Fast Regular Expression Matching Based On Dual Glushkov NFA

Ryutaro Kurai[1,2], Norihito Yasuda[1], Hiroki Arimura[2], Shinobu Nagayama[3], and Shin-ichi Minato[1,2]

[1] JST ERATO MINATO Discrete Structure Manipulation System Project
060-0814 Sapporo, Japan
{kurai, yasuda, minato}@erato.ist.hokudai.ac.jp

[2] Graduate School of Information Science and Technology
Hokkaido University, 060-0814 Sapporo, Japan
arim@ist.hokudai.ac.jp

[3] Department of Computer and Network Engineering
Hiroshima City University, 731-3194 Hiroshima, Japan
s_naga@hiroshima-cu.ac.jp

Abstract. This paper presents a new regular expression matching method by using Dual Glushkov NFA. Dual Glushkov NFA is the variant of Glushkov NFA, and it has the strong property that all the outgoing transitions to a state of it have the same labels. We propose the new matching method Look Ahead Matching that suited to Dual Glushkov NFA structure. This method executes NFA simulation with reading two input symbols at the one time. We use information of next symbol to narrow down the active states on NFA simulation. It costs additional working memory to apply Look Ahead Matching to ordinal Thompson NFA. However, we can use this method with no additional memory space if use it with Dual Glushkov NFA. Experiments also indicate that the combination of Dual Glushkov NFA with Look Ahead Matching outperforms the other methods on NFAs converted from practical regular expressions.

Keywords: regular expression matching, non-deterministic finite automata, ε-transition removal, Thompson NFA, Glushkov NFA

1 Introduction

1.1 Background

Regular expression matching is one of the fundamental research topics in computer science [13], since it plays such important role in emerging applications in large-scale information processing fields, such as: Network Intrusion Detection System (NIDS), Bioinformatics search engines, linguistic vocabulary, and pattern matching in cloud computing [10, 12, 15].

1.2 Problems with previous approaches

For regular expression matching, there are three well-known approaches: *backtracking*, *DFA*, and *NFA*. Among them, backtracking is the most widely used in practical applications. However, this approach is so slow if it manipulates some difficult patterns and texts, like $a?^n a^n$ as pattern and a^n as text, which triggers many backtracking on the input text [4]. The deterministic finite automaton (DFA) approach is extremely fast if the input regular expression can be compiled into a DFA of small size, but it is not practical if a given regular expression causes the exponential explosion of the number of states.

The Nondeterministic Finite Automaton (NFA) approach can avoid such explosion in the number of states, and is shown to be faster than the naive backtrack approach for the case that the backtracking approach suffer from many near-misses. Unfortunately, NFA approach is not so fast in practice. One of the major reasons is the cost of maintaining a set of active states; every time next input symbol comes, NFA has to update the all the active states to the next states or to just discard them. If the number of active states becomes large, the updating cost will also increase.

We can further classify the NFA into three types; Thompson NFA, Glushkov NFA and Dual Glushkov NFA. The most popular one is Thompson NFA, which is easy to construct, and its number of transitions and states are constant multiple of associated regular expression's length. Thompson NFA includes many of epsilon transitions. Those transitions make many of active states when we simulate such NFA.

Glushkov NFA is the other popular NFA. It has strong property that it has no epsilon transition and its all the incoming transitions to a state of Glushkov NFA have the same labels. Dual Glushkov NFA is a variant of Glushkov NFA, but has a special feature that is worth our attention. In an opposed manner of the Glushkov NFA, all the outgoing transitions from a state of Dual Glushkov NFA have the same labels.

1.3 Speed-up methods

We can simulate Glushkov NFA faster than Thompson NFA because of its property that it has no epsilon transition. The property causes less active states. Nevertheless, we have to manipulate amount of active state, and it slows down matching speed, if we treat complex regular expression. To cope with this problem, we propose a new method Look Ahead Matching.

That is the new matching method that reads two symbols of the input text at one time. We call the first of the two symbols the "current symbol", and second one the "next symbol". Ordinary matching methods read only current symbol, and calculate NFA active states from current active states and the symbol. Our method uses the next symbol to narrow down the active state size. We set states active only if the states have incoming transition labeled by first symbol and outgoing transition labeled by second symbol. However, fast matching by two input symbols creates large memory demands if we use the Glushkov NFA. To treat this problem, we employ a Dual Glushkov NFA. The structure of Dual Glushkov NFA is similar to that of Glushkov NFA, but is better suited to building an index of transitions for look ahead matching. We have to make transitions table for combination of two symbols to enable this new matching method. We can generate such table without additional space if we use the above Dual Glushkov NFA's property. Therefore, we propose the new look ahead matching method by using Dual Glushkov NFA.

1.4 Main Results

In this paper we propose a new matching method created by combining Dual Glushkov NFA and look ahead matching. Then we compare our proposal against other methods such as original Thompson NFA, Glushkov NFA, and a combination of Glushkov NFA and look ahead matching. For reference, we also compare our method with NR-grep [11]. In most cases our method is faster than Thompson NFA or Glushkov NFA.

