Efficient submatch addressing for regular expressions

Master's Thesis

Ville Laurikari

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyopin laboratorio

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Otaniemi 2001

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF MASTER'S THESIS

Author:	Ville Laurikari	
Name of the thesis:	Efficient submatch addressing for regular expressions	
Date:	November 1, 2001	**Number of pages:** 63
Department:	Faculty of Information Technology	**Professorship:** Tik-106
Supervisor:	Professor Eljas Soisalon-Soininen	
Instructor:	Kenneth Oksanen, M.Sc., Lic.Sc. (Tech.)	

String pattern matching in its different forms is an important topic in theoretical computer science. This thesis concentrates on the problem of regular expression matching with submatch addressing, where the position and extent of the substrings matched by given subexpressions must be provided.

The algorithms in widespread use at the time either take exponential worst-case time to find a match, can handle only a subset of all regular expressions, or use space proportional to the length of the input string where constant space would suffice. In this thesis I propose a new method for solving the submatch addressing problem using nondeterministic finite automata with transitions augmented by copy-on-write update operations.

The resulting algorithm makes a single pass over the input string, always using time linearly proportional to the input. Space consumption depends only on the used regular expression, and not on the input string. To the author's knowledge, this is a new result. A prototype of a POSIX.2 compatible regular expression matcher using the algorithm was done. Benchmarking results indicate that the prototype compares favorably against some popular implementations. Furthermore, absence of exponential or polynomial time worst cases makes it possible to use any regular expression without performance problems, which is not the case with previous implementations or algorithms.

Keywords: regular expressions, submatch addressing, parse extraction, regular expression parsing, approximate regular expression matching

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Ville Laurikari		
Työn nimi:	Säännöllisten lausekkeiden tehokas osittainen jäsentäminen		
Päivämäärä:	1.11. 2001	**Sivuja:**	63
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-106
Työn valvoja:	professori Eljas Soisalon-Soininen		
Työn ohjaaja:	tekn. lis. Kenneth Oksanen		

Hakulauseketta vastaavien osien etsiminen merkkijonoista on eri muodoissaan tärkeä tietojenkäsittelyteorian alue. Tämä diplomityö keskittyy säännöllisiin lausekkeisiin ja niiden määrittelemään kieleen kuuluvien merkkijonojen tehokkaaseen osittaiseen jäsentämiseen. Osittainen jäsentäminen tarkoittaa säännöllisen lausekkeen mielivaltaisesti valittuja osalausekkeita vastaavien osamerkkijonojen määrittämistä koko lausekkeen määrittelemään kieleen kuuluvassa merkkijonossa.

Tällä hetkellä laajassa käytössä olevat algoritmit joko kuluttavat pahimmassa tapauksessa eksponentiaalisesti aikaa merkkijonojen tutkimiseen, käyttävät tilaa suoraan verrannollisesti syötejonon pituuteen vaikka vakiotila riittäisi, tai pystyvät käsittelemään vain säännöllisten lausekkeiden osajoukkoa. Tässä diplomityössä ehdotan uutta ratkaisua osittaiseen jäsentämiseen, jossa käytetään epädeterministisiä äärellisiä automaatteja joiden siirtymiä on laajennettu funktionaalisilla sijoitusoperaatioilla.

Kehitetty algoritmi käy syötteen vain kerran läpi, ja sen aikakompleksisuus pahimmassakin tapauksessa on suoraan verrannollinen syötejonon pituuteen. Algoritmin kuluttama tila riippuu vain käytetystä säännöllisestä lausekkeesta eikä lainkaan syötejonosta. Työssä toteutettiin myös algoritmiin perustuva POSIX.2-yhteensopiva säännöllisten lausekkeiden osittaista jäsentämistä suorittava hakukirjaston prototyyppi. Suuntaa-antavien kokeellisten mittausten perusteella prototyyppi suoriutuu hyvin verrattuna eräisiin yleisesti käytössä oleviin toteutuksiin. Lisäksi algoritmin lineaarisen aikakompleksisuuden vuoksi hauissa voidaan käyttää mitä tahansa säännöllistä lauseketta ilman kohtuuttoman ajankäytön vaaraa; tämä ei ole ollut mahdollista aikaisemmilla toteutuksilla tai algoritmeilla.

Avainsanat: säännölliset lausekkeet, osittainen jäsentäminen, likimääräinen säännöllisten lausekkeiden sovittaminen

Preface

The basis work for this thesis was done in the H$^{\text{I}}B_{\text{A}}$S$^{\text{E}}$ project, a joint research project between Helsinki University of Technology (HUT) and Nokia Networks (NET). One major goal of the project was to develop a persistent and real-time functional programming environment.

A regular expression library was needed in our project, and I decided to implement one; this was in late fall of 1999. It soon became apparent that existing algorithms and implementations were not suitable, or at least had a lot of room for improvement. Thus, I began research on submatch addressing algorithms. Eventually I came up with the idea of nondeterministic finite automata with tagged transitions (TNFAs), and even wrote a paper [31] about them. Later, in the spring 2001, when the time came to start with my Master's Thesis, this was the natural subject.

I want to thank my supervisor, Professor Eljas Soisalon-Soininen, for his support, both during H$^{\text{I}}B_{\text{A}}$S$^{\text{E}}$ and afterwards. I also want to thank Kenneth Oksanen for helping me getting started and finished, and giving valuable comments on my work.

I am also grateful to Angelo Borsotti who read and commented many versions of this thesis in detail. My gratitude also goes to the members of the H$^{\text{I}}B_{\text{A}}$S$^{\text{E}}$ project for their support and testing some early implementations of a prototype deterministic finite automata with tagged transitions (TDFA) matcher: Sami-Pekka Haavisto, Jukka-Pekka Iivonen, Vera Izrailit, Jarkko Lavinen, Antti-Pekka Liedes, Marko Lyly, Petri Mäenpää, Kenneth Oksanen, Jussi Rautio, and Matti Tikkanen. I would also like to thank Elizabeth Heap-Talvela from the HUT language center for her grammatical comments.

My thanks go also to my current employer, SSH Communications Security, for letting me use some time and resources to finally finish this thesis.

Finally, I would like to thank my family and especially Henna Pietiläinen for their patience, advice, and love.

Espoo, November 1, 2001.

Ville Laurikari

Contents

Notation and Abbreviations

$\{a, b, \ldots\}$	unordered set containing the items a, b, ...
$\emptyset$	empty set
ε	empty string
L^*	closure of the language L
$L_1 \circ L_2$	concatenation of languages L_1 and L_2
$L(r)$	language represented by regular expression r
R^*	reflexive, transitive closure of binary relation R
R^+	transitive closure of binary relation R
$\mathbb{N}$	the set of natural numbers $\{0, 1, 2, \ldots\}$
$\|w\|$	length of string w
$\vdash_M$	binary relation between configurations of M "yields in one step"
$\models_M$	binary relation between configurations of M "yields tag-wise ambiguously in one step"
$\prec_T$	total order on functions from tags to their values
$\langle n_1, n_2, \ldots, n_k \rangle$	ordered k-tuple of the items n_1, n_2, ..., n_k
$\{x : P(x)\}$	the set of all x which have property P.
$O(g(n))$	$\{f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
$E_1 \| E_2$	Regular expression such that $L(E_1 \| E_2) = L(E_1) \cup L(E_2)$.
$E*$	Regular expression such that $L(E*) = L(E)^*$.
.	Regular expression matching any single symbol in the used alphabet.

AST	abstract syntax tree
DFA	deterministic finite automaton
DFAS	deterministic finite automaton with semantic actions
NFA	nondeterministic finite automaton
NFAS	nondeterministic finite automaton with semantic actions
TDFA	deterministic finite automaton with tagged transitions
TNFA	nondeterministic finite automaton with tagged transitions

Chapter 1

Introduction

Pattern matching, despite its low-key coverage, is a very important topic in computer science. It occurs naturally in many areas of science and information processing, such as data processing, lexical analysis, text editing, and information retrieval. Indeed, pattern matching is the main programming paradigm in several programming languages like Prolog, SNOBOL4, and Icon, and most programming languages provide some kind of primitives to perform different kinds of pattern matching on strings. In biology, string pattern matching problems arise in the analysis of nucleic acids and protein sequences. Considering all this, it is not a surprise that string pattern matching is one of the most widely studied problems in theoretical computer science.

This thesis concentrates on regular expression patterns. Regular expressions [28] are very popular for describing patterns for searching text, and there are numerous tools and libraries which implement regular expression pattern matching, like lex [32] and flex [47]. Most programming languages, such as Perl [54], provide some form of regular expression pattern matching. Regular expressions and regular expression matching have recently been used even for implementing type systems for programming languages [20, 21].

It is not always enough just to perform language recognition, that is, to find out whether patterns of interest occur in the text. Frequently we need to know exactly where a substring matching the pattern was found and extract parts of a successful match. For example, if a pattern matches an address, it should be easy for the programmer to access the zip code. In the extreme case, a full parse tree of the match is required. The problem of extracting partial parse information of a match is called submatch addressing and is the main focus of this thesis.

Often the searched text is very large, emphasizing the need for efficient algorithms. For example, an algorithm using time in the order $O(n^2)$ or worse is unacceptable when searching for a pattern from several megabytes of data. Space consumption should also be as low as possible, so that no more space is used than necessary. In general a full parse tree of a string w matching a regular expression r takes $O(|w|)$ space, but in most

cases a full tree is not required, and even the full parse tree often takes only $O(|r|)$ space. Searching for a simple pattern r from a very large text is best done using an algorithm which uses space depending only on r, not the length of the text being searched.

There has been some work in the area of efficient algorithms for regular expression pattern matching with full or partial parse extraction. The algorithms in widespread use at the time of this writing either take exponential worst-case time to find a match, use $O(|w|)$ space, or can handle only a subset of all regular expressions. None of these features are desirable for a general-purpose implementation, such as a POSIX.2 [23] compatible regular expression matching library.

This thesis mostly concentrates on on-line algorithms, where preprocessing of the pattern must not take long, and the searched text cannot be indexed before the search.

This thesis has the following structure:

In Chapter 2 regular expressions and the submatch addressing problem are defined. After these a brief survey of previous work on submatch addressing and regular expression parsing is given, and the most important problems of the previous techniques are shown.

In Chapter 3 I first present nondeterministic automata which may have transitions augmented with tags, give a formal definition of their semantics, and show how to solve the submatch addressing problem using nondeterministic tagged automata. Then I discuss efficient techniques to simulate these automata, and show how they can be converted to corresponding deterministic automata. Finally, a more generic model is discussed where transitions are augmented with computable functions which manipulate some arbitrary data, and full parsing and approximate regular expression matching are discussed.

In Chapter 4 an actual implementation of some of the algorithms studied in the previous chapter is discussed.

In Chapter 5 some experimental test results using the implementation described in Chapter 4 are shown, and comparison to other implementations is done.

In Chapter 6 some directions to future work and research are given.

In Chapter 7 the conclusions gained in this thesis are summarized.

Chapter 2

Submatch Addressing for Regular Expression Matching

Regular expressions, regular sets (sometimes called rational expressions and rational sets, respectively), and finite automata are central concepts in automata and formal language theory. A regular set is a set of strings matched by a regular expression. The origins of regular sets go back to the work of McCulloch and Pitts [35] who devised finite-state automata as a model for the behavior of neural networks.

The notation of regular expressions arises naturally from the mathematical result of Kleene [28] that characterizes the regular sets as the smallest class of sets of strings which contains all finite sets of strings and which is closed under the operations of union, concatenation and Kleene closure.

This chapter first defines the syntax and semantics of regular expressions. Then the submatch addressing problem is defined and some solutions by others are discussed, showing the biggest problems of these previous solutions.

2.1 Regular Expressions

Definition 2.1 *Regular expressions* over an alphabet Σ are defined as follows:

1. ε and each member of Σ is a regular expression.
2. If r_1 and r_2 are regular expressions then so is $(r_1|r_2)$.
3. If r_1 and r_2 are regular expressions then so is $(r_1 r_2)$.
4. If r is a regular expression then so is $r*$.

Nothing is a regular expression unless it follows from a finite number of applications of the rules above. The above defines only the regular expression syntax. The meaning

of a regular expression, that is, the language represented by a regular expression, is defined using a function L, which is defined recursively as follows:

1. $L(\varepsilon) = \{\varepsilon\}$ and for each symbol a in the alphabet $L(a) = \{a\}$.
2. If r_1 and r_2 are regular expressions then $L((r_1|r_2)) = L(r_1) \cup L(r_2)$.
3. If r_1 and r_2 are regular expressions then $L((r_1 r_2)) = L(r_1) \circ L(r_2)$.
4. If r is a regular expression then $L(r^*) = L(r)^*$. □

The concatenation of two languages $L_1 \circ L_2$ is defined as

$$L_1 \circ L_2 = \{w : \ w = xy \text{ for some } x \in L_1 \text{ and } y \in L_2\}$$

L^*, the Kleene closure of a language L, is the set of all strings obtained by concatenating zero or more strings from L.

Many parentheses in regular expressions can be avoided by adopting the convention that the Kleene closure operator * has the highest precedence, then concatenation, then | (alternation). The two binary operators, concatenation and alternation, are left-associative. Under these conventions the regular expressions $(a|((b(c^*))d))$ and $a|bc^*d$ are equivalent, in the sense that they match the same strings, namely, an a, or a b followed by a sequence of zero or more c's followed by a d.

Example 2.1 For example, the regular expression

$$(hot|cold)\ (apple|blueberry|cherry)\ (pie|tart)$$

matches any of the twelve delicacies ranging from *hot apple pie* to *cold cherry tart*.

The regular expression

$$the\ (very,\ )^*very\ hot\ cherry\ pie$$

matches the strings *the very hot cherry pie*; *the very, very hot cherry pie*; *the very, very, very hot cherry pie*; and so on.

The regular expression

$$(c^*(a|(bc^*))^*)$$

represents the set of all strings over $\{a,b,c\}$ that do not have the substring ac. □

There are many popular programs, tools, and libraries for performing regular expression matching. Most of these programs implement some extensions to the regular expression notation, like *awk* [3], *lex* [32], and *flex* [47]. Extensions are usually implemented in order to provide more succinct and understandable ways to represent regular languages. In fact, the relative succinctness of different notations for regular sets has been of considerable theoretical interest [19, 37].

