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

Chia-Hsiang Chang
Institute of Information Science
Academica Sinica
Teipei, Taiwan, R. O. C.
changch@iis.sinica.edu.tw

Robert Paige
Courant Institute of Mathematical Sciences
New York University
251 Mercer St. New York, NY 10012
paige@cs.nyu.edu

May 3, 1996

Abstract

There are two principal methods for turning regular expressions into NFA's - one due to McNaughton and Yamada and another due to Thompson. Unfortunately, both have drawbacks. Given a regular expression R of length r and with s occurrences of alphabet symbols, Chang and Paige (1992) and Brüggemann-Klein (1993) gave $\Theta(m+r)$ time and $O(r)$ space algorithms to produce a $\Theta(m)$ space representation of McNaughton and Yamada's NFA with $s+1$ states and m transitions. The problem with this NFA is that $m = \Theta(s^2)$ in the worst case. Thompson's method takes $\Theta(r)$ time and space to construct a $\Theta(r)$ space NFA with $\Theta(r)$ states and $\Theta(r)$ transitions. The problem with this NFA is that r can be arbitrarily larger than s.

We overcome drawbacks of both methods with a $\Theta(r)$ time $\Theta(s)$ space algorithm to construct an $O(s)$ space representation of McNaughton and Yamada's NFA. Given any set V of NFA states, our representation can be used to compute the set U of states one transition away from the states in V in optimal time $O(|V|+|U|)$. McNaughton and Yamada's NFA requires $\Theta(|V|\times|U|)$ time in the worst case. Using Thompson's NFA, the equivalent calculation requires $\Theta(r)$ time in the worst case. Comparative benchmarks show that an implementation of our method outperforms implementations

*This research was partially supported by Office of Naval Research Grant No. N00014-93-1-0924, Air Force Office of Scientific Research Grant No. AFOSR-91-0308, and National Science Foundation grant MIP-9300210. An earlier version of this paper appeared in the Conference Record of the Third Symposium on Combinatorial Pattern Matching (1992).

of competing methods with respect to time for NFA construction, NFA accepting testing, and NFA-to-DFA conversion by subset construction.

Throughout this paper program transformations are used to design algorithms and derive programs. A transformation of special importance is a form of finite differencing used previously by Douglas Smith to improve the efficiency of functional programs.

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[19]. Of special significance was their construction of the canonical minimum-state DFA that had been described non-constructively in the proof of the Myhill-Nerode Theorem[17, 18]. 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], model checking[12], lexical scanning[2], and VLSI layout design[25]; unit-time incremental acceptance testing in a DFA is also a crucial step in LR_k parsing[15]; algorithms for acceptance testing and DFA construction from regular expressions are implemented in the UNIX operating system[20].

Throughout this paper we use a uniform cost sequential RAM[1] as our model of computaton. However, our algorithms will avoid any RAM operation with hidden costs (that might show up under a logarithmic cost criterion). We report the following four results.

1. Berry and Sethi[5] used results of Brzozowski[7] to formally derive and improve McNaughton and Yamada's algorithm[16] for turning regular expressions into NFA's. NFA's produced by this algorithm have fewer states than NFA's produced by Thompson's algorithm[24], and are believed to outperform Thompson's NFA's for acceptance testing. Berry and Sethi did not publish the resource bounds of their algorithm or the details for an efficient implementation. Nevertheless, they knew that their algorithm could be implemented in worst case time $\Theta(m + r)$[21], where r is the length of the regular expression accepted as input, and m is the number of edges in the NFA produced. More recently, Brüggemann-Klein[6] presented another algorithm to compute McNaughton and Yamada's NFA, and she provided the full details and a convincing analysis that it does run in $\Theta(m + r)$ time and $\Theta(r)$ auxiliary space. The algorithms due to Berry and Sethi and to Brüggemann-Klein use multiple passes over the regular expression.

Section 3 of our paper reformulates a proof of McNaughton and Yamada's algorithm from an automaton-theoretic point of view. In Section 4 we transform this algorithm into a new algorithm that runs in the same resource bounds as Brüggemann-Klein, but offers a practical improvement by computing the same NFA in a single left-to-right scan over the regular expression without producing a parse tree. That algorithm was discovered by Chang and Paige independently of Brüggemann-Klein, and reported in the short form[10] of the current paper.

2. One disadvantage of McNaughton and Yamada's NFA is that its worst case number of edges is $m = \Theta(s^2)$. More specifically, its adjacency list representation takes up $3s + s^2$ space in the worst case. Thompson's NFA has between r and $2r$ states and between r and $4r$ edges. Its adjacency list representation takes up between $2r$ and $6r$ space, but r can be arbitrarily larger than s.

   In Section 5 we introduce a new compressed data structure, called the CNNFA, that uses only $O(s)$ 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. It supports acceptance testing in worst-case time $O(s|x|)$ for arbitrary string x, and a faster way to construct DFA's using an improved implementation of the classical subset construction of Rabin and Scott[19].

3. When using McNaughton and Yamada's NFA to perform either acceptance testing or NFA-to-DFA conversion, it is necessary to repeatedly compute the set of states U one edge away from an arbitrary set of states V. This *next-states* operation is a costly critical step of a local nature, since all we know about the two sets U and V is that they depend on the internal structure of the NFA, and that the sizes of neither set can exceed the global parameter s. Section 5 contains Theorem 5.5, our main theoretical result, which proves that the CNNFA can be used to perform this operation in in optimal time $O(|V| + |U|)$. The previous best worst-case time is $\Theta(|V| \times |U|)$. This is the essential idea that explains the computational advantage of the CNNFA over alternative NFA's in both acceptance testing and DFA construction.

4. Section 6 describes how to exploit the structure of the CNNFA to obtain an efficient implementation. It also gives 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 using our implementation of *next-states* in the classical Rabin and Scott subset construction[19] (cf. Chapter 3 of [2]) can be achieved in time one order of magnitude faster than 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.

2 Terminology and Background

With few exceptions the following basic definitions and terminology can be found in [2, 13]. 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 2.1 *Let L, L_1, L_2 be languages over Σ. The following expressions can be used to define new languages.*

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

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

Lemma 2.2

1. *$L\{\lambda\} = \{\lambda\}L = L$*
2. *$L \, \emptyset = \emptyset \, L = \emptyset$*
3. *$\emptyset^T = \lambda^T = \emptyset$*
4. *$(L_1 \cup L_2)^T = L_1^T \cup L_2^T$*
5. *$(L_1 L_2)^T = if \;\; \lambda \notin L_1 \;\; then \;\; L_1^T L_2 \;\; else \;\; L_1^T L_2 \cup L_2^T$*

Because of Lemma 2.2 identities (1) and (2), we will sometimes use $\{\lambda\}$ interchangeably with *true* and $\emptyset$ interchangeably with *false*.

Kleene[14] characterized a subclass of languages called *regular languages* in terms of *regular expressions*.

Definition 2.3 *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, regular expressions can be written with fewer parentheses by allowing star to have higher precedence than product, and product to have higher precedence than alternation. Both product and alternation are left associative. Parentheses are used to override precedence. Without loss of generality, we will assume throughout this paper that regular expressions have no occurrences of $\emptyset$.

Based on the preceding rules for defining regular expressions, we can recognize and distinguish different subexpressions occurring within a given regular expression. We say that J and K are the *immediate subexpressions* of regular expressions $J|K$ or JK, and J is the immediate subexpression of (J) or J^*. The *subexpressions* of regular expression R consists of R itself together with subexpressions of the immediate subexpressions of R.

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 2.4 *A nondeterministic finite automaton (abbr. NFA) M is a 5-tuple $(\Sigma, Q, I, F, \delta)$, where Σ is an alphabet, Q is a finite set of states, $I \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and $\delta : (Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q)$ is a state transition map, where 2^Q denotes the power set of Q. It is natural to represent transition map δ as a labeled graph. If $q \in Q$ and $y \in \Sigma \cup \{\lambda\}$, then $\delta(q, y)$ denotes the set of states $p \in Q$ reachable from state q by a single edge labeled y.*

If $V \subseteq Q$, $a \in \Sigma$, $x \in \Sigma^*$, and $B \subseteq \Sigma^*$, then it is useful to define an extended transition map δ_* so that $\delta_*(q, x)$ denotes the set of states reachable from q along any path whose labels spell x. More formally, we have,

- *$\delta_*(q, \lambda) =$ the smallest set s of states such that $\{q\} \cup \{p : \exists t \in s \mid p \in \delta(t, \lambda)\} \subseteq s$*

- $\delta_*(q,a) = \{p : \exists t \in \delta_*(q,\lambda), \exists t' \in \delta(t,a) | p \in \delta_*(t',\lambda)\}$
- $\delta_*(V,a) = \bigcup_{q \in V} \delta_*(q,a)$
- $\delta_*(q,ax) = \delta_*(\delta_*(q,a),x)$
- $\delta_*(V,x) = \bigcup_{q \in V} \delta_*(q,x)$
- $\delta_*(V,B) = \bigcup_{x \in B} \delta_*(V,x)$

The language L_M accepted by M is defined by the rule, $x \in L_M$ if and only if $\delta_*(I,x) \cap F \neq \emptyset$. In other words, $L_M = \{x \in \Sigma^* \mid \delta_*(I,x) \cap F \neq \emptyset\}$. NFA M is a deterministic finite automaton (abbr. DFA) if transition map δ has no more than one edge with the same label leading out from each state, if δ has no edge labeled λ, and if I contains exactly one state.

Kleene also characterized the regular languages in terms of languages accepted by DFA's. Rabin and Scott[19] 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. Regular expressions and NFA's that represent the same regular language are said to be *equivalent*.

There are two main practical approaches for turning regular expressions into equivalent NFA's. Thompson's approach[24] turns regular expressions into NFA's with λ-edges as described above. McNaughton and Yamada's approach[16] turns regular expressions into a slightly different kind of NFA described in the next section. There is one main approach for turning NFA's (constructed by either the method of Thompson or McNaughton and Yamada) into equivalent DFA's. This is by Rabin and Scott's subset construction[19].

3 McNaughton and Yamada's NFA

It is convenient to reformulate McNaughton and Yamada's transformation[16] from regular expressions to NFA's in the following way.

Definition 3.1 *A normal NFA (abbr. NNFA) is an NFA in which no edge can be labeled λ, and all edges leading into the same state have the same label. Thus, only states need to be labeled, and we can 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 $A : Q \to \Sigma$ 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.*

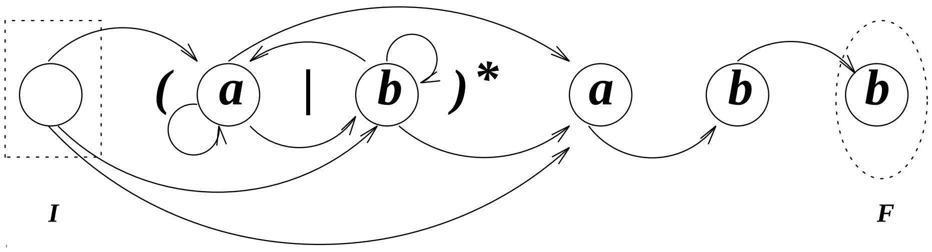


Figure 1: An MYNNFA equivalent to regular expression $(a|b)^*abb$

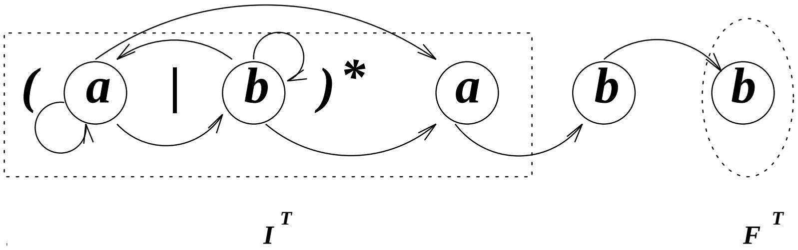


Figure 2: The tail of an MYNNFA equivalent to regular expression $(a|b)^*abb$

We sometimes omit alphabet 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 need to define A on the initial state of an MYNNFA. Fig. 1 represents an MYNNFA with six states (represented as circles that enclose their labels) and eleven transitions (represented as directed edges between states).

Definition 3.2 *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* $\Sigma^T = \Sigma$, $Q^T = Q - \{q_0\}$, $\delta^T = \{[x, y] \in \delta \mid x \neq q_0\}$, $I^T = \{y : [q_0, y] \in \delta\}$, $F^T = F - \{q_0\}$, *and* $A^T = \{[q, a] \in A \mid q \neq q_0\}$.

Fig. 2 shows the tail of the MYNNFA given in Fig. 1. This example demonstrates that the tail of an MYNNFA is an NNFA that need not be an MYNNFA. Note that our definition of a tail machine only applies to an MYNNFA, and not to NNFA's in general.

Lemma 3.3 *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.

