NFAs with Tagged Transitions, their Conversion to Deterministic Automata and Application to Regular Expressions

Ville Laurikari
Helsinki University of Technology
Laboratory of Computer Science
PL 9700, 02015 TKK, Finland
vl@iki.fi

Abstract

A conservative extension to traditional nondeterministic finite automata is proposed to keep track of the positions in the input string for the last uses of selected transitions, by adding "tags" to transitions. The resulting automata are reminiscent of nondeterministic Mealy machines. Formal semantics of automata with tagged transitions is given.

An algorithm is given to convert these augmented automata to corresponding deterministic automata, which can be used to process strings efficiently. Application to regular expressions is discussed, explaining how the algorithms can be used to implement for example substring addressing and a lookahead operator, and some informal comparison to other widely used algorithms is done.

1. Introduction

The methods of translating regular expressions to deterministic finite automata are well known [1, 6, 11]. The resulting automata are able to distinguish whether or not a given string belongs to the language defined by the original regular expression. However, it is not possible to, for example, tell the position and extent of given subexpressions of the regular expression in a matching string.

Algorithms do exist for solving the problem described above, but the widely used ones usually require simulating the operations of an NFA while processing an input string and most are interpretive backtracking algorithms which take polynomial or exponential worst-case time, nonconstant space, or some combination of these. For example, the GNU regexp-0.12 library consumes time exponentially on the regular expression `(v*)*|j*` with input `vvvvv...j`. With an input of about 25 characters the matching takes tens of seconds on a modern workstation.

Nakata and Sassa [8] have proposed a way of adding semantic rules to regular expressions and converting them to deterministic automata, but their algorithms do not resolve ambiguity and cannot handle situations where more than one application of the same semantic rule can be postponed at the same time.

Of course, using an attribute grammar and a parser generator is an option if speed is not essential. The automata-based approach is simpler and thus faster by up to several orders of magnitude, and in many cases more intuitively understandable for the user. For even simple regular expressions, several productions may be necessary to express the equivalent language. [8]

I propose here an extension to traditional NFAs, tagged transitions, which allow us to store sufficient information of the path taken by the automaton while matching so that subexpression matching information, in the style of, for example POSIX.2 `regexec`, can be extracted with minimal work. We give an algorithm for converting these augmented NFAs to deterministic automata which can be used to process strings under a linear time bound and more importantly, using an efficient and simple matcher loop reminiscent of traditional DFA engines.

This paper is organized as follows: Section 2 introduces the concept of tagged transitions along with a a formal description of their semantics. Section 3 describes the conversion from NFAs with tagged transitions to deterministic automata, and section 4 gives an algorithm for doing the conversion. Section 5 discusses application of the algorithms to regular expressions. The less technical reader may only skim through the next section and then skip right to section 5.

2. NFAs with tagged transitions

NFAs with tagged transitions, or TNFAs, are similar to normal NFAs except that the transitions may have a *tag* attached to them. Such transitions are called *tagged transi-*

tions. Tags are of the form t_x, where x is an integer. Each tag has a corresponding variable which can be set and read, and when a tagged transition is used, the current position in the input string is assigned to the corresponding variable.

If a tag is unused, it has a special value, -1; initially all tags are unused and have this value. We use the same symbol for the tag and its variable, so if we use, say, t_5 as a tag, it implies the existence of the variable t_5. Figure 1 shows how tagged transitions are marked in a graph. For untagged transitions, ω is used to denote that there is no tag. Usually the $/\omega$ is omitted from graphs so that $\mathtt{a}/\omega$ becomes $\mathtt{a}$ and ϵ/ω becomes ϵ.

TNFAs are reminiscent of nondeterministic Mealy transducers, but we are interested in a single path which results in a final state with a given input string, and want to know, in addition to which tags have been used, the places in the input string where they were last used.