In the regular expression notation defined above the symbols), (, |, and * are metacharacters that are not a part of the alphabet. In computer implementations we do not have the luxury of using extra characters out of the alphabet for the regular expression notation, and a way to match the regular expression metacharacters themselves is needed. This is usually achieved by using backslash, \, as a quoting metacharacter that permits metacharacters to be matched. The metacharacters can be denoted by prefixing them with the backslash: \), \(, \|, and * match), (, |, and * respectively. The backslash itself is matched by \\.

Often we need to specify sets of input symbols in regular expressions, and using expressions of the form $(a_1|a_2|a_3|\ldots)$ can be cumbersome. Many implementations support denoting sets of characters by surrounding them with brackets. For example, [*abc*] is equivalent to $(a|b|c)$. Character sets can be negated using a caret, so that [^*abc*] matches any character except *a*, *b*, or *c*. Character sets which consist of consecutive characters can be defined using special character range notation. For example, $[a-z]$ matches any lower case character, and $[\hat{}a-zA-Z0-9]$ matches any non-alphanumeric character. The character range notation is naturally dependent on the order in which the characters are represented internally in the implementation (typically ASCII [6] or a derivative).

Further shorthands can be defined for the most often used sets of characters, the most popular of these being . which matches any single character. The expression . can be thought of as a "don't-care" or "wildcard" symbol. Another common notation is the $^+$ operator. If *r* is a regular expression, then $(r)^+$ denotes the same language as $r(r)^*$.

None of these extensions add more descriptive power to the expressions, in the sense that the languages which can be denoted by the extended expressions are still purely regular, and only regular sets can be described with these extended expressions.

One popular extension which does extend the class of representable languages is *back referencing*. Regular expressions with back referencing, or *rewbrs*, appeared in the first version of the SNOBOL programming language [16], and have since found their way into for example the UNIX command `grep` and the Perl [54] programming language.

Rewbrs have an assignment operator %, so that if for example *r* is a regular expression, then the rewbr $r\%v_0$ matches whatever *r* matches and assigns the matched string to the variable v_0. After this, the variable can be used to match that same string again. For example, the rewbr $(a|b)^*\%v_0v_0$ denotes the language $\{w : w = xx \text{ and } x \in \{a,b\}^*\}$. Repeated strings like this are called *squares* or *tandem repeats*. As another example, the rewbr $(a|b|c)^*((a|b|c)\%v_0)(a|b|c)^*v_0(a|b|c)^*$ matches any string of *a*'s, *b*'s or *c*'s with at least one repeated character.

Surprisingly, not much theoretical study of back referencing has been done. A related but restricted class of expressions has been studied by Angluin [7]. Angluin's expressions do not have the alternation operator and only one back reference is allowed. Also Larsen [30] has studied regular expressions with back referencing and showed that

the power of the expressions increase with the number of nested levels that are allowed.

Aho has also studied rewbrs, and showed that given a pattern consisting of a rewbr r and an input string s the problem of finding out whether s contains a substring matched by r is NP-complete [1]. This is perhaps one of the main reasons for lack of broad theoretical interest in rewbrs. Back referencing constructs shall not be discussed any further in this thesis.

2.2 Submatch Addressing

The extension discussed in this section, *submatch addressing*, sometimes called *substring addressing of matches*, *substring extraction*, *parse extraction*, or just parsing regular expressions, is a very useful feature implemented in many regular expression matching programs. For example, all IEEE POSIX standard [23] compatible regexp matching libraries, and the Perl [54] and SNOBOL [16] programming languages support submatch addressing.

Instead of being an extension to the regular expression notation, submatch addressing is an extension to the amount of detail given about a successful match. Not only the information of whether a match was found is given, but the substrings matching the pattern and given subpatterns are reported. In short, submatch addressing means finding the position and extent of the substring matched by a given subexpression.

For example, the regular expression *very* $(.*)$ *stick* matches the string *Jack has a very long blue stick in his hand*. To be precise, the regular expression matches the substring *very long blue stick*. The parenthesized subexpression matches the substring *long blue*, and it is a *submatch* of the whole match. Submatches can be reported as pairs of integers $\langle s, e \rangle$, where s is the position of the first character of the submatch and e is the position of the last character of the submatch plus one. The length of the submatch in characters can then be computed by $e - s$. In the above example, the submatch addressing information for the parenthesized subexpression is $\langle 16, 25 \rangle$, and the length of the submatch is 9.

To mark subexpressions for which submatch addressing needs to be done we define a new notation; the wanted subexpression is surrounded with braces, $\{$ and $\}$. The regular expression in the above example can then be rewritten using this notation as *very* $\{.*\}$ *stick*.

2.2.1 Resolving Ambiguity

It is often the case that when matching a regular expression, a subexpression of the pattern can participate in the match of several different substrings of the input string. It is also possible that a subexpression does not match any substring even though the pattern as a whole does match.

Table 2.1: Leftmost-longest matches of $\{a^*\}\{a^*\}$ and *aaa*

first subexpression	second subexpression
$\langle 0,0\rangle$	$\langle 0,3\rangle$
$\langle 0,1\rangle$	$\langle 1,3\rangle$
$\langle 0,2\rangle$	$\langle 2,3\rangle$
$\langle 0,3\rangle$	$\langle 3,3\rangle$

For example, consider the regular expression $\{a^*\}\{a^*\}$ and string *aaa*. There are twenty possible submatch addressings in all, any of which are correct. One possibility is $\langle 0,0\rangle$ for the first subexpression and $\langle 0,3\rangle$ for the second. Another possibility is $\langle 1,2\rangle$ and $\langle 2,3\rangle$, and so on.

The following rules are used to determine which substrings are chosen:

- **Leftmost-longest rule**: In the event that a regular expression could match more than one substring of the input string, the match starting earliest in the string is chosen. If the regular expression may match more than one substring at that point, the longest substring is chosen.
- **Subexpression rule**: Subexpressions also match the longest possible substrings, subject to the constraint that the leftmost-longest rule must not be violated. Subexpressions starting earlier in the regular expression take priority over ones starting later. Note that higher-level subexpressions thus take priority over their lower-level component subexpressions. Matching an empty string is considered longer than no match at all.
- **Repeated matching rule**: If a subexpression matches more than one substring of the whole match, the last such substring is chosen. Note that the candidate substrings cannot overlap.

The rules are in order of decreasing priority. The subexpression rule is applied to each subexpression in order, regardless of which subexpressions are marked for submatch addressing.

Example 2.2 The submatch rule tells us to choose the addressing on the Let us match the regular expression $\{a^*\}\{a^*\}$ and string *aaa*. The leftmost-longest rule requires that the whole string is matched. This restriction cuts down the number of possible substring addressings to the four leftmost-longest matches shown in Table 2.1.

The submatch rule tells us to choose the addressing on the last row, because it has the longest match for the first subexpression. □

Example 2.3 As another example, consider the regular expression $(a|a^*)^*$. The syntax tree for this expression is shown in Figure 2.1. Each subtree is numbered with a number from 1 to 5.

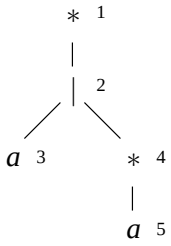


Figure 2.1: Syntax tree for $(a|a^*)^*$

Table 2.2 shows the submatches for each subtree for some input strings. Matching the empty string demonstrates the rule that an empty match is considered longer than no match at all; subtree number 4 can match the empty string and therefore it must match the empty string, although this would not be necessary to make the whole expression match.

Table 2.2: Submatch addressings for $(a|a^*)^*$ against some strings

string	1	2	3	4	5
ε	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle -1,-1\rangle$	$\langle 0,0\rangle$	$\langle -1,-1\rangle$
ba	$\langle 0,0\rangle$	$\langle 0,0\rangle$	$\langle -1,-1\rangle$	$\langle 0,0\rangle$	$\langle -1,-1\rangle$
a	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle 0,1\rangle$	$\langle -1,-1\rangle$	$\langle -1,-1\rangle$
aa	$\langle 0,2\rangle$	$\langle 0,2\rangle$	$\langle -1,-1\rangle$	$\langle 0,2\rangle$	$\langle 1,2\rangle$
aaa	$\langle 0,3\rangle$	$\langle 0,3\rangle$	$\langle -1,-1\rangle$	$\langle 0,3\rangle$	$\langle 2,3\rangle$

The second row on the table demonstrates the leftmost-longest rule. It would be possible to match the longer substring starting from the second character, but the leftmost, and in this case shorter, match is chosen.

The third row shows that subexpressions starting earlier take priority over ones starting later. In terms of a regular expression syntax tree, a depth first preorder traversal of the tree enumerates the subexpressions in order of priority. The subtrees in Figure 2.1 are numbered like this. Here, subtree number 3 takes priority over subtree number 4, so the one character is matched by subtree 3 instead of 4.

The fourth and fifth row demonstrate how higher-level subexpressions take priority over their lower-level component subexpressions. It would be possible to make the match by letting subtree 3 match the two a's by making two iterations with the topmost star operator. But since subtree 2 takes priority over its components, we must choose the match which has the longest submatch for subtree 2. □

The ambiguity resolving scheme described here is, of course, only one of numerous alternatives. The approach used here has almost identical semantics to the one used in [23]. Naturally, these rules are not good for every situation; in fact, the generally accepted leftmost-longest rule has been the subject of some criticism [11]. The main

argument is that searching for longest matching substrings usually results in more complicated patterns when searching structured text, such as XML [14].

2.3 Previous Work

The rest of this chapter describes briefly some solutions to the submatch addressing problem developed by others. Each subsection describes a different solution.

2.3.1 Backtracking Matchers

Most regular expression matching software which support substring addressing do not use the textbook NFA or DFA methods for matching regular expressions, but an interpretive backtracking algorithm and a stack of backtracking points.

There are two major advantages of the backtracking method — it is easy to implement and it allows extensions like submatch addressing and back referencing [1, 7, 30] to be incorporated easily.

There is some amount of history in the evolution of backtracking algorithms which can still be seen in the versions used today. The original backtracking algorithms supported only a subset of the regular expression syntax, the alternation operator | was not supported at all. This made it possible to implement a backtracking algorithm which finds the longest match without extra backtracking.

When | is added, it becomes possible to cheat the backtracking algorithm into making a poor choice early on that produces a less-than-longest match in the end. Many of the implementors did not notice this; their documentation still claims longest match, even though they do not always find it. In order to find the longest match, the algorithm will have to explore every possible match, and this can be spectacularly expensive even for relatively simple expressions. For example, the GNU regex-0.12 library consumes exponential time when matching the regular expression $(a^*)^*|b^*$ with input of the form $aaaaaaa\ldots b$. With an input of only approximately 25 characters the matching takes tens of seconds on a current workstation.

On the other hand, Perl [54] takes the easy way out; it does not even try to return the longest match. This can be very confusing. As an example, take the Perl regular expressions `(a|ab)(bc)?` and `(ab|a)(bc)?`, and the strings `ab` and `abc`. The Perl program

```
"ab"  =~ /(a|ab)(bc)?/; print($&, "\n");
"abc" =~ /(a|ab)(bc)?/; print($&, "\n");
"ab"  =~ /(ab|a)(bc)?/; print($&, "\n");
"abc" =~ /(ab|a)(bc)?/; print($&, "\n");
```

outputs the following:

```
a
abc
ab
ab
```

Each line in the program matches the string on the left-hand side of the `=~` operator against the regular expression between the `/` characters. The matching substring is then printed.

Even though it would be possible for each line in the program to match the whole string, it does not always happen. Namely, the first and last lines of the program do not find the longest match. This is confusing for a programmer who does not know how the Perl regular expression matcher works, and may even be misinterpreted as a bug. There are also cases which take a very long time to run, even though Perl tries to limit the amount of backtracking by not guaranteeing longest matches. For example, this program

```
"aaaaaaaaaaaaaaaaaaaaaaaaab" =~ /((a*)*b)*b/;
```

takes tens of seconds to run (using Perl version 5.005_03) on current desktop hardware. This too may be misinterpreted as an “infinite loop” bug.

The Perl regexp matcher is notoriously complex and contains a number of different tricks and optimizations to avoid situations like the above where matching takes exponential time. Still, no number of tricks will cover every possible situation, and there is a limit to the number of optimizations which can be applied until the program code becomes unmaintainable.

2.3.2 Nakata-Sassa Semantic Rules

Nakata and Sassa have proposed *regular expressions with semantic rules* [43], which can be used as tools for expressing the syntax and semantics of input data, and a method of generating programs for processing these input data. Their regular expressions can have intermixed semantic statements, which can conceivably be extended to implement submatch addressing instead of using backtracking algorithms described above.

For example, in the regular expression

$$(\alpha|\beta)\gamma$$

we can insert the semantic rules f and g by writing

$$([f](\alpha|\beta)[g])\gamma$$

The rules f and g can be implemented to store the current position in the input string to, say, the variables t_0 and t_1 respectively. After a successful match, $\langle t_0, t_1 \rangle$ would then be the submatch addressing data for the parenthesized subexpression.

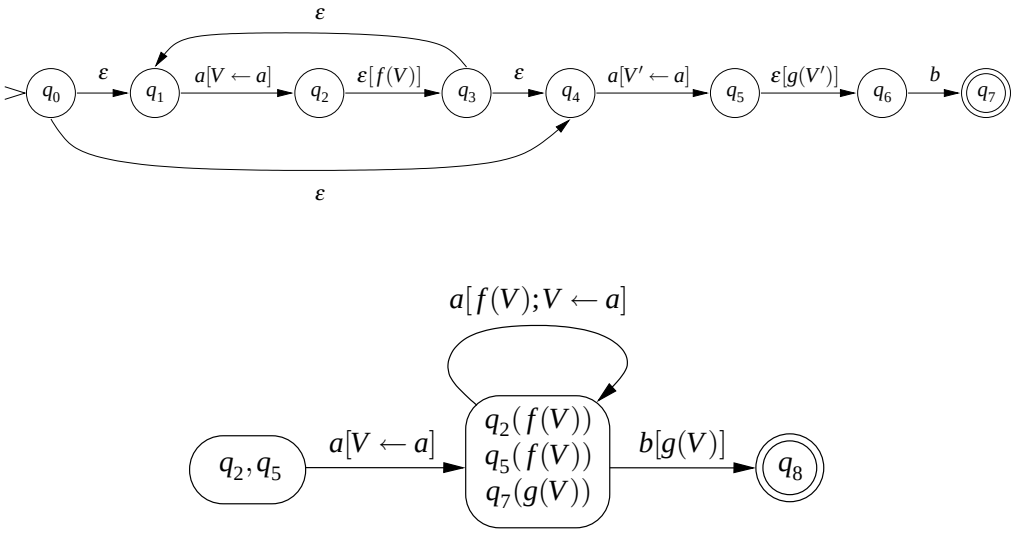


Figure 2.2: NFAS and DFAS for $(a[f(a)])^*a[g(a)]b$

The basic idea in the implementation of Nakata-Sassa semantic rules is that a processing program for an expression with semantic rules can be expressed as a finite automaton for the underlying regular expression with semantic actions attached to the proper transitions. These automata are called *nondeterministic finite automata with semantic actions* (*NFAS*) and *deterministic finite automata with semantic actions* (*DFAS*).