Proof: Let $M = (\Sigma, Q, \delta, I, F, A)$ be an MYNNFA. By Definition 2.1 (6), $x \in L_M^T$ iff $\exists a \in \Sigma \mid ax \in L_M$. By Definition of an MYNNFA, this is equivalent to saying that there is a sequence $q_0, q_1, ..., q_n$ of two or more states in Q that forms a path in δ from the unique initial state q_0 to a final state q_n in which q_1 is labeled a, and the labels on states $q_2, ..., q_n$ spell x (which is λ if $n = 1$). Since the initial state q_0 of an MYNNFA must have in-degree zero, q_0 must be different from $q_1, ..., q_n$. Hence, by Definitions 3.1 and 3.2 this is equivalent to saying that there is a sequence $q_1, ..., q_n$ with $n \geq 1$ of states in Q^T that forms a path in δ^T in which $q_1 \in I^T$, $q_n \in F^T$, and the labels for states $q_1, ..., q_n$ are the same in M^T as in M. And this is equivalent to saying that $x \in L_M^T$. □

However, given the tail of an MYNNFA M, we cannot compute an MYNNFA equivalent to M without also knowing whether $\lambda \in L_M$; i.e., we don't know whether the starting state of M is also a final state unless we know that $\lambda \in L_M$. If we know the value of

$$null_M \stackrel{\text{def}}{=} if\ \lambda \in L_M\ then\ \{\lambda\}\ else\ \emptyset,$$

then MYNNFA $M = (\Sigma, Q, \delta, I, F, A)$ can be reconstructed from its tail $M^T = (\Sigma^T, Q^T, \delta^T, I^T, F^T, A^T)$ and $null_M$ using equations,

$$\begin{aligned}&\Sigma = \Sigma^T,\ Q = Q^T \cup \{q_0\},\ \delta = \delta^T \cup \{[q_0, y] : y \in I^T\},\ I = \{q_0\},\\ &F = F^T \cup \{q_0\} null_M,\ A = A^T,\\ &\textit{where } q_0 \textit{ is a new state}\end{aligned} \tag{1}$$

Lemma 3.4 *If $M = (Q, \delta, I = \{q_0\}, F, A)$ is an MYNNFA that accepts language L, then an MYNNFA M^* that accepts language L^* can be formed from M by using the same set Q of states, the same starting state q_0, and the same label map A as M. The set of final states of M^* is $F \cup \{q_0\}$; the transition map is $\delta \cup FI^T$.*

Proof: If L_{M^*} is the language accepted by MYNNFA M^*, then it is easy to see that $L^* \subseteq L_{M^*}$. To prove that $L_{M^*} \subseteq L^*$, consider any path $P = [q_0, q_1, ..., q_n]$ that follows transitions in M^* from starting state q_0 to any final state q_n. We show that the labels $A(q_1)...A(q_n)$ along P spell a word that belongs to L^*. If $n = 0$, then the labels spell λ, which certainly belongs to L^*. Suppose $n \geq 1$. Let transition $[q_i, q_{i+1}]$ be called a reversal if it belongs to FI^T for some $i = 1, ..., n-1$. We show by mathematical induction on the number t of reversals that the concatenated labels along path P spell a word that belongs to L^{t+1}.

(Basis) If there are no reversals, then P is also a path in M, and the concatenated labels must spell a word in L.

(Induction) Suppose the claim holds for $t \geq 0$, and suppose P contains $t+1$ reversals. If transition $[q_j, q_{j+1}]$ is the last reversal in P, then labels $A(q_1)...A(q_j)$ along path $q_0, q_1, ..., q_j$ must spell a word that belongs to L^t by the induction hypothesis. Since there are no reversals along path $q_{j+1}, ..., q_n$, then the labels $A(q_{j+1})...A(q_n)$ along path $q_j, q_{j+1}, ..., q_n$ spell a word that belongs to L. Hence, the labels $A(q_1)...A(q_n)$ along path P spell a word in L^{t+1}. □

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 $\{[x, y] \in \delta \mid A(y) = a\}$ is single-valued for every alphabet symbol $a \in \Sigma$.

McNaughton and Yamada's algorithm inputs a regular expression R, and computes an MYNNFA M that accepts language L_R. To explain how the construction is done, we use the notational convention that M_R denotes an MYNNFA equivalent to regular expression R. Because of its importance in Rule (1), $null_R$ will be regarded as an essential component of M_R^T.

Theorem 3.5 *(McNaughton and Yamada[16]) 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 M_R^T for some MYNNFA M_R, where M_R^T contains one distinct state for every alphabet symbol that occurs in R; i.e., M_R^T contains s states altogether. Equations (1) can be used to obtain M_R, which has one more state than M_R^T.

We assume a fixed alphabet Σ. There are two base cases, which are easily verified from Lemma 2.2 (3) and Definition 3.2.

$$
\begin{aligned}
M_\lambda^T &= (Q_\lambda^T = \emptyset, \delta_\lambda^T = \emptyset,\ I_\lambda^T = \emptyset,\ F_\lambda^T = \emptyset,\ A_\lambda^T = \emptyset,\ null_\lambda = \{\lambda\}) \quad (2)\\
M_a^T &= (Q_a^T = \{q\},\ \delta_a^T = \emptyset,\ I_a^T = \{q\},\ F_a^T = \{q\},\ A_a^T = \{[q, a]\},\\
&\quad null_a = \emptyset), \quad (3)\\
&\quad \textit{where } a \in \Sigma, \textit{and } q \textit{ is a new distinct state}
\end{aligned}
$$

Note that the tail machine for a single alphabet occurrence contains one new distinct state introduced by Rule (3).

To use induction, we assume that J and K are two arbitrary regular expressions in which the occurrences of alphabet symbols within J are distinct from the occurrences of alphabet symbols within K. Assume also that J and K are equivalent respectively to MYNNFA's M_J and M_K with $M_J^T = (Q_J^T, I_J^T, F_J^T, \delta_J^T, A_J^T, null_J)$ and $M_K^T = (Q_K^T, I_K^T, F_K^T, \delta_K^T, A_K^T, null_K)$, where Q_J^T and Q_K^T are disjoint sets of states that correspond to the distinct sets of alphabet symbol occurrences within J and K respectively. Then we can use Lemma 2.2 (4) and (5) to verify that

$$
\begin{aligned}
M_{J|K}^T &= (Q_{J|K}^T = Q_J^T \cup Q_K^T,\ \delta_{J|K}^T = \delta_J^T \cup \delta_K^T,\ I_{J|K}^T = I_J^T \cup I_K^T,\\
&\quad F_{J|K}^T = F_J^T \cup F_K^T,\ A_{J|K}^T = A_J^T \cup A_K^T,\\
&\quad null_{J|K} = null_J \cup null_K) \quad (4)
\end{aligned}
$$

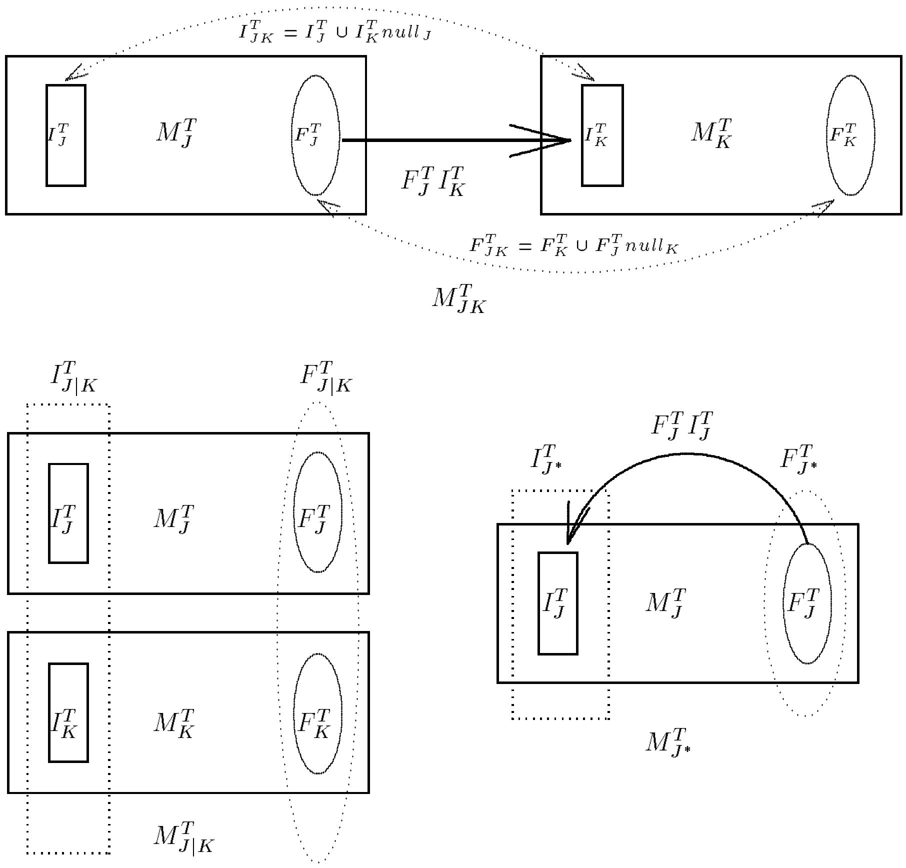


Figure 3: Tail machine construction.

$$\begin{aligned} M^T_{JK} \quad &= \quad (Q^T_{JK} = Q^T_J \cup Q^T_K,\ \delta^T_{JK} = \delta^T_J \cup \delta^T_K \cup F^T_J I^T_K, \\ &\qquad I^T_{JK} = I^T_J \cup null_J I^T_K,\ F^T_{JK} = F^T_K \cup null_K F^T_J, \\ &\qquad A^T_{JK} = A^T_J \cup A^T_K,\ null_{JK} = null_J null_K) \end{aligned} \tag{5}$$

We use Lemmas 3.3 and 3.4 to verify that

$$\begin{aligned} M^T_{J*} \quad &= \quad (Q^T_{J*} = Q^T_J,\ \delta^T_{J*} = \delta^T_J \cup F^T_J I^T_J,\ I^T_{J*} = I^T_J,\ F^T_{J*} = F^T_J, \\ &\qquad A^T_{J*} = A^T_J,\ null_{J*} = \{\lambda\}) \end{aligned} \tag{6}$$

The preceding formulas are illustrated in Fig. 3.

Since each set $Q^T_{J|K}$ and Q^T_{JK} is the disjoint union of Q^T_J and Q^T_K by Rules (4) and (5), we know by the induction hypothesis that the number of states in each set $Q^T_{J|K}$ and Q^T_{JK} equals the number of occurrences of alphabet symbols in $J|K$ and JK respectively. Since $Q^T_{J*} = Q^T_J$ by Rule (6), it follows from the induction hypothesis that $|Q^T_{J*}|$ equals the number of occurrences of alphabet symbols in J^*. The result follows. $\square$

McNaughton and Yamada's algorithm can be implemented within a left-to-right, bottom-up, shift/reduce parse (by operator precedence, for example) of regular expression R without actually producing a parse tree. The algorithm depends on a stack of elements, each of which is either a symbol from R, or a record N_P that stores δ^T_P, A^T_P, I^T_P, F^T_P, and $null_P$ for some subexpression occurrence P of R.

The following procedure can be used:

1. Initialize the stack to empty.
2. For each input symbol c in a left-to-right scan through R do the following:
   (a) Push c onto the stack.
   (b) Repeat the following step until it can no longer be applied:
   (*Reduction*) If the topmost elements of the stack match one of the following cases, replace them in the prescribed way. Otherwise, terminate the Reduction Step.
   (case λ) Replace by N_λ according to Rule (2).
   (case a, an alphabet symbol) Replace by N_a according to Rule (3).
   (case $N_J|N_K$) Replace by $N_{J|K}$ using Rule (4).
   (case $N_J N_K$) Replace by N_{JK} using Rule (5).
   (case N_J *) Replace by N_{J^*} using Rule (6).
   (case (N_J)) Replace by N_J.
3. If the stack contains only one entry, which is a record, then that entry is N_R, which stores components of M^T_R. Compute MYNNFA M_R from N_R according to Rule (1). If the stack is of any other form, then R is not a regular expression.

Lemma 3.6 *1. All unions appearing in Rules (4) and (5) are disjoint. 2. If P is a regular expression, then F^T_P and I^T_P are both nonempty iff P contains at least one alphabet symbol.*

Proof: 1. By the proof of Theorem 3.5 the unions $Q^T_J \cup Q^T_K$ appearing in Rules (4) and (5) are disjoint. Since F^T_J and I^T_J are subsets of Q^T_J, and F^T_K and I^T_K are subsets of Q^T_K, it follows that F^T_J is disjoint from F^T_K and that I^T_J is disjoint from I^T_K. It is easily shown that δ^T_J and δ^T_K are also disjoint, because they are binary relations on Q^T_J and Q^T_K respectively.

2. ($\Rightarrow$) Since F^T_P and I^T_P are both subsets of Q^T_P, then Q^T_P is nonempty whenever F^T_P or I^T_P are nonempty. Since there are $|Q^T_P|$ occurrences of alphabet

symbols within P by the proof of Theorem 3.5, then P must have at least one alphabet symbol when Q_P^T is nonempty.

($\Leftarrow$) Suppose that P has $s \geq 1$ occurrences of alphabet symbols. Using structural induction, the cases for Rules (3), (4), and (6) are easily verified. For Rule (5) suppose that J contains occurrences of alphabet symbols but K does not. Then I_J^T and F_J^T are both nonempty by hypothesis. It is trivial to show that if K has no alphabet symbols, then $null_K$ is true (.i.e, equals $\{\lambda\}$), so that $null_K F_J^T$ is nonempty. Hence, I_{JK}^T and F_{JK}^T are both nonempty. $\square$

Analysis determines that the algorithm mentioned above falls short of optimal performance, because the operation $\delta_J^T \cup F_J^T I_J^T$ within Rule (6) for M_{J*}^T is not disjoint. By Lemma 3.6 all other unions are disjoint and can be implemented in unit time. This overlapping union makes McNaughton and Yamada's algorithm use time $\theta(s^3 logs)$ to transform regular expression

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

into an MYNNFA with $s+1$ states and $s+s^2$ edges.