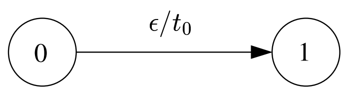

Figure 1. A tagged transition

Definition 1. *A TNFA is a 5-tuple $M = (K, T, \Sigma, \Delta, s, F)$, where*

K *is a finite set of* states*,*

T *is a finite set of* tags*,* $\omega \in T$*,*

Σ *is an* alphabet*, i.e. a finite set of symbols,*

Δ *is the* prioritized tagged transition relation*, a finite subset of* $K \times \Sigma^* \times T \times r \times K$*, where* $r \in \mathbb{N}$ *is unique for all items of the relation.*

$s \in K$ *is the* initial state*,*

$F \subseteq K$ *is the set of* final states

The meaning of a quintuple $(q, u, t, r, p) \in \Delta$ is that M, when in state q, may consume a string u from the input string, set the value of t to the current position in the input string and enter state p. The meaning of r, the priority of the transition, is explained later in this section.

A *configuration* of M is an element of $K \times \Sigma^* \times \Sigma^* \times V \times r$, where the first item is the current state, the second item is the processed part of the input string, the third item is the unprocessed part of the input string, V is a function from T to $\mathbb{N}$, i.e. from tags to their values and $r \in \mathbb{N}$ is the priority of the last used transition. The relation $\vdash_M$ between configurations (*yields in one step*) is defined as follows: $(q, p, u, v, r) \vdash_M (q', p', u', v', r')$ if and only if there are $w \in \Sigma^*$, $r' \in \mathbb{N}$ and $t \in T$ such that $u = wu'$ and $(q, w, t, r', q') \in \Delta$. Then $p' = pw$ and

$$v'(x) = \begin{cases} |p'| & \text{if } t \neq \omega \text{ and } x = t \\ v(x) & \text{otherwise.} \end{cases}$$

We define $\vdash_M^*$ to be the reflexive, transitive closure of $\vdash_M$. A string $w \in \Sigma^*$ is *accepted* by M if and only if there is a state $q \in F$, a function v and integer r such that $(s, \epsilon, w, T \times \{-1\}, 0) \vdash_M^* (q, w, \epsilon, v, r)$.

Note that the last item of a configuration, the priority of the last used transition, is not really needed and could be omitted. The input string parts are not strictly necessary, either. Read on to see why they are nevertheless included.

For a particular string w and a machine M, there may be several different q, v and r which satisfy $(s, \epsilon, w, T \times \{-1\}, 0) \vdash_M^* (q, w, \epsilon, v, r)$. In order for the results of the computation to be predictable and thus more practical, we must somehow be able to determine which particular values of q, v and r we choose as the result.

To choose between different q, we can simply assign each final state a unique priority and choose the one with the highest priority. This is basically what lexical analyzers typically do when two or more patterns match the same lexeme.

To choose between different v and r (tag value ambiguity), we must define another relation $\models_M$ between configurations (*yields tag-wise unambiguously in one step*): $(q, p, u, v, r) \models_M (q', p', u', v', r')$ if and only if there are $w \in \Sigma^*$, $r' \in \mathbb{N}$ and $t \in T$ such that

> $u = wu'$, $(q, w, t, r', q') \in \Delta$ and
>
> for every q'', p'', u'', v'', r'', v''' and r''' such that $(s, \epsilon, pu, T \times \{-1\}, 0) \models_M^* (q'', p'', u'', v'', r'')$ and $(q'', p'', u'', v'', r'') \vdash_M (q', p', u', v''', r''')$ it holds that $r' \leq r'''$.

Then $p' = pw$ and

$$v'(x) = \begin{cases} |p'| & \text{if } t \neq \omega \text{ and } x = t \\ v(x) & \text{otherwise.} \end{cases}$$

As before, $\models_M^*$ is the reflexive, transitive closure of $\models_M$. A string $w \in \Sigma^*$ is *tag-wise unambiguously accepted* by M if and only if there is a state $q \in F$, a function v and integer r such that $(s, \epsilon, w, T \times \{-1\}, 0) \models_M^* (q, w, \epsilon, v, r)$.

Note that while the above definition is recursive, it is still possible to calculate $\models_M^*$ effectively, using a non-recursive process, for a given automaton and input string.

For a particular string w and a machine M, if w is accepted by M, it is also tag-wise unambiguously accepted with uniquely determined v and r.

3. Converting TNFA to corresponding deterministic automata

There are many ways of simulating the operations of a TNFA deterministically. The main differences are in the simulation mechanisms, i.e. whether and how much backtracking is used, and how the sets of possible tag values are handled during matching. The reasonable deterministic matchers use $O(|w|)$ time and $O(1)$ space, where w is the string to be processed. Performing under an $O(|w|)$ time bound means that a backtracking algorithm cannot used.