Nakata and Sassa do not discuss efficient methods for simulating nondeterministic automata with semantic action transitions, but give an algorithm for translation from nondeterministic finite automata with semantic actions to corresponding deterministic automata. Their algorithm, however, fails to produce correct deterministic automata for classes of important nondeterministic automata, as we shall soon see.

In the Nakata-Sassa system, each state of the nondeterministic automaton to convert is assigned a temporary variable which is used to postpone execution of semantic actions in cases where look-ahead is necessary. This is the weak spot of the method, and makes it impossible to use it to implement for example a POSIX.2 [23] conformant regular expression matching library.

For example, the expression $(a[f(a)])^*a[g(a)]b$ works correctly (see Figure 2.2), whereas $(a[f(a)])^*a[g(a)]ab$ cannot be implemented (see Figure 2.3), because $V \leftarrow a_2$ is to be executed at the transition from Q_2 to Q_3, while $f(V)$ for $V \leftarrow a_2$ has not yet been evaluated. Nakata and Sassa note that the previous case could be implemented by increasing the number of variables from one to two (by changing the assignments into $V_1 \leftarrow a_1$ and $V_2 \leftarrow a_2$, and changing $2(f(V); f(V))$ in Q_3 to $2(f(V_1); f(V_2))$).

However, they fail to point out that this does not help in the general case, because if there is some finite number of n variables per state, the automaton generated from a

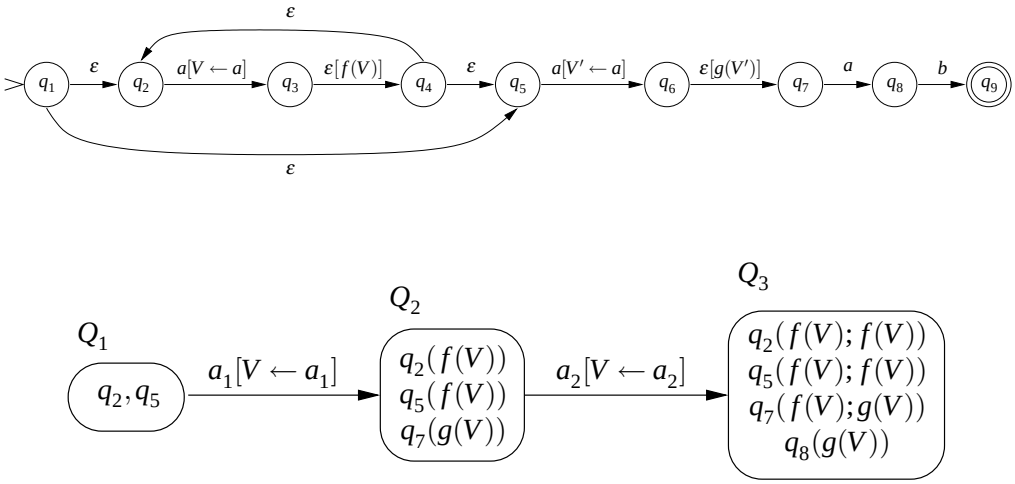


Figure 2.3: NFAS and partial DFAS for $(a[f(a)])^*a[g(a)]ab$

regular expression of the form

$$(a[f(a)])^*\overbrace{a\ldots a}^{n+1}$$

does not work, because $n+1$ variables would be needed per state to implement a matcher using the Nakata-Sassa method. All the algorithms given in their paper [43] also assume just one variable per state, and increasing the number of variables per state is only briefly mentioned. Also, Nakata and Sassa do not discuss resolving ambiguity at all; there are many important cases where submatch addressing and semantic actions can be done in different ways (see Section 2.2.1).

2.3.3 Kearns's Parse Extraction

In his paper [25], Kearns describes a method for extracting a parse after matching with a finite automaton. First he shows algorithms to find matches of regular expressions patterns in strings.

One by-product of the matching process described is a sequence of states $Q_0, Q_1, \ldots, Q_n$, such that Q_0 is the initial state and Q_n is an accepting state. The whole sequence of states is written Q and the ith state as Q_i. Each Q_i is actually a set of places in the parse tree for the regular expression pattern p being searched for.

Kearns gives a recursive algorithm which operates on the sequence of states Q and can be used to build a full parse tree of the match. He shows that the algorithm is optimal in space and time. The algorithm to build the parse tree is indeed optimal in this regard, but the sequence Q needs $O(|w|q)$ space to store for an input string w and pattern of size q. The sequence Q is not needed for anything else but parse extraction, so the actual space complexity of Kearns's algorithm is, in fact, not optimal for cases

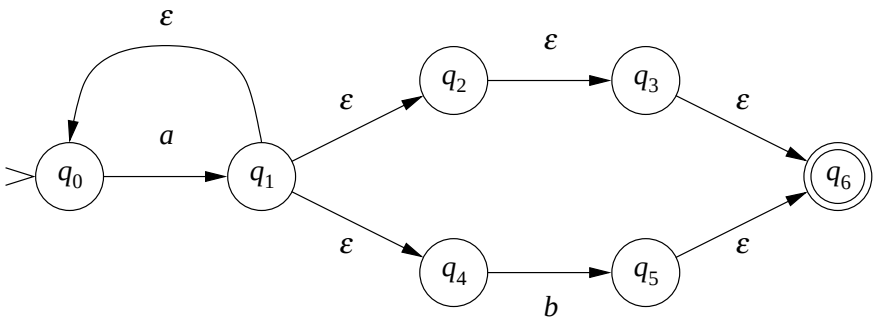


Figure 2.4: NFA for $a^+(\varepsilon|b)$

where the parse tree or partial parse tree takes less than $O(|w|q)$ space to store.

As an example we simulate the NFA in Figure 2.4, which represents the pattern $a^+(\varepsilon|b)$, on the input *baab*. The following sequence $Q_1 \ldots Q_5$ is calculated:

$$\begin{array}{lcll} Q_1 & = & \{q_0\} & !baab \\ Q_2 & = & \{q_0\} & b!aab \\ Q_3 & = & \{q_0, q_1, q_2, q_3, q_4, q_6\} & ba!ab \\ Q_4 & = & \{q_0, q_1, q_2, q_3, q_4, q_6\} & baa!b \\ Q_5 & = & \{q_0, q_5, q_6\} & baab! \end{array}$$

The exclamation mark is used to show the current position in the input string. To the left of the exclamation mark is the already processed input, and to the right is the unprocessed part.

Since the end state q_6 is in Q_3, Q_4 and Q_5, but not in Q_1 or Q_2, we conclude that the empty string and the *b* at the start of the input do not match our pattern, but some suffix of the strings *ba*, *baa*, and *baab* does. Now, using a rather simple recursive algorithm on the sequence of states Q, a full parse tree of any of these matches can be built.

Kearns's algorithms are used for example in the TLex [24, 26] code generator.

2.3.4 Others

Dubé-Feeley Parse Tree Automata

Dubé and Feeley proposed an algorithm for regular expression parsing in their paper [13]. Their algorithm uses $O(|r||w|)$ space for pattern r and string w, like Kearns's algorithm.

Combinatorial Approaches

Myers et al [40] showed an algorithm for parsing regular expressions which takes $O(c4^kPN)$ time and space, where c is the number of tagged subexpressions (subexpressions for which submatch addressing is wanted), k is the number of properly nested subexpressions in the pattern, P is the size of the regular expression pattern, and N is the length of the input string. They note that it would be possible to modify their algorithm to get an $O(cM_RPN + T_R)$ time and space algorithm, where M_R and T_R are factors which depend on the pattern searched. In the worst case, M_R and T_R still grow exponentially with P. In any case the space complexity is dependent of the length of the string and therefore the algorithm is not suitable for partial parsing needed in submatch addressing.

Chapter 3

Automata with Augmented Transitions

In this chapter I propose a new method for solving the submatch addressing problem efficiently. A new model of computation created by augmenting transitions of traditional finite automata to manipulate location data is presented. The model is applied to solve the submatch addressing problem. Algorithms to efficiently simulate the augmented automata are given.

This chapter also discusses some problems related to submatch addressing, namely full parsing and approximate regular expression matching. These problems can be solved by generalizing the augmented transition model described in the next section.

3.1 Nondeterministic Automata with Tagged Transitions

To solve the submatch addressing problem (and with some generalizations a range of related problems) using automata, I propose a model where transitions can be augmented with *tags*. These augmented transitions are called *tagged transitions*. 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.

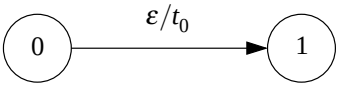


Figure 3.1: A tagged transition

If a tag is unused, it has a special value, -1. Initially all tags are unused and have this value. A tag and its variable are synonymous, so if we refer, say, to the variable t_5, we mean the variable of tag t_5. Figure 3.1 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 $\mathtt{a}/\omega$ is written $\mathtt{a}$ and ε/ω is written ε.

At first glance automata with tagged transitions are reminiscent of finite-state transducers sometimes used for parsing purposes [22, 49, 50], but the semantics are different. 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 encountered, the places in the input string where they were last seen. The following definitions formalize this idea.

Definition 3.1 A *nondeterministic finite automaton with tagged transitions*, or *TNFA*, is a 7-tuple $M = \langle K, T, \prec_T, \Sigma, \Delta, s, F \rangle$, where

K is a finite set of *states*,

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

$\prec_T$ is a total order on items of V. V is the set of all functions from $T - \{\omega\}$ to $\mathbb{N} \cup \{-1\}$. Members of V are called *tag value functions*.

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

Δ is the *transition relation*, a finite subset of $K \times \Sigma^* \times T \times K$.

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

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

The meaning of a quadruple $\langle q, u, t, p \rangle \in \Delta$ 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.

Definition 3.2 A *configuration* of M is an element of $K \times \Sigma^* \times \Sigma^* \times V$, 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, and the fourth item is a tag value function giving a value for each tag. The initial tag values are specified by $v_0 = (T - \{\omega\}) \times \{-1\}$. An *initial configuration* is a quadruple $\langle s, \varepsilon, w, v_0 \rangle$ for some input string w. □

Definition 3.3 The relation $\vdash_M$ between configurations (*yields in one step*) is defined as follows: $\langle q, p, u, v \rangle \vdash_M \langle q', p', u', v' \rangle$ if and only if there are $w \in \Sigma^*$ and $t \in T$ such that $u = wu'$ and $\langle q, w, t, q' \rangle \in \Delta$. 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 $q \in F$ and a function v such that $\langle s, \varepsilon, w, v_0 \rangle \vdash_M^* \langle q, w, \varepsilon, v \rangle$. □

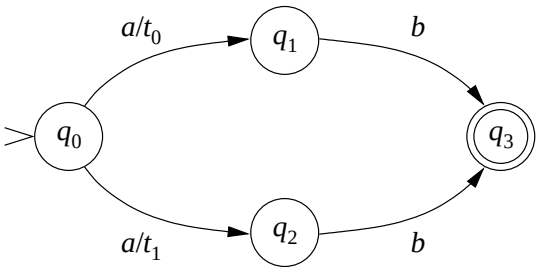


Figure 3.2: An example TNFA

Example 3.1 Figure 3.2 shows a simple example TNFA. The automaton is drawn as a directed graph with certain additional information incorporated into the picture. Like traditional finite automata, states are represented by nodes, and transitions by arrows labeled with w/t from node q to q' whenever $\langle q,w,t,q'\rangle \in \Delta$. The initial state is shown by a wedge shape, $\succ$, and final states are indicated by double circles. For the automaton in Figure 3.2 $M = \langle K,T,\prec_T,\Sigma,\Delta,s,F\rangle$, where

$$K = \{q_0,q_1,q_2,q_3\}$$

$$T = \{t_0,t_1\}$$

$$\Sigma = \{a,b\}$$

$$s = q_0$$

$$F = \{q_3\}$$

and Δ is the relation tabulated below. We do not care about $\prec_T$ for now, and can leave it undefined.

q	w	t	q'
q_0	a	t_0	q_1
q_0	a	t_1	q_2
q_1	b	ω	q_3
q_2	b	ω	q_3

Clearly the language $L(M)$ accepted by M is $\{ab\}$.

From the initial configuration $\langle q_0,\varepsilon,ab,v_0\rangle$ the following sequence of move can ensue:

$$\begin{array}{lll} \langle q_0,\varepsilon,ab,v_0\rangle & \vdash_M & \langle q_1,a,b,v_1\rangle \\ & \vdash_M & \langle q_3,ab,\varepsilon,v_1\rangle \end{array}$$

where

$$v_1(x) = \begin{cases} 1 & \text{if } x = t_0 \\ -1 & \text{if } x = t_1 \end{cases}$$

Thus $\langle q_0, \varepsilon, ab, v_0 \rangle \vdash_M^* \langle q_3, ab, \varepsilon, v_1 \rangle$ and ab is accepted by M. It is also possible to reach the final state in the following way:

$$\begin{aligned} \langle q_0, \varepsilon, ab, v_0 \rangle &\vdash_M \langle q_2, a, b, v_1' \rangle \\ &\vdash_M \langle q_3, ab, \varepsilon, v_1' \rangle \end{aligned}$$

where

$$v_1'(x) = \begin{cases} -1 & \text{if } x = t_0 \\ 1 & \text{if } x = t_1 \end{cases}$$

Therefore also $\langle q_0, \varepsilon, ab, v_0 \rangle \vdash_M^* \langle q_3, ab, \varepsilon, v_1' \rangle$. □

Theorem 3.1 *The language accepted by any TNFA is regular.*

Proof outline. The proof is by reduction from TNFA to traditional NFA without changing the matched language. A TNFA can be reduced to an NFA by replacing all tags by -1 without changing the possible configurations reached with $\vdash_M$ when tag value functions are disregarded. Then $\vdash_M$ becomes equivalent to the corresponding operator defined for NFAs (see, for example, [34]), and it is clear that the accepted language is regular. □

As demonstrated by Example 3.1, for a particular string w and a machine M, there may be several different q and v which satisfy $\langle s, \varepsilon, w, v_0 \rangle \vdash_M^* \langle q, w, \varepsilon, v \rangle$. 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 and v we choose as the result.

