Efficient Implementation of Regular Languages Using R-AFA

K. Salomaa, X. Wu and S. Yu

Department of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7

Abstract. We introduce a bit-wise representation of r-AFA transition functions and describe an efficient implementation method for r-AFA and their operations using this representation. Experiments have shown that this implementation is much more efficient than the Grail DFA implementation in both space and time.

1 Introduction

Regular languages and their representations, e.g., regular expressions and finite automata, have been used mainly for lexical analysis and string matching. However, in recent years, the applications of regular languages have been increasingly widening. For example, finite automata have been used in object-oriented modeling [10, 13], data and image compression [1, 6], parallel processing [9], and communication protocol specification.

In the applications such as object-oriented modeling, a critical problem in the use of deterministic finite automata (DFA) is that the number of states is too large to implement for many practical problems of a normal size. For example, the concatenation of a 10-state DFA and a 30-state DFA may result in a minimal DFA of 10 billion states [17]. This problem, which we call the *size-explosion problem* of DFA, has kept many ideas away from being realized in practice or only used for cases of a very limited size.

In order to solve or to avoid this size-explosion problem, let us consider several alternative forms of representations of regular languages.

It is known that an n-state nondeterministic finite automaton (NFA) is equivalent to a minimal 2^n-state DFA in the worst case. However, using NFA instead of DFA is not a solution to the size-explosion problem because (1) the problem of minimizing NFA state sets is PSPACE-complete [15, 11, 12], and the number of states of a minimal NFA may not be much smaller than the number of states of the equivalent minimal DFA (they can be equal in some cases); (2) the deterministic implementation of an NFA, e.g., using bit matrices, is very inefficient in both space and time, and most of the usual operations, e.g., intersection, are extremely awkward to implement for NFA, losing any advantage over DFA.

Regular expressions are convenient to be used as a form of user-interface. But this form of representation of regular languages is inherently unsuitable to be

used directly as a control device. In many applications, regular expressions are normally read as inputs and then transformed into certain forms of automata.

It has been shown that an n-state alternating finite automaton (AFA) is equivalent to a 2^n-state minimal NFA and a 2^{2^n}-state minimal DFA in the worst case. But, alternation is an even more general form of nondeterminism. Any implementation of AFA on real-world computers, which are deterministic machines, will lose all the conciseness and efficiency of the AFA representation.

However, a nice feature of AFA is that they are backward deterministic, i.e., they are deterministic if we consider them working on an input string from the rightmost letter to the leftmost. It has been shown that a language L is accepted by an n-state DFA if and only if the reversal of L, i.e., L^R, is accepted by a $\log n$-state AFA. So, the use of r-AFA (reversed AFA) instead of DFA guarantees a logarithmic reduction in the number of states. In addition, operations such as union, intersection, and complementation for r-AFA are much simpler and more efficient to implement than the corresponding DFA operations. All these factors make r-AFA an ideal candidate for an efficient implementation of regular languages and their operations. A problem remaining is that the transition function of an r-AFA (or an AFA) can be very complex. Expressed in the form of Boolean expressions, the size of the transition function of an r-AFA, in the worst case, can be in the order of 2^n, where n is the number of states. However, our experiments have shown that the average size of a transition function occurring in practice is much smaller than that. We believe, but have not formally proved, that in the average case the size of the transition function of an r-AFA is polynomial in the number of states of the r-AFA.

In this paper, we introduce an implementation method for r-AFA transition functions using bit-vectors. We outline an algorithm for the transformation from a DFA to an equivalent r-AFA. The algorithm minimizes the number of terms or minterms in the Boolean expression representing the r-AFA transition functions in each of the major steps. We also describe an algorithm for transforming an r-AFA to an equivalent DFA. Union, intersection, and complementation operations on r-AFA are also described. Our test results, which presented in Figure 1, show that the bit-wise r-AFA representation can be used for efficient implementations of regular languages and their operations.