The sources of the overlapping union problem are made explicit in two examples. In order to compute δ_{J**}^T, Rule (6) would be applied twice; i.e.,

$$\begin{aligned} \delta_{J*}^T &= \delta_J^T \cup F_J^T I_J^T \\ \delta_{J**}^T &= \delta_{J*}^T \cup F_{J*}^T I_{J*}^T \\ & \quad \textit{where } F_{J*}^T = F_J^T \textit{ and } I_{J*}^T = I_J^T \end{aligned}$$

which shows that product $F_J^T I_J^T$ is contained in both δ_{J*}^T and $F_{J*}^T I_{J*}^T$. For a second example, if $null_J = null_K = \{\lambda\}$, then in order to compute $\delta_{(JK)*}^T$, Rules (5) and (6) would be applied; i.e.,

$$\begin{aligned} \delta_{JK}^T &= \delta_J^T \cup \delta_K^T \cup F_J^T I_K^T \\ \delta_{(JK)*}^T &= \delta_{JK}^T \cup F_{JK}^T I_{JK}^T \\ & \quad \textit{where } F_{JK}^T I_{JK}^T = F_J^T I_J^T \cup F_J^T I_K^T \cup F_K^T (I_J^T \cup I_K^T) \end{aligned}$$

which shows that product $F_J^T I_K^T$ is contained in both δ_{JK}^T and $F_{JK}^T I_{JK}^T$.

4 Faster NFA Construction

By recognizing the overlapping union $\delta_J^T \cup F_J^T I_J^T$ within Rule (6) as the main source of inefficiency within our implementation of McNaughton and Yamada's algorithm, we can speedup the algorithm by augmenting tail machine M_P^T with a new component $nred_P$ that satisfies Invariant

$$nred_P = F_P^T I_P^T - \delta_P^T \tag{7}$$

and is stored in stack record N_P for any regular expression P. This allows us to replace the overlapping union within Rule (6) by the equivalent but more efficient disjoint union

$$\delta^T_{J*} = \delta^T_J \cup nred_J \tag{8}$$

Invariant (7) is satisfied by the following inductive definition, which is obtained by simplifying expression $F^T_P I^T_P - \delta^T_P$ in the context of equations (2) - (6).

$$\begin{aligned}
nred_\lambda &= \emptyset && (9)\\
nred_a &= F^T_a I^T_a, where\ a \in \Sigma && (10)\\
nred_{J|K} &= nred_J \cup nred_K \cup F^T_J I^T_K \cup F^T_K I^T_J && (11)\\
nred_{JK} &= F^T_K I^T_J \cup null_K nred_J \cup null_J nred_K && (12)\\
nred_{J*} &= \emptyset && (13)
\end{aligned}$$

The preceding idea of program improvement by maintaining and exploiting invariants 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 [22]).

Lemma 4.1 *Rules (9) - (13) satisfy Invariant (7). Each union operation occurring within Rules (9) - (13) and Rule (8) is disjoint.*

Proof: Correctness of Rules (9) - (13) is proved by structural induction on regular expressions based on the correctness (from the proof of Theorem 3.5) of Rules (2) - (6). Rules (9), (10) and (13) are trivial. Rule (11) follows from applying Rule (4) to the right-hand-side of Identity (7) to obtain

$$\begin{aligned}
nred_{J|K} &= F^T_{J|K} I^T_{J|K} - \delta^T_{J|K}\\
&= (F^T_J \cup F^T_K)(I^T_J \cup I^T_K) - (\delta^T_J \cup \delta^T_K)
\end{aligned}$$

which is then expanded by distributive laws, and simplified by 'folding' Definition (7). Rule (12) follows from applying Rule (5) to the right-hand-side of identity (7) to obtain

$$\begin{aligned}
nred_{JK} &= F^T_{JK} I^T_{JK} - \delta^T_{JK}\\
&= (F^T_K \cup null_K F^T_J)(I^T_J \cup null_J I^T_K) - (\delta^T_J \cup \delta^T_K \cup F^T_J I^T_K)
\end{aligned}$$

which is then expanded using distributive laws, and simplified by 'folding' Definition (7) as in the previous case.

By the proof of Lemma 3.6, sets F^T_J, I^T_J, and Q^T_J are all disjoint from sets F^T_K, I^T_K, and Q^T_K within Rules (11) and (12). By Invariant (7), we know that the union occurring in Rule (8) is disjoint, and that $nred_P$ is a binary relation defined on Q_P^T for any regular expression P. Hence, $nred_J$ and $nred_K$ are disjoint within Rules (11) and (12). The result follows immediately. □

Although we are now able to calculate MYNNFA M_R with only disjoint unions (which are $O(1)$ time implementable), our solution creates a new problem. Edges contributed to $nred_{J|K}$ and $nred_{JK}$ by potentially costly product operations within Rules (11) and (12) may never get added to transition map δ_R. This phenomenon can arise when (1) regular expression R is star-free, or (2) Rule (12) is used and either $nred_J$ is true (i.e., equals $\{\lambda\}$) and $null_K$ is false (i.e., equals $\emptyset$) or $nred_K$ is non-empty and $null_J$ is false.

To overcome this problem we will use lazy evaluation to compute products only when they actually contribute edges to δ_R. A lazy product AB is represented by a pair $[A, B]$ of sets A and B. For arbitrary regular expression P a lazy representation for $nred_P$ (which is a union of products) is a set $lazynred_P$ of pairs of sets satisfying the following invariant,

$$nred_P = \bigcup_{[X,Y] \in lazynred_P} XY \tag{14}$$

As an optimization, only pairs that contain two nonempty sets will be stored in $lazynred_P$. We will interpret the collective union over the empty set as resulting in the empty set (so the right-hand-side of formula (14) evaluates to empty whenever $lazynred_P$ is empty). By replacing component $nred_P$ within both M_P^T and N_P^T by $lazynred_P$, we can replace Rule (8) for calculating δ_{J*}^T from $nred_J$ by the equivalent calculation

$$\delta_{J*}^T = \delta_J^T \cup \Big(\bigcup_{[X,Y] \in lazynred_J} XY \Big) \tag{15}$$

We use the following macro,

$$nesets(X, Y) \stackrel{\text{def}}{=} if\ X \neq \emptyset \wedge Y \neq \emptyset\ then\ \{[X, Y]\}\ else\ \emptyset$$

to specify the inductive definition below, which satisfies Invariant (14).

$$\begin{aligned}
lazynred_\lambda &= \emptyset && (16)\\
lazynred_a &= \Big\{ \Big[F_a^T, I_a^T \Big] \Big\},\ where\ a \in \Sigma && (17)\\
lazynred_{J|K} &= lazynred_J \cup lazynred_K \cup nesets(F_J^T, I_K^T) \cup \\
&\quad\ nesets(F_K^T, I_J^T) && (18)\\
lazynred_{JK} &= null_K lazynred_J \cup null_J lazynred_K \cup \\
&\quad\ nesets(F_K^T, I_J^T) && (19)\\
lazynred_{J*} &= \emptyset && (20)
\end{aligned}$$

Lemma 4.2 *Rules (16) - (20) satisfy Invariant (14). Each union operation occurring within Rules (16) - (20) and Rule (15) is disjoint.*

Proof: Correctness of Rules (16) - (20) is proved by structural induction on regular expressions based on the correctness (by Lemma 4.1) of Rules (9) - (13). Essentially, Rules (16) - (20) are obtained from trivially turning every nonempty product that occurs in Rules (9) - (13) into a lazy product. Disjointness of unions follows immediately from Lemma 4.1, Invariant (14), and the disjoint union in Rule (8). □

Although this approach allows us to avoid computing any products until they are needed to (implicitly) reconstruct $nred_J$ and to augment δ^T_{J*} in Rule (15), a new problem arises. Copying and storing sets F^T_K and I^T_J within $lazynred_P$ for arbitrary (and not just immediate) subexpressions J and K of regular expression P can consume too much time and space. Fortunately, since, by Lemma 3.6, Rules (2) - (6) only use copy operations and disjoint unions to construct F^T_J and I^T_K, we can overcome this problem by using a binary forest as a simple space-efficient *persistent* data structure (in the sense of [11]) that stores all intermediate values of sets F^T_K and I^T_J in the order in which they are computed by Rules (2) - (6) in the construction of δ^T_R.

We represent a directed unoriented *forest* abstractly as a binary relation $succ$ over a finite set of nodes in which the inverse relation

$$succ^{-1} = \{[y, x] : [x, y] \in succ\}$$

is the graph of a finite function. For each node n, the term $succ\{n\}$ denotes the set $\{y : [n, y] \in succ\}$ of *children* of n, and the term $succ^{-1}(n)$ denotes the unique *parent* of n. Node n is a *leaf* iff $succ\{n\}$ is empty; it is *internal* if it isn't a leaf; it is a *root* iff $succ^{-1}(n)$ is undefined. To represent a binary forest, we also require that $|succ\{n\}| = 2$ for every internal node n.

Corresponding to each node n of $succ$ is a subtree represented by the restriction of $succ$ to the set of nodes reachable along paths in $succ$ from n to any leaf. The term $frontier(n, succ)$ denotes the set of leaves of this subtree. It is convenient to denote an undefined node by $\perp$ for which,

$$frontier(\perp, succ) \stackrel{\text{def}}{=} \emptyset$$

The preceding forest relation $succ$ can be implemented as an adjacency list (cf. an early chapter of any elementary algorithms text; e.g. [1]) in which each node n in $succ$ is implemented as a distinct unit-space record. It is convenient to refer to a node and its record implementation interchangeably. For each node n, its record stores a pointer to a list of pointers to the children of n. Suppose that every internal node n has more than one child; i.e., $|succ\{n\}| > 1$. Then, for any node n, we can compute $frontier(n, succ)$ in time proportional to the number $|frontier(n, succ)|$ of subtree leaves (using almost any kind of search method; e.g. depth-first-search). Creating a forest with one node n is a unit-time operation that forms a record for n with a nil pointer (since $succ\{n\}$ is empty). Augmenting the forest by adding a new root n, and adding edges $[n, x_1], ..., [n, x_k]$ to $succ$ from n to k distinct roots $x_1, ..., x_k$ is performed in $O(k)$ time by forming a new record for n with a pointer to a list of pointers to $x_1, ..., x_k$.

We consider a special case of the augmenting operation for binary forests. Let $n1$ and $n2$ be roots (possibly undefined) of binary forests $succ1$ and $succ2$ respectively, where the nodes of $succ1$ and $succ2$ are disjoint. The following macro combines $succ1$ and $succ2$ into a single binary forest $succ$. If $n1$ and $n2$ are both defined, then a new root q is created, and edges from q to $n1$ and $n2$ are added to $succ$. The macro returns a pair whose second component is the new value of $succ$. If $n1$ and $n2$ are both defined, then the first component is q; otherwise, if one of the two roots are defined, that root is returned; otherwise, $\perp$ is returned.

```
-- n1 is either undefined or a root in binary forest succ1;
-- n2 is either undefined or a root in binary forest succ2
-- PRECONDITION: succ1 nodes are disjoint from succ2 nodes
punion(n1, succ1, n2, succ2) =def --persistent union
    if n1 ≠ ⊥ ∧ n2 ≠ ⊥ then
        return [q, succ1 ∪ succ2 ∪ {[q, n1], [q, n2]}]
            where q is a new root
    else
        return [if n1 ≠ ⊥ then n1 else n2, succ1 ∪ succ2]
    end if
end
```

Operation $punion$ clearly takes unit time and space.

For regular expression P we use binary forests F^T_{P-succ} and I^T_{P-succ} as persistent data structures that store sets F^T_J and I^T_J respectively for every subexpression occurrence J within P. Each such occurrence J is associated with (possibly undefined) nodes F^T_{J-root} and I^T_{J-root} that satisfy Invariants,

$$\begin{aligned} F^T_J &= frontier(F^T_{J-root}, F^T_{P-succ}) \\ I^T_J &= frontier(I^T_{J-root}, I^T_{P-succ}) \end{aligned} \tag{21}$$

Consequently, each set F^T_J (respectively I^T_J) is represented most efficiently by a single node F^T_{J-root} (respectively I^T_{J-root}). We will exploit this idea by implementing each lazy product $\left[F^T_K, I^T_J\right]$ belonging to $lazynred_P$ more efficiently using a persistent product $\left[F^T_{K-root}, I^T_{J-root}\right]$. A persistent representation of $lazynred_P$ is a set $pnred_P$ of pairs $[x, y]$ of nodes satisfying invariant,

$$lazynred_P = \left\{\left[frontier(x, F^T_{P-succ}), frontier(y, I^T_{P-succ})\right] : [x, y] \in pnred_P\right\} \tag{22}$$

We will reformulate the tail MYNNFA in terms of the preceding persistent representations, and improve the algorithm to compute MYNNFA M_R from its tail M_R^T. We use macro *punion* from the discussion above, and the following macro (similar to *nesets*)

$$nnpairs(X,Y) \stackrel{\text{def}}{=} if\ X \neq \perp \wedge Y \neq \perp\ then\ \{[X,Y]\}\ else\ \emptyset$$

in order to specify the inductive definition below of the tail MYNNFA satisfying Invariants (21) and (22).