Indeed, there are cases for which computing all possible configurations reachable from the initial configuration by consuming an input string is not even computationally feasible. The number of different possible configurations can be exponentially large.

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. For example, *lex* [32] chooses the pattern specified earliest in the pattern list whenever several patterns match the same string. We can also leave the decision to the user of the automaton and make the automaton return a set of possible pairs $\langle q, v \rangle$ where q is a final state and v is the corresponding tag value function.

When choosing between different v (tag value ambiguity), the situation is similar; we need some kind of ordering for tag values also. This is where $\prec_T$ comes in. It is used as a way to prioritize different tag value configurations over others.

Definition 3.4 We define another binary relation $\models_M$ on configurations, (*yields tag-wise unambiguously in one step*): $\langle q, p, u, v \rangle \models_M \langle q', p', u', v' \rangle$ if and only if for any configuration α for which $\langle s, \varepsilon, pu, v_0 \rangle \models_M^* \alpha$ and $\alpha \vdash_M \langle q', p', u', v'' \rangle$ it holds that either $v' = v''$ or $v' \prec_T v''$.

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 $q \in F$ and a function v such that $\langle s, \varepsilon, w, v_0 \rangle \models_M^* \langle q, w, \varepsilon, v \rangle$. □

Note that the definitions of $\vDash_M$ and $\vDash_M^*$ are mutually recursive. It is still possible to compute $\vDash_M^*$ effectively, using an iterative process, for any automaton and input string. Examining the definition a little further reveals that the initial configuration can be used as the starting point of the computation. This is because the initial configuration c_i is the only configuration in the beginning for which we know that $c_i \vDash_M^* c_i$. Proceeding in a breadth-first manner always choosing at most one path reaching any state is a fairly efficient strategy in computing $\vDash_M^*$. Algorithms 3.4 and 3.5 later in this chapter show a way to compute $\vDash_M^*$ efficiently.

Example 3.2 For the automaton of the previous example (see Figure 3.2) and string ab, the initial configuration is $\langle q_0, \varepsilon, ab, v_0 \rangle$. Due to reflexivity, $\langle q_0, \varepsilon, ab, v_0 \rangle \vDash_M^* \langle q_0, \varepsilon, ab, v_0 \rangle$. Because $\langle q_0, \varepsilon, ab, v_0 \rangle \vdash_M \langle q_1, a, b, v_1 \rangle$ and $\langle q_0, \varepsilon, ab, v_0 \rangle \vdash_M \langle q_2, a, b, v_1' \rangle$ (see the previous example), we have also

$$\langle q_0, \varepsilon, ab, v_0 \rangle \vDash_M \langle q_1, a, b, v_1 \rangle$$

and

$$\langle q_0, \varepsilon, ab, v_0 \rangle \vDash_M \langle q_2, a, b, v_1' \rangle$$

We do not need to choose the "winners" for states q_1 and q_2, since there is only one path from the initial configuration to each of these states.

Note that if $\alpha \vDash_M \beta$ then also $\alpha \vDash_M^* \beta$. The previous example shows also that $\langle q_1, a, b, v_1 \rangle \vdash_M \langle q_3, \varepsilon, ab, v_1 \rangle$ and $\langle q_2, a, b, v_0 \rangle \vdash_M \langle q_3, \varepsilon, ab, v_1' \rangle$. Now we need to use $\prec_T$ to choose one to be the tag-wise unambiguous step which reaches state q_3. If $v_1 \prec_T v_1'$, then

$$\langle q_1, a, b, v_1 \rangle \vDash_M \langle q_3, ab, \varepsilon, v_1 \rangle$$

and

$$\langle q_0, \varepsilon, ab, v_0 \rangle \vDash_M^* \langle q_3, ab, \varepsilon, v_1 \rangle$$

The other possibility is that $v_1' \prec_T v_1$, then

$$\langle q_2, a, b, v_0 \rangle \vDash_M \langle q_3, ab, \varepsilon, v_1' \rangle$$

and

$$\langle q_0, \varepsilon, ab, v_0 \rangle \vDash_M^* \langle q_3, ab, \varepsilon, v_1' \rangle$$

□

Theorem 3.2 *For a string w and a TNFA M, if $v_0 \vDash_M^* \langle q, p, u, v \rangle$ for some $q \in K$, p, u, and $v \in V$, then v is unique.*

Proof. The proof follows trivially from Definition 3.4. If $v_0 \vDash_M^* \langle q, p, u, v \rangle$, then v is the minimum tag value function, as per $\prec_T$, for which the otherwise same configuration can be reached with $\vdash_M$ from some previous configuration reached by $\vDash_M^*$. Therefore v must be unique, since $\prec_T$ is a total order. □

Theorem 3.3 *If a string w is accepted by M, it is also tag-wise unambiguously accepted by M.*

Proof outline. As can be seen from the definition of $\models_M$, a configuration c' can be reached from another configuration c if $c \vdash_M c'$. There is an additional restriction that the tag value function in c' must be the minimal one for the state reached with $\models_M^*$ for the same input string prefix. This restriction does not prevent any state from being reached with $\models_M$ if it is reached with $\vdash_M$, it only cuts down the number of possible tag value functions to exactly one. The conclusion is that if a state is reachable with $\vdash_M$, it is also reachable with $\models_M$, and the theorem follows. □

The point of $\models_M^*$ is that it can be used to efficiently compute the minimum tag value functions of final configurations reachable with an automaton for an input string. However, depending on the properties of the automaton, $\models_M^*$ does not always find the correct minimum tag value function that would be found by computing all possible final configurations with $\vdash_M^*$ and finding from these the minimum tag value function.

Let us explore in more detail what properties of the automaton and $\prec_T$ are necessary for $\vdash_M^*$ and $\models_M^*$ to give the same answer when searching for the minimum tag value function.

Definition 3.5 (consistency) Let W be the set of strings which are tag-wise unambiguously accepted by an automaton M. That is, for every string $w \in W$

$$\langle s, \varepsilon, w, v_0 \rangle \models_M^* \langle q, w, \varepsilon, v \rangle$$

for some $q \in F$ and $v \in V$. Then M is *consistent* if for every $q' \in F$ and $v' \in V$ for which

$$\langle s, \varepsilon, w, v_0 \rangle \vdash_M^* \langle q', w, \varepsilon, v' \rangle$$

we have that if $q' = q$ then $v \prec_T v'$ or $v = v'$. □

In other words, an automaton is consistent if $\models_M^*$ yields the same tag value functions in the final states as the minimum tag value functions computed with $\vdash_M^*$.

It is not immediately obvious that any usable class of consistent nontrivial TNFAs exist. But, as it turns out, there is a class of consistent TNFAs which can be used to solve the submatch addressing problem, which is quite enough for most practical applications.

Let v_a and v_b be two tag value functions such that $v_a \prec_T v_b$. Let *pos* be some integer such that $pos \geq v_a(t_x)$ and $pos \geq v_b(t_y)$ for any $t_x \in T$ and $t_y \in T$. Also let $t_k \in T$ be some tag and

$$v'_a(t) = \begin{cases} pos & \text{if } t = t_k \\ v'_a(t) & \text{otherwise.} \end{cases}$$

$$v'_b(t) = \begin{cases} pos & \text{if } t = t_k \\ v'_b(t) & \text{otherwise.} \end{cases}$$

The above is a formal description of a situation where the tag value function v_a wins another tag value function v_b. A change to the functions later, by changing the value of some tag t_k to the current position given by *pos*, would yield the modified tag value functions v'_a and v'_b. If we want to find the globally minimal tag value function, it must then hold that $v'_a \prec_T v'_b$, or $v'_a = v'_b$. For if it were that $v'_b \prec_T v'_a$, then v'_a would certainly not be the minimum value. But $\vDash_M$ would have already chosen v_a earlier, and v'_b would never even be computed.

So to summarize, in a consistent automaton M, if $\vDash_M$ chooses some tag value function v_a over another tag value function v_b, then it must be certain that no later tag encountered would yield a situation where v_b should in fact have been chosen instead of v_a.

From now on we will restrict ourselves to $\prec_T$ of the following form. Let $v_a \in V$ and $v_b \in V$ be some tag value functions. Then $v_a \prec_T v_b$ if and only if

$$\begin{aligned} \exists t_x \in T: \quad & (t_x \in \textit{minimized} \text{ and } (v_a(t_x) < v_b(t_x) \\ & \text{and } \forall t_y \in T,\ 0 \le y < x:\ v_a(t_y) = v_b(t_y))) \\ \text{or} \quad & (t_x \notin \textit{minimized} \text{ and } (v_a(t_x) > v_b(t_x) \\ & \text{and } \forall t_y \in T,\ 0 \le y < x:\ v_a(t_y) = v_b(t_y))) \end{aligned} \tag{3.1}$$

Here *minimized* is a set which contains the tags whose values we want to minimize. The values of tags which are not in *minimized* are maximized.

Another restriction is put on tags, we will allow each tag occur at exactly one transition. The TNFA definition would allow for multiple occurrences of the same tag, although it is not immediately clear whether this could be useful.

Now we are ready to analyze when $v_a \prec_T v_b$ if and only if $v'_a \prec_T v'_b$ or $v'_a = v'_b$. In equation 3.1 there is always some minimum x for which $v_a(t_x)$ and $v_b(t_x)$ differ, and for all y less than x the values $v_a(t_y)$ and $v_b(t_y)$ are the same. If we define new tag value functions v'_a and v'_b like above by changing the value of some tag t_k, there are three cases. The first two cases are trivial, the third is less so.

- If $k < x$ then $v'_a \prec_T v'_b$, because $v'_a(t_k) = v'_b(t_k)$ and $v_a(t_k) = v_b(t_k)$.
- If $k > x$ then $v'_a \prec_T v'_b$, because x is the minimum number for which $v'_a(t_x) \neq v'_b(t_x)$, and $v'_a(t_x) = v_a(t_x)$ and $v'_b(t_x) = v_b(t_x)$.
- If $k = x$ then the situation is a bit more complicated. Each tag occurs exactly once in the automaton, and t_x has already been encountered at least once, because it has different values in v_a and v_b. If $k = x$ happens, then the same tag is encountered again. Then there must be a cycle in the automaton containing t_k. But now $v'_a(t_k) = v'_b(t_k) = pos$, and it seems to be difficult to make any assumptions on the values of the rest of the tags t_r, $r > k$, which determine whether $v'_a \prec_T v'_b$.

  Figure 3.3 illustrates this situation. The arbitrary path P_2 along with the transition

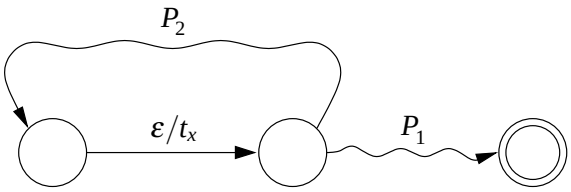


Figure 3.3: Illustration to analyze TNFA consistency. P_1 and P_2 are arbitrary paths.

labeled ε/t_x constitutes a cycle. P_1 is a path from the target state of the t_x transition to a final state. There may be several different P_1 in a TNFA.

Because it seems difficult to reason anything clever about tags t_r such that $r > k$, we will resort to an easy way out. We look for situations such that the values of t_r, $r > k$ do not actually matter. There are at least three relatively simple cases:

- All t_r, $r > k$ occur in all P_2. Then whatever values each t_r have would be overwritten to the same values by $\vdash_M^*$, and $v'_a = v'_b$.
- All t_r, $r > k$ occur in all P_1. Then it does not matter whether $v'_a \prec_T v'_b$, $v'_a = v'_b$, or $v'_b \prec_T v'_a$, because the tags which decide this will be overwritten anyway by the time a final state is reached.
- For any path from the initial state to any of the states on P_2 no tag t_r, $r > k$ must occur. In this case $v'_a = v'_b$, because all tags t_r, $r > k$ are unused.

Now we have learned some simple restrictions which guarantee the consistency of a TNFA which meets these restrictions. The next section shows how to construct a consistent TNFA for any submatch addressing problem.

3.1.1 Solving the Submatch Addressing Problem Using Tags

Automata with tagged transitions provide an elegant solution to the submatch addressing problem. It is well known that as a formalism for specifying strings, regular expressions and finite automata are equivalent in that they both describe the same sets of strings [34, 48, 51]. There are many ways to transform regular expressions into nondeterministic finite automata which recognize the language defined by the regular expression. Perhaps the most well-known method is Thompson's construction [5] and similar inductive methods [34, 51].

Regular expressions with tags are similar to normal regular expressions (see Section 2.1) with one addition; one may write tags of the form t_x straight into the regular expressions. A tag matches the empty string and has the side-effect that the current position in the input string is assigned to the tag's variable.

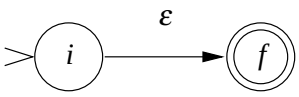
TNFAs can be constructed for regular expressions with tags by modifying Thompson's construction [5] to handle tags.

Definition 3.6 (Modified Thompson's construction) A regular expression E over an alphabet T is transformed into an nondeterministic finite automaton $M(E)$ with input alphabet T. For all E, $M(E)$ has exactly one final state. The final state is distinct from the initial state and has no transitions leaving from it. Similarly, there are no transitions to the initial state.

To avoid redundancy in the drawings, a partial automaton $M'(E)$ is usually shown instead of $M(E)$. The difference between M' and M is such that $M'(t_a(E)t_b) = M(E)$. In other words, in $M(E)$ the first and last transition are tagged with tags t_a and t_b, respectively. The tags are such that a and b are smaller than the number of any tag occurring in $M(E)$, and $a \neq b$. Tag t_a is minimized and t_b is maximized, so that $\prec_T$ can be written down in the form of Equation 3.1 on page 21.

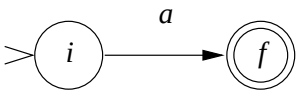
The following is a list of recursive rules to construct a consistent TNFA for any regular expression.