Our method is only slightly slower than the combination of Glushkov NFA and look ahead matching, but it uses far less memory.

1.5 Related Works

Many regular expression matching engines use the backtracking approach. They traverse the syntax tree of a regular expression, and backtrack if they fail to find a match. Backtracking has a small memory footprint and high throughput for small and simple regular expressions. However, in worst case, it takes exponential time in the size of the regular expression [4].

Another approach, compiling regular expression has been used from the 1970s [1]. Such an algorithm converts a regular expression into an NFA, which is then converted into a DFA. This intermediate NFA (called Thompson NFA) has linear size memory in the length of the input regular expression. However, the final subset-constructed DFA takes exponential space in the size of the NFA and overflows the main memory of even recent computers.

In recent years, NFA evaluation for regular expression matching has been attracting much attention. Calculations performed on GPU, FPGA, or some special hardware cannot use abundant memory, but their calculation speed is much faster and concurrency is larger than typical computers. Therefore, using just the basic approach is adopted in some fields [3,7]. Cox proposed the NFA based regular expression library RE2 [14]. For fast evaluation, it caches a DFA generated from an NFA on the fly. RE2 seems to be the NFA based library that is being used widely; it has guaranteed computation time due to the NFA-oriented execution model.

Berry and Sethi indeed popularized Glushkov NFA [2]. Watson has finely researched Glushkov NFA and its application. He also showed the relationship between Thompson NFA and Glushkov NFA [16–18].

1.6 Organization

Sec. 2 briefly introduces regular expression matching, non-deterministic automata, and their evaluation. Sec. 3 presents our methods including preprocessing and runtime methods. Sec. 4 shows the results of computer experiments on NFA evaluation. Sec. 5 concludes this paper.

2 Preliminaries

2.1 Regular Expression

In this study, we consider regular expressions as follows. This definition is from [9].

Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.

1. $\emptyset$ is a regular expression and denotes the empty set.
2. ε is a regular expression and denotes the set ε.
3. For each symbol a in Σ is a regular expression and denotes the set a.
4. If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$, respectively then $(r|s)$, (rs), and $(r*)$ are regular expressions that denote the sets $L(r) \cup L(s)$, $L(r)L(s)$ and $L(r)*$, respectively.

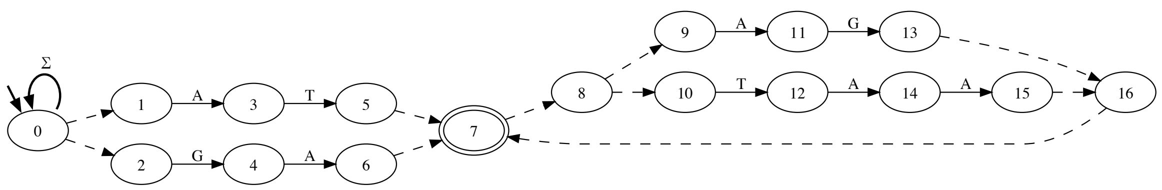

Figure 1. T-NFA for $R = (AT|GA)((AG|TAA)^*)$

2.2 Thompson NFA

The NFA constructed by Thompson's algorithm for regular expression R is called the *Thompson NFA* (or *T-NFA*, for short). It precisely handles the language statement $\Sigma^* L(R)$, which represents the substring match of R against a substring of a text.

Formally, T-NFA for R is a 5-tuple $N_R = (V, \Sigma, E, I, F)$, where V is a set of *states*, Σ is an *alphabet* of symbols, $E \subseteq V \times \Sigma \cup \{\varepsilon\} \times V$ is a set of symbol- and ε-transitions, called the *transition relation*, I and $F \subseteq V$ are the sets of *initial* and *final states* respectively. Each transition e in E is called a *symbol-transition* if its label is a symbol c in Σ, and an *ε-transition* if the label is ε. Each transition is described as $(s, char, t) \in E$, in this expression, s and t mean source state and target state. $char$ is a label of the transition.

The T-NFA N_R for R has nested structure associated with the syntax tree for R.

Let the length of associated regular expression be m. This length means the number of all symbols that appeared in the associated regular expression. The number includes the special symbols like "*", "(", or "+". For instance, the length of a regular expression "abb*" is 4.

It has at most $2m$ states, and every state has in-degree and out-degree of two or less. Specifically, state s in V can have at most one symbol-transition or two ε-transitions from s. We show an example of T-NFA when $R = (AT|GA)((AG|TAA)^*)$ in Fig. 1.

2.3 Glushkov NFA

Another popular method of constructing an NFA from a regular expression is Glushkov's algorithm [2]. We call the automaton constructed by this method Glushkov NFA (also known as Position Automata); abbreviated here to G-NFA. However, G-NFA can also be converted from T-NFA by using the algorithm in Fig. 4 (Watson showed in [18]). This algorithm removes the ε-transitions from T-NFA. For instance, we show the new transitions that skip ε-transition by the bold transitions in Fig. 2 and the fully converted G-NFA from T-NFA in Fig. 3. Both examples show NFAs that precisely handle $R = (AT|GA)((AG|TAA)^*)$.