$$\begin{aligned}
M_\lambda^T &= (\delta_\lambda^T = \emptyset,\ I_{\lambda-succ}^T = \emptyset,\ I_{\lambda-root}^T = \perp,\ F_{\lambda-succ}^T = \emptyset,\ F_{\lambda-root}^T = \perp, \\
&\quad pnred_\lambda = \emptyset,\ A_\lambda^T = \emptyset,\ null_\lambda = \{\lambda\}) && (23) \\
M_a^T &= (\delta_a^T = \emptyset,\ I_{a-succ}^T = \emptyset,\ I_{a-root}^T = q,\ F_{a-succ}^T = \emptyset,\ F_{a-root}^T = q, \\
&\quad pnred_a = \{[q,q]\},\ A_a^T = \{[q,a]\},\ null_a = \emptyset) && (24) \\
&\quad where\ a \in \Sigma, and\ q\ is\ a\ new\ distinct\ state \\
M_{J|K}^T &= (\delta_{J|K}^T = \delta_J^T \cup \delta_K^T, \\
&\quad \left[I_{J|K-root}^T, I_{J|K-succ}^T\right] = punion(I_{J-root}^T, I_{J-succ}^T, I_{K-root}^T, I_{K-succ}^T), \\
&\quad \left[F_{J|K-root}^T, F_{J|K-succ}^T\right] = punion(F_{J-root}^T, F_{J-succ}^T, F_{K-root}^T, F_{K-succ}^T, \\
&\quad pnred_{J|K} = pnred_J \cup pnred_K \cup nnpair(F_{J-root}^T, I_{K-root}^T) \cup \\
&\quad\quad nnpair(F_{K-root}^T, I_{J-root}^T),\ A_{J|K}^T = A_J^T \cup A_K^T, \\
&\quad null_{J|K} = null_J \cup null_K) && (25) \\
M_{JK}^T &= (\delta_{JK}^T = \delta_J^T \cup \delta_K^T \cup frontier(F_{J-root}^T, F_{J-succ}^T) frontier(I_{K-root}^T, I_{K-succ}^T), \\
&\quad \left[I_{JK-root}^T, I_{JK-succ}^T\right] = punion(I_{J-root}^T, I_{J-succ}^T, \\
&\quad\quad if\ null_J\ then\ I_{K-root}^T\ else\ \perp, I_{K-succ}^T), \\
&\quad \left[F_{JK-root}^T, F_{JK-succ}^T\right] = punion(if\ null_K\ then\ F_{J-root}^T\ else\ \perp, \\
&\quad\quad\quad F_{J-succ}^T, F_{K-root}^T, F_{K-succ}^T), \\
&\quad pnred_{JK} = null_K pnred_J \cup null_J pnred_K \cup \\
&\quad\quad nnpair(F_{K-root}^T, I_{J-root}^T),\ A_{JK}^T = A_J^T \cup A_K^T, \\
&\quad null_{JK} = null_J null_K) && (26) \\
M_{J*}^T &= (\delta_{J*}^T = \delta_J^T \cup (\bigcup_{[x,y] \in pnred_J} frontier(x, F_{J-succ}^T) frontier(y, I_{J-succ}^T), \\
&\quad I_{J*-succ}^T = I_{J-succ}^T,\ I_{J*-root}^T = I_{J-root}^T,\ F_{J*-succ}^T = F_{J-succ}^T, \\
&\quad F_{J*-root}^T = F_{J-root}^T,\ pnred_{J*} = \emptyset,\ A_{J*}^T = A_J^T,\ null_{J*} = \{\lambda\}) && (27)
\end{aligned}$$

The preceding inductive definitions for constructing the tail MYNNFA together with the following new Rule, analogous to Rule (1), for constructing an

MYNNFA from its tail,

$$\begin{aligned} M_R \quad = \quad & (\delta_R = \delta^T_R \cup \{q_0\} frontier(I^T_{R-root}, I^T_{R-succ}),\ I_R = q_0, \\ & I_{R-succ} = I^T_{R-succ},\ F_R = null_R\{q_0\} \cup frontier(F^T_{R-root}, F^T_{R-succ}), \\ & F_{R-succ} = F^T_{R-succ}, A = A^T), \\ & where\ \ q_0\ is\ a\ new\ distinct\ state \end{aligned} \tag{28}$$

lead to our first theoretical result.

Theorem 4.3 *For any regular expression R we can compute an equivalent MYNNFA with $s+1$ states in time $O(r+m)$ and auxiliary space $O(r)$, 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.*

Proof: As before, by applying Rules (23) - (27) during each reduction step of a shift/reduce regular expression parse, and subsequently applying Rule (28), we can construct MYNNFA M_R equivalent to regular expression R. Each stack record N_P will now store all components of our persistent reformulation of M^T_P for any regular expression P.

The correctness of Rules (23) - (28) is proved by structural induction of regular expressions P based on the correctness (by Theorem 3.5) of Rules (1) - (6), and the correctness (by Lemma 4.2) of Rule (15) and Rules (16) - (20). The inductive definitions of I^T_{P-root}, I^T_{P-succ}, F^T_{P-root} F^T_{P-succ} within Rules (23) - (27) are shown to satisfy Invariants (21) based on the semantics of *punion* and the inductive definitions for F^T_P and I^T_P within Rules (1) - (6). Invariants (21) justify the inductive definitions of δ^T_{JK} within Rule (26), and the specifications of δ_R and F_R within Rule (28), which are derived from their counterparts within Rules (5) and and Rule (1) by replacing occurrences of F^T_J and I^T_K by equivalent $frontier$ expressions.

Because it is derived by trivially replacing every lazy product that occurs in Rules (16) - (20) with persistent products, the inductive definition for $pnred_P$ within Rules (23) - (27) must satisfy Invariant (22). Invariant (22) allows us to directly derive the inductive definition of δ^T_{J*} within Rule (27) based on Rule (15).

All of the union operations appearing in Rules (23) - (28) are disjoint, and so unit-time implementable. For any regular expression P disjointness of unions within rules for $pnred_P$ follows immediately from Lemma 4.2 and Invariant (22). Disjointness of unions within rules for δ^T_P follows from Lemma 3.6, Lemma 4.2 (which proves disjointness of unions appearing in Rule (15)), and Invariants (21).

Analysis of the time bound for the algorithm is straightforward. Scanning regular expression R from left to right (and shifting each symbol onto the stack) takes $O(r)$ time altogether. A reduction, which triggers application of one of the rules (23) - (27), is performed once for each occurrence of λ, an alphabet symbol, or an operator appearing in R. Each *punion* and *nnpair* operaton appearing in Rules (23) - (28) takes unit time, and each *frontier* calculation takes linear time in the set it computes. The cross product calculations in Rules (26), (27), and (28) take linear time in the edges they compute. Since the final MYNNFA δ_R is the disjoint union of such products, the cumulative cost of these products is $O(m)$. The remaining *frontier* calculation in Rule (28) computes the final states F_R in $O(|F_R|) = O(s)$ time, and would be performed only once. Hence, the cumulative time to perform all reductions is $O(m + r)$, and MYNNFA M_R can be constructed in $O(m + r)$ time.

Next, consider auxiliary space. Because of occurrences of λ and parentheses appearing in regular expression R, the stack can take up $\Theta(r)$ space. Recall that each record N_P stored on the stack represents some subexpression P occurring in the input regular expression R. Since each record component for I^T_{P-root}, F^T_{P-root}, and $null_P$ takes up $O(1)$ space, $O(r)$ space is needed overall to store these components on the stack.

Each record N_P for which P contains no alphabet symbols is equivalent to N_λ (as implied by Lemma 3.6), which can be stored in unit space by Rule (23). Hence, $O(r)$ space is a bound on all such records. Since regular expression R contains s occurrences of alphabet symbols, the stack cannot contain more than s records N_P in which P contains one or more alphabet symbol. Since *punion* maintains I^T_{J-succ} and F^T_{J-succ} as binary forests, since the nodes of I^T_{J-succ} (respectively F^T_{J-succ}) are disjoint for distinct records N_J appearing on the stack at any one time, and since the union of the frontiers of these forests appearing on the stack at any one time have $O(s)$ nodes, then the total space needed to store these forests on the stack is also $O(s)$.

If P is a regular expression with $c \geq 1$ occurrences of alphabet symbols, then we show by structural induction on regular expressions that $|pnred_P| \leq 3(c-1)+1$. The claim is true for any alphabet symbol a, since $|pnred_a| = 1$ by Rule (24). The claim holds for J^*, since $|pnred_{J^*}| = 0$ by Rule (27). The claim also holds if P is of the form JK or $J|K$, and either argument has no occurrences of alphabet symbols. In these cases each *nnpair* operation performed in Rules (25) and (26) evaluates to $\emptyset$, which implies that

$$|pnred_P| = |pnred_J| + |pnred_K| \leq 3(c-1)+1$$

by induction. Suppose that J and K have $c_J \geq 1$ and $c_K \geq 1$ occurrences of alphabet symbols respectively. If P is of the form $J|K$, then the two *nnpair* operations performed in Rule (25) contribute one edge apiece to $pnred_P$. Hence, by Rule (25) and induction,

$$|pnred_{J|K}| = |pnred_J| + |pnred_K| + 2 \leq 3(c_J-1) + 3(c_K-1) + 4 = 3(c-1)+1$$

A similar inductive argument using Rule (26) proves the claim when P is of the form JK. Hence, the total space used to store $pnred_P$ for all records N_P

appearing on the stack at any one time is $O(s)$. We conclude that the algorithm uses $O(r)$ auxiliary space overall. $\square$

Theorem 4.3 leads to a new algorithm that computes the adjacency list form of MYNNFA M_R in a single left-to-right, bottom-up, shift/reduce parse of the regular expression R. It has the same asymptotic resource bounds as Brüggemann-Klein's algorithm[6], but has the advantage of using only one pass.

5 Improving Space for McNaughton and Yamada's NFA

As was remarked earlier, McNaughton and Yamada's NFA has certain theoretical disadvantages over Thompson's simpler NFA. For regular expressions

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

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 given in Theorem 4.3 so that in $O(r)$ time it produces an $O(s)$ space data structure that encodes McNaughton and Yamada's NFA, and still supports acceptance testing in $O(s|x|)$ time for input strings x. In the same way that $nred_P$ was represented in lazy, persistent form as $pnred_P$ for regular expression P, we can represent δ_P^T, which is the union of cartesian products, as a set $lazy\delta_P^T$ of pairs of nodes from persistent data structures F_{P-succ}^T and I_{P-succ}^T satisfying invariant,

$$\delta_P^T = \bigcup_{[x,y]\in lazy\delta_P^T} frontier(x, F_{P-succ}^T) frontier(y, I_{P-succ}^T) \tag{29}$$

Invariant (29) is satisfied by the following inductive definition, which results from turning every nonempty product that occurs in the inductive definition of δ_P^T for each form of P within Rules (23) - (27) by an equivalent persistent product.

$$\begin{aligned}
lazy\delta_\lambda^T &= \emptyset && (30)\\
lazy\delta_a^T &= \emptyset && (31)\\
lazy\delta_{J|K}^T &= lazy\delta_J^T \cup lazy\delta_K^T && (32)\\
lazy\delta_{JK}^T &= lazy\delta_J^T \cup lazy\delta_K^T \cup nnpair(F_{J-root}^T, I_{K-root}^T) && (33)\\
lazy\delta_{J*}^T &= lazy\delta_J^T \cup pnred_J && (34)
\end{aligned}$$

Once $lazy\delta_R^T$ is computed, we can compute the compressed form $lazy\delta_R$ of δ_R by the rule,

$$\begin{aligned}
lazy\delta_R &= lazy\delta_R^T \cup nnpair(\{q_0\}, I_{R-root}^T), && (35)\\
& \quad \textit{where } q_0 \textit{ is the new state generated in } Rule(28)
\end{aligned}$$

which satisfies invariant

$$\delta_R = \bigcup_{[x,y] \in lazy\delta_R} frontier(x, F_{R-succ}) frontier(y, I_{R-succ}) \tag{36}$$

Rule (35) is obtained by turning the nonempty product within Rule (28) into persistent form.

Lemma 5.1 *Rules (30) - (34) satisfy Invariant (29), and Invariant (36) is correctly established by Rule (35). Each union operation occurring within Rules (30) - (35) is disjoint.*

Proof: Correctness of Rules (30) - (35) is proved by structural induction on regular expressions based on the correctness (by Theorem 4.3) of Rules (23) - (28). Disjointness of unions follows immediately from Theorem 4.3 and Invariants (21). $\Box$

In order to explain how $lazy\delta_R$ can be used to simulate δ_R, it is convenient to regard $lazy\delta_R$ as a binary relation, and to introduce a few new operations on such relations. Let E be a binary relation, and let S be a set. Then:

$$\begin{array}{rcl}
\mathsf{domain}\ E & = & \{x : \exists y \mid [x,y] \in E\} \\
\mathsf{range}\ E & = & \{y : \exists x \mid [x,y] \in E\} \\
E^{-1} & = & \{[y,x] : [x,y] \in E\} \\
E\{x\} & = & \{y : [u,y] \in E \mid u = x\} \\
E[S] & = & \{y : \exists x \in S \mid [x,y] \in E\} = \bigcup_{x \in S} E\{x\} \\
E|_S & = & \{[x,y] \in E \mid x \in S\}
\end{array}$$

The term $E[S]$ is called the *image* of S under E; $E|_S$ is called the *restriction* of E to S.