- $M'(\varepsilon)$ is



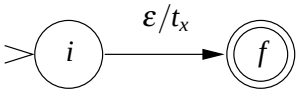
  Here i is a new initial state and f a new final state. Clearly, the language recognized by this TNFA is $\{\varepsilon\}$.

- For $a \in \Sigma$, $M'(a)$ is



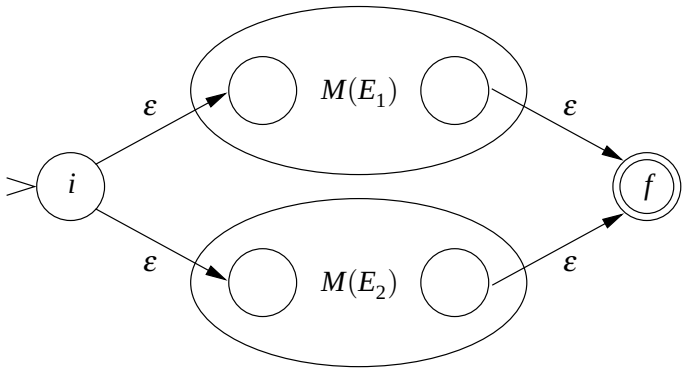
  Again i is a new initial state and f a new final state. This machine recognizes $\{a\}$.

- For $t_x \in T$, $M'(t_x)$ is



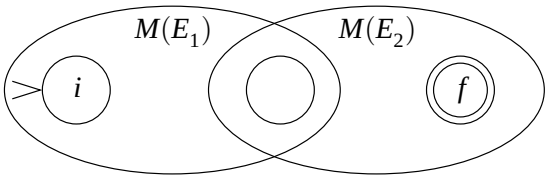
  This machine recognizes $\{\varepsilon\}$, with the side-effect that the current position in the input string is assigned to t_x.

- For the regular expression $E_1|E_2$, construct the following composite TNFA $M'(E_1|E_2)$.



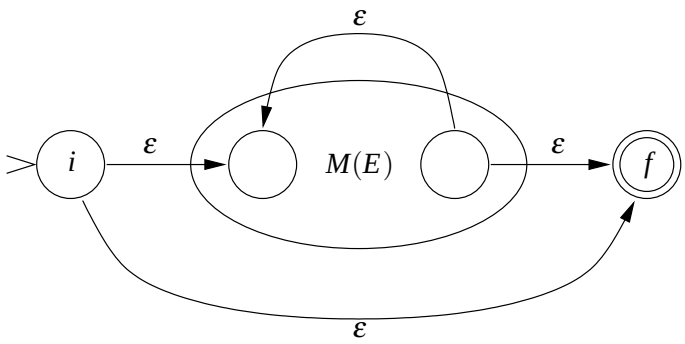
Here i is a new initial state and f a new final state. There is a transition on ε from i to the start states of $M(E_1)$ and $M(E_2)$. There is a transition on ε from the final states of $M(E_1)$ and $M(E_2)$ to the new final state f. The initial and final states of $M(E_1)$ and $M(E_2)$ are not initial or final states of $M(E_1|E_2)$. Note that any path from i to f must pass through either $M(E_1)$ or $M(E_2)$ exclusively. Thus, we see that the composite TNFA recognizes $L(E_1) \cup L(E_2)$.

- For the regular expression E_1E_2, construct the composite TNFA $M'(E_1E_2)$:



The initial state of $M(E_1)$ becomes the initial state of the composite TNFA and the final state of $M(E_2)$ becomes the final state of the composite TNFA. The final state of $M(E_1)$ is merged with the initial state of $M(E_2)$; that is, all transitions from the initial state of $M(E_2)$ become transitions from the final state of $M(E_1)$. The new merged state loses its status as a start or accepting state in the composite TNFA. A path from i to f must go first through $M(E_1)$ and then through $M(E_2)$ and no edge enters the initial state of $M(E_1)$ or leaves the final state of $M(E_2)$, there can be no path from i to f that travels from $M(E_2)$ back to $M(E_1)$. Thus, the composite TNFA recognizes $L(E_1) \circ L(E_2)$.

- For the regular expression $E*$, construct the composite TNFA $M'(E*)$:



Here i is a new initial state and f a new final state. In the composite TNFA, we can go from i to f directly, along an edge labeled ε, representing the fact that ε is in $(L(E))^*$, or we can go from i to f passing through $M(E)$ one or more times. Clearly, the composite TNFA recognizes $(L(E))^*$.

- For the parenthesized regular expression (E), use $M(E)$ itself as the TNFA.
- For a regular expression marked for submatch addressing, $\{E\}$, use $M(E)$ as the TNFA. The tags in the first and last transition of $M(E)$ will give the submatch for E after a successful match.

□

3.1.2 Efficient Simulation

Simulating a TNFA means computing $\vDash_M^*$ using some algorithm. This section discusses algorithms to compute $\vDash_M^*$, starting from a simple but inefficient version and gradually improving the algorithm to finally get a sufficiently efficient algorithm.

As already suggested in conjunction with Definition 3.4, the best way of computing $\vDash_M^*$ is to follow all possible paths in parallel. Since we are interested in only one set of tag values, it is possible to throw away paths which will result in unwanted tag values, so that the total number of paths we consider at each instant does not grow over a certain limit. To be precise, this pruning can be done at each state after each consumed input symbol so that we need to remember at most as many paths as there are states in our automaton. This idea is already incorporated into the definition of $\vDash_M$, and in this section a pseudo-code algorithm is given to efficiently calculate $\vDash_M^*$ for an automaton and input string.

All the algorithms in this section work on a nondeterministic tagged automaton $M = \langle K, T, \Sigma, \Delta, s, F \rangle$.

The following is an algorithm to calculate the *ε-closure* of a set of states, taken from [5]. It takes as an argument a set of TNFA states $Q \subseteq K$. The algorithm computes the set of all nodes reachable from Q using only ε-labeled edges of the TNFA. The stack holds states whose edges have not yet been checked for ε-labeled transitions.

Algorithm 3.1 (ε-closure)

```
1  push each state in Q onto stack
2  initialize result to Q
3  while stack is not empty do
4      pop q_1, the top element, off of stack
5      for each q_2 such that ⟨q_1, ε, t, q_2⟩ ∈ Δ for some t do
6          if q_2 is not in result then
7              add q_2 to result
8              push q_2 onto stack
9          endif
10     done
11 done
12 return result
```

This is a fairly efficient algorithm, taking $O(|\Delta|)$ worst-case time and $O(|K|)$ worst-case space when implemented reasonably. When simulating a TNFA, we also need to calculate the set of tags encountered on the path to each reachable state. The following algorithm calculates the *tagged ε-closure* of a set of TNFA states $Q \subseteq K$. The algorithm was obtained by modifying the ε-closure algorithm to operate on pairs $\langle q, k \rangle$ where $q \in Q$ is a state and $k \subseteq T$ is the set of tags seen so far.

Algorithm 3.2 (tagged ε-closure)

```
1  for each state q in Q, push ⟨q, ∅⟩ onto stack
2  initialize result to the items in stack
3  while stack is not empty do
4      pop ⟨q1, k⟩, the top element, off of stack
5      for each q2 and t such that ⟨q1, ε, t, q2⟩ ∈ Δ do
6          if ⟨q2, k ∪ {t}⟩ is not it result then
7              add ⟨q2, k ∪ {t}⟩ to result
8              push ⟨q2, k ∪ {t}⟩ onto stack
9          endif
10     done
11 done
12 return result
```

Algorithm 3.2 returns the set of all pairs $\langle q,k\rangle$ where q is a state reachable from some state p in Q using only ε-transitions and k is the set of tags encountered on the path from p to q. There may be several $\langle q,k\rangle$ with the same q but different k, because there may be several different paths with different tags to q from the states in Q.

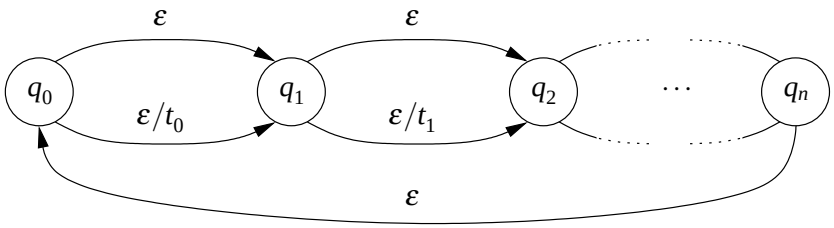


Figure 3.4: Worst case for Algorithm 3.2

The time and space complexity of Algorithm 3.2 is $O(|K|2^{|T|})$. The set of all possible subsets of T is 2^T, so the result can contain at most $|K|$ times $|2^T|$ elements. Figure 3.4 shows an example of a TNFA with which this worst case behavior occurs. From any state q in $\{q_0,q_1,\ldots,q_n\}$ any state can be reached by following a path which contains any subset of the tags in $\{t_0,t_1,\ldots,t_{n-1}\}$. Thus tagged-$\varepsilon$-closure$(q)$ for any q is of size $(n+1)2^n$.

The next algorithm uses $\prec_T$ to choose exactly one set of tags for each reachable state in an attempt to keep the space requirements reasonable. After all, we are interested only in the minimal tag value functions.

Algorithm 3.3 (tagged $\prec_T$-minimal ε-closure)

initialize *result* to $\emptyset$
for each item $\langle q_0, v_0 \rangle$ in W **do**
  for each item $\langle q, t \rangle$ in *tagged-ε-closure*$(\{q_0\})$ **do**
    let $v(x) = \begin{cases} pos & \text{if } x \in t \\ v_0(x) & \text{otherwise} \end{cases}$
    if *result*(q) is defined **then**
      if $v \prec_T$ *result*(q) **then**
        replace *result*(q) with v
      endif
    else
      set *result*(q) to v
    endif
  done
done
return *result*

In this algorithm, *result* is a function from K to V. As input the algorithm takes a set of pairs W. Each item $\langle q, v \rangle$ in W consists of a TNFA state $q \in K$ and a tag value function v associated with that state.

The algorithm calls the (ambiguous) tagged ε-closure for each item in W, and computes the new tag value functions according to what tags have been encountered. In *result* the winning tag values for the reached states as per $\prec_T$ are kept. Since calls to *tagged-ε-closure* are made, Algorithm 3.3 takes $O(|W|C_T|K|2^{|T|})$ time, where C_T is the time to perform a $\prec_T$ comparison.

The culprit of this algorithm is the way it gathers exponential worst-case size sets of items and then compares their elements to find out the minimum tag value functions. The following algorithm computes the unambiguous tagged ε-closure as defined by $\vDash^*_M$, which is equivalent to Algorithm 3.3 if the automaton is consistent (see Definition 3.5).

Algorithm 3.4 ($\models_M$ ε-closure)

for each pair $\langle q, v\rangle$ in W, add q to *queue*
initialize *result* to W
for each q in K set $count(q)$ to the input order of q
while *queue* is not empty **do**
    remove the first item, q_1, from *queue*
    for each q_2 and t such that $\langle q_1, \varepsilon, t, q_2\rangle \in \Delta$ **do**
        let $v_2(x) = \begin{cases} pos & \text{if } x = t \text{ and } t \neq \omega \\ v_1(x) & \text{otherwise} \end{cases}$
        if $result(q_2)$ is defined **and** $v_2 \prec_T result(q_2)$
          or $result(q_2)$ is undefined **then**
            set $result(q_2)$ to v_2
            decrease $count(q_2)$ by one
            if $count(q_2) = 0$ **then**
                prepend q_2 to *queue*
                set $count(q_2)$ to the input order of q_2
            else
                append q_2 to *queue*
            endif
        endif
    done
done
return *result*

This algorithm handles the case in Figure 3.4 in linear time, which is naturally a significant improvement to Algorithm 3.3. Note, however, that Algorithm 3.3 and this algorithm do not solve the same problem, and therefore do not always return the same result. Algorithm 3.4 solves a different, more restricted, problem.

To be specific, this algorithm computes the reflexive transitive closure of $\models_M$ over ε-transitions, while Algorithm 3.3 computes the closure of $\vdash_M$ over ε-transitions and then uses $\prec_T$ to choose at most one tag value function for each state. If the automaton is consistent (see Definition 3.5) then these problems are the same; in general they are not.

The complexity of Algorithm 3.4 is $O(|T||\Delta|C_T \log|T|)$. The term $\log|T|$ comes from using a functional data structure [15, 44, 45] for tag value functions, $|T|$ is present because every tag may need to be set. $|\Delta|$ and C_T are present because the whole graph may need to be traversed C_T times. Figure 3.5 shows a worst case for Algorithm 3.4.

The following algorithm simulates a consistent TNFA $M = \langle K, T, \Sigma, \Delta, s, F\rangle$ on an input string. The algorithm steps through the set of possible $\models_M^*$ configurations by consuming one input symbol at a time.

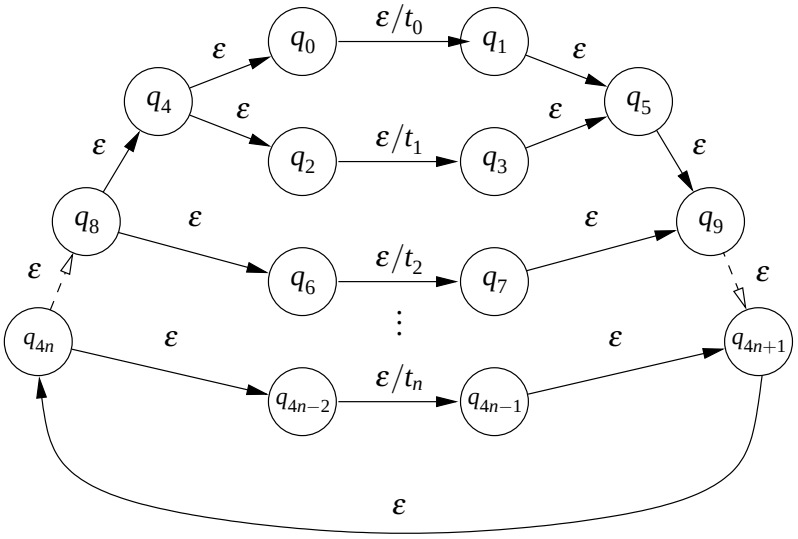


Figure 3.5: A worst case for Algorithm 3.4

Algorithm 3.5 (Simulating a TNFA)

```
1  initialize reach to ⊨_M-ε-closure({⟨s, v_0⟩})
2  initialize pos to 0.
3  while pos < |w| do
4      fetch the next input symbol c from w
5      initialize reachNext to ∅
6      for each item ⟨q, v⟩ in reach do
7          for each transition ⟨q, c, ω, p⟩ in Δ do
8              add ⟨p, v⟩ to reachNext
9          done
10     done
11     set reachNext to ⊨_M-ε-closure(reachNext)
12     swap reach and reachNext
13     set pos to pos + 1
14 done
15 return {⟨q, v⟩ | q ∈ F, ⟨q, v⟩ ∈ reach}
```

Given a TNFA and an input string w, this algorithm computes the set of pairs $\langle q, v\rangle$ such that $\langle s, \varepsilon, w, v_0\rangle \vDash_M^* \langle q, w, \varepsilon, v\rangle$. In other words, the algorithm returns all ways that the string w is tag-wise unambiguously accepted by the automaton (see Definition 3.4 on page 18). If w is not accepted, the algorithm returns an empty set.

