dptr->build = TRUE;

}
void build_next( dptr )
DFA *dptr;
{
MYCHAR *ip, *ip2;
NSET **obase, **ohead, **otail;
NSET **dhead;
NSET *ns, *ns1, *ns2;
EDGE *edge, *edge2;
int firsttime;
    register MYCHAR cch;
    register struct mset *m;

    /* initial Mset data structure */
    Mptr = Mbase;
```

```
    Midx = 0;

    obase = dptr->state;
    ohead = obase + dptr->cnt;
    dhead = dirty;

    /* find adjacent state from those states in obase to ohead */
    /* and tch buffer */
    for( otail = obase, firsttime = TRUE; otail < ohead; ) {
        if( firsttime ) {
            ns = Nstart;
            firsttime = FALSE;
        } else {
            ns = *otail++;
        };

        edge2 = ns->edge;
        while( edge2 ) {
            if( edge2->mark )  break;
            edge2->mark = TRUE;
            ns1 = edge2->start;
            ns2 = edge2->stop;
/* --------------------------------------------------
 *          SMARK_AND_PICK;
 * -------------------------------------------------- */
{   NSET *nns1, *nns2;
    while( ns1->mnext < ns2 ) {
        if( ns1->mnext == ns1 ) {
            /* node is unmark */
            *dhead++ = ns1;
            ns1->mnext = ns2;
```

```
            switch( ns1->ch_size) {
                case 0:
                /* external node */
                nns1 = ns1->ext;
                nns2 = nns1 + ns1->ext_cnt;
                for(; nns1 < nns2; nns1++ ) {
                    if( nns1->ch_size == 1) {
    Mptr->st = nns1;
    if( ! (m = Mset[cch = (MYCHAR ) nns1->ch]) ) {
        Mset_dirt[Midx++] = cch;
    };
    Mptr->next = m;
    Mset[cch] = Mptr++;
                    } else {
                    ip = nns1->ch;
                    ip2 = ip + nns1->ch_size;
                    for( ; ip < ip2; ) {
    Mptr->st = nns1;
    if( ! (m = Mset[cch = *ip++]) ) {
        Mset_dirt[Midx++] = cch;
    };
    Mptr->next = m;
    Mset[cch] = Mptr++;
                    };
                    };
                };
                break;
            case 1:
    Mptr->st = ns1;
    if( ! (m = Mset[cch = (MYCHAR ) ns1->ch])) {
        Mset_dirt[Midx++] = cch;
```

```
    };
    Mptr->next = m;
    Mset[cch] = Mptr++;
                break;
                default:
                ip = ns1->ch;
                ip2 = ip + ns1->ch_size;
                for( ; ip < ip2; ) {
    Mptr->st = ns1;
    if( ! (m = Mset[cch = *ip++] )) {
        Mset_dirt[Midx++] = cch;
    };
    Mptr->next = m;
    Mset[cch] = Mptr++;
                };
                break;
            };
            ns1++;
        } else {
/* printf("ns1 %x mnext %x\n", ns1, ns1->mnext); */
            nns1 = ns1->mnext;
            ns1->mnext = ns2;
            ns1 = nns1;
        };
    };
}
/* ------------ end of SMARK_PICK ----------------- */
            edge2 = edge2->next;
        };
    };
```

```
    /* clear edge mark */
    for(otail=obase; otail < ohead; otail++) {
        edge = (*otail)->edge;
        for( ; edge ; edge = edge->next) {
            if( ! edge->mark ) break;
            edge->mark = FALSE;
        };
    };

    if(! Midx ) { dptr->build = TRUE; return; };

    for(otail= dirty; otail < dhead; otail++)
        (*otail)->mnext = *otail;
    Nstart->mnext = Nstart;


    build_follow(dptr);
}
```

B.2.7 nfa2dfa.c

```
/*
 *  This file contains routines for regular expression matching;
 *  we build DFA on-the-fly, and use it for acceptance test.
 */
int st_cntt = 0;
#include <stdio.h>
#include "cnfa.h"
```

```
#include "fly.h"
void build_next(), compact_hash(), compact_state();
void tmsd(), build_follow(), print_dstate();
void clear_mdirt();
DFA *build_first();
NSET **new_nstate();
int Uninit = TRUE;
int Sys_dirty = FALSE;
DFA *Dfirst;
DFA ST[MAX_DSIZE];      /* state pool --- keep at most */
NSET    *Nsetpool[MAX_NSET_POOL];   /* to store collection of
                       NFA states that --- */
DFA *Avail;         /* Available Dfa state pool */
NSET    **Npp;          /* Netpool pointer */
struct mset *Mset[MAX_CHAR];
struct mset Mbase[MAX_MBASE];
struct mset *Mptr = Mbase;
MYCHAR Mset_dirt[MAX_CHAR];
int Midx = 0;
DFA     *Hash[MAX_HASH];
void light_dfa_init()
{
    register DFA *sptr;

    Uninit = FALSE;
    /* link all the dfa states */
    Avail = sptr = ST;
    for( ;sptr < ST+(MAX_DSIZE-1 );sptr++) sptr->next = sptr+1;
    sptr->next = NULL;
    Npp = Nsetpool;
}
```

```
void dfa_init()
{
    register DFA *sptr;
    register DFA **hptr;
    register struct mset **mptr;

    Uninit = FALSE;
    /* link all the dfa states */
    Avail = sptr = ST;
    for(;sptr < ST+(MAX_DSIZE-1);sptr++) sptr->next = sptr+1;
    sptr->next = NULL;

    for(mptr=Mset; mptr < Mset+MAX_CHAR;) *mptr++ = NULL;

    /* initialize hash table */
    for(hptr=Hash;hptr < Hash+MAX_HASH;) *hptr++ = NULL;
    Npp = Nsetpool;

}
int nsc = 0;
DFA * new_dstate()
{
    register DFA *rtn, **sptr;

    st_cntt++;
    if( (rtn = Avail) == NULL ) {
            return( NULL );
    };
    Avail = rtn->next;

    /* clean transition table */
```

```
    if( Sys_dirty ) {
        rtn->final = FALSE;
        rtn->build = FALSE;
        rtn->reserve = FALSE;
        rtn->next = NULL;
        rtn->state = NULL;
        rtn->cnt = 0;
        for( sptr = rtn->trans; sptr < rtn->trans + MAX_CHAR; )
            *sptr++ = NULL;
    };
    return(rtn);
}
DFA *In_use[MAX_DSIZE];
int In_idx;
void compact( dptr )
DFA *dptr;
{
    register DFA **dp, **dp1, **dp2, *dp3;
    int i;

    /* mark all the reseved state */
    Sys_dirty = TRUE;
    dptr->reserve = TRUE;
    Dfirst->reserve = TRUE;
    In_idx = 0;
    In_use[In_idx++] = Dfirst;
    In_use[In_idx++] = dptr;
    if( dptr == Dfirst ) {
        printf("something funny here\n");
    };
    for(i=0; i < 2; i++) {
```