2 Basic notations and definitions

The concept of alternating finite automata (AFA) was introduced in [3] and [2] at the same period of time under different names. A more detailed treatment of AFA operations can be found in [8]. In this paper, we introduce h-AFA and r-AFA, which are modified versions of AFA. An h-AFA is an s-AFA of [17] with a generalized acceptance condition. An r-AFA is defined in this paper exactly as an h-AFA except that it reads its input reversely. We use r-AFA for efficient implementations of regular languages and their operations.

An h-AFA A is a quintuple (Q, Σ, g, h, F), where

Q is the finite set of states,

Σ is the input alphabet,

$g : Q \times \Sigma \times B^Q \to B$ is the transition function, where B is the two-element Boolean algebra, B^Q refers to all the Boolean functions from Q to B,

$h : B^Q \to B$ is the accepting Boolean function,

$F \subseteq Q$ is the set of final states.

We use $g_q : \Sigma \times B^Q \to B$ to denote that g is restricted to state q, i.e., $g_q(a, u) = g(q, a, u)$, for $a \in \Sigma$, $u \in B^Q$, $q \in Q$.

We also use g_Q to denote the function from $\Sigma \times B^Q$ to B^Q that is obtained by combining the functions g_q, $q \in Q$, i.e.,

$$g_Q \equiv (g_q)_{q \in Q}.$$

We will write g instead of g_Q whenever there is no confusion.

Let $u \in B^Q$. We use $u(q)$ or u_q, $q \in Q$, to denote that u is restricted to q.

We extend the definition of g of an h-AFA to a function: $Q \times \Sigma^* \times B^Q \to B$ as follows:

$g(q, \lambda, u) = u_q$

$g(q, a\omega, u) = g(q, a, g(\omega, u))$

for all $q \in Q, a \in \Sigma, \omega \in \Sigma^*, u \in B^Q$.

Similarly, the function g_Q, or simply g, is extended to a function $\Sigma^* \times B^Q \to B^Q$.

Given an h-AFA $A = (Q, \Sigma, g, h, F)$, for $w \in \Sigma^*$, w is accepted by A if and only if $h(g(w, f)) = 1$, where f is the characteristic vector of F, i.e., $f_q = 1$ iff $q \in F$.

An r-AFA A is an h-AFA such that for each $w \in \Sigma^*$, w is accepted by A if and only if $h(g(w^R, f)) = 1$, where f is the characteristic vector of F.

An example of an h-AFA is given below. Note that for each $q \in Q$, we use q also to denote the Boolean value associated with it.

Example 1. We define an h-AFA $A = (Q, \Sigma, g, h, F)$ where $Q = \{q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$, $h(q_1, q_2) = q_1 \wedge \overline{q_2}$, and g is given by

State	a	b
q_1	$q_1 \vee \overline{q_2}$	$q_1 \wedge \overline{q_2}$
q_2	$\overline{q_1} \wedge \overline{q_2}$	$\overline{q_1} \vee \overline{q_2}$

Note that we use $\overline{q}$ instead of $\neg q$ for convenience.

The above table can be denoted by the following system of equations:

$$\begin{cases} X_1 = a \cdot (X_1 \vee \overline{X_2}) + b \cdot (X_1 \wedge \overline{X_2}) \\ X_2 = a \cdot (\overline{X_1} \wedge \overline{X_2}) + b \cdot (\overline{X_1} \vee \overline{X_2}) + \lambda \end{cases}$$

where a variable X_i represents the state q_i, $i = 1, 2$, and λ appearing in the second equation specifies that q_2 is a final state.