2.4 G-NFA Properties

G-NFA has some very interesting properties.

– It has no ε-transitions. We call this property ε-free.
– For any state, the state's incoming transitions are labeled by the same symbol.
– It has only one initial state.
– It has one or more final states.

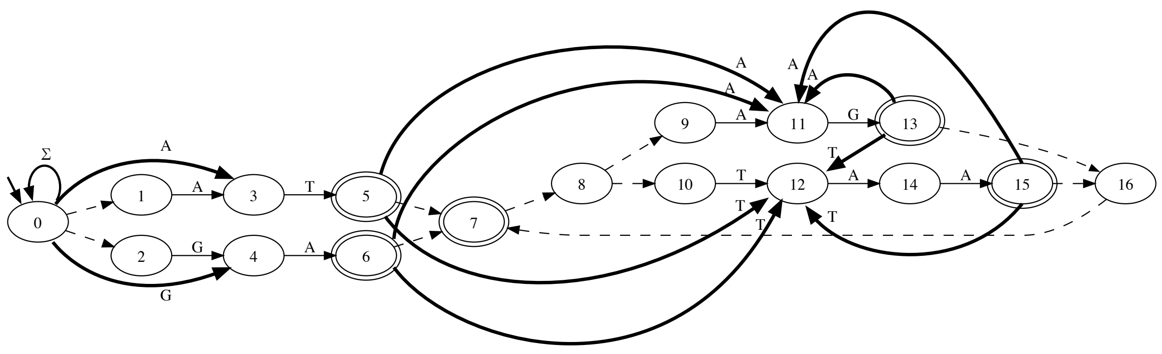

Figure 2. T-NFA and skip transitions of ε-transitions

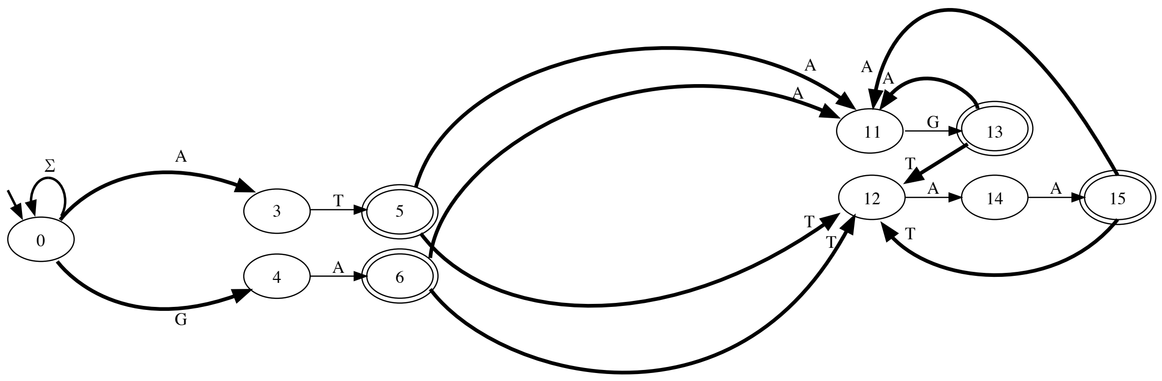

Figure 3. G-NFA for $R = (AT|GA)((AG|TAA)^*)$

– Its number of states is $\tilde{m} + 1$. $\tilde{m}$ is the length of the associated regular expression, but the number excludes the special symbols like "$*$", "(", or "+".
– The number of transitions is $\tilde{m}^2$ at worst.

2.5 Dual Glushkov NFA

As a variation of G-NFA, Dual Glushkov NFA is known [16]. We call it Dual G-NFA for short. The algorithm that converts T-NFA into Dual G-NFA (Fig. 5) is similar to the algorithm that converts T-NFA into G-NFA (fig. 4). The base algorithm of Fig. 5 was also shown by Watson [18].

When we convert T-NFA into G-NFA, we generate a skip transition as follows. First, we search the path that started by epsilon-path and ended only one symbol-transition. Then we create skip transition from the start state to the end state for each such path. The label of new skip transition is taken from the last transition of the path.

When we convert T-NFA into Dual G-NFA, we generate a skip transition as follows. First, we search the path that started only one symbol-transition and ended epsilon-path. Then we create skip transition from the start state to the end state for each such path. The label of new skip transition is taken from the first transition of the path.

In addition, an original T-NFA has same number of outgoing and incoming transitions for all section of T-NFA. In fact, Conjunction, Concatenation, and Kleene

```
procedure BuildG-NFA($N = (V, \Sigma, E, I, F)$)
    $V' \leftarrow \emptyset, E' \leftarrow \emptyset, F' \leftarrow F$
    $GlushkovState \leftarrow V \setminus \bigcup_{s', \exists s \in V, (s, \varepsilon, s') \in E} s'$
    for all $s \in GlushkovState$ do
        for $s' \in Eclose(s)$ do
            if $s' \in F$ then
                $F' \leftarrow F' \cup \{s'\}$
            end if
            for $(s', char, t) \in E$ do
                if $char \neq \varepsilon$ then
                    $E' \leftarrow E' \cup \{(s, char, t)\}$
                end if
            end for
        end for
    end for
    return $(V', \Sigma, E', I, F')$
end procedure
procedure Eclose($s \in E$)
    $Closure \leftarrow \{s\}$
    for $(s, char, t) \in E$ do
        if $char = \varepsilon$ then
            $Closure \leftarrow Closure \cup Eclose(t)$
        end if
    end for
    return $Closure$
end procedure
```