```
        dptr = In_use[i];
        for( dp = dptr->trans, dp2 = dptr->trans + MAX_CHAR ;
                            dp < dp2; ) {
            if( dp3 = *dp++ ) {
                if(  dp3->reserve ) continue;
                dp3->reserve = TRUE;
                In_use[In_idx++] = dp3;
                if( ! dp3->build ) continue;
                dp3->build = FALSE;
                for( dp1= dp3->trans; dp1 < dp3->trans +
                        MAX_CHAR; ) {
                    *dp1++ = NULL;
                };
            };
        };
    };
    (void) compact_hash();
    (void) compact_state();
}
void compact_hash()
{
    register DFA **hp, **hp2;
    register DFA *dp, *dp2, *dp3;

    for(  hp = Hash, hp2 = hp + MAX_HASH; hp < hp2; hp++ ) {
        if( dp = *hp ) {
            for( dp2 = NULL; dp ; dp = dp3 ) {
                dp3 = dp->next;
                if( dp->reserve ) {
                    dp->reserve = FALSE;
                    dp->next = dp2;
```

```
                dp2 = dp;
            } else {
                dp->next = Avail;
                Avail = dp;
            };

        };
        *hp = dp2;
    };
  };
}
void print_dstate( dptr )
DFA *dptr;
{
    NSET **nptr;
    int i, cnt;

    printf("dfa state %x final %d:", dptr, dptr->final);
    cnt = dptr->cnt;
    nptr = dptr->state;
    for(i=0; i < cnt; i++) printf( " %x", *nptr++);
    printf("\n");
}
DFA * find_state( hashval, cnt)
int hashval, cnt;
{
    register DFA  *dptr;
    register NSET **nptr, **nptr2;

    for( dptr = Hash[hashval]; dptr; dptr = dptr->next ) {
        if( dptr->cnt != cnt) continue;
```

```
        for(nptr=dptr->state,nptr2=nptr+cnt;nptr<nptr2;){
            if( ! (*nptr++)->mark ) goto nxt;
        };
        return( dptr );
    nxt:    ;
    };
    return( NULL );
}
/* sort In_use array accroding to state pointer */
void in_sort()
{
    register int i,j, min;
    DFA *tmp;

    /* simple selection sort */
    for(i =0; i < In_idx - 1; i++) {
        min = i;
        for(j = i+1; j < In_idx ; j++) {
            if( In_use[j]->state < In_use[min]->state ) {
                min = j;
            };
        };
        tmp = In_use[min];
        In_use[min] = In_use[i];
        In_use[i] = tmp;
    };
}
/*
 *  compact the Nsetpool array
 */
void compact_state()
```

```
{
    register i;
    register NSET **sptr1, **sptr2, **sptr3;

    (void) in_sort();
    sptr1 = Nsetpool;
    for(i=0; i < In_idx; i++) {
        sptr2 = In_use[i]->state;
        sptr3 = sptr2 + In_use[i]->cnt;
        In_use[i]->state = sptr1;
        while( sptr2 < sptr3 ) {
            *sptr1++ = *sptr2++;
        };
    };
    Npp = sptr1;
}
NSET * dirty [ MAX_NFA ];
DFA * build_first()
{
    register MYCHAR cch;
    register struct mset *m;
    DFA *first;
    MYCHAR *ip, *ip2;
    NSET  *ns1, *ns2, *ns3;

    Nstart->mark = TRUE;
    if( first = find_state(0,1) ) {
        return(first);
    };
    Nstart->mark = FALSE;
    /* initial Mset data structure */
```

```
    Mptr = Mbase;
    Midx = 0;

    first = new_dstate();
    first->final = Nstart->final;
    first->cnt = 1;
    first->build = FALSE;
    first->state = Npp;
    *Npp++ = Nstart;
    Hash[0] = first;
    return( first );
}
#define UNMARK_NSET {                    \
for( nsptr2 = Npp; nsptr2 < nsptr; nsptr2++ )   \
(*nsptr2)->mark = FALSE;                 \
}
```

B.2.8 pack.c

```
/*
 *  this file contains routines for packing transformation
 */
/*
    packing algorithm
    pack( Fset ) = Iset_promote( Fset )     if Fset is a leaf
    pack( F1 + F2 ) = Fset_promote( pack(F1), pack(F2) )
*/
#include <stdio.h>
#include <stdlib.h>
#include "cnfa.h"
```

```
int Fclass_idx;
SET *No_edge_rep = NULL;
SET *AFset, *AIset;
void print_edge(), del_head(), del_tail(), pack_set();
void comp_Fclass(), pin_Iset();
void group_no_edge( );
void pack( )
{
    register SET *Fset, *tset, *root;
    struct forest *ff;

    /* allocate F-class queue */
    Fclass_idx = 0;
    ff = Fforest;
    pack_set( ff->set);
    comp_Fclass(ff->set, ff->set);

    ff = Fforest->next; /* the real Fset of reg exp */
    AFset = ff->set;
    pack_set( AFset);
    comp_Fclass(AFset, AFset);
    pin_Iset(AFset->edge);
    (void) group_no_edge(AFset);
    ff = ff->next;      /* the only left over Fset */
    if( ff != NULL ) {
        pack_set( ff->set );
        comp_Fclass(ff->set, ff->set);
        pin_Iset(ff->set->edge);
        AFset = join_sets(FSET, AFset, ff->set);
    };
```

```
}
void pack_set( root )
SET *root;
{
    register EDGE *edge1, *edge2, *edge3;
    register SET *parent, *root2;
    EDGE *edge4, *edge_save;

    if( root->left ) {
        /* internal node */
        /* step 0 -- pack subset -- */
        pack_set(root->left);
        pack_set(root->right);

        /* ============================ */
        /*  Fset Promotion      */
        /* ============================ */
        edge3 = edge4 = NULL;
        edge1 = root->left->edget;
        edge2 = root->right->edget;
        if( (edge1 == NULL) || (edge2 == NULL) ) goto nofp;
        if(edge1->iset == edge2->iset ) {
            del_tail(root->left);
            del_tail(root->right);
            edge3 = edge4 = edge1;
        };
        edge1 = root->left->edge;
        edge2 = root->right->edge;
        if( (edge1 == NULL) || (edge2 == NULL) ) {
            if( edge3 != NULL ) goto fp; else goto nofp;
        };
```

```
        if(edge1->iset == edge2->iset ) {
            del_head(root->left);
            del_head(root->right);
            if( edge3 ) {
                edge3 = edge1;
                edge3->next = edge4;
                edge4->back = edge3;
            } else {
                edge3 = edge4 = edge1;
            };
        };
fp:
        if( edge3 == NULL ) goto nofp;
        if( edge1 = root->edget ) {
            if( edge3->iset < edge1->iset ) {
                /* one step ahead of tail */
                edge4->next = edge1;
                edge3->back = edge1->back;
                edge1->back = edge4;
                if(edge1 == root->edge) root->edge = edge3;
            } else {
                /* insert at tail */
                edge1->next = edge3;
                edge3->back = edge1;
                root->edget = edge4;
            };
        } else {
            root->edge = edge3;
            root->edget = edge4;
        };
```

```
nofp:
        /* compute F-classes */
        if( edge1 = (root2 = root->right)->edge ) {
            comp_Fclass(root2, root2);
            pin_Iset(edge1);
        };
        if( edge1 = (root2 = root->left)->edge ) {
            comp_Fclass(root2, root2);
            pin_Iset(edge1);
        };
    } else {
        /* looks as if has f-promotion */
        edge3 = edge4 = root->edget;
    };