For simplicity, the algorithm assumes that only ε-transitions can be tagged, and that only ε-transitions and transitions on single input symbols are allowed. Any TNFA can be easily converted to another TNFA which follows this restriction, so generality is not lost by imposing these restrictions.

The formation of *reachNext* on lines 4–10 takes $O(\Delta)$ time. Each call to *unambiguous-tagged-ε-closure* on line 11 takes $O(|T||\Delta|C_T \log |T|)$ time, as discussed above. For each input symbol, both of the above are done exactly once, so the time complexity of the whole algorithm is $O(T \log TMC_T N)$, where N is the length of the input string. Particularly, if $C_T = O(T)$, then the algorithm takes $O(NMT^2 \log T)$ time in the worst case.

3.2 Deterministic Automata with Tagged Transitions

There are many ways to simulate the operations of a TNFA deterministically, and Algorithm 3.5 in the previous section is one. As is the case with traditional nondeterministic and deterministic automata, computations that can be performed by an TNFA can be precomputed to form a deterministic automaton. Naturally, all possible tag values cannot be enumerated finitely, but fortunately this is not necessary.

As with traditional finite automata, the usual time-space trade-offs 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 be implemented to process characters faster than the algorithm in the previous section. A deterministic automaton may need much more space to store than a corresponding nondeterministic automaton, and time and space can be wasted in 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 number of previously calculated transitions in a cache.

3.2.1 Converting Nondeterministic Tagged Automata to Deterministic Tagged Automata

To account for the fact that a TNFA can be in many different states after reading some input symbols, a state in the deterministic counterpart, TDFA, is a set of items. Each item in the set describes one possible configuration the TNFA can be in. A situation is the combination of the current state and tag values, and can be represented by a pair $\langle s,t \rangle$, where s is a TNFA state and t is a value which describes the current value of all tags.

Actually, t does not need to be an explicit description of the values, it can be just a reference to a location (a pointer, if you will) which contains the actual description. If we used explicit tag value descriptions as values of t, the number of different sets of situations would be infinite. By using references instead, we gain two things. First, all possible TDFA states can be finitely enumerated if we restrict ourselves to a finite set of locations. Second, by swapping the contents of different memory locations we can change a TDFA state to appear different without changing its meaning. This makes the

TDFA matcher easier to implement.

Definition 3.7 To represent the idea of locations and references formally, we define an *address* to be a symbol a_i, where $i \in \mathbb{N}_k$ for some k. The set of all addresses is denoted by A. We also define a function m from A to V representing *memory*. Here V denotes the set of tag value functions as in Definition 3.2 on page 16. □

For example, to get the tag value function stored in m at address a_n, we simply look up $m(a_n)$.

Definition 3.8 To describe operations on m and the tag value functions stored there, we define C to be the set of possible *instructions*. C consists of two parts, C_s and C_c, so that $C = C_s \cup C_c$. C_s is the set of all strings of the form $\texttt{set}(n,t)$ where $n \in N_k$ and $t \in T - \{\omega\}$. C_c is the set of all strings of the form $\texttt{copy}(a,b)$ where a and b are in N_k. □

The meaning of $\texttt{set}(n,t)$ is that the tag value function $m(a_n)$ is changed so that t is mapped to *pos*. It may be that t already maps to *pos* in which case nothing changes when $\texttt{set}(n,t)$ is performed.

The meaning of $\texttt{copy}(x,y)$ is that the value at address a_x is copied to address a_y. The copy does not interfere with the original, so that `set`-operations on $m(a_x)$ do not change $m(a_y)$ or vice versa.

Instructions can be concatenated together to form sequences of instructions. These sequences are bounded with brackets, and the instructions are separated by commas. For example, $[\texttt{copy}(0,1),\ \texttt{set}(1,0)]$ first copies the tag value function $m(a_0)$ to $m(a_1)$, and then changes the copy so that $m(a_1)(0) = pos$. The set of all possible instruction sequences is denoted by $\mathscr{C}$.

Definition 3.9 A *deterministic finite automaton with tagged transitions*, or *TDFA*, is a 7-tuple $M = \langle K, \Sigma, \delta, s, m_0, F, V \rangle$, 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 $K \times \mathscr{C}$.

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

m_0 is a function from addresses to V specifying the *initial tag values*, and

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

V is the *final tag value selector*, a function from F to A.

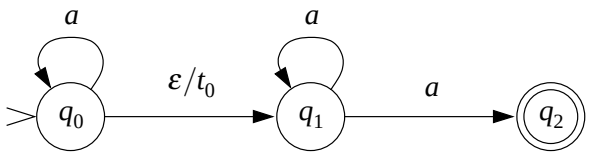


Figure 3.6: An example TNFA

Example 3.3 The algorithm is outlined by means of an example. The example TNFA is shown in figure 3.6. The TNFA corresponds to the regular expression $\{a^*\}a^*a$ so that $\langle 0, t_0 \rangle$ gives the submatch.

Now we begin to generate the TDFA, and the first step is to find the initial state. The initial state of the TNFA is q_0, and there is a tagged ε-transition from q_0 to q_1. Following the definition of $\vDash_M^*$, the TNFA can stay in state q_0 ($\vDash_M^*$ is reflexive) or use the transition labeled ε/t_0 and enter state q_1. Formally, $\langle q_0, \varepsilon, w, v_0 \rangle \vDash_M \langle q_1, \varepsilon, w, v_1 \rangle$ for any w, where $v_0 = \{\langle t_0, -1 \rangle\}$ and $v_1 = \{\langle t_0, 0 \rangle\}$.

From these considerations we form the initial state of the TDFA:

$$Q_0 = \{\langle q_0, a_0 \rangle,\ \langle q_1, a_1 \rangle\}$$

and the initial tag value functions

$$m_0 = \{\langle a_0, \{\langle t_0, -1 \rangle\} \rangle,\ \langle a_1, \{\langle t_0, 0 \rangle\} \rangle\}$$

TDFA states are represented as sets of pairs $\langle q_i, a_n \rangle$, where q_i is a TNFA state and a_n is an address such that $m(a_n)$ is a tag value function specifying the current tag value function for state q_i. This particular state can be interpreted to mean that a TNFA can be either in state q_0 with $m(a_0)$ as the tag value function, or in state q_1 with $m(a_1)$ as the tag value function.

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

- Move from $\langle q_0, a_0 \rangle$ back to q_0. We take a copy of $m(a_0)$ to some location, say x.
- Move from $\langle q_0, a_0 \rangle$ back to q_0 and then move to q_1 using t_0. We again take a copy of $m(a_0)$ to some location y. Since a t_0 was encountered, we also need to modify the copy so that $m(a_y)(t_0) = pos$.
- Move from $\langle q_1, a_1 \rangle$ back to q_1, and take a copy of $m(a_1)$ to z.
- Move from $\langle q_1, a_1 \rangle$ to q_2, and take a copy of $m(a_1)$ to w.

From this we get the second state of the TDFA: $\{\langle q_0, a_x \rangle, \langle q_1, a_y \rangle, \langle q_1, a_z \rangle, \langle q_2, a_w \rangle\}$. Note that a pair with q_1 as the left item occurs twice in this set. This means that there are two different ways we could reach q_1, and we must choose one. Now we will make an assumption; we assume that $\{\langle t_0, a \rangle\} \prec_T \{\langle t_0, b \rangle\}$ always, if $a > b$. While this is

not true in general, we assume that it is true for the $\prec_T$ that we are using. In this case $m(a_y)(t_0) > m(a_z)(t_0)$ always because $m(a_y)(t_0) = pos$ and pos is the largest tag value so far. Therefore always $m(a_y) \prec_T m(a_z)$, and the unambiguous state is: $Q_1 = \{\langle q_0, a_x\rangle, \langle q_1, a_y\rangle, \langle q_2, a_w\rangle\}$.

We have not yet assigned concrete values for x, y and w. Now that we have the whole unambiguous state in sight, we can freely choose any suitable locations for the tag value functions. In this case, we can let $x = 0$, $y = 1$, and $w = 2$, and the final unambiguous state is:

$$Q_1 = \{\langle q_0, a_0\rangle, \langle q_1, a_1\rangle, \langle q_2, a_2\rangle\}$$

We must add the instructions to create the proper tag value functions to $m(a_0)$, $m(a_1)$, and $m(a_2)$ during the transition from Q_0 to Q_1. So, we add to our transition function the entry $\delta(Q_0, a) = \langle Q_1, [\texttt{copy}(1,2), \texttt{copy}(0,1), \texttt{set}(1,0)]\rangle$.

Finally, we notice that Q_1 contains q_2, which is a final state. Thus, Q_1 is also final, and we add Q_1 to F. If the input string ends with the TDFA in state Q_1, then corresponding TNFA would have to be in state q_2 in order to produce a match. The final tag values will then be in the tag value function associated with q_2, that is, at a_2. To reflect this, we add the entry $V(Q_1) = a_2$ to the final tag value selector function V.

When the symbol a is read while in state Q_1, the TNFA can choose any of the following four actions:

- Move from $\langle q_0, a_0\rangle$ back to q_0. We take a copy of $m(a_0)$ to location x.
- Move from $\langle q_0, a_0\rangle$ back to q_0 and then move to q_1 using t_0. We take a copy of $m(a_0)$ to location y and modify it so that $m(a_y)(t_0) = pos$.
- Move from $\langle q_1, a_1\rangle$ back to q_1, and take a copy of $m(a_1)$ to location z.
- Move from $\langle q_1, a_1\rangle$ to q_2, and take a copy of $m(a_1)$ to location w.

In the same way as before, we get the ambiguous state $\{\langle q_0, a_x\rangle, \langle q_1, a_y\rangle, \langle q_1, a_z\rangle, \langle q_2, a_w\rangle\}$. Like before, $m(a_y) \prec_T m(a_z)$ always, and the unambiguous state is $\{\langle q_0, a_x\rangle, \langle q_1, a_y\rangle, \langle q_2, a_w\rangle\}$. But this is just the same as Q_1 if we let $x = 0$, $y = 1$ and $w = 2$. Thus we have a loop in our TDFA from Q_1 to Q_1 on reading a. The corresponding transition function entry is $\delta(Q_1, a) = \langle Q_1, [\texttt{copy}(1,2), \texttt{copy}(0,1), \texttt{set}(1,0)]\rangle$.

Now the construction of the TDFA is complete. The TDFA is $\langle K, \Sigma, \delta, s, m_0, F, V\rangle$, where

$$K = \{Q_0, Q_1\}$$

$$\Sigma = \{a\}$$

$$s = Q_0$$

$$m_0 = \{\langle a_0, \{\langle t_0, -1\rangle\}\rangle, \langle a_1, \{\langle t_0, 0\rangle\}\rangle\}$$

$$F = \{Q_1\}$$
$$V = \{\langle Q_1, a_2 \rangle\}$$

and δ is the function tabulated below.

q	w	q'	c
Q_0	a	Q_1	$[\texttt{copy}(1,2), \texttt{copy}(0,1), \texttt{set}(1,0)]$
Q_1	a	Q_1	$[\texttt{copy}(1,2), \texttt{copy}(0,1), \texttt{set}(1,0)]$

□

During the HIBASE project I implemented a TNFA to TDFA compiler prototype. The compiler didn't use the lazy transition evaluation approach, but always created the full TDFA before processing input. The compiler source code, written in a prototype functional programming language Shines [45], should be available from the WWW sometime in the future at `http://hibase.cs.hut.fi/`. Pseudo-code for the conversion algorithm can be found in [31].

Because Shines is not optimized for computationally intensive tasks, but rather for database applications, the performance of the TDFA implementation is modest. However, it did pass all tests for correctness, and shows that the algorithm outlined above is feasible. The inner loop of the TDFA simulator is quite simple suggesting that an implementation using a lower-level language, such as C [27], would probably be efficient.

3.3 Related Problems

The tagged transition model can be extended to a more generic model where transitions are augmented with computable functions which manipulate some arbitrary data. This makes it possible to create for example an automaton which counts the number of times a certain transition is used. Using functional data structures [15, 44, 45] this more generic model can be simulated efficiently. The following two sections show two good examples of the ways the tagged transition model can be extended to solve related problems.

3.3.1 Full Parsing

The submatch addressing algorithm can be extended to store full parse data and still retain the same time complexity. Space complexity will rise to $O(|w|)$, since an explicit representation of a full parse tree cannot be stored in less space in the worst case.

To get full parse data, we must not discard old tag values when a tag is encountered repeatedly, but store all the positions in the input string where tags were encountered. This can be easily achieved by changing the new tag value function in the definition of $\vdash_M$ (on page 16) to the following:

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

This new definition will accumulate all positions in the input string where tags were seen into a list (lisp programmers will find this representation of lists as nested pairs familiar) where -1 marks the end of the list. The definition of $\prec_T$ will of course need to be changed to compare the first values of the lists.

After this simple change a concrete parse tree can be built from the lists of tag values easily in $O(|w|)$ time.

3.3.2 Approximate Regular Expression Matching

The submatch addressing algorithm can be easily extended to an approximate regular expression matching algorithm. Approximate pattern matching allows matches to be approximate, that is, allows the matches to be close to the searched pattern under some measure of closeness. One commonly used measure is *edit-distance*, also known as the *Levenshtein distance* [33], where characters can be inserted, deleted, or substituted in the searched text in order to get an exact match. Each insertion, deletion, or substitution adds the distance, or cost, of the match.