Figure 4. Algorithm Converting T-NFA to G-NFA

```
procedure BuildDualG-NFA($N = (V, \Sigma, E, I, F)$)
    $V' \leftarrow \emptyset, E' \leftarrow \emptyset, I' \leftarrow \emptyset$
    $DualGlushkovState \leftarrow V \setminus \bigcup_{s', \exists s \in V, (s', \varepsilon, s) \in E} \{s'\}$
    for $s \in Eclose(I)$ do
        if $s \in DualGlushkovState$ then
            $I' \leftarrow I' \cup \{s\}$
        end if
    end for
    for all $s \in DualGlushkovState$ do
        for $(s, char, t) \in E$ do
            for $t' \in Eclose(t)$ do
                if $t \neq t'$ and $char \neq \varepsilon$ and $t' \in DualGlushkovState$ then
                    $E' \leftarrow E' \cup \{(s, char, t')\}$
                end if
            end for
            $E' \leftarrow E' \cup \{(s, char, t)\}$
        end for
    end for
    return $(V', \Sigma, E', I, F')$
end procedure
```

Figure 5. Algorithm Converting T-NFA to Dual G-NFA

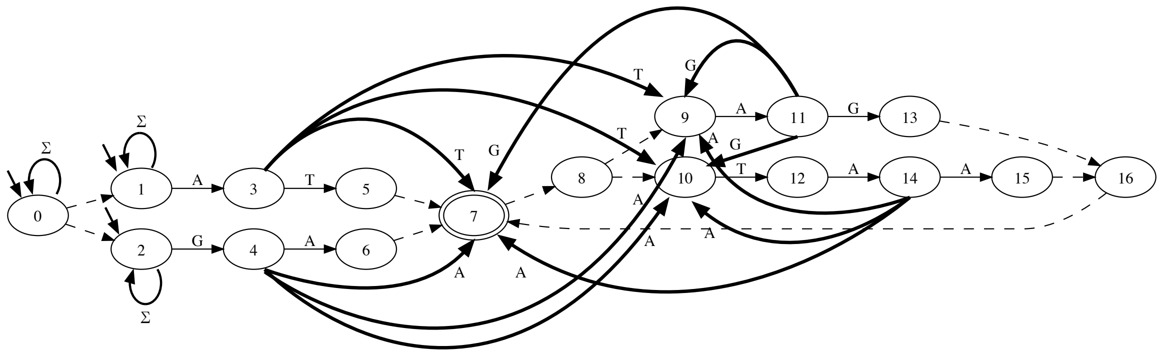

Figure 6. T-NFA to Dual G-NFA

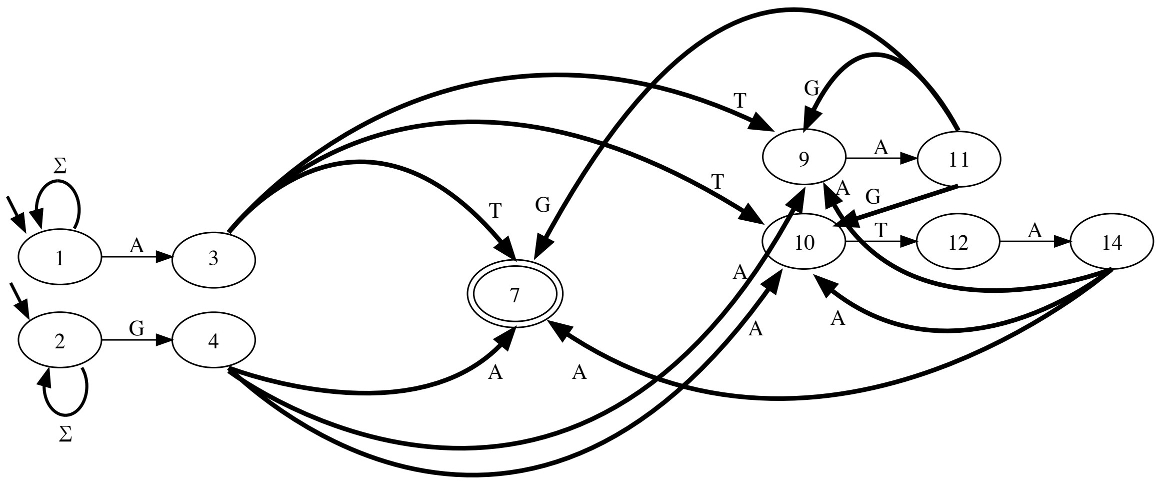

Figure 7. Dual G-NFA for $R = (AT|GA)((AG|TAA)^*)$

Closure section of T-NFA have same in-degree and out-degree. Because of this T-NFA's property, we can consider that above "Dual G-NFA" is dual of "G-NFA".