    /* ============================ */
    /*  Iset Promotion      */
    /* ============================ */
    if( edge3 == NULL ) goto pack_ret;
    edge_save = ( edge4 ) ? edge4->next : NULL;
    /* start I-promotion */
    edge1 = edge3;
    edge2 = edge1->back;
    if( edge2 == NULL ) goto test_save;
    if( (parent = edge1->iset->iparent) == NULL) goto pack_ret;
    if( edge2->iset->iparent != parent ) goto pack_ret;

    /* now edge1 and edge2 has the same parent */
    do {
        /* merge edge1 and edge2 */
        edge1->iset = parent;
```

```
        edge1->back = edge2 = edge2->back;
        if( edge2 != NULL ) {
            edge2->next = edge1;
        } else {
            /* reach the end */
            root->edge = edge1;
            goto test_save;
        };
        if( (parent = parent->iparent) == NULL ) break;
        if( edge2->iset->iparent != parent ) break;
    } while(1);
    goto pack_ret;
test_save:
    if( edge_save == NULL ) goto pack_ret;
    parent = edge1->iset->iparent;
    if( parent != edge_save->iset->iparent ) goto pack_ret;
    root->edge = edge1->next;
    root->edge->back = NULL;
    edge_save->iset = parent;
    goto pack_ret;
pack_ret:
        return;
}
void print_edge( root )
SET *root;
{
    EDGE *e;

    for(e=root->edge; e ; e = e->next) printf( " %x(%x %d)",
            e, e->iset, e->attrib);
    putchar('\n');
```

```
}
void del_head( set )
SET *set;
{
    register EDGE *edge1, *edge2;

    if( ! set ) return;
    if( ! (edge1 = set->edge) ) return;
    set->edge = edge2 = edge1->next;
    edge1->next = edge1->back = NULL;
    if( edge2 == NULL ) {
            set->edget = NULL;
    } else {
        edge2->back = NULL;
    };
}
void del_tail( set )
SET *set;
{
    register EDGE *edge1, *edge2;

    if( ! set ) return;
    if( ! (edge1 = set->edget) ) return;
    set->edget = edge2 = edge1->back;
    edge1->next = edge1->back = NULL;
    if( edge2 == NULL ) {
            set->edge = NULL;
    } else {
        edge2->next= NULL;
    };
}
```

```
void comp_Fclass( root, class )
SET *root, *class;
{
    if( ! root ) return;
    if( root->Fclass == NULL) {
        if( root->left ) {
            comp_Fclass(root->left, class);
            comp_Fclass(root->right, class);
        } else {
            /* This is the way we count number of Fclasses */
            if( class->Fclass == NULL ) {
                class->Fclass = class;
                Fclass_idx++;
            };
        };
        root->Fclass = class;
    };
}
void pin_Iset( edge )
EDGE *edge;
{
    register EDGE *edge1;

    edge1 = edge;
    while( edge1 ) {
        edge1->iset->pin = TRUE;
        edge1 = edge1->next;
    };
}
void group_no_edge( root )
SET *root;
```

```
{
    if( ! root ) return;
    if( root->edge ) return;
    if( root->left ) {
        group_no_edge(root->left);
        group_no_edge(root->right);
    } else {
        if( No_edge_rep ) {
            root->Fclass = No_edge_rep;
        } else {
            root->Fclass = No_edge_rep = root;
        };
    }
}
```

B.2.9 path.c

```
/*
 *  This file path.c contains routines used in path compression
 *  transformation.
 */
#include <stdio.h>
#include <stdlib.h>
#include "cnfa.h"

void comp_Iclass(), find_class(), fix_edges();
void nch_init(), fix_ss(), build_nch(), layout();
NSET *build_ext(), *new_nset();

struct set **Fclass;
```

```
int Fclass_idx;
struct nset *Nstart;

/*
 *  CNFA
 */
NSET    *NFset,     /* redundant */
    *NIset,
    *NFforest;  /* redundant */
NSET *NIforest = NULL;  /* To link new I-forest */

int NIset_size;
void path_compression()
{
    register SET *AIset, *tset, *root;
    register NSET *bptr, *base;

    /* Fclass -- as temp base map array */
    Fclass = (SET **) xmalloc( sizeof( *Fclass) * Fclass_idx);

    nch_init();
    AIset = Iforest->set;
    AIset->pin = TRUE;  /* artifical made it */
    Start->pin = TRUE;
    comp_Iclass( AIset, AIset );

    find_class(AIset);

    NIforest = base = new_nset( AIset->internal + 1);
    base->size = AIset->internal;
    (base + AIset->internal)->mnext = base + AIset->internal ;
```

```
    layout(AIset, base);

    Nstart = Start->Frep_addr;

    /* fix_edges(Fforest->set, NULL); */
    fix_edges(AFset, NULL);
    fix_edges(Start, NULL);

    /* patch start and stop fields of NSET records */
    /* (void) fix_ss(NIforest, NIforest->size); */
}

void comp_Iclass(root, class)
SET *root, *class;
{
    register SET *nclass, *root2;

    root->Iclass = NULL;
    nclass = (root->pin) ? root : class;
    if( root2 = root->left ) {
        comp_Iclass(root2, nclass);
        comp_Iclass(root->right, nclass);
    } else {
        /* Here we might create a self loop at leaf */
        root->Inext= nclass->Iclass;
        nclass->Iclass = root;
    };
}
void find_class( root )
SET *root;
{
```

```
register SET *root1, *root2;
register SET *Ic;
register int idx = 0;
SET *Fc, *Rep, *cl, *rep, *Ic2;
int i;

if( root->left ) {
    /* internal node */
    if( Ic = root->Iclass ) {
        /* Fclass[] contains all the F-classes */
        /* Fclass->class point to a list of Iset which */
        /* is linked by Iclass field with Isets */
        /* orginally Iclass field was used to link all */
        /* the member in the same I-class */
        /* -- see comp_Iclass() */
            for( ; Ic ; Ic = Ic2) {
            /* Ic must be a leaf node */
            Ic2 = Ic->Inext;
            Fc = Ic->Fclass;
            /* Fc->class must be NULL when first met */
            if( ! Fc->class){
                Fclass[idx++] = Fc;
            };
            Ic->Inext = Fc->class;
            Fc->class = Ic;
            }; /* for Ic */

        if( idx == 1 ) {
            /* only one class in a I-class */
            cl = Fclass[0]->class;
            rep = root;
```

```
        /* seems to be redundant code, Yes, it is */
        while(cl != NULL) {
            cl->rep = rep;
            cl = cl->Inext;
        };

        build_nch(rep, Fclass[0]->class);

        /* create a loop intensionally */
        /* here we have non-NULL Iclass */
        /* but 0 external node */
        root->Iclass = root;
        root->external = 0;
        root->internal = 1;
        Fclass[0]->class = NULL;
        root->Fclass = Fclass[0];
        root->final = Fclass[0]->final; /* important */
    } else {
        /* two or more classes */
        Rep = NULL;
        /* seems to be redundant code, Yes, it is */
        for(i=0; i < idx; i++) {
            /* arbitary chose one in each class
            as rep */
            rep = cl = Fclass[i]->class;
            do {
                cl->rep = rep;
                cl = cl->Inext;

            } while( cl );
```

```
                build_nch( rep, Fclass[i]->class);

                /* link all the class reps */
                rep->Inext= Rep;
                Rep = rep;