Let $w = aaab$. Then w is accepted by A as follows:

$$
\begin{aligned}
& h(g(aaab, f)) \\
= & h(g(a, g(aab, f))) \\
= & h(g(a, g(a, g(ab, f)))) \\
= & h(g(a, g(a, g(a, g(b, f))))) \\
= & h(g(a, g(a, g(a, g(b, (0,1)))))) \\
= & h(g(a, g(a, g(a, (0,1))))) \\
= & h(g(a, g(a, (0,0)))) \\
= & h(g(a, (1,1))) \\
= & h((1,0)) \\
= & 1
\end{aligned}
$$

If we define A as an r-AFA rather than an h-AFA, then $w^R = baaa$ is accepted by A. □

The following results can proved similarly as in in [3, 8, 17].

Theorem 1 *If L is accepted by an n-state r-AFA, then L is accepted by a DFA with at most 2^n states.*

Theorem 2 *If L is accepted by an n-state complete DFA, then L is accepted by an r-AFA with at most $\lceil \log n \rceil$ states.*

3 A bit-wise representation of r-AFA

The transition functions of h-AFA and r-AFA are denoted by Boolean functions. In this section, we consider a bit-wise representation of Boolean functions, which is much more convenient to implement than any symbolic representation of Boolean functions, as well as more efficient in both space and time.

Given a set of independent Boolean variables

$$S = \{x_1, x_2, \ldots, x_n\},$$

and its dual set

$$\overline{S} = \{\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n}\},$$

we have the following definitions:

Definition 1 *A term t defined on $S \cup \overline{S}$ is a conjunction*

$$t = y_1 \wedge \ldots \wedge y_k, \quad 1 \leq k \leq n$$

where $y_i \in S \cup \overline{S}$, $y_i \neq y_j$, $y_i \neq \overline{y_j}$, for $1 \leq i < j \leq k$, or t is a constant, i.e., $t = 1$ or $t = 0$. The conjunction of $y_1, \ldots, y_k$ is denoted also simply as $y_1 \cdots y_k$. A term t is called a minterm if $t = y_1 \cdots y_n$ where y_i is either x_i or $\overline{x_i}$, $1 \leq i \leq n$.

Definition 2 *A Boolean expression f is said to be in disjunctive normal form if $f = \bigvee_{i=1,\ldots,k} t_i$, where t_i, $i = 1, \ldots, k$, is a term defined on $S \cup \overline{S}$.*

Theorem 3 *For every Boolean function f defined on S that can be expressed as a single term, there exist two n-bit vectors α and β such that*

$$f(u) = 1 \Longleftrightarrow (\alpha \& u) \uparrow \beta = \mathbf{0}, \quad \textit{for all } u \in B^n,$$

where & is the bit-wise AND operator, $\uparrow$ the bit-wise EXCLUSIVE-OR operator, and $\mathbf{0}$ is the zero vector $(0, \ldots, 0)$ in B^n.

Proof. First, we consider the cases where f is not a constant function. By the given condition, $f = y_{i_1} \wedge \ldots \wedge y_{i_k}$, $1 \leq k \leq n$, where y_{i_j} is x_{i_j} or $\overline{x_{i_j}}$, $i_j \neq i_{j'}$ if $j \neq j'$. Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$, where

$\alpha_k = 1 \Longleftrightarrow x_k$ or $\overline{x_k}$ appears in f,

$\beta_k = 1 \Longleftrightarrow x_k$ appears in f,

for $1 \leq k \leq n$.

Let us denote $\alpha \& u$ by t. Then

$$t_k = 1 \Longleftrightarrow u_k = 1 \text{ and } (x_k \text{ or } \overline{x_k} \text{ appears in} f)$$

i.e., $t_i = u_i$ if variable x_i or $\overline{x_i}$ appears in f and $t_i = 0$ otherwise.

Let $t' = t \uparrow \beta$. Then $t'_i = \overline{t_i}$ if x_i appears in f and $t'_i = t_i$ otherwise. So, (1) if neither x_i nor $\overline{x_i}$ appears in f, $t'_i = 0$; (2) if x_i appears in f, then $t'_i = 0$ iff $x_i = 1$; (3) if $\overline{x_i}$ appears in f, then $t'_i = 0$ iff $x_i = 0$.