For instance, we show the new transitions that skip ε-transition by bold transitions in Fig. 6, and the fully converted Dual G-NFA from T-NFA in Fig. 7. Both examples show NFAs that precisely handle $R = (AT|GA)((AG|TAA)^*)$.

2.6 Dual G-NFA Properties

Dual G-NFA has properties similar to those of G-NFA.

- It is ε-free.
- For any state, the state's outgoing transitions are labeled by the same symbol.
- It has only one final state.
- It has one or more initial states.
- Its number of states is $\tilde{m} + 1$.
- The number of transitions is $\tilde{m}^2$ at worst.

There is a duality between G-NFA and Dual G-NFA in the sense of the properties of initial states, final states, and labels of transitions.

```
procedure G-NFACountMatching($N = (V, \Sigma, E, I, F), T = t_1 t_2 t_3 ... t_n$)
    $CurrentActive \leftarrow \emptyset$
    $NextActive \leftarrow \emptyset$
    $MatchCount \leftarrow 0$
    $Index \leftarrow BuildIndex(E)$
    for $pos \in 1, \ldots, n$ do
        $CurrentActive \leftarrow CurrentActive \cup I$
        for $s \in CurrentActive$ do
            $NextActive \leftarrow NextActive \cup Index[t_{pos}][s]$
        end for
        if $NextActive \cap F \neq \emptyset$ then
            $MatchCount \leftarrow MatchCount + 1$
        end if
        $CurrentActive \leftarrow NextActive$
        $NextActive \leftarrow \emptyset$
    end for
    return $MatchCount$
end procedure
procedure BuildIndex($V, \Sigma, E$)
    for $s \in V$ do
        for $char \in \Sigma$ do
            $Index[char][s] = \emptyset$
        end for
    end for
    for $(s, char, t) \in E$ do
        $Index[char][s] = Index[char][s] \cup \{t\}$
    end for
    return Index
end procedure
```

Figure 8. Regular Expression Matching Using NFA

2.7 Regular Expression Matching Method

For both G-NFA and Dual G-NFA, ε-free NFAs have the same simulation algorithm like that of Fig. 8. The basic idea of this algorithm was also shown by Watson [17].

This algorithm reads input symbol t_i one by one, then searches for a state that has outgoing transition labeled t_i from current active state set (*CurrentActive* in Fig. 8). For fast search we use the index created by *BuildIndex*. If such states are found, we add a transitive state to next state set (*NextActive* in Fig. 8). At the end of a step, we check if the *NextActive* includes a final state. If a final state is found, we recognize that the input symbols match a given regular expression.

3 Our Method

3.1 Look ahead matching

The above NFA simulation method reads input symbols one by one, and calculates state transitions. However, it is quite easy to read a next input symbol. We consider how to more effectively calculate state transitions. Let the current input symbol be t_i, next input symbol t_{i+1}. When we know t_{i+1}, we want to treat the states that satisfy the next formula as active states.

$$LookAheadActive(s, t_i, t_{i+1}) = \{s' : (s, t_i, s') \in E, (s', t_{i+1}, s'') \in E\}$$

And we formally define normal active states as follows.

$$Active(s, t_i) = \{s' : (s, t_i, s') \in E\}$$

For any $LookAheadActive(s, t_i, t_{i+1})$, the size of $LookAheadActive(s, t_i, t_{i+1})$ is equal or less than the size of $Active(s, t_i)$. Because of this difference in size of active states, we consider that look ahead matching can calculate transitions faster than normal matching. We formally show this algorithm in Fig. 9. This look ahead mathing idea have been used in some studies [5, 6].

The problem of this matching method is the large size of the state transition table associated with t_i and t_{i+1}. The state transition table has duplicate transitions and costs $O(|E|^2)$ space to build from G-NFA.

For example, we show the transition table of Fig. 1 as Table 1. This table has 19 records, more than the number of original G-NFA's transitions. The difference is due to the duplication of transitions.

id	t_i	t_{i+1}	source state	target state
1	T	A	3	5
2	T	A	5	12
3	T	A	6	12
4	T	A	13	12
5	T	A	15	12
6	T	T	3	5
7	A	A	4	6
8	A	A	12	14
9	A	A	14	15
10	A	T	4	6
11	A	T	14	15
12	A	T	0	3
13	G	A	11	13
14	G	A	0	4
15	G	T	11	13
16	A	G	5	11
17	A	G	6	11
18	A	G	13	11
19	A	G	15	11

Table 1. Look Ahead Transition Table for G-NFA and Dual G-NFA

id	t_i	t_{i+1}	source state	target state
1	A	T	1	3
2	A	T	4	10
3	A	T	14	10
4	G	A	2	4
5	G	A	11	9
6	A	G	9	11
7	T	A	10	12
8	T	A	3	9
9	A	A	12	14
10	A	A	4	9
11	A	A	14	9
12	T	*	3	7
13	T	T	3	10
14	A	*	4	7
15	A	*	14	7
16	G	*	11	7
17	G	T	11	10

Table 2. Look Ahead Transition Table for Dual G-NFA

3.2 Dual G-NFA Look Ahead Transition Function

As shown in the above section, Dual G-NFA has the very desirable property that all outgoing transition of a state have the same label. Because of this property, when the source state and t_i are given, the pairs of t_{i+1} and the target state are determined uniquely. Therefore, the transition tables size is $O(|E|)$. This is effectively smaller than G-NFA's size of $O(|E|^2)$.