                Fclass[i]->class = NULL;
            };
            root->internal = 1;
            root->external = idx;
            root->Iclass = Rep;
        };
    } else {
        /* no Iclass */
        ;
    };
    find_class(root1 = root->left);
    find_class(root2 = root->right);
    root->internal += (root1->internal + root2->internal);
} else {
    /* leaf node */
    if( root->Iclass ) {
        root->internal = 1;
        root->external = 0;
        root->Iclass = NULL;    /* since pointed to iself */
        build_nch( root, root);
        root->Iclass = root;    /* recover */
    } else {
        ;
    };
};
```

```
}
void layout( root, base)
SET *root;
NSET *base;
{
    register SET *root2;
    register SET *class;
    register NSET *nbase;

    if( root->internal == 0 ) return;   /* nothing to do */

    /* use dfs to assign storage */
    nbase = base;
    if( root2 = root->left) {
        (void) layout(root2, nbase);
        nbase += root2->internal;
        root2= root->right;
        (void) layout(root2, nbase);
        nbase += root2->internal;
    };

    if( class = root->Iclass ) {
        /* we consume a internal node, and create external set
            if necessary */
        if( root->external ) {
            nbase->ext = build_ext( root->external, class );
            nbase->ext_cnt = root->external;
            nbase->ch = NULL;
            nbase->ch_size = 0;
            nbase->mnext = nbase;
            nbase->edge = NULL;
```

```
            nbase->final = root->final;
            /* nbase->start = nbase->stop = NULL; */
            nbase->mark = FALSE;
            nbase->real = nbase;
        } else {
            EDGE *edge1;
            /* only one class and class rep is itself */
            nbase->ext_cnt = 0;
            nbase->ext = NULL;
            nbase->ch = root->nch;
            nbase->ch_size = root->ch_size;
            nbase->mnext = nbase;
            nbase->final = root->final;

            nbase->mark = FALSE;
            nbase->edge = class->Fclass->edge;
            if( class->Fclass->Frep_addr == NULL ) {
                class->Fclass->Frep_addr = nbase;
                nbase->real = nbase;
            } else {
                nbase->real = class->Fclass->Frep_addr;
            };
        };

        nbase++;
    };

    if( root->pin ) {
        root->start = base;
        root->stop = nbase;
    } else {
```

```
        root->start = root->stop = NULL;    /* redundant */
    };
}
void fix_edges( root , anc_edge )
SET *root;
EDGE *anc_edge;
{
    register EDGE *edge, *edge2, *nanc_edge;
    register SET *iset;

    if( ! root ) return;
    nanc_edge = anc_edge;
    if( edge = root->edge ) {
        nanc_edge = edge;
        do {
            iset = edge->iset;
            edge->start = iset->start;
            edge->stop = iset->stop;
            edge->mark = FALSE;
            edge2 = edge;

            edge = edge->next;
        } while( edge );

        root->edget->next = anc_edge;
        edge2->next = anc_edge;
    };

    fix_edges(root->left, nanc_edge);
    fix_edges(root->right, nanc_edge);
}
```

```
MYCHAR nch_map[MAX_CHAR];
void nch_init()
{
    register MYCHAR *ptr;

    for( ptr=nch_map; ptr < &nch_map[MAX_CHAR]; )
        *ptr++ = FALSE;
}
void build_nch( rep, class)
SET *rep, *class;
{

    register SET *cp;
    register MYCHAR ch, *ptr, *ptr2;
    register int idx = 0, i;
    MYCHAR chq[MAX_CHAR];

    cp = class;
    do {
        ptr = cp->ch;
        ptr2 = ptr + cp->ccnt;
        while( ptr < ptr2 ) {
            if( ! nch_map[ch=*ptr++] ) {
                nch_map[ch] = TRUE;
                chq[idx++] = ch;
            };
        };

        cp = cp->Inext;
    } while( cp );
```

```
    if( idx > 1 ) {
        ptr = rep->nch = (MYCHAR *) new_ch( idx );
        for(i=0, ptr2 = ptr + idx; ptr < ptr2; ) {
            ch = chq[i++];
            *ptr++ = ch;
            nch_map[ch] = FALSE;
        };
    } else {
        /* only one character */
        ch = chq[0];
        rep->nch = (MYCHAR *) ch;
        nch_map[ch] = FALSE;
    };

    rep->ch_size = idx;

}
struct nset * build_ext( cnt, class )
int cnt;
struct set *class;
{
    struct nset *rtn, *rtn1;

    rtn = rtn1 = new_nset(cnt);
    for( ; class != NULL ; class = class->Inext) {
        rtn1->ext_cnt = 0;
        rtn1->ext = NULL;
        rtn1->ch = class->nch;
        rtn1->ch_size = class->ch_size;
        rtn1->mark = FALSE;
        rtn1->final = class->final;
```

```
        rtn1->size = 0;
        if( class->Fclass->Frep_addr == NULL ) {
            class->Fclass->Frep_addr = rtn1;
            rtn1->real = rtn1;
        } else {
            rtn1->real = class->Fclass->Frep_addr;
        };
        rtn1->mnext = rtn1;
        rtn1->edge = class->Fclass->edge;
        rtn1++;
    };
    return(rtn);
}


{
    int i;
    MYCHAR *p;
    EDGE *e;

    printf("SET %x l %x r %x, ip %x fp %x ccnt %d \n\tedge:",
    r, r->left, r->right, r->iparent, r->fparent, r->ccnt);
    for(e=r->edge; e ; e = e->next) printf( " %x(%x %d)",
            e, e->iset, e->attrib);
    printf("\n\tch:");
    for(p=r->ch, i=0; i < r->ccnt; i++) putchar(*p++);

    printf(
    "\n\tFc %x Ic %x In %x Cl %x rep %x pin %d ext %d int %d\n",
    r->Fclass, r->Iclass, r->Inext, r->class, r->rep,
         r->pin, r->external, r->internal);
```

```
    printf("\tstart %x stop %x chszie %d final %d\n\tnch:",
        r->start, r->stop, r->ch_size, r->final);
    if( r->ch_size == 1 ) putchar( (MYCHAR) r->nch );
    else for(p=r->nch, i=0; i < r->ch_size; i++) putchar(*p++);

    printf("\n\n");

    if(r->left) {
        print_tree(r->left);
        print_tree(r->right);
    };
}
#endif

print_nset( start, cnt, type)
NSET *start;
int cnt, type;
{
    MYCHAR *c;
    EDGE *e;
    NSET *b;

    if( type ) printf("internal : cnt %d \n", cnt);
    else    printf("External : cnt %d \n", cnt);

    for(b=start; b < start + cnt; b++) {
        printf("%x fin %d sz %d ext %x ext_cnt %d mx %x rl %x\n",
        b,b->final,b->ch_size,b->ext,b->ext_cnt,b->mnext,b->real);
        printf("\tch:");
        if( b->ch_size == 1 ) putchar( (MYCHAR) b->ch );
```

```
        elsefor(c=b->ch;c<b->ch+b->ch_size;)putchar(*c++);
        printf("\n\tedge: ");
        for(e=b->edge; e ; e=e->next)
            printf("<%x, %x> ",e->start, e->stop);
        printf("\n");
        if(b->ext_cnt) print_nset(b->ext, b->ext_cnt, 0);
    };
}
print_nnset(b)
NSET *b;
{
    printf("addr %x ch %x,  ext %x ",
    b, b->ch, b->ext);
    printf(" ed %x mn %x r %x sz %d\n",
    b->edge, b->mnext,b->real,b->size);
}
```

B.2.10 uty.c

