A Simple Way to Construct NFA with Fewer States and Transitions

Guangming Xing
Department of Computer Science
Western Kentucky University
Bowling Green, KY 42101
Guangming.Xing@wku.edu

ABSTRACT

The problem of converting a regular expression to NFA is a fundamental problem that has been well studied. However, the two basic construction algorithms: (1) Thompson, (2) McNaughton-Yamada and Glushkov, does not yield the best solution in terms of the number of states and transitions. In this paper, we show: For a regular expression with l literals, we can construct an NFA with $2l$ states and $4l$ transitions in the worst case. Our algorithm runs in linear time with respect to the length of the regular expression. This improves the construction algorithm given by Chang in [5], which constructs an NFA with $5l/2$ states and $(10l-5)/2$ transitions in the worst case. The method presented here is much simpler and easier to understand, as we use only naive ideas: divide and conquer.

1. INTRODUCTION

The construction of finite automata from regular expressions is of central importance to string pattern matching [1, 10], lexical scanning [3], content-based network service [8], and computational biology [10].

There are two basic methods converting a regular expression to an NFA, one is due to Thompson [11] and the other is due to McNaughton and Yamada [9] and Glushkov [7]. Based on these two constructions, many papers were published reporting the optimizing techniques for improvement. Chang presented an algorithm in [5] that computes the same NFA in the same asymptotic time $O(n)$ as Berry and Sethi [4], but it improves the auxiliary space to $O(l)$, where n is the length of the regular expression and l is the number of literals. In Chang's construction [5], the result has $5l/2$ states and $5l$ transitions. It is a version of McNaughton and Yamada's construction which he called CNNFA. And he proved that there are no more transitions than in Thompson's construction without optimization, and the CNNFA is more efficient than the Thompson's original construction for string matching. Antimirov gave a construction algorithm in [2] using partial derivatives. He showed that for a regular expression with l literals, the number of states is bounded by $l+1$ and the size of the NFA is smaller in some cases than those by McNaughton and Yamada [9], Chang [5], and Berry and Sethi [4].

This paper is organized as follows: First, a parsing algorithm is presented that can produce a parse tree with fewer nodes. Second, based on the parsing algorithm, it is shown that NFA with fewer states and transitions can be constructed by a simple methods.

2. DEFINITIONS AND TERMINOLOGY

We follow the same notations as used in [3]. By an *alphabet*, we mean a finite non-empty set of symbols. In this paper, we use Σ to denote an alphabet. If Σ is an alphabet, Σ^* denotes the set of all finite strings of symbols in Σ. The empty string is denoted by ϵ. Any subset of Σ^* is a *language* over Σ.

Definition 1 A *regular expression* over an alphabet Σ is defined as follows [3]:

1. ϵ, ϕ and a for each $a \in \Sigma$ are regular expressions denoting the regular language $\{\epsilon\}$, the empty set and $\{a\}$ respectively;
2. If r_1, r_2 are regular expressions denoting the languages L_1, L_2, respectively, then $(r_1 + r_2)$, $(r_1 r_2)$ and (r_1^*) are regular expressions, denoting $L_1 \cup L_2$, $L_1 L_2$ and L_1^*, which we call alternation, concatenation, and star, respectively;
3. All regular expressions can be defined by the above rules.

For each symbol in $\Sigma \cup \{\epsilon\}$ occurrence in a regular expression, we call it a *literal.*

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

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

where

1. S is the finite set of states of the control;
2. Σ is the alphabet from which input symbols are chosen;
3. δ is the *state transition function* which maps $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of S;

4. s_0 in S is the *initial state* of the finite control;
5. $F \subseteq S$ is the set of *final (or accepting) states.*

The function δ can be extended to a function $\hat{\delta}$ mapping $Q \times \Sigma^*$ to the set of subsets of Q as follows [3]:

1. $\hat{\delta}(q, \epsilon) = \{q\}$
2. $\hat{\delta}(q, wa) = \{p \mid \text{for some state} r \in \hat{\delta}(q, w), p \in \delta(r, a)\}$

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

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

We illustrate Thompson's method by Figure 1.