There has been some previous work on approximate regular expression matching. In [38] Mužátko presents nondeterministic automata for approximate regular expression matching, but concludes that "simulation of a nondeterministic automaton is of a high time complexity" without doing any concrete complexity analysis.

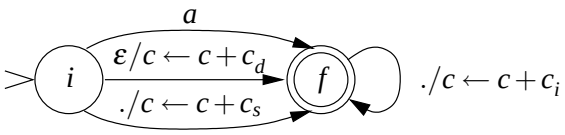
In [39] Myers and Miller give an algorithm to solve the problem in $O(MP)$ time, given a string of length M and a regular expression of length P. This is asymptotically no worse than for the simpler problem of approximate matching of simple keywords. The paper also gives an $O(MP(M+P)+N^2 \log N)$ time algorithm for arbitrary increasing gap penalties. In [29] Knight and Myers describe an $O(MP(\log M + \log^2 P))$ algorithm for approximate regular expression matching with concave gap penalties [40].

Definition 3.10 (Approximate RE match) A string w matches the regular expression E approximately with cost c if some $w' \in L(E)$ can be transformed to w with c insertions, deletions, or substitutions. □

Any string matches any regular expression with some cost, so a useful algorithm is one that can be used to tell whether there is a match with a cost lower than some threshold value, or to find the minimum cost. Some algorithms let the relative costs of insertions, deletions and substitutions to be changed arbitrarily. These costs are denoted by c_i, c_d, and c_s, respectively.

The approximate matching algorithm is constructed by changing the modified Thompson's construction in Section 3.1.1 as follows:

For $a \in \Sigma$, $M(a)$ is



Here i is a new initial state and f a new final state. This machine recognizes:

1. $\{a\} \circ \overbrace{\Sigma \circ \ldots \circ \Sigma}^{n}$ with the side-effect that c is increased by nc_i.
2. $\overbrace{\Sigma \circ \ldots \circ \Sigma}^{n}$ with the side-effect that c is increased by $c_d + nc_i$.
3. $\Sigma \circ \overbrace{\Sigma \circ \ldots \circ \Sigma}^{n}$ with the side-effect that c is increased by $c_s + nc_i$.

Now, if we define T, the set of tags, to contain only c and use plain integer comparison as $\prec_T$, the TNFA simulation algorithm becomes an algorithm which finds the minimum cost for which the input string matches the regular expressions. Since $|T| = O(1)$, the algorithm takes $O(MN)$ time to match a string of length N against a regular expression of size M.

Chapter 4

An Implementation

This chapter describes my implementation of a regular expression matcher which applies the algorithms studied in this thesis. The aim was to create a general purpose regular expression matching library; the library should be robust and sufficiently good for a wide variety of uses. The TNFA matcher implementation, including the C language source code, is available as free software. It can be downloaded from the WWW at `http://www.iki.fi/vl/libtre/`. The proof-of-concept TDFA implementation discussed in Section 3.2.1 should be available from the WWW at `http://hibase.cs.hut.fi` sometime at the future.

A typical use for a general purpose matcher is searching for all non-overlapping occurrences of relatively simple patterns from a long text. For example, a search-and-replace utility in a text editor could be implemented in this way. The matcher should not scan more text that absolutely necessary to find the next match — if the matcher would scan the whole text even though the first match is returned, searching for successive occurrences of the pattern will then take quadratic time. The implementation may not even use `strlen()` or similar for finding out the length of the text.

Another typical use case is searching for texts which match a pattern from a large number of short texts. For example, the popular UNIX utility `grep` works this way; each line of the input data is searched for a match and the matching lines are output. Note that this use does not require any kind of submatch addressing.

A third typical use case is dividing a text into words or tokens which are described using regular expressions. Traditionally this kind of processing has been done using specialized tools, but there are situations where it makes sense to avoid using lexer generators in favor of a library.

Most regular expression matching libraries require that the patterns must be compiled into some internal representation before they can be used for matching. Some applications use a large number of regular expressions for various purposes, and compile them when the application is started. If compilation takes a very long time, then

the application takes a very long time to start. Therefore, compiling regular expressions should be as fast as possible.

The POSIX standard is a widely used and accepted API for regular expression libraries, so it seemed natural to implement a POSIX compatible matcher. This gives also the benefit that are numerous other implementations to compare against.

A TNFA based implementation would be suitable for a POSIX compatible matcher, because of the restriction that compiling regular expressions should not take long. A lazy TDFA generating algorithm might also be acceptable, but would be much more complex and use a lot more memory, so I decided to go ahead with a TNFA implementation.

There are numerous methods for converting regular expressions to finite automata [8, 9, 10, 46, 36], making an NFA matcher run faster [2, 41], reducing the space requirements for the transition tables [4, 5, 12, 17, 52], and other useful methods and tricks [18, 42, 53]. Most of these are probably applicable to TNFAs and TDFAs perhaps with slight modifications.

4.1 Sacrificing Complexity

Any NFA with ε-transitions can be converted to an NFA without ε-transitions. In the worst case, the modified NFA has $O(n^2)$ transitions if n is the number of transitions in the original NFA. This happens for example with NFA's converted from regular expressions of the form $(a|a|\ldots|a)^*$ with Thompson's construction. However, it is easier to implement a fast simulation routine for an NFA without ε-transitions.

Functional data structures [15, 44, 45] are also hard to implement very efficiently. A tree-like functional $O(\log n)$ time data structure is slower than a copying $O(n)$ time routine for small n, due to overhead from reference counting or garbage collection, memory allocation and freeing, and other constant factors rising from the more complicated implementation.

Taking the above into consideration, I decided to implement an algorithm which is based on TNFA's without ε-transitions. I also decided to use a copying $O(n)$ routine for tag value sets, since the number of tags is usually very low in practice, and modern computers are capable of copying small memory blocks very efficiently.

The resulting algorithm is described in the next section. It uses $O(NM^2T)$ time, but is presumably faster than an implementation of the $O(NMT^2 \log T))$ time algorithm for most practical patterns.

4.2 Generating ε-free Tagged Automata from Regular Expressions

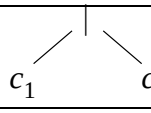
ε-free nondeterministic automata with tagged transitions can be generated from regular expressions using a modified version of the method described in [5], Section 3.9. Note that the aim here is to create an ε-free nondeterministic automaton, not a deterministic automaton. Section 3.9 of [5] targets for a deterministic automaton by first creating an ε-free nondeterministic automaton as an intermediate phase.

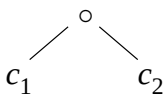
A regular expression is represented by a syntax tree with basic symbols and tags at the leaves and operators at the interior nodes. Symbol leaves in the syntax tree for a regular expression are labeled by symbols in the alphabet. To each alphabet leaf (a leaf not labeled by ε or a tag) we attach a unique integer and refer to this integer as the *position* of the leaf.

To create an ε-free TNFA for a tagged regular expression E we first augment it by forming the expression $(E)\#$. The symbol # is not a part of the original alphabet and is used to get a unique final state later.

The functions *nullable*, *firstpos*, and *lastpos* are calculated for each syntax tree node. These can be formed using the inductive rules in Table 4.1 by working up the syntax tree from the bottom; in each case the inductive rules correspond to the three operators, alternation, concatenation, and repetition. The rules for *lastpos* are the same as those

Table 4.1: Rules for computing *nullable* and *firstpos*.

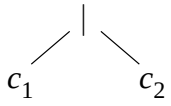
Node n	*nullable*(n)	*firstpos*(n)
ε	**true**	$\emptyset$
t_x	**true**	$\emptyset$
leaf at position i	**false**	$\{\langle i, \emptyset \rangle\}$
	*nullable*(c_1) **or** *nullable*(c_2)	*firstpos*$(c_1) \cup$ *firstpos*(c_2)

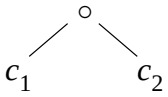
	*nullable*(c_1) **and** *nullable*(c_2)	**if** *nullable*(c_1) **then**
		firstpos$(c_1) \cup$
		addtags(*firstpos*(c_2),
		emptymatch(c_1))
		else
		firstpos(c_1)
		endif

	**true**	*firstpos*(c_1)

Table 4.2: Rules for computing *emptymatch*.

Node n	$emptymatch(n)$
ε	$\emptyset$
t_x	$\{t_x\}$
leaf	$\emptyset$

	**if** $nullable(c_1)$ **then**
	  $emptymatch(c_1)$
	else
	  $emptymatch(c_2)$
	endif

	$emptymatch(c_1) \cup emptymatch(c_2)$

	**if** $nullable(c_1)$ **then**
	  $emptymatch(c_1)$
	else
	  $\emptyset$
	endif

for *firstpos*, but with c_1 and c_2 reversed, and are not shown.

The function *emptymatch* is defined in Table 4.2.

The function *addtags* takes as arguments a set of pairs $\langle p,t\rangle$ called P and a set of tags T, where p is a position and t is a set of tags. The function returns a new set of pairs

$$\{\langle p,t'\rangle \mid \langle p,t\rangle \in P \text{ and } t' = t \cup T\}$$

The first and second rules for *nullable* state that if n is a leaf labeled ε or a tag t_x, then $nullable(n)$ is true. The third rule states that if n is a leaf labeled by an alphabet symbol, then $nullable(n)$ is false. In this case, each leaf corresponds to a single input symbol, and therefore cannot generate ε. The rest of the rules for *nullable* follow directly from the algebraic properties of the corresponding operators.

As another example, the fifth rule for *firstpos* says that if in an expression rs, r generates ε, then the first positions of s "show through" r and are also first positions of rs. Any tags which will be used in r when generating ε are added to the result. If r does not generate ε, then only the first positions of r are the first positions of rs. The reasoning behind the remaining rules of *firstpos* are similar.

When the functions *firstpos* and *lastpos* have been computed for each node in the tree, we can proceed to generate the transition relation Δ of the ε-free TNFA. Basically,

the transition relation tells us what positions can follow a position in the syntax tree and which tags are used to get there. The transition relation is a set of quadruples $\langle q_a, u, T, q_b \rangle$, where T is a set of tags. The meaning of such a quadruple is that the TNFA, when in state q_a, may consume the input symbol u from the input string, set the values of tags in T to the current position in the input string, and enter state q_b. Two rules define can be used to compute all transitions from an annotated syntax tree:

1. If n is a catenation node with left child c_1 and right child c_2, and $\langle p, t \rangle$ is an item in $\mathit{lastpos}(c_1)$, then for each item $\langle p', t' \rangle$ in $\mathit{firstpos}(c_2)$, add $\langle q_p, u, t \cup t', q_{p'} \rangle$ to Δ. Here u is the input symbol corresponding to position q_p.

2. If n is a repetition node, and $\langle p, t \rangle$ is an item in $\mathit{lastpos}(n)$, then for all items $\langle p', t' \rangle$ in $\mathit{firstpos}(n)$, add $\langle q_p, u, t \cup t', q_{p'} \rangle$ to Δ. As before, u is the input symbol corresponding to position q_p.

If *firstpos* and *lastpos* have been computed for each node, Δ can be computed by making one depth-first traversal of the syntax tree.

The initial states and initial tag values are determined by the *firstpos* of the root node. For example, if $\mathit{firstpos} = \{\langle 1, \{t_0\}\rangle, \langle 2, \{t_1\}\rangle\}$, then q_1 and q_2 are initial states. The initial tag values at q_1 are 0 for t_0 and -1 for all other tags. The initial tag values at q_2 are 0 for t_1 and -1 for all other tags.

Example 4.1 Figure 4.1 shows *firstpos* and *lastpos* for the nodes in a syntax tree for $\{(a|b)^*\}abb\#$. Figure 4.2 shows the ε-free TNFA computed from the annotated syntax tree.

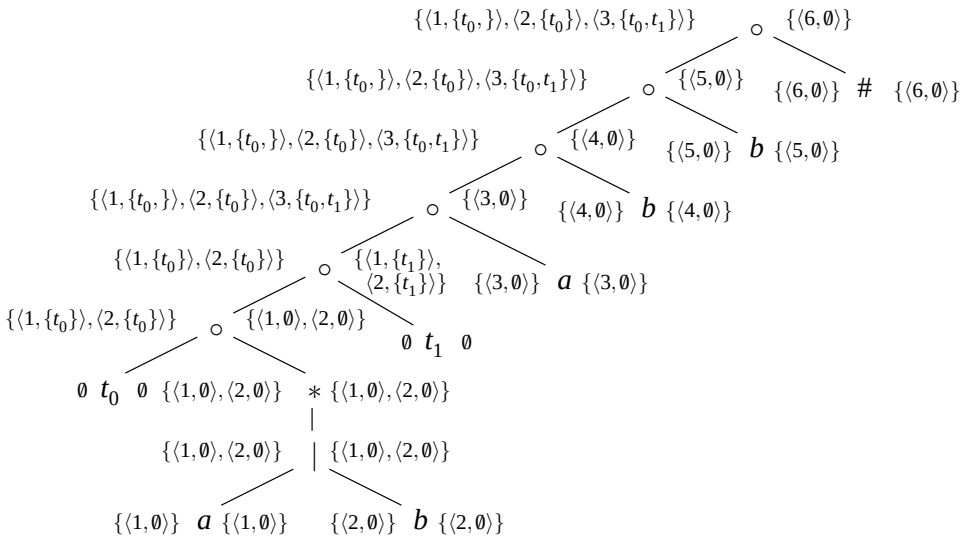


Figure 4.1: *firstpos* and *lastpos* for nodes in syntax tree for $\{(a|b)^*\}abb\#$.

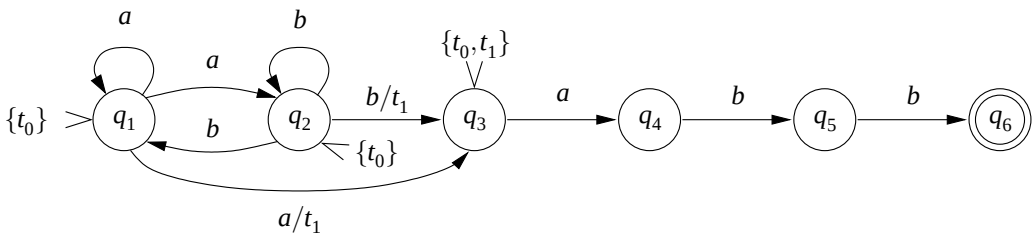


Figure 4.2: The ε-free TNFA computed from the tree in Figure 4.1

4.3 Eliminating Unnecessary Tags

It is often possible to remove some tags from a syntax tree without losing any submatch addressing information. The used submatch addressing rules (the rules which are used to decide which one of the set of possible submatches are chosen) affect tag elimination in subtle, but complicated ways. Therefore I will not present an algorithm for eliminating tags from an annotated syntax tree. Instead, a few examples are shown to give a general idea of how such an algorithm might work.