```
/*
 *  This file contains some utiltiy routines
 */
#include <stdio.h>
#include <stdlib.h>
#include "cnfa.h"
/* dynamic memory allocation  */
struct set_pool *Set_pool;
struct nset_pool *Nset_pool;
struct edge_pool *Edge_pool;
struct ch_pool *Ch_pool;
```

```
#define SPSIZE 128  /* set pool size */
#define NPSIZE 128  /* set pool size */
#define EPSIZE 128  /* edge pool size */
#define CPSIZE 1024
#define max(x,y)    ((x > y) ? (x) : (y))
mmax(x,y)
{
    return( ((x) > (y)) ? (x) : (y));
}
void mem_init()
{
    Set_pool = NULL;
    Edge_pool = NULL;
    Ch_pool = NULL;
}
error(msg)
char *msg;
{
    fprintf(stderr, "%s\n", msg);
    exit(0);
}
char *xmalloc(size)
int size;
{
    char *rtn;

    if( (rtn = malloc(size)) == NULL)
        error("no memory avaiable");
    return(rtn);
}
SET *new_set()
```

```
{
    struct set_pool *tmp;
    SET *rtn;

    if( (Set_pool) && (Set_pool->cnt--) ) {
        rtn = (Set_pool->pool)++ ;
    } else {
        tmp = (struct set_pool *) xmalloc( sizeof(*tmp));
        tmp->next = Set_pool;
        tmp->pool = (SET *) xmalloc(sizeof( SET ) * SPSIZE );
        tmp->cnt = SPSIZE - 1;
        Set_pool = tmp;
        rtn = (Set_pool->pool)++ ;
    };
    rtn->class = NULL;
    rtn->final = FALSE;
    rtn->edge = rtn->edget = NULL;
    /* add for protection apr 14, 92*/
    rtn->internal = rtn->external = 0;
    /* end add for protection apr 14, 92*/
    rtn->fparent = rtn->iparent = NULL;
    rtn->left = rtn->right = NULL;
    rtn->ch = rtn->nch = NULL;
    rtn->ccnt = rtn->ch_size = 0;
    rtn->start = rtn->stop = rtn->nset = rtn->Frep_addr = NULL;
    rtn->rep = rtn->class = NULL;
    rtn->Iclass = rtn->Inext = NULL;
    rtn->Fclass = NULL;
    return(rtn);
}
EDGE *new_edge(fset, iset, kind)
```

```
SET *fset, *iset;
int kind;
{
    struct edge_pool *tmp;
    EDGE *rtn;

    if( (Edge_pool) && (Edge_pool->cnt--) ) {
        rtn = (Edge_pool->pool)++ ;
    } else {
        tmp = (struct edge_pool *) xmalloc( sizeof(*tmp));
        tmp->next = Edge_pool;
        tmp->pool = (EDGE *) xmalloc(sizeof( SET ) * SPSIZE );
        tmp->cnt = SPSIZE - 1;
        Edge_pool = tmp;
        rtn = (Edge_pool->pool)++ ;
    };
    rtn->fset = fset;
    rtn->iset = iset;
    rtn->attrib = kind;
    rtn->next = rtn->back = NULL;
    rtn->anc = NULL;
    return(rtn);
}
MYCHAR *new_ch(size)
int size;
{
    struct ch_pool *tmp;
    MYCHAR *rtn;

    if( (Ch_pool) && (Ch_pool->cnt-size >= 0) ) {
        rtn = Ch_pool->pool;
```

```
        Ch_pool->pool += size;
        Ch_pool->cnt -= size;
    } else {
        tmp = (struct ch_pool *) xmalloc( sizeof(*tmp) );
        tmp->next = Ch_pool;
        tmp->pool = (MYCHAR *) xmalloc(CPSIZE * sizeof( MYCHAR) );
        tmp->cnt = CPSIZE - size;
        Ch_pool = tmp;
        rtn = Ch_pool->pool;
        Ch_pool->pool += size;
    };
    return(rtn);
}
struct nset *new_nset(size)
int size;
{
    struct nset_pool *tmp;
    struct nset *rtmp, *rtn;

    if( (Nset_pool) && (Nset_pool->cnt-size >= 0) ) {
        rtn = Nset_pool->pool;
        Nset_pool->pool += size;
        Nset_pool->cnt -= size;
    } else {
        tmp = (struct nset_pool *) xmalloc( sizeof(*tmp) );
        tmp->next = Nset_pool;
        tmp->pool = (NSET *) xmalloc(max(NPSIZE,size) *
                sizeof( *rtn ) );
        tmp->cnt = max(NPSIZE,size) - size;
        Nset_pool = tmp;
        rtn = Nset_pool->pool;
```

```
        Nset_pool->pool += size;
    };
    /* add for protection -- apr 14, 92 */
    for(rtmp=rtn; rtmp < rtn + size; rtmp++) {
        rtmp->ext_cnt = rtmp->size = rtmp->ch_size = 0;
        rtmp->mnext = rtmp;
    };
    /* end add for protection -- apr 14, 92 */
    return(rtn);
}
void free_set( head )
struct set_pool *head;
{
    if( head == NULL ) return;
    free_set(head->next);
    free(head->pool);
    free(head);
}
void free_edge( head )
struct edge_pool *head;
{
    if( head == NULL ) return;
    free_edge(head->next);
    free(head->pool);
    free(head);
}
void free_ch( head )
struct ch_pool *head;
{
    if( head == NULL ) return;
    free_ch(head->next);
```

```
    free(head->pool);
    free(head);
}
/*
 *  join two set
 */
SET * join_sets( type, set1, set2)
int type;
SET *set1, *set2;
{
    SET *root;

    if( set1 == NULL ) return(set2);
    if( set2 == NULL ) return(set1);
    root = new_set();
    root->left = set1;
    root->right = set2;
    root->ch = NULL;
    root->ccnt = 0;
    root->edge = root->edget = NULL;
    /* parent pointer */
    if( type == FSET ) {
        set1->fparent = set2->fparent = root;
    } else {
        set1->iparent = set2->iparent = root;
    };
    return( root );
}
/*
 *  Concatenate two line [h1, t1] and [h2, t2] into [h1, t1]
 *  -- for lazy edges
```

```
 */
void cat_edges( h1, t1, h2, t2)
EDGE **h1, **h2, **t1, **t2;
{
    register EDGE *tail1, *head2;

    if( (tail1 = *h1) == NULL ) {
        *h1 = *h2;
        *t1 = *t2;
    } else if( head2 = *h2 ) {
        (*t1)->next = head2;
        (*h2)->back = tail1;
        *t1 = *t2;
    };
}
void add_edge( head, tail, fset, iset)
EDGE **head, **tail;
SET *fset, *iset;
{
    register EDGE *rtail, *edge;

    if( fset == NULL || iset == NULL ) return;
    edge = new_edge(fset, iset, LAZY);
    if( rtail= *tail ) {
        edge->next = NULL;
        edge->back = rtail;
        rtail->next = edge;
        *tail = edge;
    } else {
        edge->next = edge->back = NULL;
        *head = *tail = edge;
```

```
    };
}
/*
 *  treat regular expression [^a-b]
 */
void geneps( fset, iset, lazyh, lazyt, null )
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{
    *fset = *iset = NULL;
    *lazyh = *lazyt = NULL;
    *null = TRUE;
}
void genleaf( ch, fset, iset, lazyh, lazyt, null )
MYCHAR ch;
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{
    register MYCHAR *cptr;
    register SET *set;