As with traditional finite automata, the usual time-space tradeoffs apply; converting a TNFA into a deterministic automaton may take a lot of time, but needs to be done only once, and the resulting automaton can process characters very fast. A deterministic automaton may also take much more space to store than a corresponding nondeterministic automaton, and time and space can be wasted into computing transitions that are never used. Simulating a TNFA takes less space, but is slower than with a deterministic automaton. Finally, the lazy transition evaluation approach can be used, where a deterministic automaton is constructed transition by transition as needed, possibly keeping only a limited amount of previously calculated transitions in a cache.

Before going further, we need some definitions. For saving possible tag values during TDFA operation we use finite maps (or functions), one per tag. The map for tag t_x is m_x. The values of maps are are set with operations of the form $m_x^i \leftarrow k$, which assign the value of k to m_x^i. The subscript is the tag identifier, the superscript is the index to the map. Writing $m_x^i \leftarrow k$ is thus just a shorthand for writing

$$m'_x(j) = \begin{cases} k & \text{if } j = i \\ m_x(j) & \text{otherwise,} \end{cases}$$

and then letting $m_x = m'_x$.

Sometimes we will need the tag value mappings to be reordered. We use a similar notation for this, for example $m_0^0 \leftarrow m_0^1$ will set the value of m_0^1 to m_0^0.

Also, we define C to be the set of all possible lists of reordering commands and assignment commands of the form $m_x^i \leftarrow k$, and define the variable p, which always contains the number of read input symbols, i.e. the current position in the input string.

Definition 2. *A TDFA is a 7-tuple $M = (K, \Sigma, \delta, s, F, c, e)$, where*

> K *is a finite set of* states*,*
>
> Σ *is an alphabet, i.e. a finite set of symbols,*
>
> δ *is the* transition function*, a function from $K \times \Sigma$ to $K \times C$*
>
> $s \in K$ *is the* initial state*,*
>
> $F \subseteq K$ *is the set of* final states*,*
>
> $c \in C$ *is the* initializer*, a list of commands to be executed before the first symbol is read, and*
>
> $e \in F \times C$ *is the set of* finishers*, a list of commands to be executed after the last symbol is read. Each final state has its own finisher list.*

The algorithm for converting TNFAs to corresponding deterministic automata has the same basic principle as the subset construction algorithm used to convert traditional NFAs to traditional DFAs: it enumerates all possible situations which may occur when the nondeterministic automaton consumes an input string. Of course, all possible tag values cannot be enumerated finitely. Fortunately, it is not necessary, all we need to keep track of is where tag values are stored, the actual values can be ignored, and incremental changes to the stored value sets can then be made in the TDFA transitions.

3.1. A small example

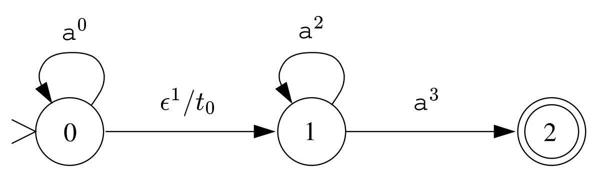

Figure 2. An example TNFA

We present the algorithm first by means of a simple example, and pseudo code will follow in the next section. The example TNFA is shown in figure 2. The priorities of the literals are marked with superscript integers. The TNFA corresponds to the POSIX.2 leftmost matching regular expression `(a*)a*a` so that t_0 gives the extent of the parenthesized subexpression.

Now we begin to generate the TDFA, the first step is to find the initial state. The initial state of the TNFA is 0, and there is a tagged ϵ-transition from 0 to 1, so following the definition of $\models_M^*$ in the previous section, the TNFA can stay in state 0 ($\models_M^*$ is reflexive) or use t_0 and enter state 1 (because $(0, w, \epsilon, v, x) \models_M (1, w, \epsilon, v', 1)$ for any w, v, v' and x). Since the tag could be used, we need to save the current position in the input string.