If V is a subset of the MYNNFA states Q_R, 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 the set

$$F\_domain(V) = \{x \in \mathsf{domain}\ lazy\delta_R \mid V \cap frontier(x, F_{R-succ}) \neq \emptyset\}$$

of nodes in F_{R-succ} whose frontiers overlap with V. Next, we compute the set

$$I\_image(V) = lazy\delta_R[F\_domain(V)]$$

of nodes in I_{R-succ} that form the image of $F\_domain(V)$ under $lazy\delta_R$. After that we compute the set

$$\delta_R(V, \Sigma) = \{z : \exists y \in I\_image(V) \mid z \in frontier(y, I_{R-succ})\}$$

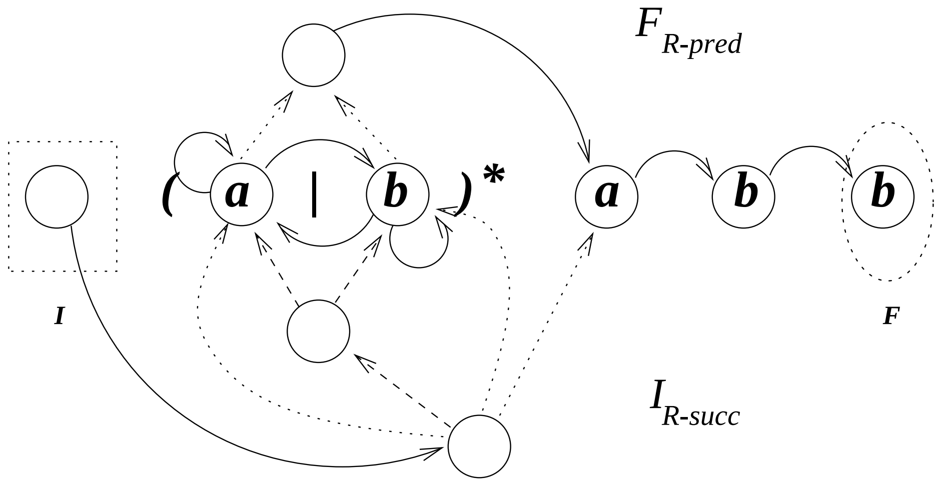


Figure 4: A Compressed MYNNFA (CNNFA) equivalent to regular expression $(a|b)^*abb$. Solid lines represent edges that belong to $lazy\delta_R$. Dotted lines represent edges that belong to F_{R-pred} and I_{R-succ}. Dashed lines represent edges removed by useless node elimination.

of states one transition away from V in δ_R. Finally, for each alphabet symbol $a \in \Sigma$, we compute the set

$$\delta_R(V, a) = \{q \in \delta_R(V, \Sigma) \mid A(q) = a\}$$

of states one transition away from V and labeled with symbol a.

We call our compressed MYNNFA $M_R = (lazy\delta_R, I_R, I_{R-succ}, F_R, F_{R-succ})$ a pre_CNNFA. To make simulation efficient, we need to modify the pre_CNNFA only slightly. To compute $F\_domain(V)$ efficiently, it is better to use a representation for parent pointers $F_{R-pred} = F^{-1}_{R-succ}$ than child pointers F_{R-succ}. In order to implement $lazy\delta_R$, we will represent every node n in F_{R-pred} as a record containing a single parent pointer, and a pointer to a list of pointers to the set $lazy\delta_R\{n\}$ of nodes in I_{R-succ}. We also gain efficiency by compressing forests F_{R-pred} and I_{R-succ} so that their internal nodes are contained in domain $lazy\delta_R$ and range $lazy\delta_R$ respectively. Such compression, which we call *useless node elimination*, removes intermediate nodes generated by successive applications of the alternation Rule (32). The representation that results from these modifications is called the CNNFA. Fig. 4 illustrates a CNNFA equivalent to regular expression $(a|b)^*abb$.

Procedure *convert* below is used to convert a pre_CNNFA to a CNNFA in $O(s)$ time and space. It performs useless node elimination together with the conversion of F_{R-succ} into F_{R-pred} in place; i.e., without relocating any node record that belongs to the domain or range of $lazy\delta_R$. Procedure *convert* makes use of a predicate $busy(n)$, which is true if and only if a node n belongs to Q_R or to the domain or range of $lazy\delta_R$.

To perform useless node elimination on I_{R-succ} we execute $convert(rt)$ for each root rt in forest I_{R-succ}.

```
procedure convert(n)
    if leaf(n) then
        return [n] -- busy(n) is true for leaf; return singleton
    else
        nodelist := append lists convert(y) for each y ∈ I_{R-succ}(n)
    end if
    if busy(n) then
(*)     I_{R-succ}(n) := nodelist --compression takes place here
        return [n]
    else
        return nodelist
    end if
end procedure
```

To perform useless node elimination and transform F_{R-succ} into F_{R-pred}, we modify procedure $convert$ by replacing line (*) with

```
    for x in nodelist loop
        F_{R-pred}(x) := n --compression takes place here
    end loop
```

and by replacing the remaining occurrence of I_{R-succ} by F_{R-succ}. If we allow F_{R-succ} and F_{R-pred} to be aliases, then simultaneous compression and conversion (implemented so that a pointer to a list of children is replaced by a parent pointer) is achieved in place.

Thus, we have

Theorem 5.2 *For any regular expression R, its equivalent CNNFA, consisting of F_{R-pred}, I_{R-succ}, I_R, F_R, A_R, and $lazy\delta_R$, takes up $O(s)$ space and can be computed in time $O(r)$ and auxiliary space $O(r)$.*

Proof: Let us first consider the cost of constructing a pre_CNNFA. As in Theorem 4.3, F_{R-succ} and I_{R-succ} are computed in $O(r)$ time and take up $O(s)$ space. Each of the unions in Rules (30) - (35) is disjoint by Lemma 5.1, and, hence, unit-time implementable for any regular expression P. In particular, $pnred_J$ can be combined with $lazy\delta_J^T$ destructively (i.e., without hidden costs due to copying) in unit time in Rule (34), since $pnred_{J^*}$ is subsequently assigned empty (in Rule (27), and $pnred_J$ is never needed again. Hence, the overall time bound for the rules to construct $lazy\delta_R$ is $O(r)$.

By Theorem 4.3 $O(s)$ space is needed to store F_{R-succ}^T and I_{R-succ}^T. Analysis of the space needed to store $lazy\delta_R$ is tied to the analysis of $pnred_R$. By the same argument used to analyze the overall space for $pnred$ contributed by Rule (26) in the proof of Theorem 4.3, we see that Rule (33) contributes $O(s)$ space overall to the calculation of $lazy\delta_R^T$. The $O(s)$ space bound for $pnred_R$ was coarse in the proof of Theorem 4.3, since it only bounded the cumulative number of pairs that were added to $pnred_R$ in every possible application of Rules (23) - (28). Since the union operation in Rule (34) is disjoint, the collection of sets $pnred_J$ that ever get added to $lazy\delta_R^T$ as a result of applying this rule must be mutually disjoint. Hence the cardinality of their union must be $O(s)$. We conclude that $O(s)$ space is needed to store $lazy\delta$. By the proof of Theorem 4.3 the overall space exclusive of the stack is $O(s)$. The overall space bound due to the stack size is $O(r)$. Finally, the conversion from a pre_CNNFA to a CNNFA takes $O(s)$ time and space. $\square$

CNNFA $lazy\delta_R$ also supports an efficient simulation of transition map δ_R. The best previous worst case time bound for computing the collection of sets $\delta_R(V,a)$ for an arbitrary subset $V \subseteq Q_R$ and all of the alphabet symbols $a \in \Sigma$ is $\Theta(|V| \times |\delta_R(V,\Sigma)|)$ using an adjacency list implementation of McNaughton and Yamada's NFA, or $\Theta(r)$ using Thompson's NFA.

In Theorem 5.5 we improve this bound, and obtain, essentially, optimal asymptotic time. 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 6.

Before proving the theorem, we will first prove two technical lemmas.

Lemma 5.3 *Let R be a regular expression containing occurrences of subexpressions J, K, and P, each of which contains an occurrence of an alphabet symbol. The following two properties hold for pre_CNNFA's.*

1. *If J is not a subexpression of K, and K is not a subexpression of J, then $F^T_{J-root} \neq F^T_{K-root}$ and $I^T_{J-root} \neq I^T_{K-root}$.*
2. *If J is a subexpression of K, and K is a subexpression of P, then (i) $F^T_{J-root} \neq F^T_{P-root}$ whenever $F^T_{K-root} \neq F^T_{J-root}$ or $F^T_{K-root} \neq F^T_{P-root}$, and (ii) $I^T_{J-root} \neq I^T_{P-root}$ whenever $I^T_{K-root} \neq I^T_{J-root}$ or $I^T_{K-root} \neq I^T_{P-root}$.*

Proof: Since J, K, and P contain alphabet symbols, then I^T_{J-root}, F^T_{J-root}, I^T_{K-root}, F^T_{K-root}, I^T_{P-root}, and F^T_{P-root} are all defined nodes, according to Lemma 3.6 (2) and Invariants (21).

(1) Since J and K have no common occurrences of alphabet symbols, then Q^T_J and Q^T_K are disjoint. Hence, so is F^T_J and F^T_K, so is I^T_J and I^T_K, and the inequalities are established.

(2) It suffices to prove implication (i). The proof is by structural induction on regular expression P. Suppose implication (i) holds for any immediate subexpression S of P. If P is of the form S^*, then $F^T_{P-root} = F^T_{S-root}$ according to Rule (27), so impliciation (i) must also hold for P. Suppose P is either of the forms $S|H$ or $S\ H$. By Rules (25) and (26) F^T_{P-root} is either the same as F^T_{S-root}, the same as F^T_{H-root}, or different from F^T_{S-root}, F^T_{H-root}, and from any node F^T_{J-root} for any subexpression occurrence J within S or H. In any case F^T_{S-root} is different from F^T_{H-root} by part (1) of the lemma. Hence, if F^T_{P-root} is different from F^T_{H-root}, then implication (i) must hold for P. If $F^T_{P-root} = F^T_{H-root}$, then implication (i) must also hold for P, because $F^T_{H-root} \neq F^T_{J-root}$ by part (1) of the lemma, .

□

Lemma 5.4 *Let V be any subset of states in the pre_CNNFA built from regular expression R, and let* $lazy\delta_V = \{[x, y] \in lazy\delta_R \mid frontier(x, F_{R-succ}) \cap V \neq \emptyset\}$. *Then* $|lazy\delta_V| = O(|I\_image(V)|)$.

Proof: We note, first of all, that

$$\begin{aligned} I\_image(V) &= lazy\delta_R[F\_domain(V)] \\ &= lazy\delta_R[\mathsf{domain}\ lazy\delta_V] \end{aligned}$$

The Lemma is proved by showing that $lazy\delta_V$ is the union of three one-to-one maps, one one-to-many map, and one many-to-one map whose cardinality is less than or equal to the cardinality of one of the other maps. Since the cardinality of a one-to-many map equals the cardinality of its own range, the cardinality of each of the five maps must be bounded by $|I\_image(V)|$.

Any pair that belongs to binary relation $lazy\delta_V$ is contributed by either Rule (33) (for product) or Rule (34) (for star). First consider the set of pairs $f_1 \subseteq lazy\delta_V$ contributed by Rule (33). Whenever pair $\left[F^T_{J-root}, I^T_{K-root}\right]$ is added to $lazy\delta^T_{JK}$ by Rule (33), we know that both F^T_{J-root} and I^T_{K-root} are defined. By Lemma 3.6 (2) and by Invariants (21), this implies that I^T_{J-root} and F^T_{K-root} are also defined. Hence, by Rule (26) we know that $F^T_{JK-root}$ must be either a new node or the same node as F^T_{K-root}, and $I^T_{JK-root}$ must be either a new node or the same node as I^T_{J-root}, depending on the values of $null_J$ and $null_K$. Let PS be a subexpression of R different from JK, and consider the pair $\left[F^T_{P-root}, I^T_{S-root}\right]$ contributed to $lazy\delta^T_{PS}$ by Rule (33). If neither PS or JK is a subexpression of the other, then $F^T_{J-root} \neq F^T_{P-root}$ and $I^T_{K-root} \neq I^T_{S-root}$ by Lemma 5.3 (1). Suppose that JK is a subexpression of P. Then $I^T_{K-root} \neq I^T_{S-root}$ by Lemma 5.3 (1). Since $F^T_{JK-root} \neq F^T_{J-root}$ by Rule (26), we know that $F^T_{J-root} \neq F^T_{P-root}$ by Lemma 5.3 (2). A similar argument is used for the case where JK is a subexpression of S. Hence, the set of pairs $\left[F^T_{J-root}, I^T_{K-root}\right]$ that gets added to $pnred_{JK}$ over every subexpression JK of

R forms a one-to-one map whose restriction f_1 to $\mathsf{domain}\ lazy\delta_V$ must also be one-to-one.

We split all of the pairs contributed by $pnred_J$ to $lazy\delta_V$ via Rule (34) (over all such J where Rule (34) is applied by our algorithm) into three subsets, based on the way that $pnred_J$ is built up according to Rules (24), (25), and (26) respectively. The set f of pairs $[q, q]$ added to $pnred_a$ by Rule (24) over all alphabet symbol occurrences a within R is clearly one-to-one, since a unique pair is added for each occurrence a of an alphabet symbol within R. Its restriction f_2 to $\mathsf{domain}\ lazy\delta_V$ must also be one-to-one.

