Minimized Thompson NFA

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

Key Words: Regular Expression, NFA, Minimization, Topological Sorting

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, both have disadvantages. In this paper: First, a "smart" parsing algorithm is developed which constructs a parse tree with at most $(3l - 1)$ nodes form a regular expression with l literals; Second, we propose an algorithm that works on the resulting NFA from Thompson's construction, eliminating as many auxiliary states as possible while maintaining Thompson's properties. It is shown that the resulting NFA is minimized. This means that no auxiliary states can be eliminated without violating the defining properties of Thompson NFA. The time and space requirement for the above algorithms are linear with respect to the length of the regular expression. To the author's knowledge, this is the first linear time algorithm minimizing an NFA in a precise technical sense.

1 Background

The construction of finite automata from regular expressions is of central importance to string pattern matching [3, 12], approximate string pattern matching [10], lexical scanning [1], computational biology [12], and DFA construction from regular expressions.

There are two basic methods converting a regular expression to an NFA, one is due to Thompson [7] and the other is due to McNaughton and Yamada [5] and Glushkov [6]. Based on these two constructions, many papers were published reporting the optimizing techniques for improvement. Chang presented an algorithm in [8] that computes the same NFA in the same asymptotic time $O(n)$ as Berry and Sethi [11], 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 [8], the result has $5l/2$ states and $5l$ transitions. It is a version of McNaughton and Yamada's construction which he called CNNFA. He proved that there are no more transitions than in Thompson's construction without optimization, and the CNNFA is more efficient than the Thompson's original construction for string matching. Antimirov gave a construction algorithm in [9] 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 [5], Chang [8], and Berry and Sethi [11].

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 we can minimize the Thompson NFA in linear time.

2 Definitions and Terminology

We follow the same notations as used in [1]. 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 [1]:

1. ϵ, ϕ and a for each $a \in \Sigma$ are regular expressions denoting the regular language $\{\epsilon\}$, the empty set and $\{a\}$ respectively;
2. If r_1, r_2 are regular expressions denoting the languages L_1, L_2, respectively, then $(r_1|r_2)$, (r_1r_2) and (r_1^*) are regular expressions, denoting $L_1 \cup L_2$, L_1L_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 $\Sigma \cup \{\epsilon\}$, an 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

$$(Q, \Sigma, \delta, q_0, F),$$

where

1. Q 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 $Q \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of Q;
4. q_0 in Q is the *initial state* of the finite control;
5. $F \subseteq Q$ 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 [1]:

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 Fig. 1. Unlabeled edges are used to represent ϵ-transitions.

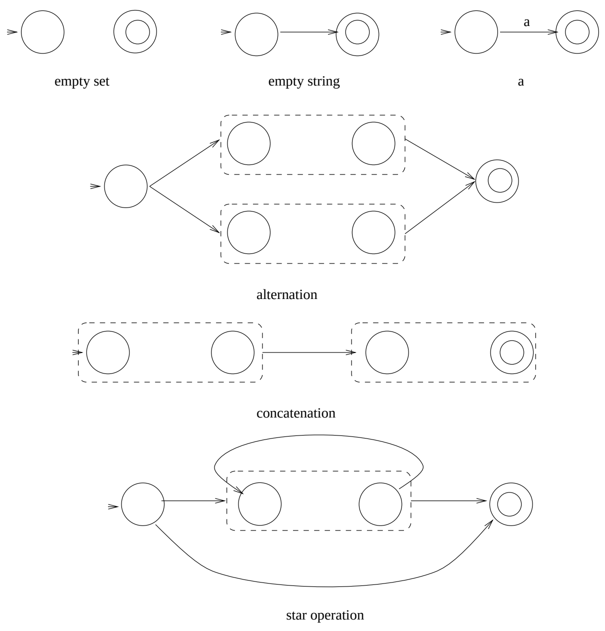

Figure 1: Thompson's Construction of NFA

For the above diagram, we call the procedure constructing the NFAs for empty set, empty string and single character as *atomic_nfa*, the procedure to construct NFA that is an alternation, concatenation and star of the sub-NFAs as *mk_alt*, *mk_cat* and *mk_star* respectively.

In an NFA, we call a state *auxiliary* if it has only ϵ-transitions from other states; otherwise, we call it a *transition* state. Auxiliary states are the candidates in our algorithm for deletion.

In [1], 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 [13], shortest matching is useful for searching text formatted with markup languages, and they gave an algorithm in [13] for shortest matching that runs in time $O(emn)$, where e is the upper bound of $|\delta(q, a)|$, where $q \in Q$ and $a \in \Sigma \cup \{\epsilon\}$.

For the NFAs constructed using Thompson's method, we know $e \leq 2$. Because this property is important for NFA simulation, we call this property Thompson's property, and an NFA with this property a Thompson NFA (TNFA for short). From [1, 13], we know e is an important factor for an NFA. Also, when $e \leq 2$, the representation can be simplified, and the size of an NFA is bounded by $O(em|\Sigma|)$.