Therefore, $f(u) = 1 \Longleftrightarrow (\alpha \& u) \uparrow \beta = \mathbf{0}$.

Now we consider the cases where f is a constant function. If $f = 1$, we can choose $\alpha = \mathbf{0}, \beta = \mathbf{0}$. If $f = 0$, we define $\alpha = \mathbf{0}, \beta = \mathbf{1}$. □

Each n-bit vector $v = (v_1, \ldots, v_n)$, $n \leq 32$ (32-bit is the normal size of a word), can be represented as an integer

$$I_v = \sum_{i}^{n} v_i 2^{i-1}.$$

We can also transform an integer I_v back to a 32-bit vector v in the following way:

$$v_i = (I_v \& 2^{i-1})/2^{i-1}, \quad 1 \leq i \leq n.$$

Note that since a language is accepted by an n-state DFA iff it is accepted by a $\log n$-state r-AFA, a 32-state AFA is equivalent to a 2^{32}-state DFA, which is normally big enough for any practical purpose.

A Boolean function, which is in disjunctive normal form, can be represented as a list of terms, while each term is represented by two integers and a link (pointer).

For an r-AFA $A = (Q, \Sigma, g, h, F)$, where $Q = \{q_1, \ldots, q_n\}$ and $\Sigma = \{a_1, \ldots, a_m\}$, we can represent g as a table of functions of size $n \times m$ with (i, j) entry corresponding to the function $g_{q_i}(a_j) : B^Q \rightarrow B$ defined by $[g_{q_i}(a_j)](u) = g_{q_i}(a_j, u)$, for $q_i \in Q$, $a_j \in \Sigma$, and $u \in B^Q$.

4 Transformation between DFA and r-AFA

In this section, we show how to transform an n-state DFA into an equivalent $\log n$-state r-AFA, and also show the transformation in the opposite direction. Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA. Assume that $Q = \{p_1, \ldots, p_n\}$. We construct an r-AFA $A' = (Q', \Sigma, g, h, F')$ as follows:

First, let's assume that $n > 1$.

(1) Find the minimal integer m such that $n \leq 2^m$, i.e., $m = \lceil \log n \rceil$. Let $d = 2^m - n$. Define $Q' = \{q_1, \ldots, q_m\}$ to be the state set of the target r-AFA.

(2) Let n_f be the number of final states of A, i.e., $n_f = |F|$. Find the number n'_f such that $n_f \leq n'_f \leq n_f + d$ and n'_f has the smallest number of 1's in its binary form. Assume that n'_f has l, $l \leq m$, positions holding a 1 and $i_1, \ldots, i_l$ are those positions. Then we define the Boolean function h for A':

$$h(q_1, \ldots, q_m) = \overline{q_1} \cdots \overline{q_{i_1-1}} q_{i_1} \vee \ldots \vee \overline{q_1} \cdots \overline{q_{i_l-1}} q_{i_l}.$$

Denote by $P_F \subseteq B^{Q'}$ the set $\{u \in B^{Q'} \mid h(u) = 1\}$ and by P_N the set $B^{Q'} - P_F$. Note that the cardinality of P_F is n'_f.

(3) Sort P_F and P_N, respectively, in the ascending order of the number of 1's in each element. Sort F and $Q - F$, respectively, in the descending order of the number of incoming transitions of each state. Define a one-to-one mapping $\rho : Q \to B^{Q'}$ by matching the ith element of the sorted F to the ith element of the sorted P_F, $1 \leq i \leq n_f$, and the jth element of the sorted $Q - F$ to the jth element of the sorted P_N, $1 \leq j \leq n - n_f$.

(4) Define $F' = \{q_i \mid \rho(s)_{q_i} = 1\}$.