    *fset = *iset = set = new_set( );
    set->ch = cptr = new_ch(1); /* allocate one character */
    *cptr = ch;
    set->ccnt = 1;
    set->edge = set->edget = NULL;
    set->fparent = set->iparent = NULL;
    *lazyh = *lazyt = new_edge( set, set , LAZY);
    *null = FALSE;
```

```
}
MYCHAR rmap[MAX_CHAR];  /* base array for genrange() */
void range_init( )
{
    register MYCHAR *ptr;

    for(ptr=rmap; ptr < rmap + MAX_CHAR; ) *ptr++ = FALSE;
}
/*
 *  Remark: We do not perform error checking in this routine
 */
void genrange(reg, fset, iset, lazyh, lazyt, null)
MYCHAR **reg;
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{
#define MAX_RANGE 20
    register MYCHAR *ptr;
    register MYCHAR ch;
    register i, idx;
    register SET *set;
    MYCHAR range[MAX_RANGE][2];
    int cnt, negate;
    int ich;

    idx = 0;
    cnt = 0;
    negate = FALSE;
    ptr = *reg;
    if( *ptr == '^' ) {
```

```
        ptr++; negate = TRUE;
    };
    for( ; *ptr != ']'; ) {
        range[idx][0] = *ptr++;
        if( *ptr++ == '-' ) {
            range[idx++][1] = *ptr++;
        } else {
            ptr--;
            range[idx++][1] = NULL;
        };
    };
    /* set reg */
    *reg = ++ptr; cnt = 0;
    for(i=0; i < idx; i++) {
        if( range[i][1] ) {
            /* range pattern 'a-b' */
            for(ich= (int) range[i][0]; ich <= (int) range[i][1];
                ich++)  {
                if( ! rmap[ich] ) {
                    rmap[ich] = TRUE;
                    cnt++;
                };
        };
        } else {
            /* range pattern 'a' */
            if( ! rmap[ (ch = range[i][0]) ] ) {
                rmap[ch] = TRUE;
                cnt++;
            };
        };
    };
```

```
    *fset = *iset = set = new_set();
    set->ccnt = cnt = (negate ) ? (MAX_CHAR  - cnt) : cnt;
    if( cnt == 0 ) error("no character");
    set->ch = ptr = (MYCHAR *) new_ch(cnt);

    /* a stupid algorithm, but keep it simple for now */
    if( negate ) {
        for(ich=0; ich < MAX_CHAR; ich++) {
            if( rmap[ich] ) rmap[ich] = FALSE;
            else {
                *ptr++ = (MYCHAR ) ich;
            };
        };
    } else {
        for(ich=0; ich < MAX_CHAR; ich++) {
            if( rmap[ich] ) {
                rmap[ich] = FALSE;
                *ptr++ = (MYCHAR) ich;
            };
        };
    };
    *lazyh = *lazyt  = new_edge(set, set, LAZY);
    set->left = set->right = NULL;
    set->edge = set->edget = NULL;
    set->fparent = set->iparent = NULL;
    *null = FALSE;
}
/*
 *  accepts everything except NEWLINE
 */
void gendot(fset, iset, lazyh, lazyt, null)
```

```
SET **fset, **iset;
EDGE **lazyh, **lazyt;
int *null;
{
    register MYCHAR *ptr, ch;
    register int ich;
    register SET *set;

    *fset = *iset = set = new_set();
    set->ccnt = MAX_CHAR - 1;
    set->ch = ptr = (MYCHAR *) xmalloc( MAX_CHAR -1);
    for(ich=0; ich < '\n'; ich++) *ptr++ = (MYCHAR )ich;
    for(ich='\n'+1;ich<MAX_CHAR;ich++)*ptr++=(MYCHAR)ich;

    *lazyh = *lazyt = new_edge(set, set, LAZY);
    set->left = set->right = NULL;
    set->edge = set->edget = NULL;
    set->fparent = set->iparent = NULL;
    *null = FALSE;
}
void mark_final( set )
SET *set;
{
    if( ! set ) return;
    set->final = TRUE;
    mark_final(set->left);
    mark_final(set->right);
}
/*
 *  To build a compressed NFA
 */
```

```
struct forest *Iforest, *Fforest;
void append_iset( iset, kind )
SET *iset;
{
    SET *set;
    struct forest *tmp;

    if( (kind == INSERT) || (Iforest == NULL )) {
        tmp = (struct forest *) xmalloc( sizeof(*tmp));
        tmp->set = iset;
        tmp->next = Iforest;
        Iforest = tmp;
    } else {
        /* link with previous fset */
        set = new_set();
        set->edge = set->edget = NULL;
        set->iparent = NULL;
        iset->iparent = Iforest->set->iparent = set;
        set->left = iset;
        set->right = Iforest->set;
        Iforest->set = set;
    };
}
void append_fset( fset , kind)
SET *fset;
{
    SET *set;
    struct forest *tmp;

    if( (kind == INSERT ) || (Fforest == NULL) ) {
        tmp = (struct forest *) xmalloc( sizeof(*tmp));
```

```
        tmp->set = fset;
        tmp->next = Fforest;
        Fforest = tmp;
    } else {
        /* link with previous fset */
        set = new_set();
        set->edge = set->edget = NULL;
        set->fparent = NULL;
        fset->fparent = Fforest->set->fparent = set;
        set->left = fset;
        set->right = Fforest->set;
        Fforest->set = set;
    };
}
/*
 *  read regular expression from file -- trim all the white space
 *  and newlines
 */
void read_input( fname, str)
char *fname;
MYCHAR *str;
{
#define BUF 256
    int cnt,fd;
    MYCHAR *ptr, buf[BUF];

    if( (fd = open(fname, 0)) <= 0) {
        printf("can not open %s\n", fname);
        exit(0);
    };
    /* read regular expression */
```

```
    do {
        cnt = read(fd, buf, BUF);
        for(ptr = buf; ptr < buf + cnt; ptr++) {
            switch(*ptr) {
                case ' ':
                case '\t':
                case '\n':
                case '\r':
                    break;
                default:
                    *str++ = *ptr;
            };
        };
    } while( cnt );
    *str = '\0';
    close(fd);
}
#include <sys/stat.h>
#include <sys/types.h>
MYCHAR *read_rexp( fname )
char *fname;
{
    int cnt, fd;
    struct stat s;
    MYCHAR *rtn;

    if( (fd = open(fname, 0)) <= 0) return( NULL);
    if( fstat(fd, &s) ) return( NULL);
    rtn = (MYCHAR *) xmalloc( cnt = s.st_size );
    cnt = read(fd, rtn, cnt);
    /* eliminate last NEWLINE -- same as egrep */
```