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 list of 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, the goal is to make sure there is no star over a nullable node. In fact, if we add more complicated properties (like $(A|A')^* = A^*$ if $L(A') \subseteq L(A)$), it is possible to get even smaller parse trees at the expense of running time, but as we know regular expression minimization problem in general is a PSPACE-hard problem.

Use the above properties, we can reduce the size of the parse tree as we will see later, but this optimization is localized in the parse tree.

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

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.

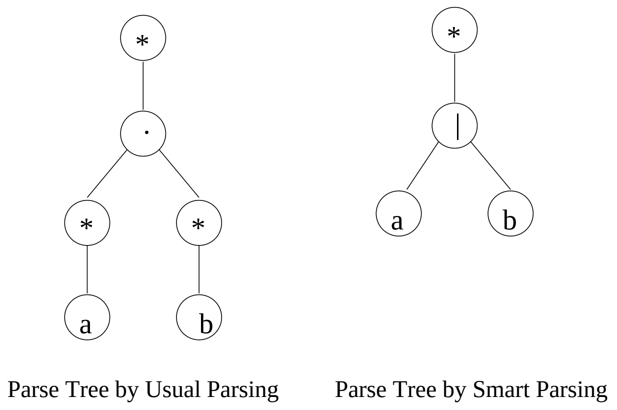

Figure 2: Parse Tree Comparison between Smart and Usual Parsing

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.

Note 1: 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 `epsilon`, to the node in the parse tree.

1. Whenever we do an alternation between a node r with ϵ, make `epsilon` to be true unless the node is already nullable (because if r is nullable, then $r|\epsilon = r$) and then return the node.

2. Whenever we do a concatenation between a node r with ϵ, just return node (because $r\epsilon = r$).

3. 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 parenthesis to make the regular expression unambiguous, and we have the following lemma about the property of a regular expression:

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

This is a better result than the $14(l - 1) + 5$ upper bound in Chang's thesis [8].

3.3 Time and Space Analysis

The following theorem bounds the time needed for this algorithm:

Theorem 2 *The construction of the parse tree can be done in linear time w.r.t the size of the regular expression.*

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

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

□

For space complexity, Chang gave an algorithm in [8] based on operator grammar [2] and showed the extra space needed is bounded by the l, which is the number of literals in the regular expression. We can apply the same technique to our parsing algorithm as "smart parsing" is a just ordinary parsing with *denull* procedure making sure that no star has a nullable child.

4 Minimized Thompson NFA

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

4.1 Algorithm

Based on smart parsing, we could use Thompson's method presented in the previous section to construct an NFA, this is the NFA to which we will apply the minimization algorithm.

To proceed, let's analyze how transition states and auxiliary states get generated. From Fig. 1, we know there are at most l transition states in TNFA for a regular expression with l literals, which are the final states when we generate the TNFA for a character. All other states are auxiliary states.

When we try to delete some auxiliary states, we need to make sure the resulting NFA will not violate the defining property of TNFA. Intuitively, we need to examine all subsets of the auxiliary states, which will make the overall run time exponential. To get a polynomial (in fact linear) time algorithm, we will show a modified topological sorting algorithm to get the order of the auxiliary state to examine for deletion. Just as with the ordinary topological sorting algorithm, the sorting result is not unique. Based on the order (which is not unique), we may not get a unique minimized Thompson NFA. But each minimized Thompson NFA is minimized in the sense that no auxiliary states may be deleted without violating the defining property of Thompson NFA. This may be regarded as the most significant contribution in this paper.

We call a set of auxiliary states *deletable* if the resulting NFA after the deletion is still a TNFA. And we call a set *undeletable* if it is not deletable. If there are more than two transitions labeled with c leaving state q, we say state q has a c-violation. One property about the deletable set is that there may be some undeletable states in a deletable set. Because when we talk about the set, we only care about the fact that the resulting NFA is a TNFA, but we need not guarantee that it is a TNFA after deletion of a single auxiliary state.

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

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

and

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

We define a relation $brother(x, y)$ (illustrated by Fig. 3) over auxiliary states if there exists another state z such that $z \succ x$ and $z \succ y$. And we use $B(x)$ to denote x's brother if its brother exists and $F(x)$ to denote the set of states z such that $z \succ x$.

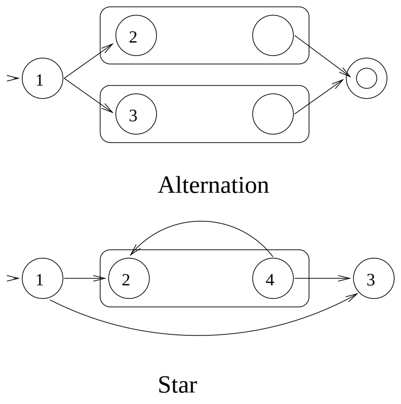

Figure 3: Illustration of Brother Relation

In Fig. 3, for alternation and star cases, state 2 and state 3 are brothers with each other. One comment about the star construction case is that 2 and 3 have two parents, which are states 1 and 4.