For instance, we show the transition table of Fig. 7 in Table 2. This table has 17 records, equaling the number of original Dual G-NFA's transitions. A final state of Dual G-NFA has no outgoing transition, so we show the "$*$" on t_{i+1} column for the transitions that go to final state.

```
procedure DualG-NFACountLookAheadMatching($N = (V, \Sigma, E, I, F), T = t_1 t_2 t_3 ... t_n$)
    $CurrentActive \leftarrow \emptyset$
    $NextActive \leftarrow \emptyset$
    $MatchCount \leftarrow 0$
    $(Index, FinalIndex) \leftarrow BuildLookAheadIndex(E)$
    for $pos \in 1, \ldots, n-1$ do
        $CurrentActive \leftarrow CurrentActive \cup I$
        for $s \in CurrentActive$ do
            for $(t \in Index[t_{pos}][t_{pos+1}][s])$ do
                $NextActive \leftarrow NextActive \cup \{t\}$
            end for
        end for
        for $s \in CurrentActive$ do
            for $(t \in FinalIndex[t_{pos+1}][s])$ do
                $NextActive \leftarrow NextActive \cup \{t\}$
            end for
        end for
        if $NextActive \cap F \neq \emptyset$ then
            $MatchCount \leftarrow MatchCount + 1$
        end if
        $CurrentActive \leftarrow NextActive$
        $NextActive \leftarrow \emptyset$
    end for
    return $MatchCount$
end procedure
procedure BuildLookAheadIndex($V, \Sigma, E, F$)
    for $s \in V$ do
        for $char_1 \in \Sigma$ do
            for $char_2 \in \Sigma$ do
                $Index[char_1][char_2][s] = \emptyset$
            end for
            $FinalIndex[char_1][s] = \emptyset$
        end for
    end for
    for $(s, char_1, t) \in E$ do
        for $(t, char_2, t') \in E$ do
            $Index[char_1][char_2][s] = Index[char_1][char_2][s] \cup \{t\}$
        end for
        if $t \cap F \neq \emptyset$ then
            $FinalIndex[char_1][s] = FinalIndex[char_1][s] \cup \{t\}$
        end if
    end for
    return (Index, FinalIndex)
end procedure
```

Figure 9. Look Ahead Regular Expression Matching Using Dual G-NFA

4 Experiments and Results

To confirm the efficiency of Dual G-NFA with Look Ahead matching (for short, Dual G-NFA with LA), we conducted three experiments. All experiments use "English.100MB" text in Pizza&Chili Corpus [8] as the input texts, and we compared our method with G-NFA, and G-NFA with Look ahead matching (for short G-NFA with LA). For reference, the results of a simple T-NFA implementation by Russ Cox [4], and NR-grep, a bit parallel implementation of G-NFA by Gonzalo Navarro are shown. All experiments were executed 10 times and the average time is shown.

The first experiment examines fixed string patterns. In this experiment, patterns were generated as follows. n fixed strings were randomly chosen from a fixed strings dictionary and then patterns were joined by conjunction symbol "|". We used `/usr/share/dict/words` file on Mac OS X 10.9.2 as the fixed strings dictionary. None of patterns included special symbols of regular expressions like "$*$", "?", or "+". Thus, the Aho-Corasick algorithm is clearly the most suited method for this problem. However, to measure trends of our methods, we make this experiment.

Table 3 shows the time (in seconds) needed to convert regular expression to NFAs. From Table 3, the converting time is so shorter than matching time. T-NFA (by Cox) was so fast to measure the converting time accurately (It was under micro seconds).

n	G-NFA	Dual G-NFA
20	immeasurable	immeasurable
40	0.01	0.01
60	0.01	0.01
80	0.02	0.02
100	0.02	0.02
120	0.03	0.03
140	0.04	0.04
160	0.05	0.05
180	0.06	0.06
200	0.07	0.07

Table 3. Needed time converting regular expression to NFAs in seconds

Fig. 10 shows the time (in seconds) needed to match with the whole text of "English.100MB". From Fig. 10, the time taken linearly increases with the number of patterns with T-NFA, G-NFA or Dual G-NFA. In contrast, G-NFA with LA or Dual G-NFA with LA, which uses look ahead matching method took almost constant time regardless of n. We assume this is because look ahead matching kept the active state size small.

NR-grep could not treat large regular expressions, so we only measured patterns for $n = 20$, 40, and 60;

Fig. 11 shows the average active state size, total number of active states divided by the number of all input symbols. As the graph shows, there is strong correlation between the average active state size and matching time.