(5) For each $a \in \Sigma$ and i, $1 \leq i \leq m$, define $S_i^a = \{p \in Q \mid (\rho(\delta(p, a)))_{q_i} = 1\}$. We also define a mapping θ from $B^{Q'}$ to Boolean terms by $\theta(u) = \bigwedge_{q \in Q'} t(u_q)$, where $t(u_q) = q$ if $u_q = 1$ $t(u_q) = \overline{q}$ if $u_q = 0$. Then we define $g(q_i, a, (q_1, \ldots, q_m)) = \bigvee_{p \in S_i^a} \theta(\rho(p))$, $q_i \in Q'$ and $a \in \Sigma$ if $S_i^a \neq \emptyset$, or $g(q_i, a, (q_1, \ldots, q_m)) = 0$ otherwise.

(6) Simplify each Boolean function by a known algorithm, e.g., the Quine-McCluskey two-level minimization algorithm [5].

If $n = 1$, we can define that the state set of the AFA is empty, and the head function is either constant 1 or constant 0 according to whether the only state of the DFA is a final state or not.

We omit the proof of correctness for the algorithm, i.e., for that the resulting r-AFA A' is equivalent to the given DFA A.

It can be shown that the above algorithm achieves the following:

- It minimizes the number of terms in the Boolean function h when it is represented in disjunctive normal form.
- Under the definition of h, it also minimizes the total number of minterms in g represented in disjunctive normal form in Steps (3) and (5).

However, the algorithm does not necessarily minimize the total size of the resulting r-AFA.

The transformation from a given r-AFA to an equivalent DFA is straightforward. We outline the algorithm in the following without giving any proof.

Let $A_A = (Q_A, \Sigma, g, h, F_A)$ be an r-AFA. We construct a DFA $A_D = (Q_D, \Sigma, \delta, s, F_D)$ in the following. Note that each state in Q_D is a Boolean vector in B^{Q_A}. $NewQ$ is a temporary variable to store new states of Q_D.

(i) Initially, we set $s = f$, where f is the characteristic vector of F_A, $Q_D = \emptyset$, $F_D = \emptyset$, and $NewQ = \{s\}$.
(ii) While $NewQ \neq \emptyset$ do the following:
Choose $u \in NewQ$.
$Q_D = Q_D \cup \{u\}$.
If $h(u) = 1$, then $F_D = F_D \cup \{u\}$.
For each $a \in \Sigma$, define $\delta(u, a) = g(a, u)$.
If $g(a, u) \notin Q_D$, $NewQ = NewQ \cup \{g(a, u)\}$.
$NewQ = NewQ - \{u\}$.

5 Other operations of r-AFA

The operations of union, intersection, and complementation on regular languages using r-AFA are straightforward.

Let $A^{(1)} = (Q^{(1)}, \Sigma, g^{(1)}, h^{(1)}, F^{(1)})$ and $A^{(2)} = (Q^{(2)}, \Sigma, g^{(2)}, h^{(2)}, F^{(2)})$ be two r-AFA. We assume that $Q^{(1)} \cap Q^{(2)} = \emptyset$.

The union of $L(A^{(1)})$ and $L(A^{(2)})$ is accepted by the r-AFA $A = (Q, \Sigma, g, h, F)$ where

$$Q = Q^{(1)} \cup Q^{(2)};$$

$$g_q(a, u) = \begin{cases} g_q^{(1)}(a, u|_{Q^{(1)}}), & \text{if } q \in Q^{(1)}, \\ g_q^{(2)}(a, u|_{Q^{(2)}}), & \text{if } q \in Q^{(2)} \end{cases}$$

for $q \in Q$, $a \in \Sigma$, and $u \in B^Q$, where $u|_P$ denotes that u is restricted to P;

$$h = h^{(1)} \vee h^{(2)};$$

$$F = F^{(1)} \cup F^{(2)}.$$

The intersection operation is the same as above except that $h = h^{(1)} \wedge h^{(2)}$. The complementation operation can be implemented by changing h to its complement.