Property 1: There is no auxiliary state cycle in the TNFA constructed from the tree by smart parsing.

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

If there is no cycle at the beginning, as we delete auxiliary states, we know the remaining auxiliary states are still separated by the transition states. So there is no auxiliary state cycle in the process of deleting auxiliary states.

From this property, we know $\succ^+$ is a partial order, and by topological sorting, we can get the order of the auxiliary states that we could examine for deletion from the TNFA.

In the following pseudo-code, for an auxiliary state p, we mark it as *Ready* if "All of p's children are either transition states or have been visited (can not be deleted)". Each continue will transfer the control to the next iteration (repeat).

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

Algorithm 2 Procedure to Eliminate ϵ-transitions

Note 2: One property about the queue used in the above algorithm is that we need to access the elements that are not at the front of the queue. So we need to have pointers to each element in the queue. We still have the same Enqueue and Dequeue procedure as in other queue implementation, but each element in the queue will be marked as `ready, deleted` (`ready` means that it is enqueued, `deleted` means that it had been deleted with its brother at the same time, the other possibility is `unvisited` means that it had not been enqueued yet. Although this is not related with the implementation of the queue, it is about the elements in the queue). When we do the Dequeue in the above procedure, first examine whether or not the dequeued node was marked `deleted` (because it may be deleted at the same time with its brother), if yes, just Dequeue and do nothing else. Overall, the queue is a normal queue, but the procedure is a modified topological sorting, the states are not examined in the order they get dequeued, the brother relation may make a state examined before it get dequeued.

Based on the above algorithm, we have the following properties about the auxiliary states in the TNFA.

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

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

1. the start state of an alternation,
2. the start state of a star,
3. the final state of an NFA to which the last star operation is applied.

For each state with two ϵ-transitions, the two auxiliary states are either in two sub-NFAs that do not have any transition in between (case i), or are separated by one or more transition states (cases ii and iii). □

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

Lemma 2 *For an auxiliary state set Q and a state $q \in Q$, where there is no $r \in Q$ such that $q \succ r$, if we can neither delete q first, and then delete $Q - \{q\}$, nor delete $\{q, B(q)\}$ first, and then delete $Q - \{q, B(q)\}$, then Q is not a deletable set.*

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

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

If $p \notin Q$, if the violation is from a symbol other than ϵ, we know the deletion of the other states will not remove this violation. If the violation is from ϵ, the only way that can remove the violation is delete some auxiliary states that this state has ϵ-transition to. But from Property 1, we know such states do not exist. $\square$

Property 3: Each auxiliary state has at most one brother state.

This can be verified by Thompson's construction.

We have the following theorem about the TNFA after applying the $Minimization$ algorithm:

Theorem 3 *The TNFA produced by procedure $Minimization$ does not have a non-empty deletable auxiliary state subset.*

Before proving the theorem, we need the following lemma:

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

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

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

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

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

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

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

So the non-empty deletable set does not exist. $\square$

From this theorem, we know that the resulting TNFA is minimized in the sense that no more auxiliary states could be deleted without violating Thompson's property.

5 Time and Space Analysis of the Algorithm

We know the algorithm consists of three parts: smart parsing, Thompson's construction and the minimization procedure.

Theorem 4 *The overall time for the deletion of the auxiliary states is bounded by the number of the states in the TNFA.*

Proof Notice the number of ϵ-transitions is bounded by the number of the nodes in the parse tree, and the overall number of states in the list will never increase. The work on each node is bounded a constant (it is related with the size of the alphabet).

So the overall time needed is $O(\#\text{of } \epsilon\text{-transitions} + \#\text{ of auxiliary states}) = O(l)$. $\square$

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

6 Conclusion

We presented a TNFA minimization algorithm taking a TNFA as input and delete as many auxiliary state as possible. It is shown that the resulting TNFA is minimized in a sense that no auxiliary states could be deleted without violating the defining property of TNFA. As for future work, it is interesting to consider a generalized version of this problem: if we insist that the resulting NFA have certain property, could we minimize NFA in polynomial time.

7 Acknowledgements

The author would like to thank Dr. Rodney Canfield for many helpful discussions about this work. I would also like to thank members of Cisco LocalDirector group, especially Curt Cursey, Bruce Wong and Justin Pecqueur for their valuable comments, where I was an Intern.

References

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

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

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

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

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

[6] V.M. Glushkov *The Abstract Theory of Automata* Russian Math. Surveys, 16, 1-53, (1961).

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

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

[9] V. Antimirov, *Partial Derivatives of Regular Expressions and Finite Automata Construction*, Theoretical Computer Science, 155, 291-319, (1996).

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

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

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

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

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

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

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