From these considerations we form the initial state of the TDFA: $\{(0, \{\}), (1, \{m_0^0\})\}$ and the initialization list: $[m_0^0 \leftarrow p]$. TDFA states are represented as sets of pairs (s, t), where s is a TNFA state and t is a set of map items, pointing to the map locations where tag values are stored. If a tag is not present in a map item set, the value of the tag is -1. This particular state can be interpreted to mean that a TNFA can be either in state 0 (no tags used), or in state 1 with m_0^0 containing the value of t_0. We give the initial state the number 0 as a shorthand.

Next, when the symbol a is read, the TNFA can choose any of the following actions:

- move from $(0, \{\})$ to 0
- move from $(0, \{\})$ to 0, and then move to 1 using t_0
- move from $(1, \{m_0^0\})$ to 1
- move from $(1, \{m_0^0\})$ to 2

From this we get the second state of the TDFA: $\{(0, \{\}), (1, \{m_0^1\}), (1, \{m_0^0\}), (2, \{m_0^0\})\}$. Note the item m_0^1; the position for the second usage of t_0 is stored in the second item in the map, since we do not want to overwrite the previously stored value, as it may still be needed. Also notice that there are two possible ways the TNFA could have reached state 1: either by using t_0 before reading the symbol, or after it. The value of t_0 could be 0 or 1, respectively, so the situation is ambiguous.

Ambiguities like this are resolved by taking use of the priorities we have assigned for the transitions of the TNFA, following the definition of $\models_M^*$ in the last section. In this case, $(1, \{m_0^1\})$ is prioritized over $(1, \{m_0^0\})$, since the latter state is reached by taking a lower-priority transition last. To resolve the ambiguity, we simply remove all but the highest priority item from the set, and the TDFA state becomes $\{(0, \{\}), (1, \{m_0^1\}), (2, \{m_0^0\})\}$. We give this state the number 1. Since a new occurrence of the tag, m_0^1, is introduced in this state, the current position must be saved again with $m_0^1 \leftarrow p$. So, we add to our TDFA's transition function the entry $\delta(0, \mathtt{a}) = (1, [m_0^1 \leftarrow p])$.

Finally, we notice that the state contains the TNFA state number 2, which is a final state. Thus, state number 1 in our TDFA is also final. If the input string ends when the TDFA is in this state, the TNFA would have to be in state 2 in order to produce a match. To give tags their correct values in this situation, we add the item $(1, [t_0 \leftarrow m_0^0])$ to the finisher set.

When the symbol a is read in this state, the TNFA can choose any of the following actions:

- move from $(0, \{\})$ to 0
- move from $(0, \{\})$ to 0, and then move to 1 using t_0
- move from $(1, \{m_0^1\})$ to 1
- move from $(1, \{m_0^1\})$ to 2

In the same way as before, we get the state $\{(0, \{\}), (1, \{m_0^2\}), (2, \{m_0^1\})\}$, and the command $m_0^2 \leftarrow p$. Now comes the tricky part: since it does not matter what values the tags actually have, we can freely move the values in the maps behind different keys if we reflect the changes to the TDFA state, too. In this particular situation, the map indices may all be subtracted by one, and the resulting state is the same as state 1. Thus, we add the transition $\delta(1, \mathtt{a}) = (1, [m_0^0 \leftarrow m_0^1, m_0^1 \leftarrow p])$ to the transition function.

Now there is nothing more to do and our TDFA is complete. It is depicted in figure 3. The finisher for state 1 is drawn as an arrow which leaves state 1 but enters no other state. The reader may wish to verify that the TDFA indeed matches the same language and always produces the same value for t_0 as the original TNFA.

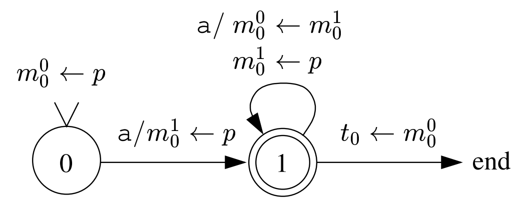

Figure 3. A TDFA corresponding to the TNFA in figure 2

In the next section an algorithm is given for doing this kind of conversions for arbitrary TNFAs mechanically.