In the second experiment, we generated patterns as follows. We inserted special symbols of regular expressions such as "$*$","?", or "+" into the patterns used in the first experiment. Insert positions were randomly selected excluding the first and last pattern positions. We then joined these generated regular expression patterns by conjunction. In this case, the Aho-Corasick algorithm is clearly the most suited

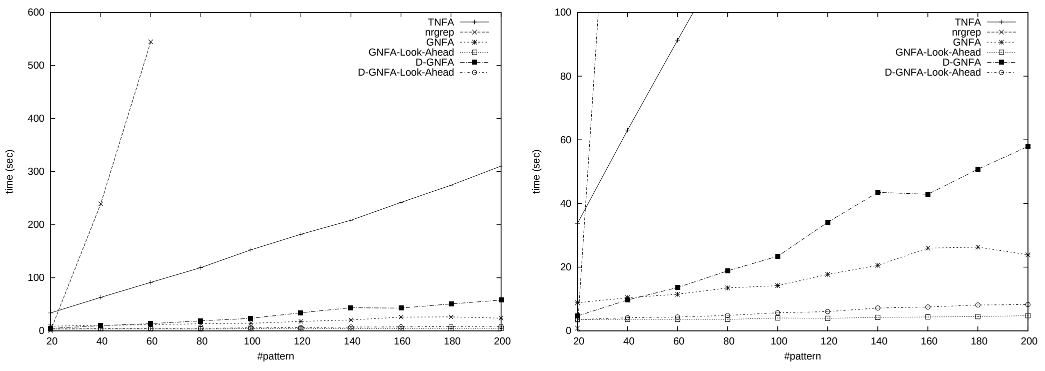

Figure 10. Needed time (sec) to matching whole text of "English.100MB" for 1st experiment. Right part is the part of left part.(Right pert is scaled-up)

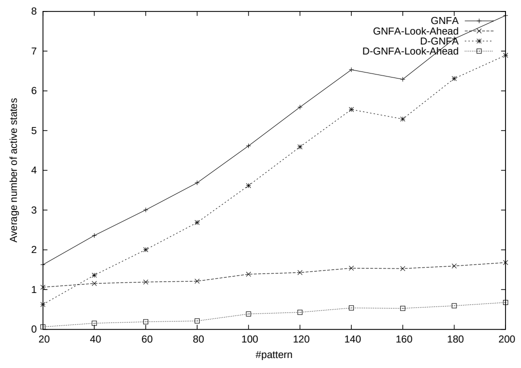

Figure 11. Average number of active states for 1st experiment.

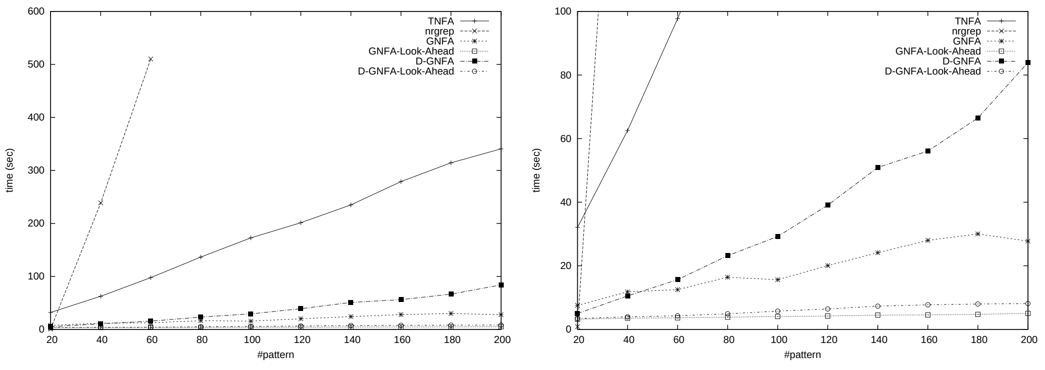

Figure 12. Needed time (sec) to matching whole text of "English.100MB" for 2nd experiment. Right part is the part of left part.(Right pert is scaled-up)

method since the pattern is a set of fixed string. However, we can see the basic speed of pattern matching by treating the pattern as a regular expression.

Fig. 12 shows the results. The trends resemble those of first experiment. G-NFA-Look-Ahead or Dual G-NFA-Look-Ahead was superior in terms of calculation time.

In the third experiment, we challenged our method with some actual regular expression patterns in Table 4. First pattern "suffix" matches to words that have some specific suffixes. There were 35 suffixes. Second pattern "prefix" matches to words that have some specific prefixes. There were 32 prefixes. Third pattern "name" matches

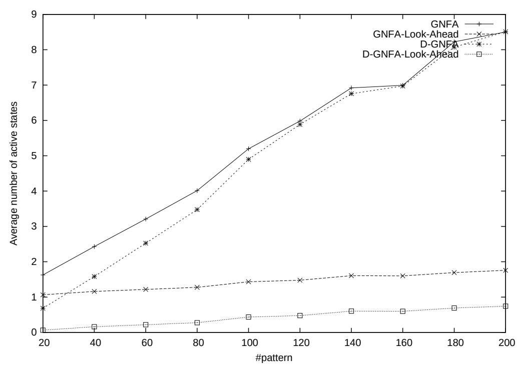

Figure 13. Average number of active states for 2nd experiment.