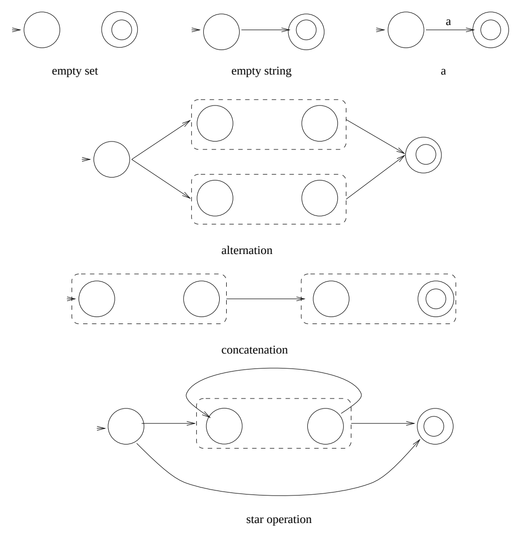

Figure 1: Thompson's Construction of NFA

It is easy to see that the number of states and transitions from above construction are linear with respect to the length of the regular expression, and this is usually true for the number of literals unless arbitrary number of Kleene stars are added to the regular expression.

In [3], it is shown that the NFA simulation time is $O(emn)$, where n is the length of the string, m is the number of the states and e is the upper bound of the number of transitions leaving each state with the same label. It is also shown that eliminating all auxiliary states and ϵ-transitions does not necessarily yield a better NFA for simulation, because although m gets smaller when we delete some auxiliary states, e may get larger.

As pointed in [6], shortest matching is useful for searching text formatted with markup languages, and they gave an algorithm in [6] for shortest matching that runs in time $O(emn)$, where e is the upper bound of $|\delta(q, a)|$, where $q \in Q$ and $a \in \Sigma \cup \{\epsilon\}$. So the number of states and transitions of an NFA is important for NFA simulation.

3. SMART PARSING

In this section, we show: From a regular expression with l literals, we can construct a parse tree with l leaf nodes (corresponding to the l literals in the regular expression), $(l - 1)$ alternation and concatenation nodes and at most l stars.

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

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

$$(A^*)^* = A^* \tag{1}$$

$$(A^*|B^*)^* = (A|B)^* \tag{2}$$

$$(A^*B^*)^* = (A|B)^* \tag{3}$$

$$(A^*|B)^* = (A|B)^* \tag{4}$$

It should be noted that the above list is not exhaustive, more rules can be added to the list by symmetry. Also, it should be noted that if we add more complicated properties, it is possible to get smaller parse trees at the expense of running time. Use the above properties, we can reduce the size of the parse tree as we will see later, but this optimization is local in the parse tree. It should be noted that that rewriting a regular expression to its simplest form is a PSPACE-complete problem.

Fig. 2 illustrates the difference between the smart parsing algorithm and the usual parsing algorithm.

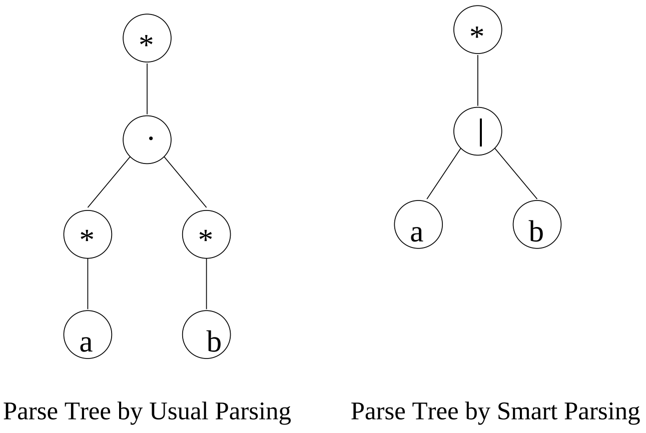

Figure 2: Parse Tree Comparison between Smart and Usual Parsing

3.1 Algorithm Description

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

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

1. a star node in the parse tree,

2. an alternation of a node with an empty string,
3. a concatenation of which both children are nullable,
4. an alternation of which at least one of the children is nullable.

A regular expression is nullable, if the root of its parse tree is nullable. It is easy to see that r is nullable iff $\epsilon \in L(r)$. And a node *non-nullable* if it is not nullable.

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

```
algorithm  $denull(root)$
Input: A Parse Tree
Output: A Parse Tree without Star over nullable node
    if not $nullable(root)$ then
        // do nothing
    else if $root.Op$ =‘*’ then
        $root = root.Child$
    else if $root.Op$ =‘.’ then
        // both Lchild and Rchild are nullable,
        // so $(Lchild.Rchild)^* = (Lchild|Rchild)^*$
        $root.Op$ =‘|’
        $denull(root.Lchild)$
        $denull(root.Rchild)$
    else if $root.Op$ = ‘|’ then
        $denull(root.Lchild)$
        $denull(root.Rchild)$
    fi
    return $root$
end
```

Algorithm 1 *denull* Procedure

Because the algorithm described above behaves smarter than the “usual” parsing algorithm, we call it the “smart” parsing algorithm. It is easy to show that Smart parsing produces a parse tree which is equivalent to the one produced by the usual parsing algorithm. Moreover, the new parse tree contains no star nodes with a nullable child.