4. The conversion algorithm

The *reach* procedure below takes as arguments a TDFA state t which is a set of pairs $(state, tags)$ in S, where $state$ is a TNFA state, and $tags$ is a set of map items m_n^i which tell where tag values are stored. The other argument is an input symbol a. The procedure calculates which states can be reached from the states in t using a transition labelled with a, and returns the result as a set of pairs like t.

Note that when map items of the form m_i^n are used here, and in the other two procedures, they do not refer to any values, but act as symbolic names for the places where the actual tag values are stored.

reach(t, a)
  $r \leftarrow$ empty list
  for each transition from u to some state u' labelled
    with a, such that $(u, k) \in t$ for some k **do**
   add (u', k) to r
  return r

The *tϵ_closure* procedure below takes as argument a set of pairs, and calculates which states are tag-wise unambiguously reached from the given states, and returns the result as set of pairs (u, k), where u is a TNFA state and k is a set of map items.

```
tϵ_closure(S)
    for each (u, k) ∈ S do
        push (u, 0, k) to stack
    initialize closure to S
    while stack is not empty
        pop (s, p, k), the top element, off of stack
        for each ϵ-transition from s to some state u do
            if the ϵ-transition was tagged with t_n then
                if ∃m : m_n^m ∈ k then
                    remove m_n^m from k
                add m_n^x to k, where x is the smallest
                nonnegative integer such that m_n^x does not
                occur in S
            if ∃p', k' : (u, p', k') ∈ closure and p < p' then
                remove (u, p', k') from closure
            if (u, p, k) ∉ closure then
                add (u, p, k) to closure
                push (u, p, k) onto stack
    remove the middle element, the priority, from all
    triples in closure
    return closure
```

The next procedure is the main conversion algorithm, and is basically just the subset construction (see for example [1] page 118) modified for tagged automata.

```
TNFA_to_TDFA()
    i ← tϵ_closure(s)
    for each map item m_n^0 in i
        add m_n^0 ← p to the initializer
    add i as an unmarked state to states
    while there is an unmarked state t in states do
        mark t
        for each input symbol a do
            u ← tϵ_closure(reach(t, a))
            c ← empty list
            for each map item m_n^i in u do
                if m_n^i is not in k then
                    append m_n^i ← p to c
            if there is a state u' ∈ states such that u can
              be converted to u' with a series of reordering
              commands R then
                append R to c
            else
                u' ← u
                add u' as an unmarked state to states
            δ(t, a) = (u', c)
            if u' contains a final state then
                choose the smallest state and set the finisher
                of u' correspondingly
```

For simplicity, the above algorithm assumes that only ϵ-transitions can be tagged, if more than one transition lead to the same state they are all ϵ-transitions and that there are no transitions with strings longer than one symbol. Any TNFA can be easily modified without changing its behavior to follow the above restrictions, so generality is not lost here.

Once the algorithm is understood, it is not difficult to see that it always terminates. In the following some fairly simple theorems and proofs are given:

Theorem 1. *The algorithm generates a finite amount of TDFA states for any TNFA.*

Proof. There are only a finite amount of TNFA states. A TDFA state can have only one item for each TNFA state, so letting T denote the number of states in the TNFA, for any TDFA state S it holds that $|S| \leq T$. There are also a finite amount of different tags, because there are a finite amount of transitions. Furthermore, in any map item set there is at most one item per tag. From these it follows that there can be at most as many used items in any map as there are states in the TNFA. Thus, there are a finite amount of possible map item sets, and therefore a finite amount of possible TDFA states. □

As a consequence of theorem 1, the algorithm always terminates.

Theorem 2. *A TDFA generated from a TNFA with the algorithm processes an input string s in $O(|s|)$ time.*

Proof. The TDFA is deterministic and the amount of operations for all transitions is constant during operation. The theorem immediately follows. □

Perhaps a more interesting result is the following, which assumes that a reorder of n items takes n operations:

Theorem 3. *The maximum amount of operations associated with any TDFA transition generated with the algorithm is $s \times t$, where s in the amount of states in the original TNFA and t is the amount of different tags in the TNFA.*

Proof. First we note that the algorithm obviously creates at most one $m_a^n \leftarrow p$ operations per tag per transition, so they account for at most t operations per transition. Also, at most s reordering operations per tag per transition need to be made, since there are at most s used items in any map at any time and they can be permuted in any order with s operations. There are t maps, so there can be at most $s \times t$ operations for any transition. □

It should be noted that better theoretic upper bounds for operations per transition could be reached by using more elaborate mechanisms for storing tag values. The constant factors involved with more complicated methods far outweigh the theoretical maximum amount in practice, so that simple solutions are usually better. Using deletion relaxed queues with unidirectional flow would yield an upper bound of $O(t^2)$ for work per transition and a slight penalty in the number of TDFA states, but in practice the penalty will be small and the work will be very near $O(t)$, except in pathological cases.