In the case of Rule (25), each application of this rule adds either two new pairs $\left[F^T_{J-root}, I^T_{K-root}\right]$ and $\left[F^T_{K-root}, I^T_{J-root}\right]$ or no new pairs to $pnred_{J|K}$. Let $P|S$ be a subexpression of R different from $J|K$, and suppose that the two pairs $\left[F^T_{P-root}, I^T_{S-root}\right]$ and $\left[F^T_{S-root}, I^T_{P-root}\right]$ are contributed to $pnred_{S|P}$ by Rule (25). If neither $P|S$ or $J|K$ is a subexpression of the other, then the four pairs of nodes have no node in common by Lemma 5.3 (1). Suppose that $J|K$ is a subexpression of P. Then $I^T_{K-root} \neq I^T_{S-root}$, $I^T_{J-root} \neq I^T_{S-root}$, $F^T_{K-root} \neq F^T_{S-root}$, and $F^T_{J-root} \neq F^T_{S-root}$ by Lemma 5.3 (1). By Rule (25) we know that $F^T_{J|K-root} \neq F^T_{J-root}$, $F^T_{J|K-root} \neq F^T_{K-root}$, $I^T_{J|K-root} \neq I^T_{J-root}$, $I^T_{J|K-root} \neq I^T_{K-root}$, $F^T_{K-root} \neq F^T_{J-root}$, and $I^T_{K-root} \neq I^T_{J-root}$. It follows from Lemma 5.3 (2) that $F^T_{J-root} \neq F^T_{P-root}$, $F^T_{K-root} \neq F^T_{P-root}$, $I^T_{J-root} \neq I^T_{P-root}$, and $I^T_{K-root} \neq I^T_{P-root}$. A similar argument is used for the case where $J|K$ is a subexpression of S. Hence, all of these pairs forms a one-to-one map whose restriction f_3 to $\mathsf{domain}\ lazy\delta_V$ must also be one-to-one.

Finally, consider the set f of pairs $\left[F^T_{K-root}, I^T_{J-root}\right]$ contributed by Rule (26) to $pnred_{JK}$ over all subexpressions JK of R. Whenever any such pair $\left[F^T_{K-root}, I^T_{J-root}\right]$ is added to $pnred_{JK}$, then F^T_{K-root} and I^T_{J-root} must be defined. By Lemma 3.6 (2) and by Invariants (21), F^T_{K-root} and I^T_{J-root} are defined if and only if F^T_{J-root} and I^T_{K-root} are defined. Finally, F^T_{J-root} and I^T_{K-root} are defined if and only if the pair $\left[F^T_{J-root}, I^T_{K-root}\right]$ is added to $lazy\delta^T_{JK}$ by Rule (33).

Consider any pair $\left[F^T_{K-root}, I^T_{J-root}\right] \in f$ contributed by subexpression JK. For any other pair $\left[F^T_{H-root}, I^T_{G-root}\right] \in f$ to have $I^T_{G-root} = I^T_{J-root}$, it must be that either GH is a subexpression of J or JK is a subexpression of G by Lemma 5.3 (1). This implies that for any $I \in \mathsf{range}\ f$ the set $\{J_1K_1, ..., J_kK_k\}$ of all those subexpressions JK of R such that $F^T_{J-root} \in f^{-1}\{I\}$ and $I^T_{K-root} = I$ can be totally ordered such that J_iK_i is a subexpression of J_{i+1} for $i = 1, ..., k-1$. Among the k pairs contributed by these subexpressions to f, we call pair $\left[F^T_{K_k-root}, I^T_{J_k-root}\right]$ the *representative* for I. Let g be the set of representative pairs in f, and let $h = f - g$. Clearly g is one-to-many, and its restriction f_4

to $\mathsf{domain}\ lazy\delta_V$ must also be one-to-many. Unfortunately, the restriction f_5 of h to $\mathsf{domain}\ lazy\delta_V$ is many-to-one. For example, regular expression $(GH)K$ contributes pairs $\left[F^T_{H-root}, I^T_{G-root}\right]$ and $\left[F^T_{K-root}, I^T_{GH-root}\right]$ to $pnred_{(GH)K}$ such that $F^T_{H-root} \neq F^T_{K-root}$ and $I^T_{G-root} = I^T_{GH-root}$ whenever $null_G$ is false, $null_H$ is true, $null_K$ is true, and G, H, and K all have occurrences of alphabet symbols.

We will show that the cardinality of f_5 is bounded by the cardinality of f_1. The proof rests on two observations. The first is that $F^T_{J_iK_i-root}$ either is equal to or is the parent of $F^T_{K_i-root}$ in F^T_{R-succ} for $i = 1, ..., k$. This observation follows immediately from Rule (26), and implies that $F^T_{J_iK_i-root} \in \mathsf{domain}\ lazy\delta_V$ whenever $F^T_{K_i-root} \in \mathsf{domain}\ lazy\delta_V$. The second observation is that $F^T_{J_iK_i-root} = F^T_{J_{i+1}-root}$ for $i = 1, ..., k-1$. Thus, for $i = 1, ..., k-1$, if $\left[F^T_{K_i-root}, I^T_{J_i-root}\right]$ belongs to both f and $lazy\delta_V$, then $F^T_{J_iK_i-root} \in \mathsf{domain}\ lazy\delta_V$ by the first observation, $F^T_{J_{i+1}-root} \in \mathsf{domain}\ lazy\delta_V$ by the second, and $\left[F^T_{J_{i+1}-root}, I^T_{K_{i+1}-root}\right]$ belongs to one-to-one map f_1 by previous analysis. Hence, we have a one-to-one correspondence between f_5 and a subset of f_1.

To prove the second observation let $1 \leq i \leq k-1$. Since $I^T_{J_i-root} = I^T_{J_{i+1}-root}$, and since J_i is a subexpression of J_iK_i, which is a subexpression of J_{i+1}, then $I^T_{J_i-root} = I^T_{J_iK_i-root} = I^T_{J_{i+1}-root}$ by Lemma 5.3 (2). Either J_{i+1} equals J_iK_i or not. If it does, then $F^T_{J_{i+1}-root} = F^T_{J_iK_i-root}$ trivially. Otherwise, let $E_0 = J_iK_i$, let $E_j = J_{i+1}$, and let E_i be the immediate subexpression of E_{i+1} for $i = 0, ..., j-1$. We note, first of all, that since $I^T_{J_iK_i-root} = I^T_{J_{i+1}-root}$, then $I^T_{E_i-root} = I^T_{E_j}$ for $i = 0, ..., j$ by Lemma (5.3) (2). We show by induction that $F^T_{E_0} = F^T_{E_i}$ for $i = 1, ..., j$.

Assume that $F^T_{E_0} = F^T_{E_i}$ for $i \geq 0$. By hypothesis E_{i+1} cannot be of the form JK where neither J or K is equivalent to λ. Hence, by Rules (23) - (27), only when E_{i+1} is $L|E_i$, $E_i|L$, LE_i, E_iL, or $E_i{}^*$, where L is equivalent to λ, does $I_{E_i-root} = I_{E_{i+1}-root}$. And in all these cases we also have $F_{E_i-root} = F_{E_{i+1}-root}$.
□

Since $lazy\delta_V$ and $I\_image(V)$ are the same values in both the pre_CNNFA and the CNNFA (i.e., useless node elimination only removes nodes not incident to $lazy\delta_V$), the bound $|lazy\delta_V| = O(|I\_image(V)|)$ established by Lemma 5.4 for pre_CNNFA's also holds for CNNFA's. Hence, we have

Theorem 5.5 *Given any subset V of the CNNFA states, we can compute all of the sets $\delta_R(V, a)$ for every alphabet symbols $a \in \Sigma$ in time $O(|V| + |\delta_R(V, \Sigma)|)$.*

Proof: Because of useless node elmination, the set of all nodes in the CNNFA along the paths in F_{R-pred} starting from nodes in V is just the set $V \cup F\_domain(V)$, which can be found in $O(|V| + |F\_domain(V)|)$ time by a marked

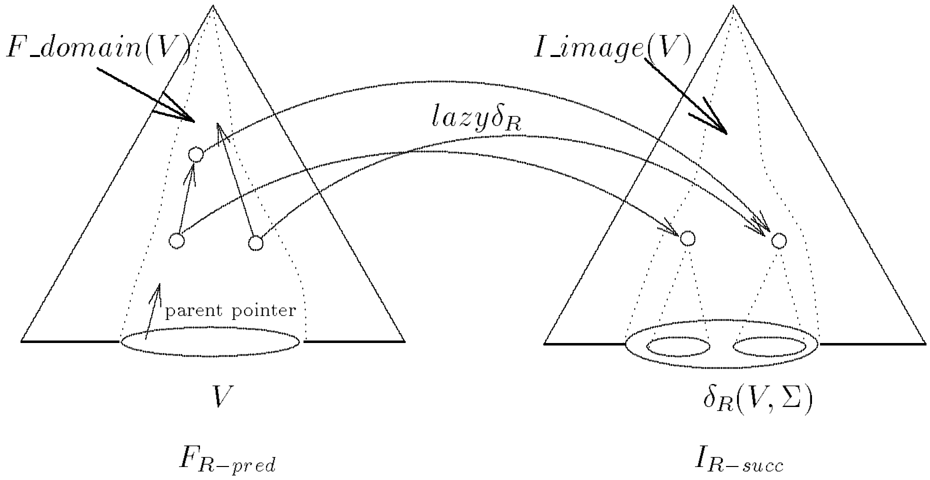


Figure 5: To compute $\delta_R(V, a)$ in a CNNFA

traversal of parent pointers in forest F_{R-pred}. Although $|F\_domain(V)|$ can be much larger than $|V|$, the fact that $F\_domain(V) = \textsf{domain}\ lazy\delta_V$ together with Lemma 5.4 guarantees that

$$|F\_domain(V)| = |\textsf{domain}\ lazy\delta_V| \leq |lazy\delta_V| = O(|I\_image(V)|$$

Computing $I\_image(V)$ involves taking the union of sets $lazy\delta_R\{n\}$ for each n that belongs to $F\_domain(V)$. That is, for each $n \in F\_domain(V)$ we traverse the list storing $lazy\delta_R\{n\}$, and mark every unmarked node in I_{R-succ} pointed to by an element of the list. $I\_image(V)$ is the set of all such marked nodes. Overall, this takes time linear in the sum of the lengths of these lists, which is $O(|lazy\delta_V|)$.

Calculating $\delta_R(V, \Sigma)$ involves computing the union of the sets $frontier(n, I_{R-succ})$ for each n that belongs to $I\_image(V)$. This is achieved in $O(|\delta_R(V, \Sigma)|)$ time using a depth-first-search through I_{R-succ} starting from $I\_image(V)$, marking all unmarked leaves. The set of marked leaves is $\delta_R(V, \Sigma)$. Multiset discrimination[8] can be used to separate out all of the sets $\{q \in \delta_R(V, \Sigma) \mid A(q) = a\}$ for each $a \in \Sigma$ in time $O(|\delta_R(V, \Sigma)|)$. See Fig. 5 for an illustration of the $\delta_R(V, a)$ computation. □

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 $\Theta(n^2)$ 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.

We end this section with the following theoretical result, originally proved by Chang[9], which improves the auxiliary space given in Theorems 4.3 and 5.2.

Theorem 5.6 *Given any regular expression R with length r and with s occurrences of alphabet symbols, its equivalent CNNFA can be computed in time $O(r)$ and auxiliary space $O(s)$.*

Proof: See the Appendix. □

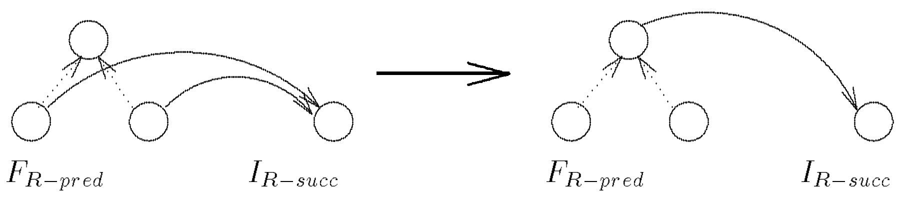
6 Computational Results

The unique structure of the CNNFA lends itself to further compression suitable for a practical implementation. Our implementation is developed starting from binary forests F_{R-pred} and I_{R-succ} without useless node elimination. For practical reasons we also equip forest I_{R-succ} with parent pointers as well as pointers to children. Then, in a single linear time bottom-up traversal through forests F_{R-pred} and I_{R-succ}, we repeatedly perform the following two *promotion* transformations, which reduce the number of edges in $lazy\delta_R$. In F_{R-pred} *promotion* we replace edges $[x_1, y]$ and $[x_2, y]$ within $lazy\delta_R$ by the single edge $[z, y]$ when edges $[x_1, z]$ and $[x_2, z]$ belong to F_{R-pred}. In I_{R-succ} *promotion* we replace edges $[x, y_1]$ and $[x, y_2]$ within $lazy\delta_R$ by the single edge $[x, z]$ when edges $[z, y_1]$ and $[z, x_2]$ belong to I_{R-succ} (see Fig. 6). In the case of regular expression $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ promotion can simplify $lazy\delta_R$ from $3s-1$ edges to two edges.

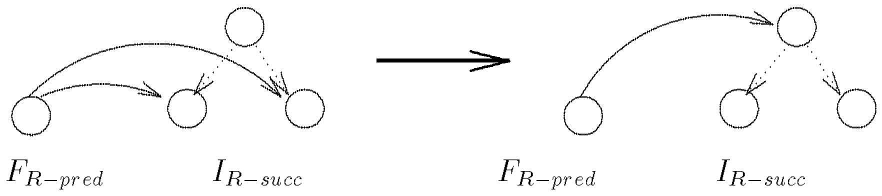
Recall from Section 5 that useless node elimination removes forest nodes so that the internal nodes of F_{R-pred} (respectively I_{R-succ}) are contained in the domain (respectively range) of $lazy\delta_R$. In developing our implementation, we carry out useless node elimination at the same time as promotion.