For the whole paper, we are assuming that no ϵ occurs in a regular expression. To handle ϵ in a regular expression, we could add a flag, say `null`, to the node in the parse tree. Whenever we do an alternation between a node with ϵ, make `null` to be true unless the node is already nullable (because if r is nullable, then $r|\epsilon = r$); whenevr we do a concatenation between a node with ϵ, just return the node (because $r\epsilon = r$); whenever we do a star over ϵ, just return ϵ. By doing this, the result parse tree will not have any ϵ node except the case it is the only node in the tree.

3.2 Upper Bound on the Number of Nodes in the Parse Tree

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

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

PROOF. Let’s analyze the generation of the parse tree of a regular expression. There are three kinds of nodes in a parse tree:

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

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

By a standard property of trees, we know

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

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

Because we have at most l leaf nodes,

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

The key new property of our parse tree is: No star node has a nullable child. By using this property, we show that a tree with l leaves having l stars must be nullable.

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

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

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

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

Based on the above theorem, for a regular expression with l literals, we rewrite the regular expression from a parse tree. We add at most $(l-1)$ pairs of parentheses to make the regular expression unambiguous, and we have the following lemma about the property of a regular expression:

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

This is an improvement over the $14(l-1)+5$ upper bound based on the optimization algorithm proposed in Chang’s thesis [5]. It is also easier to understand.

4. CONSTRUCTION ALGORITHM

In this section, we give a construction method similiar to Thompson’s methods. It takes the parse tree from smart parsing, recursively constructs the NFA similiar to Thompson’s construction, but with different base cases and fewer states and transitions added in recursion step.

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

For 1-literal and 2-literal regular expressions (this is the case for small regular expressions), we construct NFAs as illustrated in Figure 3:

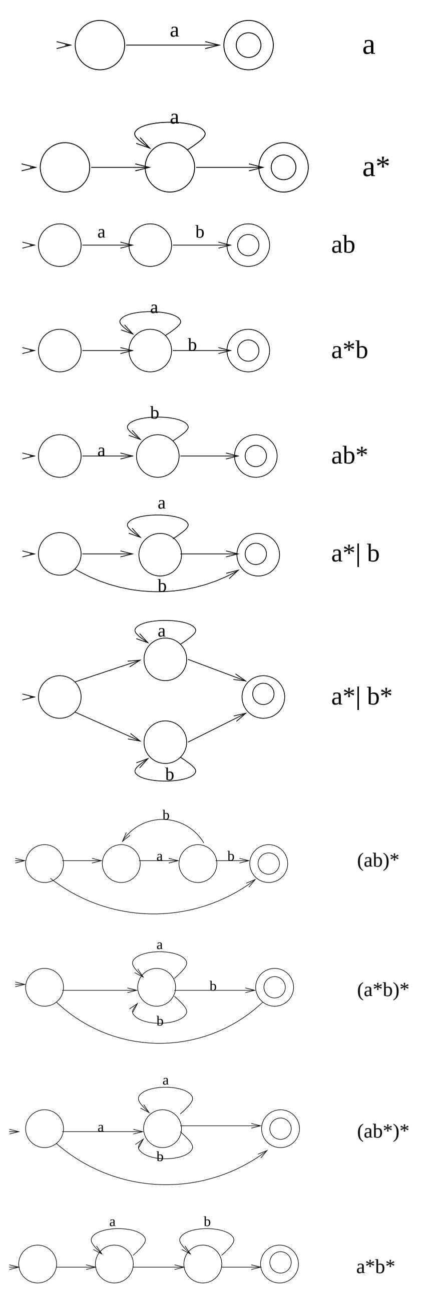

Figure 3: NFA from 1,2-literal r.e.

The combining procedure is very similar with naive Thompson's construction, but it does special construction for alternation and concatenation illustrated by Figure 4, 5 and 6. In alternation, the start and final states of the two NFA are merged, instead of creating two new states and four $\epsilon-$transitions in original Thompson's construction. In concatenation case, we will merge the final state of first NFA with the start state of the second NFA (illustrated by Figure 5), and when $\chi(n_1, n_2) = 1$, the final state of the first NFA and start state of the second NFA are removed (illustrated by Figure 6).