Simply leaving out all reordering operations a maximum of t operations per transition can be easily achieved. Doing this will blow up the amount of states in the TDFA to be $O(s!)$ for all but the simplest cases, which is not acceptable for most uses.

5. Applications

TNFAs behave very similarly to ordinary NFAs, so that many of the applications of NFAs can be readily adapted for TNFAs while still retaining the advantage of using tags. As the primary motivation for my work on TNFAs was using them with regular expressions, they are what this section mostly considers.

5.1. Regular expressions with tags

Thompson's construction (see for example [1] pages 122–125) can be used to create NFAs from regular expressions. To make use of tags, we give the symbol t_x a special meaning: instead of being interpreted as a literal symbol, an ϵ-transition with tag t_x is created.

For example, say we want to match a string of one or more digits, followed by any number of characters, and want to know where the last digit was without having to scan the first part of the string again. We can use the tagged regular expression `[0-9]*`t_0`[a-z]*`, and the value of t_0 tells us the index of the first character after a successful match.

Consider the regular expression `(a|b)*`t_0`b(a|b)*`, and the input string `abba`. Clearly this is ambiguous, and t_0 has two possible values, 1 or 2. To resolve this ambiguity, we may choose to prefer the *leftmost match*, that is, input symbols are matched as much as possible to the left in the regular expression. In this case, the first `a` and `b` would be matched by the first part of the regular expression (to the left of t_0), and second `b` and `a` would be matched by the part after t_0. Thus, t_0 would get the value 2.

In the case of automata generated with this extended Thompson's construction from regular expressions, the places where ambiguity may arise are as follows:

- The state where the two subautomata of a union are combined. Either the left subexpression or the right subexpression must be preferred.
- The last state of the subautomaton of a repeated subexpression, and the state where the looping ϵ-transition leads to. Either zero repetitions or one or more repetitions must be preferred.

To follow, for example, POSIX.2 `regexec` semantics, the left subexpression should always have higher priority than the right, and thus also zero repetitions should be preferred over one or more repetitions (while still searching for the longest match). If the semantics need to be changed, the automaton method is probably more flexible than a backtracking algorithm, because the matching algorithm or automaton generator itself need not be modified, just the input data or the Thompson's construction phase, which is fairly simple. For example, to get the rightmost match instead of the leftmost match, choosing exactly the opposite preferences is enough. It is further possible to use, for example, rules which apply to individual characters instead of whole subexpressions by giving each literal a unique priority and propagating them to the decision transitions listed above.

Tags also provide a mechanism to implement the *lookahead* operator `/` in the style of Lex [5] and Flex [10], so that the regular expression `abc/def` matches `abc` only when followed by `def`, but `def` is not part of the matched string. According to Appel [2], Lex uses an incorrect algorithm described by Aho et al. [1], which fails, for example, on `(a|ab)/ba` with input `aba`, matching `ab` where it should match `a`. Flex uses a better mechanism which works in the above case, but fails with a warning message on `zx*/xy*`. Using a tag in place of `/` to tell the length of the match the lookahead operator can be implemented correctly.

Regular expressions with tags can be used virtually everywhere where normal regular expressions can be used. These applications include lexical analyzers, parsers for grammars with regular right parts and a multitude of other things. I have implemented a regular expression matching and searching library and a lexical analyzer library which support tags.