We also modify useless node elimination so that it can remove leaves of F_{R-pred} and I_{R-succ}. Whenever there is an internal node z of F_{R-pred} all of whose children are leaves $q_1, ..., q_k$, none of which belong to domain $lazy\delta_R$ or to range $lazy\delta_R$, and all of which share a common parent w in I_{R-succ}, then these leaves can be replaced by a single leaf z. For $i = 1, ..., k$, edge $[q_i, z]$ is removed from F_{R-pred}, and edge $[w, q_i]$ is replaced by $[w, z]$ within I_{R-succ}. We also need to assign the set $\{A(q_i) : i = 1, ..., k\}$ of symbols to $A(z)$ (cf. [9] for details). Finally, if $q_1, ..., q_k$ are all final states (they must be all final or none final), then they must be replaced in the set F_R of final states by z.

As an example, consider expression $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^*\cdots)^*)^*$ once again. After promotion the CNNFA has $4s-1$ states; after useless node elimination it has only two (cf. Fig. 7). In using our CNNFA to simulate an NFA, the transition edge t can be taken only if the input symbol being scanned belongs



F_{R-pred} Promotion



I_{R-succ} Promotion

Figure 6: F_{R-pred} and I_{R-succ} promotion

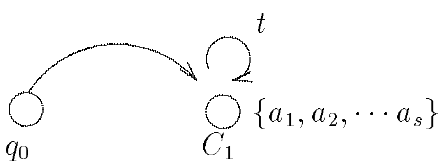


Figure 7: A CNNFA equivalent to $((a_1|\lambda)(\cdots((a_{s-1}|\lambda)(a_s|\lambda)^*)^* \cdots)^*)^*$ resulting from promotion and useless node elimination.

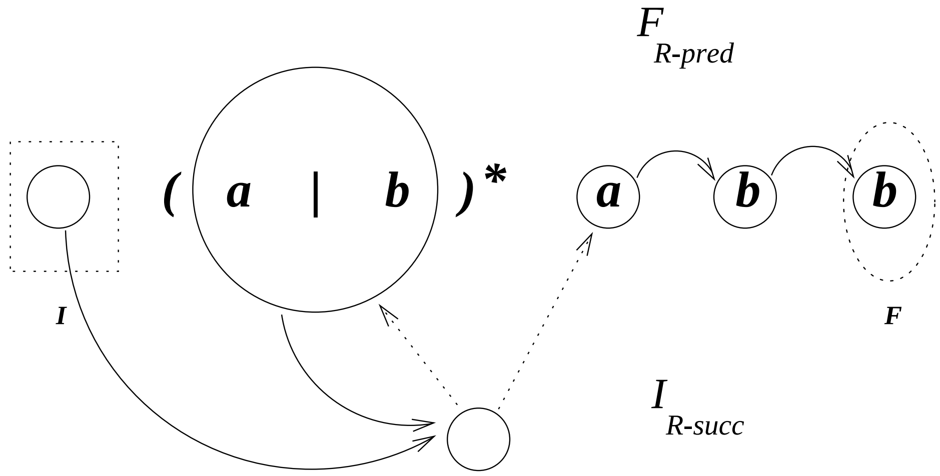


Figure 8: A CNNFA equivalent to regular expression $(a|b)^*abb$ resulting from promotion and useless node elimination

to $\{a_1, a_2, \cdots a_s\}$, which labels node C_1. Fig. 8 illustrates a CNNFA resulting from applying promotion and useless node elimination to the CNNFA of Fig. 4.

We mention one more useful transformation called *tree contraction*, which we did not implement. Tree contraction considers the union of edges in $lazy\delta_R$, F_{R-pred}, and I_{R-pred}. When an internal node n of F_{R-pred} 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 the $k_1 + k_2$ edges incident to n by $k_1 k_2$ edges (see Fig. 9); and (2) when an internal node n of I_{R-succ} has k_1 incoming edges and k_2 outgoing edges, and if $k_1 k_2 \leq k_1 + k_2$, then we can replace node n and the $k_1 + k_2$ edges incident to n by $k_1 k_2$ edges (see Fig. 9).

After applying tree contraction to the CNNFA of Fig. 8, one I_{R-succ} node is eliminated (see Fig. 10). It contains 5 states and 6 edges in contrast to the 9 states and 14 edges found in the pre_CNNFA of Fig. 4.

Experiments to benchmark the performance of our CNNFA implementation 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. We build Thompson's NFA according to the construction rules described in [2]. 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 some simple but effective transformations that eliminate redundant states and edges in most of the test cases.

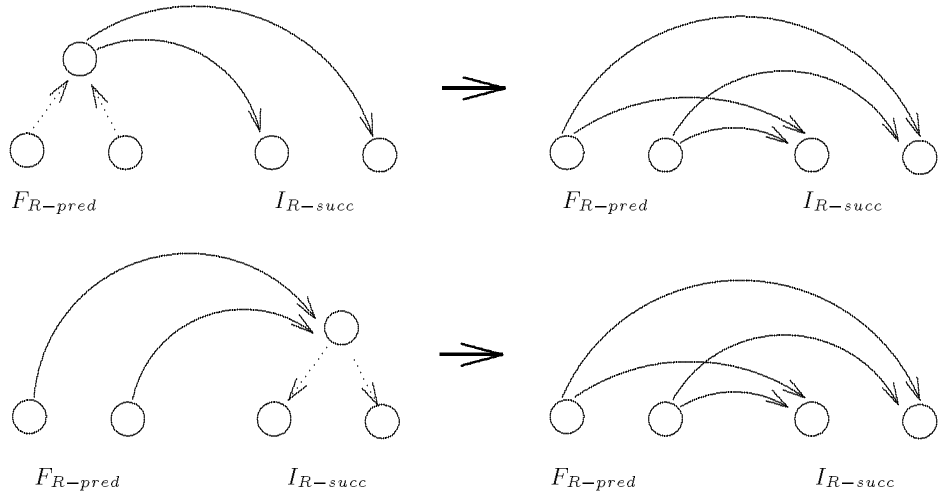


Figure 9: Tree contraction

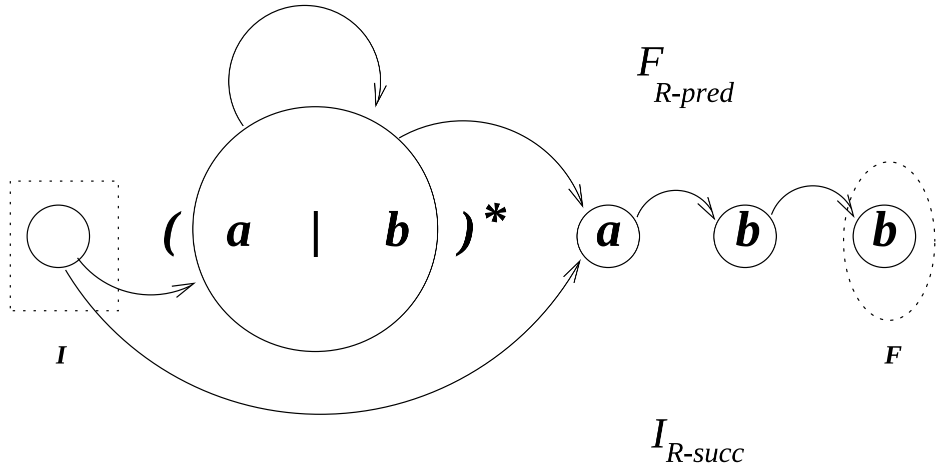


Figure 10: A CNNFA equivalent to regular expression $(a|b)^*abb$

pattern	TNFA	opt. TNFA	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	60% faster	2 times faster

Figure 11: CNNFA acceptance test speedup ratio

pattern	TNFA	TNFA (kernel)	opt. TNFA	MYNFA
$(abc\cdots)^*$	5 times faster	comparable	comparable	comparable
$(a\|b\|\cdots)^*$	quadratic speedup	linear speedup	linear speedup	linear speedup
$(0\|1\cdots\|9)^n$	70 times faster	10 times faster	20% faster	10 times faster
$((a\|\lambda)(b\|\lambda)\cdots -)^*$	linear speedup	20% faster	linear speedup	5% faster
$((a\|\lambda)(b\|\lambda)\cdots)^*$	quadratic speedup	linear speedup	linear speedup	linear speedup
$(a\|b)^*a(a\|b)^n$	2.5 times faster	comparable	10% slower	50% faster
prog. lang.	800 times faster	6 times faster	40% faster	6 times faster

Figure 12: CNNFA subset construction speedup ratio

Our acceptance testing experiments show that the CNNFA outperforms Thompson's NFA, Thompson's optimized NFA, and McNaughton and Yamada's NFA. See Fig. 11 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. This is an anomalous shortcoming of our current implementation, which can be overcome easily.

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 on the following five starting machines: the CNNFA, Thompson's NFA, Thompson's optimized NFA, Thompson's NFA using the kernel items heuristic[2], and McNaughton and Yamada's NFA.

We implemented subset construction tailored to the CNNFA and other machines. The only differences in these implementations is in the calculation of $\delta_R(V, \Sigma)$, where we use the efficient procedure described by Theorem 5.5 for the CNNFA, and needed the standard procedure for traversing λ edges for Thompson's NFA and Thompson's optimized NFA. The CNNFA achieves linear speedup and constructs a linearly smaller DFA in many of the test cases. See Fig.'s 12 and 13 for benchmark summaries. The raw timing data is found in [9]. All the tests described in this paper are performed on a lightly loaded SUN 3/250 server. We used `getitimer()` and `setitimer()` primitives[23] to measure program execution time. It is interesting to note that the CNNFA has a better speedup ratio on SUN Sparc-based computers.

Recently at Columbia University's Theory Day, Aho reported a highly efficient heuristic for deciding whether a given string belongs to the language

pattern	TNFA	TNFA (kernel)	opt. TNFA	MYNFA
$(abc\cdots)^*$	comparable	comparable	comparable	comparable
$(a\|b\|c\cdots)^*$	linearly smaller	linearly smaller	comparable	linearly smaller
$(0\|1\|\cdots 9)^n$	200 times smaller	10 times smaller	comparable	10 times smaller
$((a\|\lambda)(b\|\lambda)\cdots -)^*$	3 times smaller	comparable	comparable	comparable
$((a\|\lambda)(b\|\lambda)\cdots)^*$	linearly smaller	linearly smaller	comparable	linearly smaller
$(a\|b)^*a(a\|b)^n$	4 times smaller	comparable	comparable	comparable
prog. lang.	10 times smaller	5 times smaller	comparable	5 times smaller

Figure 13: DFA size improvement ratio starting from the CNNFA

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 as *egrep*. Aho's heuristic constructs McNaughton and Yamada's NFA first, and subsequently builds a DFA specialized to the input string incrementally as the string is scanned from left to right. Benchmarks showing substantial computational improvement in adapting the CNNFA to Aho's heuristic are found in [9].

7 Conclusion

Theoretical analysis and confirming empirical evidence demonstrates that our proposed CNNFA leads to a substantially more efficient way of turning regular expressions into DFA's than other NFA's in current use. It would be interesting future research to analyze the effect of promotion and useless node elimination on the CNNFA. It would also be worthwhile to obtain a sharper analysis of the constant factors in comparing the CNNFA with other NFA's.

8 Acknowledgement

We are grateful for helpful comments from Deepak Goyal, Anne Brüggemann-Klein, and Ravi Sethi.

A Appendix. CNNFA Construction In $O(s)$ Auxiliary Space

In order to reduce the auxiliary space from $O(r)$ to $O(s)$ in CNNFA construction, we will parse regular expressions so that the stack will only grow when an alphabet symbol is encountered. Our approach depends on distinguishing between regular expressions R_C formed by product, and regular expressions R_D formed by alternation. We also distinguish between regular expressions with and without occurrences of alphabet symbols. That is, regular expressions R_C

(respectively R_D) are partitioned into expressions A_C (respectively A_D) that contain occurrences of alphabet symbols and expressions L_C (respectively L_D) that do not. The following grammar with start symbol Z generates delimited regular expressions in terms of the syntactic categories just mentioned:

$$\begin{array}{rcl} Z &=& \text{`\%'} R_D \text{`\%'} \\ R_D &=& L_D \mid A_D \\ R_C &=& L_C \mid A_C \\ L_D &=& L_C \mid L_D \text{`}|\text{'} L_D \\ L_C &=& \lambda \mid L_C \text{`*'} \mid L_C L_C \mid \text{`('} L_D \text{`)'} \\ A_D &=& A_C \mid L_D \text{`}|\text{'} A_D \mid A_D \text{`}|\text{'} R_D \\ A_C &=& A \mid L_C A_C \mid A_C R_C \\ A &=& a \mid A \text{`*'} \mid \text{`('} A_D \text{`)'} \end{array} \tag{37}$$

where a represents an alphabet symbol, and literal symbols are quoted to distinguish them from meta-symbols.

Let *input* be a sequence of symbols storing the delimited regular expression $\%R\%$, where the first symbol of R is stored at the second *position* denoted by $input(2)$. Our algorithm consists of three procedures that read but never modify *input*.

Procedure $right\_re(barrier)$ accepts a single parameter *barrier*, which is the leftmost position of the longest subexpression P of R of the form (either R_C or R_D) uniquely determined by the context in which this procedure is called. If the symbol just to the left of *barrier* is either % or |, then the context is either $\%P\%$ or A_D'|'P, where P is of the form R_D. Otherwise, the symbol just to the left of *barrier* must be '*', ')', or 'a', and the context is $A_C P$, where P is of the form R_C. Procedure $right\_re(barrier)$ returns record N_P representing the tail of the CNNFA for P together with the rightmost position of P. The tail of the CNNFA for input R is computed by the top-level function call $right\_re(2)$.