The concatenations and star operations are more involved. We omit them due to the limitation on the length of the paper.

References

1. J. Berstel and M. Morcrette, "Compact representation of patterns by finite automata", *Pixim 89: L'Image Numérique à Paris*, André Gagalowicz, ed., Hermes, Paris, 1989, pp.387-395.

2. J.A. Brzozowski and E. Leiss, "On Equations for Regular Languages, Finite Automata, and Sequential Networks", *Theoretical Computer Science* 10 (1980) 19-35.
3. A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, "Alternation", *Journal of the ACM* 28 (1981) 114-133.
4. H.K. Cheung, *An Efficient Implementation Method for Alternating Finite Automata*, MSc Project Paper, Dept. of Computer Science, Univ. of Western Ontario, Sept. 1996.
5. Olivier Coudert, "Two-Level Logic Minimization: An Overview", *The VLSI Journal* 17 (1994) 97-140.
6. K. Culik II and J. Kari, "Image Compression Using Weighted Finite Automata", *Computer and Graphics*, vol. 17, 3, (1993) 305-313.
7. A.Fellah, *Alternating Finite Automata and Related Problems.* Ph.D dissertation, Kent State Univ. 1991.
8. A.Fellah, H.Jürgensen, and S.Yu, "Constructions for Alternating Finite Automata", *Intern J. Comp. Math.* 35 (1990) 117-132.
9. L. Guo, K. Salomaa, and S. Yu, "Synchronization Expressions and Languages", *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing* (1994) 257-264
10. D. Harel, "Statecharts: a visual formalism for complex systems", *Science of Computer Programming* 8 (1987) 231-274.
11. H.B. Hunt, D.J. Rosenkrantz, and T.G. Szymanski, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages", *Journal of Computer and System Sciences* 12 (1976) 222-268.
12. T. Jiang and B. Ravikumar, "Minimal NFA Problems are Hard", *SIAM Journal on Computing* 22 (1993) 1117-1141.
13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
14. D. Raymond and D. Wood, *Release Notes for Grail Version 2.5*, Dept. of Computer Science, Univ. of Western Ontario, 1996.
15. L. Stockmeyer and A. Meyer, "Word problems requiring exponential time (preliminary report)", *Proceedings of the 5th ACM Symposium on Theory of Computing*, (1973) 1-9.
16. D. Wood, *Theory of Computation*, John Wiley & Sons, New York, 1987.
17. S. Yu, "Regular Languages", Chapter 2, *Handbook of Formal Languages*, vol. 1, Springer 1997.

6 Appendix

File Names		File Size		DFA:r-AFA
DFA	r-AFA	DFA	r-AFA	
testfm0	bitafa0	174	83	2.10
testfm1	bitafa1	348	156	2.23
testfm8	bitafa8	867	271	3.20
testfm21	bitafa21	1446	518	2.79
testfm9	bitafa9	1686	386	4.37
testfm10	bitafa10	2118	483	4.38
testfm13	bitafa13	4930	1178	4.18
testfm24	bitafa24	5832	1291	4.52
testfm17	bitafa17	6221	1629	3.82
testfm31	bitafa31	10546	2026	5.21
testfm18	bitafa18	10729	2166	4.95
testfm32	bitafa32	12487	2608	4.79
testfm15	bitafa15	15124	1603	9.43
testfm26	bitafa26	17020	2374	7.17
testfm27	bitafa27	20020	2685	7.46
testfm33	bitafa33	21058	4675	4.50
testfm25	bitafa25	21354	2396	8.91
testfm28	bitafa28	23170	3199	7.24
testfm30	bitafa30	46460	4080	11.38
testfm29	bitafa29	47760	4251	11.24
testfm36	bitafa36	58497	7191	8.31
testfm34	bitafa34	66208	6852	9.66
testfm35	bitafa35	108694	13833	7.86
testfm38	bitafa38	131022	16807	7.80

Fig. 1. File size comparison