5.2. Searching for matching substrings

We introduce the any-symbol, $\bullet$, to the syntax of our regular expressions. With this, it is easy to specify the language which contains all strings which contain some substring specified by a regular expressions. For example, if we want to recognize strings which contain a substring which matches the regular expression R, we can use $\bullet{*}R$. If we give the $\bullet$-loop a lower priority than any other transition and add a tag to mark the position where the matching substring begins, after the first loop, we have a working solution. [7]

We need to ensure, though, that the implementation does something reasonable. Usually searching is defined so that it finds the first and longest matching substring, which is reasonable in most applications. Matching, on the other hand, is usually defined so that it finds the longest matching prefix of the string. If we simply use the matching procedure as-is for searching, we end up finding the last and longest matching substring, instead of the first and longest.

The automata generated for searching are equivalent to the Knuth-Morris-Pratt pattern-matching algorithm, only so that instead of fixed strings, arbitrary regular expressions can be used. This means that we have also automatically a searching algorithm equivalent to the Aho-Corasick algorithm for searching several fixed string patterns simultaneously. And, when we use TNFAs, the searching algorithm is further extended to support substring addressing with the use of tags. [7, 12]

Of course, the method for searching matching substring stated here is not the most efficient one, and it is possible to achieve sublinear expected time for any regular expression, even logarithmic expected time, if preprocessing of the text to be searched is allowed using the algorithms developed for regular expressions by R. Baeza-Yates [4, 3], or in the case of regular grammars by B. W. Watson [13, 14].

6. Conclusions

I have described an efficient way to match regular languages and extract last points of use for selected transitions in a single pass. The algorithms are used in a regular expression matching and search library, and a lexical analyzer library, for a high-level, functional prototype language called Shines [9] which is being developed in a research project joint between Helsinki University of Technology and Nokia.

The main reason why we did not settle for existing libraries is that Shines is real-time but existing libraries are not. Speed is essential for us, so an exponential time backtracking algorithm was not a viable option, even though the pathological cases are rare in practice.

It has been claimed that it is easier to write a backtracking matcher than an automaton based one. This claim is contrasted by my experience with writing the TNFA to TDFA compiler; it probably would have been easier to write a backtracking matcher, but it was much easier to be sure that the automaton based matcher does what it is supposed to do and it is easier to understand what it does and why.

I tested the library by generating pseudo-random regular expressions and strings which should or should not match. The regular expressions and strings were fed to existing matchers and the TDFA matcher. I immediately discovered numerous bugs and performance problems in the GNU regexp-0.12 library, and two bugs in the GNU rx-1.8a library. Both libraries are widely used.

It is still an open problem whether the backreference operator, defined for example in POSIX.2, can be incorporated into the TNFA method of regular expression matching. Having a backreference operator means moving out of the realm of regular languages, which has proved to be a nuisance for many regexp package implementors.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1999.

[3] R. A. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 46–62, Berlin, July 1989. Springer.

[4] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *ACM*, 43(6):915–936, Nov. 1996.

[5] M. Lesk. Lex — a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, NJ, 1975.

[6] H. R. Lewis and C. H. Papadimitrou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[7] I. Nakata. Generation of pattern-matching algorithms by extended regular expressions. *Advances in Software Science and Technology*, 5:1–9, 1993.

[8] I. Nakata and M. Sassa. Regular expressions with semantic rules and their application to data structure directed programs. *Advances in Software Science and Technology*, 3:93–108, 1991.

[9] K. Oksanen. *Real-time Garbage Collection of a Functional Persistent Heap*. Licentiates Thesis, Helsinki University of Technology, Faculty of Information Technology, Department of Computer Science, 1999. Also available at `http://hibase.cs.hut.fi/`.

[10] V. Paxson. *Flex — Fast Lexical Analyzer Generator*. Lawrence Berkeley Laboratory, Berkeley, California, 1995. `ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz`.

[11] S. Sippu and E. Soisalon-Soininen. *Languages and Parsing*, volume 1 of *Parsing Theory*, pages 65–113. Springer, 1988.

[12] G. A. Stephen. *String Searching Algorithms*, volume 3 of *Lecture Notes Series on Computing*. World Scientific Publishing, 1994.

[13] B. Watson. Implementing and using finite automata toolkits. *Journal of Natural Language Engineering*, 2(4):295–302, Dec. 1996.

[14] B. Watson. A new regular grammar pattern matching algorithm. In J. Diaz and M. Serna, editors, *Proceedings of the European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 364–377, Berlin, Sept. 1996. Springer.