In operational terms procedure $right\_re(barrier)$ scans and validates subexpression P (determined by *barrier*) from left to right. It begins by assigning *barrier* - 1 to *rpos*, which represents the rightmost position in P that has been scanned. It also sets the parentheses count *pcount* to 0, and sets the current state to $S1$. After that, it scans symbols of P from left to right (starting with position *barrier*), and takes actions depending on the character *rsymb* (the next character on the right to be scanned) stored in position $input(rpos + 1)$ and a possible condition as specified in Table 14.

If no alphabet symbol is encountered during scanning, then P is equivalent to λ, and N_λ is returned along with the rightmost position of P. Otherwise, the first time an alphabet symbol a is encountered, say at position *pos*, function $expand\_a(barrier, pos)$ is called to compute and return record N_P and the rightmost position of P, which is also returned by $right\_re(barrier)$. If the procedure fails to terminate (i.e., it gets stuck), then input R is invalid.

rsymb	*condition*	*action*
		State S1
(		$rpos$ +:= 1; $pcount$ +:= 1
a		*return* $expand\_a(barrier, rpos+1)$
λ		$rpos$ +:= 1; *goto State S2*
		State S2
[)%] |	$pcount = 0$ *or* ($syntyp = C$ *and* $pcount = 0$)	*return* $[N_\lambda, rpos]$
)	$pcount \neq 0$	$rpos$ +:= 1; $pcount$ -:= 1
*		$rpos$ +:= 1
|	($pcount \neq 0$ *or* $syntyp = D$)	$rpos$ +:= 1; *goto State S1*
$[a\lambda(]$		*goto State S1*

Figure 14: $right\_re(barrier)$

Procedure $expand\_a(barrier, pos)$ does most of the work computing the CN-NFA. Its first parameter $barrier$ defines the context P for which N_P is computed. This procedure begins by computing record N_a for symbol a in the initial one-symbol context at position pos. It then recomputes the single record N_A as context A is repeatedly extended until the context becomes P. If N_A is the current record representing context A, then the following are the different ways of extending A. Because of the way that $right\_re(barrier)$ is called, $expand_a$ cannot move outside of the context P determined by $barrier$. We use the notation [(|λ] to denote a single symbol belonging to symbols listed between square brackets. The notation [^*] denotes any symbol but *.

1. (case: A *) Compute N_{A*} from N_A using Rules (27) and (34), and extend the context to $A*$.

2. (case: (A)) Leave the current record alone, but extend the context to (A).

3. (case: A $[(a\lambda]$) A must be of the form A_C in the context AR_C. If $rpos$ is the position just to the right of A, then execute the call $right\_re(rpos)$, which returns N_{R_C}. Compute N_{AR_C} using Rules (26) and (33) and extend the context to AR_C.

4. (case: [(|%] A |) A must be of the form A_D in the context of $A|R_D$. If $rpos$ is the position just to the right of A, then execute the call $right\_re(rpos{+}1)$, which returns N_{R_D}. Compute $N_{A|R_D}$ using Rules (25) and (32), and extend the context to $A|R_D$.

5. (case: $[)*\lambda]$ A [^*], where A doesn't begin at $barrier$) A must be of the form A_C in the context of $L_C A$, where L_C is restricted to be the

longest subexpression that doesn't extend to the left of *barrier*. If *lpos* is the position just to the left of A, then execute our third procedure $left\_\lambda(barrier, lpos)$ (which finds L_C), and extend the context to $L_C A$. Record N_A remains unchanged, because $\lambda A = A$.

6. (case: $\mid A$ [)|%], where A doesn't begin at *barrier*) A must be of the form A_D in the context of $L_D|A$, where L_D is restricted to be the longest subexpression that doesn't extend to the left of *barrier*. If *lpos* is the position just to the left of A, then execute $left\_\lambda(barrier, lpos-1)$ to locate L_D, compute $N_{\lambda|A}$ using Rules (25)and (32), and extend the context to $L_D|A$.

When the context can no longer be extended according to the preceding cases, then the final context A is correct if it begins at *barrier*, and its surrounding symbols are either (1) % A % (where N_A represents the tail CNNFA for R), (2) $\mid A$ [)%] (where A is of the form A_D), or (3) [)*λa] A [|%] (where A is of the form A_C). In these cases return record N_A and the rightmost position of A. Otherwise, regular expression R is invalid.

The simplest procedure is $left\_\lambda(barrier, pos)$. Whenever it is called by *expand_a*, it returns a maximal subexpression of the form either L_C or L_D (depending on the symbol that occurs in $input(pos+1)$) whose right boundary is at position *pos*, and whose left boundary doesn't extend to the left of *barrier*. It begins by setting the leftmost position *lpos* of the scanned context to *pos*, and assigning zero to parentheses count *pcount*. We also set variable *syntyp* to D if $input(pos-1) =$ '|', which indicates that we are looking for a subexpression of the form L_D. Otherwise, we assign C to *syntyp*, which indicates that we are looking for a subexpression of the form L_C. After initialization, the procedure scans symbols from right to left, taking actions depending on the character *lsymb* in position $input(lpos-1)$ and a condition according to Table 15. Within Table 15 the term *any* means any symbol. Observe that no validation is necessary, since the symbols that are found have already been validated by a right-to-left scan within *right_re*.

lsymb	*condition*	*action*
)	$lpos \neq barrier$	*lpos* -:= 1; *pcount* +:= 1
| *any* (	$(syntyp = C$ *and* $pcount = 0)$ *or* $lpos = barrier$ *or* $pcount = 0$	return *lpos*
(	$pcount \neq 0$	*lpos* -:= 1; *pcount* -:= 1
	otherwise	*lpos* -:= 1;

Figure 15: $left\_\lambda(barrier, pos)$

The intuition behind the $O(r)$ time $O(s)$ auxiliary space of the preceding algorithm stems from the fact that regular expression R is scanned once from left to right by calls to *right_re* and *expand_a*. It is scanned no more than once from right to left by calls to *left_*λ and *expand_a*. No more than unit time is spent scanning each symbol, and no more than unit space is consumed for each alphabet symbol. The depth of recursive calls is restricted to $O(s)$, because *expand_a* is called only once for each occurrence of an alphabet symbol, and *right_re* can only be called once at the top level or from *expand_a*. These arguments are made more formally in the following lemma, which leads to Theorem 5.6.

Lemma A.1 *Let P be the context uniquely determined by barrier in the function call right_re(barrier). Let s_P be the number of alphabet symbol occurrences in P, and let r_P be the length of P. Then the call computes record N_P in time $O(r_P)$ and auxiliary space $O(1 + s_P)$.*

Proof: The proof uses rule induction[26] with respect to grammar rules (37). Whenever *right_re*(*barrier*) is called, the context P determined by position *barrier* must be of the form R_D in the greater contexts $\%R_D\%$ and $A_D|R_D[)\%]$, or of the form R_C in the greater context $A_C R_C[|)\%]$. We will only prove complexity here. The correctness proof is tedious but straightforward.

Referring to grammar (37), we see that R_D may be either L_D or A_D, and R_C may be either L_C or A_C. Since record N_A used to store the tail CNNFA in procedure *expand_a* takes $O(s_A)$ space by Theorem 5.2, then there is a constant $b_1 > 0$ such that $b_1 s_A$ bounds the space for N_A. Since both *left_*λ and the portion of *right_re* prior to its call to *expand_a* take unit space, then there is a constant $b_2 > 0$ that bounds both the cost of calling and executing *left_*λ and the cost of calling and executing *right_re* prior to the call to *expand_a*. Finally, there is a constant $b_3 > 0$ that bounds the unit space used by any invocation of *expand_a* without considering any of its calls or the storage of record N_A.

Letting $b = b_1 + b_2 + b_3$, we will prove that *right_re* uses space bounded by $b_2 + bs_P$ and time $O(r_P)$. When P is of the form L_D or L_C (so that $s_P = 0$) it is easy to see that *right_re*(*barrier*) scans each symbol of P from left to right in linear time and that b_2 bounds the space. It remains to show that the lemma holds when P is of the form A_C or A_D.

First consider when P is of the form A_C. According to grammar (37) A_C must be of the form A, $L_C A_C$, or $A_C R_C$. First, suppose that $P = P_1 P_2$, where P_1 is of the form L_C and P_2 is of the form A_C. In this case function *right_re* carries out three tasks. First it scans symbols of P_1 from left to right in linear time and space bounded by b_2. Second, it proceeds to scan P_2 from left to right until it encounters an alphabet symbol occurrence. At this point *expand_a* is called, and record N_{P_2} is computed. This second task goes through the same steps as when *right_re* is called with context P_2. By induction this second task takes time $O(r_{P_2})$ and space bounded by $b_2 + bs_{P_2}$. At the end of the second task after N_{P_2} is computed (when the call tree contains only the top level call to *right_re* and its call to *exand_a*), the total space being used is bounded by

$b_2 + bs_{A_C} - b_2$. At this point we begin the third task, which is the right-to-left scan over P_1 when *expand_a* calls $left\_\lambda$. This task consumes b_2 additional units of space, and takes $O(r_{P_1})$ time. Procedure *left_lambda* releases its b_2 units of space on return, and *expand_a* returns record N_{P_2}, which equals $N_{\lambda P_2}$, and takes no more than $b_1 s_{P_2}$ space. Hence, the overall space bound is $b_2 + bs_P$.

Next, suppose that $P = P_1 P_2$, where P_1 is of the form A_C and P_2 is of the form R_C. In this case function *right_re* initially goes through the same steps as when P is P_1; that is, when the first alphabet symbol is encountered, *expand_a* is called to produce record N_{P_1}. These steps take time $O(r_{P_1})$ and space bounded by $b_2 + bs_{P_1}$ by induction. After N_{P_1} is computed, the total space being used is $b_2 + bs_{A_C} - b_2$. Subsequently, *expand_a* calls *right_re* to handle the inner expression P_2. If P_2 is of the form L_C, then b_2 additional units of space are consumed, and the overall space is bounded by $b_2 + bs_{P_2}$. If P_2 is of the form A_C, then this call takes time $O(r_{R_C})$, and space bounded by $b_2 + bs_{P_2}$ by induction. Record N_{P_2} is computed and returned with space bounded by $b_1 s_{P_2}$. Record N_{P_2} is combined with N_{P_1} by *expand_a* in unit time to produce $N_{P_1 P_2}$, which consumes no more than $b_1 s_P$ space. Thus, the overall space is bounded by $b_2 + bs_P$.

The last case needed to prove that the lemma holds for A_C (and, hence, R_C) is A. Grammar (37) allows A to be either an alphabet symbol a, $A*$, or (A_D). It is easy to prove the axiom that unit time and space are used when *right_re* is called in the context a. Straightforward analysis of *expand_a* is enough to prove the inductive argument in the other two cases.

Finally, in order to prove that the lemma holds for A_D, grammar (37) indicates that A_D must be either A_C, $L_D|A_D$, or $A_D|R_D$. Case A_C has already been taken care of. Proofs of the other two cases mirror the proofs for the earlier cases $L_C A_D$ and $A_C R_C$. □

References

[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*, Vol. 48, 1986, pp. 117-126.

[6] Brüggemann-Klein, A., "Regular Expressions into Finite Automata", *Theoretical Computer Science*, Vol. 120, 1993, 197-213.

[7] Brzozowski, J., "Derivatives of Regular Expressions", *JACM*, Vol. 11, Num. 4., Oct. 1964, pp. 481-494.

[8] Cai, J. and Paige, R., "Using multiset discrimination to solve language processing problems without hashing," *Theoretical Computer Science*, Vol. 145, 1995, 189-228.

[9] Chang, C., Ph. D. Thesis, "From Regular Expressions to DFA's Using Compressed NFA's", 1992.

[10] Chang, C. and Paige, R., "From Regular Expressions to DFA's Using Compressed NFA's," *LNCS Num. 644*, Springer-Verlag, eds. A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, 1992, pp. 88-108.

[11] Driscoll, J., Sarnak, N., Sleator, D., and Tarjan, R., "Making Data Structures Persistent", *Proc. 8th ACM STOC*, May, 1986, pp. 109 - 121.

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

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

[14] Kleene, S., "Representation of events in nerve nets and finite automata", in *Automata Studies, Ann. Math. Studies Num. 34*, Princeton U. Press, 1956, pp. 3 - 41.

[15] Knuth, D., "On the translation of languages from left to right", *Information and Control*, Vol. 8, Num. 6, 1965, pp. 607 - 639.

[16] McNaughton, R. and Yamada, H. "Regular Expressions and State Graphs for Automata", *IRA Trans. on Electronic Computers*, Vol. EC-9, Num. 1, Mar. 1960, pp 39-47.

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

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

[19] Rabin, M. and Scott, D., "Finite automata and their decision problems" *IBM J. Res. Develop.*, Vol. 3, Num. 2, Apr., 1959, pp. 114 - 125.

[20] Ritchie, D. and Thompson, K. "The UNIX Time-Sharing System" *Communications of the ACM*, Vol. 17, Num. 7, Jul., 1974, pp. 365 - 375.

[21] Sethi, R., private communication, 1989.

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

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

[24] Thompson, K., "Regular Expression search Algorithm", *Communications of the ACM*, Vol. 11, Num. 6, 1968, pp. 419-422.

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

[26] Winskel, G., "The Formal Semantics of Programming Languages", *MIT Press*, 1993.