```
    if( (cnt) && (rtn[cnt-1] == '\n')) rtn[cnt-1] = '\0';

    close(fd);
    return(rtn);
}
```

Appendix C

cgrep Benchmark Raw Timing Data

All test are performed in a SUN 3/50. Benchmark time isin mini-second.

NFA Construction Time

$a_1 \cdots a_n$

	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	20	100	1.00	5.00
50	40	40	280	1.00	7.00
100	40	120	680	3.00	17.00
150	80	280	1160	3.50	14.50
200	120	480	1720	4.00	14.33

$-(a? \cdots a?)-$

	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	40	140	2.00	7.00
50	40	200	600	5.00	15.00
100	60	820	2620	13.67	43.67
150	100	1720	7220	17.20	72.20
200	120	3020	15640	25.17	130.33

$-((a_1b_1)? \cdots (a_nb_n)?)^*-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	60	220	3.00	11.00
50	80	380	860	4.75	10.75
100	160	1400	3220	8.75	20.12
150	180	3140	8160	17.44	45.33
200	260	5460	16960	21.00	65.23

$-((a_1b_1)\vert \cdots \vert(a_nb_n))^*-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	80	220	2.00	5.50
50	120	280	600	2.33	5.00
100	160	1040	1500	6.50	9.38
150	300	2260	2500	7.53	8.33
200	360	3920	3960	10.89	11.00

$-((a_1b_1)? \cdots (a_nb_n)?)-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	60	260	1.50	6.50
50	100	300	900	3.00	9.00
100	220	1000	3380	4.55	15.36
150	280	2120	8740	7.57	31.21
200	420	3700	18180	8.81	43.29

$-((a_1b_1)^+ \cdots (a_nb_n)^+)^+-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	80	200	2.00	5.00
50	100	320	560	3.20	5.60
100	120	1220	1360	10.17	11.33
150	220	2680	2400	12.18	10.91
200	260	4500	3860	17.31	14.85

$-(a_1? \cdots a_n?)^*-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	60	160	3.00	8.00
50	40	220	620	5.50	15.50
100	60	760	2700	12.67	45.00

$-(a_1 \| \cdots \| a_n)^*-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	40	100	n/a	n/a
50	60	140	280	2.33	4.67
100	80	440	640	5.50	8.00

$-(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^n(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^*-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	60	140	3.00	7.00
50	60	180	380	3.00	6.33
100	140	600	780	4.29	5.57
150	180	1140	1240	6.33	6.89
200	280	2000	1840	7.14	6.57

$-(a_1? \cdots a_n?)-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	20	120	1.00	6.00
50	40	120	540	3.00	13.50
100	80	400	2380	5.00	29.75

$-(a_1^+ \cdots a_n^+)^+-$					
	NFA construction time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	40	100	2.00	5.00
50	60	180	320	3.00	5.33
100	60	660	840	11.00	14.00

DFA Construction Time

$a_1 \cdots a_n$

	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	20	n/a
50	0	80	n/a
100	20	300	15.00
150	40	600	15.00
200	40	1060	26.50

$-(a? \cdots a?)-$

	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	0	n/a
50	0	20	n/a
100	0	60	n/a
150	0	80	n/a
200	0	160	n/a

$-((a_1b_1)? \cdots (a_nb_n)?)^*-$

	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	40	n/a
50	20	160	8.00
100	20	420	21.00
150	20	800	40.00
200	40	1240	31.00

$-((a_1b_1)| \cdots |(a_nb_n))^*-$

	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	60	n/a
50	0	180	n/a
100	20	420	21.00
150	60	800	13.33
200	20	1220	61.00

$-((a_1 b_1)? \cdots (a_n b_n)?)-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	20	200	10.00
50	140	1160	8.29
100	360	4440	12.33
150	700	9100	13.00
200	1020	16500	16.18

$-((a_1 b_1)^+ \cdots (a_n b_n)^+)^+-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	60	n/a
50	40	200	5.00
100	20	520	26.00
150	20	900	45.00
200	20	1360	68.00

$-(a_1? \cdots a_n?)^*-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	40	n/a
50	0	120	n/a
100	20	480	24.00

$-(a_1 \| \cdots \| a_n)^*-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	20	n/a
50	0	100	n/a
100	20	420	21.00

$-(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^n(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^*-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	20	20	1.00
50	0	180	n/a
100	20	620	31.00
150	40	1340	33.50
200	60	2320	38.67

$-(a_1? \cdots a_n?)-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	20	120	6.00
50	100	1300	13.00
100	360	9180	25.50

$-(a_1^+ \cdots a_n^+)^+-$			
	DFA construction time (msec)		speedup ratio
length	cgrep2	egrep2	egrep2/cgrep2
20	0	40	n/a
50	20	200	10.00
100	40	720	18.00

On-line Simulation Time

$a_1 \cdots a_n$

	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	20	140	1.00	7.00
50	0	60	280	n/a	n/a
100	20	200	560	10.00	28.00
150	20	420	840	21.00	42.00
200	60	740	1180	12.33	19.67

$-(a? \cdots a?)-$

	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	0	60	n/a	n/a
50	20	0	320	0.00	16.00
100	0	20	2020	n/a	n/a
150	0	0	6420	n/a	n/a
200	0	0	14700	n/a	n/a

$-((a_1 b_1)? \cdots (a_n b_n)?)^*-$

	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	40	220	n/a	n/a
50	20	140	800	7.00	40.00
100	20	400	3220	20.00	161.00
150	20	760	8660	38.00	433.00
200	40	1320	18340	33.00	458.50

$-((a_1 b_1)| \cdots |(a_n b_n))^*-$

	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	20	220	1.00	11.00
50	20	120	840	6.00	42.00
100	20	360	3260	18.00	163.00
150	40	740	8880	18.50	222.00
200	40	1160	18460	29.00	461.50

$-((a_1b_1)?\cdots(a_nb_n)?)-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	80	720	2.00	18.00
50	140	580	4140	4.14	29.57
100	400	2860	19360	7.15	48.40
150	640	84980	53640	132.78	83.81
200	1200	245820	115400	204.85	96.17

$-((a_1b_1)^+\cdots(a_nb_n)^+)^+-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	40	240	n/a	n/a
50	20	160	840	8.00	42.00
100	20	440	3260	22.00	163.00
150	20	800	8760	40.00	438.00
200	20	1300	18520	65.00	926.00

$-(a_1?\cdots a_n?)^*-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	20	160	n/a	n/a
50	0	40	880	n/a	n/a
100	0	140	4920	n/a	n/a

$-(a_1\|\cdots\|a_n)^*-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	0	160	n/a	n/a
50	0	40	880	n/a	n/a
100	20	80	4940	4.00	247.00

$-(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^n(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^*-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	20	260	1.00	13.00
50	40	40	1480	1.00	37.00
100	20	160	7440	8.00	372.00
150	40	280	21180	7.00	529.50
200	40	480	46200	12.00	1155.00

$-(a_1? \cdots a_n?)-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	20	600	0.50	15.00
50	80	40	4420	0.50	55.25
100	200	140	25180	0.70	125.90

$-(a_1^+ \cdots a_n^+)^+-$					
	on-line simulation time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	20	220	n/a	n/a
50	20	80	1300	4.00	65.00
100	40	100	7300	2.50	182.50

Total Elapsed Time

$a_1 \cdots a_n$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	40	240	1.00	6.00