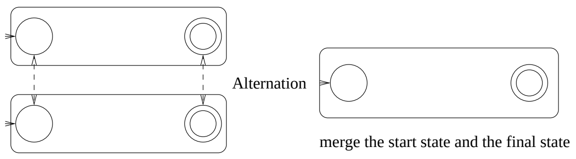

Figure 4: Illustration of Alternation

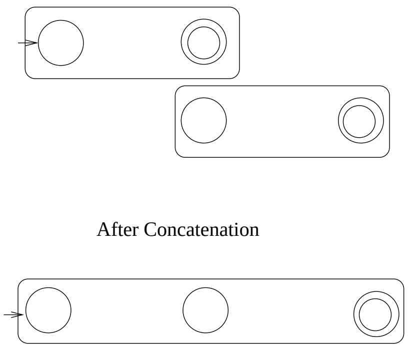

Figure 5: Illustration of Concatenation

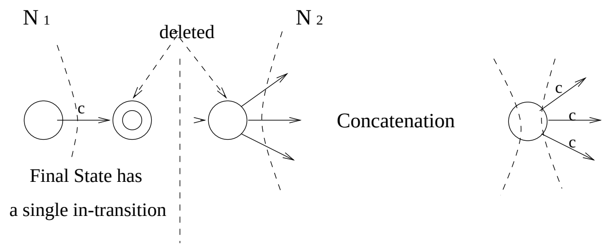

Figure 6: Illustration of Concatenation when $\chi(n_1, n_2) = 1$

5. ANALYSIS

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

Going through the NFAs in Figure 3, it is easy to verify:

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

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

Based on the combining algorithm illustrated in, we have the following recursive relation:

$$S_{n_1 n_2} = S_{n_1} + S_{n_2} - 1 - \chi(n_1, n_2) \quad (5)$$

$$S_{n_1|n_2} = S_{n_1} + S_{n_2} - 2 \quad (6)$$

$$S_{n_1^*} = S_{n_1} + 2 \quad (7)$$

$$T_{n_1 n_2} = T_{n_1} + T_{n_2} \quad (8)$$

$$T_{n_1|n_2} = T_{n_1} + T_{n_2} \quad (9)$$

$$T_{n_1^*} = T_{n_1} + 4 \quad (10)$$

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

Based on the above recursive relation and the facts about the NFAs from regular expressions with 1 or 2 literals, we have the main theorem in this note:

THEOREM 2. *For each regular expression with $l \geq 3$ literals, the resulting NFA from the above construction has*

1. *$2l - 2$ states and $4l - 6$ transitions if it is not nullable;*
2. *$2l - 1$ states and $4l - 3$ transitions if it is nullable but the root is not a star;*
3. *$2l$ states and $4l - 2$ transitions if the root is a star.*

6. TIME AND SPACE ANALYSIS

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

THEOREM 3. *The construction of the parse tree can be done in linear time w.r.t the size of the regular expression.*

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

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

THEOREM 4. *The overall time for the NFA construction from a parse tree is bounded by the number of nodes in the parse tree.*

This is obvious because each operation in Thompson's construction takes constant time, and there are at most $3l$ steps.

7. CONCLUSION

We presented a very simple algorithm to construct NFA with fewer states and transitions which improves Chang's work in [5]. No elaborate data structures are required in this algorithm, and it is very easy to implement.

8. REFERENCES

[1] A. Aho. Pattern matching in strings. In *Formal Language Theory(R. Book ed)*. Academic Press, 1980.

[2] V. Antimirov. Partial derivatives of regular expressions and finite automata construction. *Theoretical Computer Science*, 155:291–319, 1996.

[3] J. H. A. Aho and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[4] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[5] C. Chang. *Ph.D Thesis*. Department of Computer Science, Courant Institute, New York, 1992.

[6] G. C. C. Clark. On the use of regular expressions for searching text. *ACM Trans. on Programming Languages and Systems*, 19(3):413–426, 1997.

[7] V. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.

[8] P. P. G. Apostolopoulos, V. Peris. L5: A self learning layer 5 switch. In *Technical Report RC21461*. IBM, T.J. Watson Research Center, 1999.

[9] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Trans. IRS*, EC(9):39–47, 1960.

[10] E. Meyer. A four-russians algorithm for regular expression pattern matching. *Journal of ACM*, 39(2):432–448, 1992.

[11] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):410–422, 1968.