to some people's names. The names were combination of ten common given names and ten common surnames. Fourth pattern "user" matches to popular expression of user and computer name. Fifth pattern "title" matches strings that composed of capitalized words like a chapter title in books. These patterns include symbol classes like "[a-zA-Z]".

name	pattern sample
suffix	`[a-zA-Z]+(able|ible|al|...|ise)`
prefix	`(in|il|im|infra|...|under)[a-zA-Z]+`
names	`(Jackson|Aiden|...|Jack) (Smith|Johnson|...|Rodriguez|Wilson)`
user	`[a-zA-Z]+@[a-zA-Z]+`
title	`([A-Z]+ )+`

Table 4. Regular expression patterns used in third experiment.

As shown in Table 5, Dual G-NFA with LA is the fastest in some cases, once again to the reduction in active state size. Look ahead methods never match slower than T-NFA, G-NFA and Dual G-NFA. If that input consists of only small regular expression like pattern "name", "user" or "title", NR-grep is the fastest. For such patterns, bit parallel method implemented in NR-grep can manipulate G-NFAs effectively.

pattern	T-NFA (by Cox)	ngrep	G-NFA	G-NFA with LA	Dual G-NFA	Dual G-NFA with LA
suffix	113.48	20.24	9.74	7.51	106.64	3.35
prefix	14.33	5.295	2.74	3.97	78.39	3.82
names	12.95	0.216	2.97	2.74	3.21	2.76
user	78.14	0.08	12.11	7.41	185.22	3.36
title	38.88	0.186	2.93	2.38	2.68	2.21

Table 5. Needed time (sec) to matching with whole text of "English.100MB"

5 Conclusion

We proposed the new regular expression matching method that based on Dual G-NFA and Look Ahead Matching. We have shown that Dual G-NFA can construct a look ahead matching index without additional space. Simulations have shown the effectiveness of look ahead matching in accelerating NFA. From the experimental

pattern	G-NFA	G-NFA with LA	Dual G-NFA	Dual G-NFA with LA
suffix	2.33	1.65	50.44	1.15
prefix	1.50	1.15	8.11	0.33
names	1.01	1.00	0.01	0.001
user	1.77	1.59	40.67	0.59
title	1.03	1.01	0.75	0.01

Table 6. Average number of active states

results, our method can be useful for regular expression matching in practical usage. G-NFAs are used in some bit parallel methods, so we now plan to apply bit parallel techniques to Dual G-NFA methods.

References

1. A. V. Aho and J. E. Hopcroft: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1st ed., 1974.
2. G. Berry and R. Sethi: *From regular expressions to deterministic automata.* Theoretical computer science, 48 1986, pp. 117–126.
3. N. Cascarano, P. Rolando, F. Risso, and R. Sisto: *iNFAnt: NFA pattern matching on GPGPU devices.* ACM SIGCOMM Computer Comm. Review, 40(5) 2010, pp. 20–26.
4. R. Cox: *Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).* `http://swtch.com/~rsc/regexp/regexp1.html`, January 2007.
5. N. de Beijer: *Stretching and jamming of automata.* Masters thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 2004.
6. N. de Beijer, L. G. Cleophas, D. G. Kourie, and B. W. Watson: *Improving automata efficiency by stretching and jamming*, in Proceedings of the Fifteenth Prague Stringologic Conference, September 2010, pp. 9–24.
7. P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes: *An efficient and scalable semiconductor architecture for parallel automata processing.* IEEE Transactions on Parallel and Distributed Systems, 2013.
8. P. Ferragina and G. Navarro: *The Pizza & Chili Corpus.* `http://pizzachili.dcc.uchile.cl/`.
9. J. E. HOPCROFT, R. MOTWANI, and J. D. ULLMAN: *Introduction to Automata Theory, Languages, and Computation, Third Edition*, Addison Wesley, 2006.
10. Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga: *Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching*, in Proc. FPT'10, Dec 2010, pp. 21–28.
11. G. Navarro: *Nr-grep: a fast and flexible pattern-matching tool.* Software: Practice and Experience, 31(13) 2001, pp. 1265–1312.
12. G. Navarro and M. Raffinot: *Flexible pattern matching in strings — practical on-line search algorithms for texts and biological sequences.*, Cambridge, 2002.
13. D. Perrin: *Finite automata*, in Handbook of Theor. Comput. Sci, Vol.B, Chap. 1, J. van Leeuwen, ed., 1990, pp. 1–57.
14. *RE2 an efficient, principled regular expression library*: `https://code.google.com/p/re2/`.
15. Y. Wakaba, M. Inagi, S. Wakabayashi, and S. Nagayama: *An efficient hardware matching engine for regular expression with nested kleene operators*, in Proc. FPL'11, 2011, pp. 157–161.
16. B. W. Watson: *A taxonomy of finite automata construction algorithms*, tech. rep., Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1993.
17. B. W. Watson: *The design of the FIRE engine: A C++ toolkit for FInite automata and Regular Expressions*, tech. rep., Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1994.
18. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, September 1995.