50	40	100	560	2.50	14.00
100	60	320	1240	5.33	20.66
150	100	700	2000	7.00	20.00
200	180	1220	2900	6.77	16.11

$-(a? \cdots a?)-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	40	200	2.00	10.00
50	60	200	920	3.33	15.33
100	60	840	4640	14.00	77.33
150	100	1720	13640	17.20	136.40
200	120	3020	30340	25.16	252.83

$-((a_1b_1)? \cdots (a_nb_n)?)^*-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	100	440	5.00	22.00
50	100	520	1660	5.20	16.60
100	180	1800	6440	10.00	35.77
150	200	3900	16820	19.50	84.10
200	300	6780	35300	22.60	117.66

$-((a_1b_1)\| \cdots \|(a_nb_n))^*-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	60	100	440	1.66	7.33
50	140	400	1440	2.85	10.28
100	180	1400	4760	7.77	26.44
150	340	3000	11380	8.82	33.47
200	400	5080	22420	12.70	56.05

$-((a_1b_1)?\cdots(a_nb_n)?)-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	80	140	980	1.75	12.25
50	240	880	5040	3.66	21.00
100	620	3860	22740	6.22	36.67
150	920	87100	62380	94.67	67.80
200	1620	249520	133580	154.02	82.45

$-((a_1b_1)^+\cdots(a_nb_n)^+)^+-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	120	440	3.00	11.00
50	120	480	1400	4.00	11.66
100	140	1660	4620	11.85	33.00
150	240	3480	11160	14.50	46.50
200	280	5800	22380	20.71	79.92

$-(a_1?\cdots a_n?)^*-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	80	320	4.00	16.00
50	40	260	1500	6.50	37.50
100	60	900	7620	15.00	127.00

$-(a_1\vert\cdots\vert a_n)^*-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	0	40	260	n/a	n/a
50	60	180	1160	3.00	19.33
100	100	520	5580	5.20	55.80

$-(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^n(0\|1\|2\|3\|4\|5\|6\|7\|8\|9)^*-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	40	80	400	2.00	10.00
50	100	220	1860	2.20	18.60
100	160	760	8220	4.75	51.37
150	220	1420	22420	6.45	101.90
200	320	2480	48040	7.75	150.12

$-(a_1? \cdots a_n?)-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	60	40	720	0.66	12.00
50	120	160	4960	1.33	41.33
100	280	540	27560	1.92	98.42

$-(a_1^+ \cdots a_n^+)^+-$					
	total elapsed time (msec)			speedup ratio	
length	cgrep	egrep	gnu	egrep/cgrep	gnu/cgrep
20	20	60	320	3.00	16.00
50	80	260	1620	3.25	20.25
100	100	760	8140	7.60	81.40

Programming Language Test Pattern

Raw Timing Data (in mini-second)

	NFA	DFA	simu	elapsed time
cgrep	80	0	840	920
egrep	420	300	1260	1680
e?grep	740	n/a	2640	3380

Speedup Ratio

	NFA	DFA	simu	elapsed time
cgrep /egrep	5.2	n/a	1.5	1.82
cgrep/ e?grep	9.3	n/a	3.14	3.67

Bibliography

[1] Aho, A., Hopcroft, J. and Ullman J., "Design and Analysis of Computer Algorithms", *Reading*, Addison-Wesley, 1974.

[2] Aho, A., Sethi, R. and Ullman, J., "Compilers Principles, Techniques, and Tools", *Reading*, Addison-Wesley, 1986.

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

[4] Berry, G. and Cosserat, L., "The Esterel synchronous programming language and its mathematical semantics" in *Seminar in Concurrency*, S. D. Brookes, A. W. Roscoe, and G. Winskel, eds., LNCS 197, Springer-Verlag, 1985.

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

[6] Brüggemann-Klein, A., "Regular Expressions into Finite Automata", To appear in *Theoretical Computer Science*, 1992.

[7] Brüggemann-Klein, A., private communication, 1992.

[8] Brzozowski, J., "Derivatives of Regular Expressions", *JACM*, Vol. 11, No. 4., Oct. 1964, pp. 481-494.

[9] Cai, J. and Paige, R., "Look Ma, No Hashing, And No Arrays Neither", *ACM POPL*, Jan. 1991, pp. 143 - 154.

[10] Emerson, E. and Lei, C., "Model Checking in the Propositional Mu-Calculus", *Proc. IEEE Conf. on Logic in Computer Science*, 1986, pp. 86 - 106.

[11] Fredman, M., Komlos, J., and Szemeredi, E., "Storing a Sparse Table with O(1) Worst Case Access Time", *JACM*, vol. 31, no. 3, pp. 538-544, July, 1984.

[12] Haertel, M., "GNU e?grep", *USENET archive comp.sources.unix*, Vol. 17, 1989.

[13] Hopcroft, J., " An nlogn Algorithm for Minimizing states in a Finite Automata", in *Theory of Machines and Computation*, ed. Kohavi and Paz, pp. 189-196, Academic Press, New York, 1971.

[14] Hopcroft, J. and Ullman, J., "Formal Languages and Their Relation to Automata",*Reading*, Addison-Wesley, 1969.

[15] Hume, A., "A Tale of Two Greps", Vol. 18, Num. 11, pp. 1063-1072, Nov., 1988. *Software-Practice and Experience*,

[16] Kleene, S., "Representation of events in nerve nets and finite automata", in *Automata Studies, Ann. Math. Studies No. 34*, Princeton U. Press, 1956, pp. 3 - 41.

[17] Knuth, D., "On the translation of languages from left to right", *Information and Control*, Vol. 8, No. 6, 1965, pp. 607 - 639.

[18] McNaughton, R. and Yamada, H. "Regular Expressions and State Graphs for Automata", *IRA Trans. on Electronic Computers*, Vol. EC-9, No. 1, Mar. 1960, pp 39-47.

[19] Meyer, E., "A Four-Russians Algorithm for Regular Expression Pattern Matching", *JACM*, vol. 39, no. 2, April, 1992.

[20] Myhill, J., "Finite automata and representation of events," WADC, Tech. Rep. 57-624, 1957.

[21] Nerode, A., "Linear automaton transformations," *Proc. Amer. Math Soc.*, Vol. 9, pp. 541 - 544, 1958.

[22] Paige, R., "Programming with Invariant", *IEEE Software*, 3:1, Jan, 1986, pp. 56-69.

[23] Paige, R. and Koenig, S. "Finite Differencing of Computable Expressions", *TOPLAS*, Vol. 4, Num. 3, 1982, pp. 402-452.

[24] Paige, R. and Tarjan, R., "Three Efficient Algorithms Based on Partition Refinement", *SIAM Journal on Computing*, Vol. 16, No. 6, Dec, 1987.

[25] Rabin, M. and Scott, D., "Finite automata and their decision problems" *IBM J. Res. Develop.*, Vol. 3, No. 2, Apr., 1959, pp. 114 - 125.

[26] Ritchie, D. and Thompson, K. "The UNIX Time-Sharing System" *Communication ACM*, Vol. 17, No. 7, Jul., 1974, pp. 365 - 375.

[27] Sharir, M., "Some Observations Concerning Formal Differentiation of Set Theoretic Expressions", *ACM TOPLAS* 4:2 (1982), pp. 196-225.

[28] Smith, D., "KIDS - A Knowledge-Based Software Development System", in Proc. Workshop on Automating Software Design, AAAI-88, 1988.

[29] "SunOS Reference Manual VOL. II", *Programmer's Manual*, SUN microsystems, 1989.

[30] Thompson, K., "Regular Expression search Algorithm", *Communication ACM* 11:6 (1968), pp. 419-422.

[31] Ullman, J., "Computational Aspects of VLSI", *Computer Science Press*, 1984.

[32] Wu, S. and Manber, U., "Fast Text Seaching with Errors", submitted to *CACM*. Also, Tech. report TR-91-11, dept. of computer science, University of Arizona.