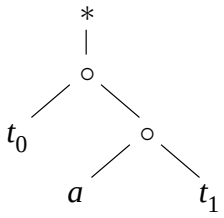


Figure 4.3: AST for $\{a\}*$

$$\begin{array}{lcl} t_0 & = & e-1 \\ t_1 & = & e \end{array}$$

Figure 4.4: Optimized AST for $\{a\}*$

Example 4.2 The regular expression $\{a\}*$ has a syntax tree shown in Figure 4.3. This can be changed to the one in Figure 4.4 without losing any submatch addressing capabilities. In the box beside Figure 4.4, e signifies the position of the next symbol after the match. If the match has zero length, then $e-1 < e$ and the submatch addressing data computed would be invalid. This situation can be checked as a special case. □

Example 4.3 The regular expression $a\{b\{c\}|\{d\}*\}*$ has a syntax tree shown in Figure 4.5. This can be changed to the one in Figure 4.6 without losing any submatch addressing capabilities. As can be seen from the figures, tags t_2 and t_3 are combined into t_1', and tags t_4 and t_5 are combined into t_2' and lifted outside the scope of the iteration operator. Tag t_0 has been left in its original position, and t_1 has been removed altogether. □

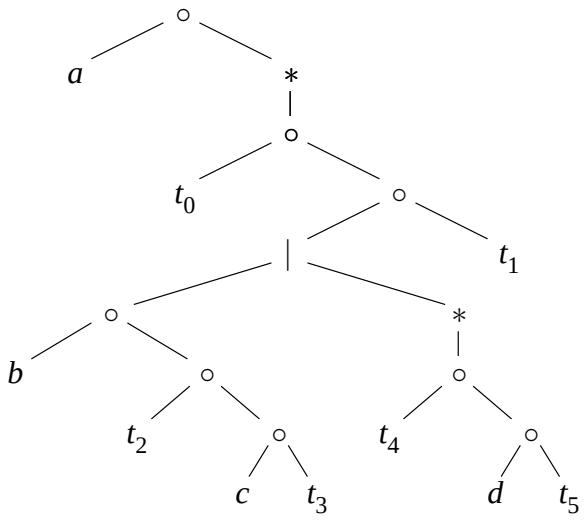
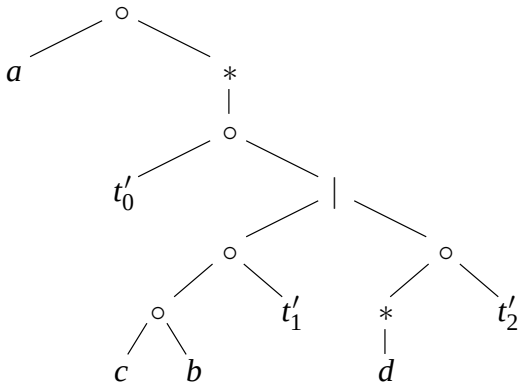


Figure 4.5: AST for $a\{b\{c\}|\{d\}^*\}^*$



$$
\begin{array}{lcl}
t_0 & = & t'_0 \\
t_1 & = & e \\
t_2 & = & t'_1 - 1 \\
t_3 & = & t'_1 \\
t_4 & = & t'_2 - 1 \\
t_5 & = & t'_2
\end{array}
$$

Figure 4.6: Optimized AST for $a\{b\{c\}|\{d\}^*\}^*$

Chapter 5

Experiments

This chapter gives some experimental results which were obtained using the implementation discussed in the previous chapter.

The performance characteristics of regular expression matchers are complex matters. Depending on the used regular expressions and the strings being searched, the performance of an implementation may vary significantly. Each implementation employs a different set of optimizations and tricks which can be applied in different situations.

In addition to performance, another important characteristic of an implementation is correctness. Surprising as it may seem, performance and correctness are often intimately related. Some implementations have bugs which speed up matching in some cases, but cause incorrect results in some other cases. Therefore it does not make sense to compare implementations with different semantics; the semantics of the matcher have profound influence on inherent performance problems and optimizations.

My implementation is POSIX compatible. There is no industry-wide agreement on a realistic set of benchmarks for POSIX regexp matchers. None have even been proposed. Therefore, it would be possible to show results which suggest that my implementation seems to be always faster than other implementations, or results which seem to indicate that my implementation is typically slower than others.

For these reasons, I have tried to be very careful about what conclusions to draw from the benchmark results. The results shown in this chapter should be mostly regarded only as demonstrations of some of the characteristics of my implementation and some other implementations.

5.1 Test Setup

In addition to my TNFA implementation, the same benchmarks were also done for GNU regex-0.12[1] and hackerlab version 20010609. Both libraries claim to be POSIX.2 compatible, and are generally regarded to be of good quality. Both libraries are written in the C programming language [27], and so is the TNFA matcher.

The tests consisted of timing the matching operation `regexec` for different patterns and input strings of different lengths. The time used by the regex compilation operation `regcomp` for different patterns was not tested.

The tests were performed on a PC with a Celeron 300A processor (running at 450MHz, with 128 KB L2 cache and a 100 MHz front side bus), 128 MB memory, and running Linux 2.4.4. The used C compiler was the GNU C compiler (`gcc`), version 2.95.

Standard statistics techniques were used to calculate 95% confidence intervals for the test results using the T-distribution. The deviations were negligible, so the results presented in the next section can be considered quite accurate.

[1] There are many different versions of GNU regex with the version label 0.12. I used the version available from `ftp://ftp.gnu.org/pub/gnu/prep/regex/`.

5.2 Test Results

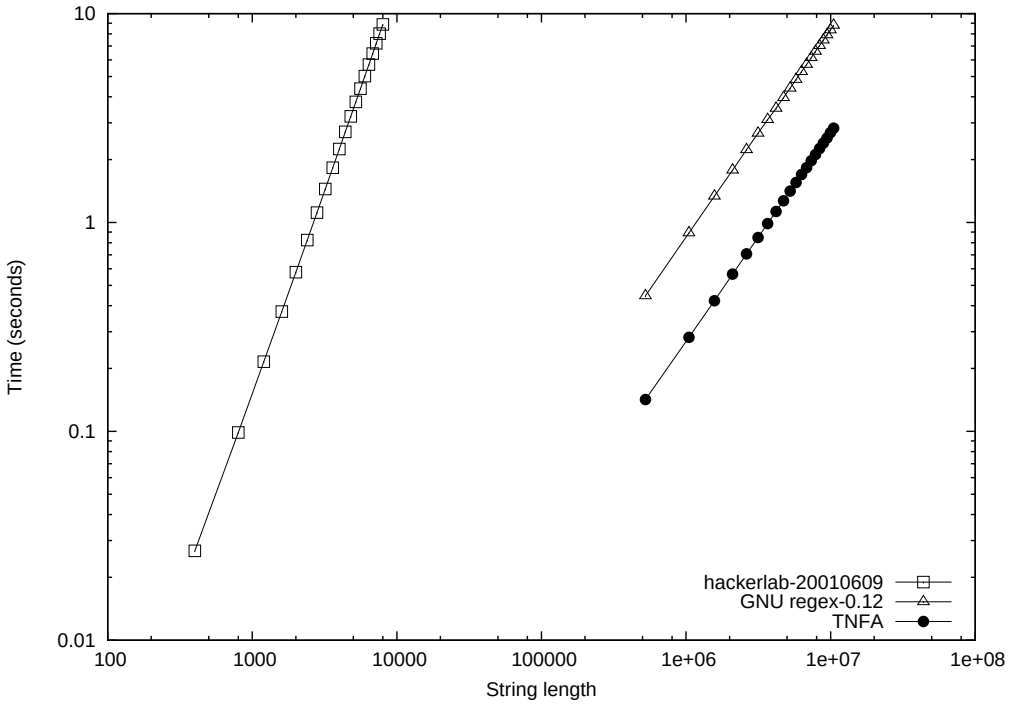


Figure 5.1: Test results for pattern `(a)*` and string `aaaa`...

Figure 5.1 shows the results for a very basic regular expression, `(a)*`, and string `aaaa`.... Note the logarithmic scale on both axes. As can be seen from the figure, the difference between hackerlab and the others is huge. Hackerlab performs very badly for some reason. It takes over ten seconds to match a one kilobyte string with hackerlab where the TNFA implementation scans something like 40 megabytes in the same time.

Table 5.1: Matching speeds for test 1

TNFA	GNU regex	hackerlab
3710000 cps	1190000 cps	901 cps

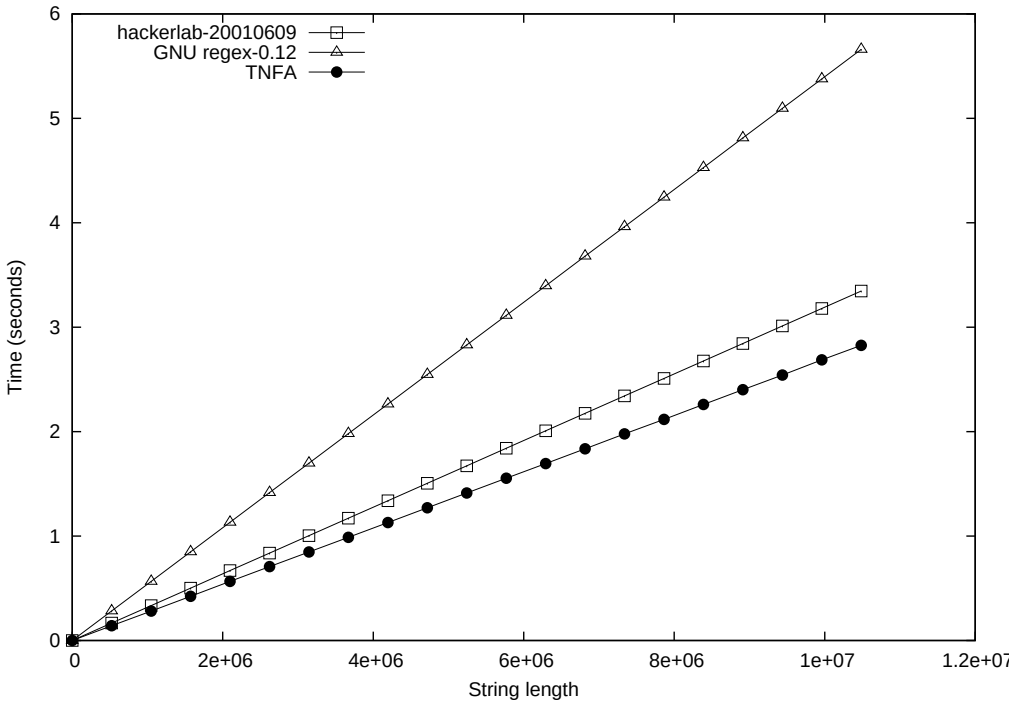


Figure 5.2: Test results for pattern `(a*)` and string `aaaa`...

Figure 5.2 shows the results for regular expression `(a*)`, slightly different from the regular expression in test 1 in terms of submatch addressing, and string `aaaa`.... The slow behavior of hackerlab does not apply to this case, and it fares much better this time. GNU regex is now the slowest implementation taking about twice as much time as the TNFA implementation and hackerlab.

Table 5.2: Matching speeds for test 2

TNFA	GNU regex	hackerlab
3710000 cps	1850000 cps	3130000 cps

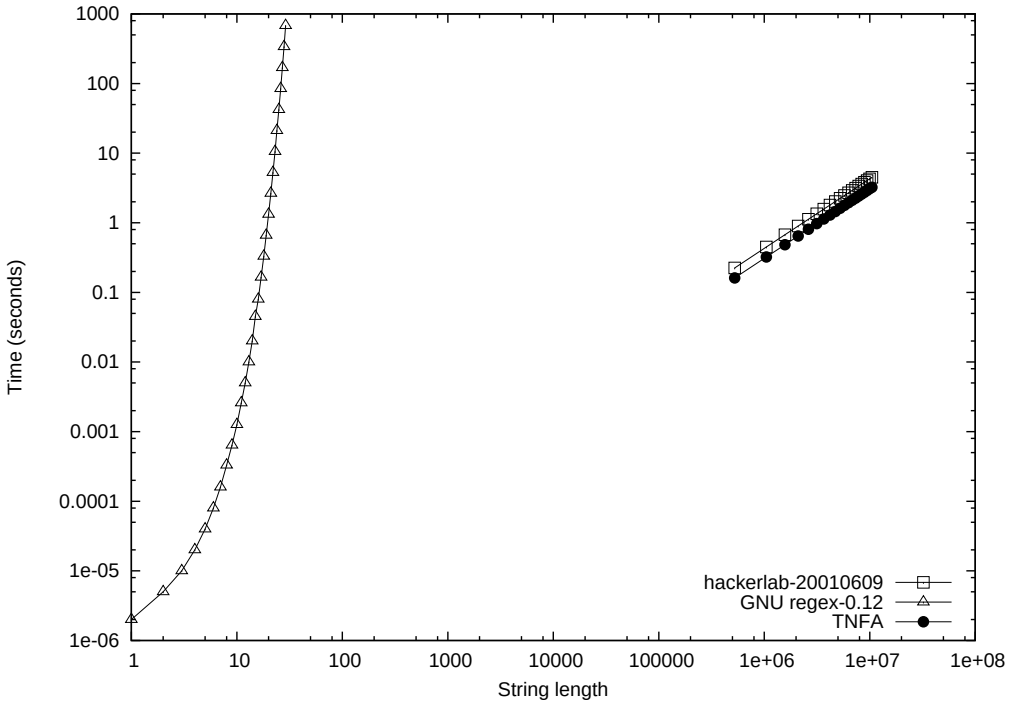


Figure 5.3: Test results for pattern `(a*)*|b*` and string `aaaa`...

Figure 5.3 shows the results for pattern `(a*)*|b*` and string `aaaa`.... This test illustrates a weakness in the backtracking algorithm used by GNU regex. Note the logarithmic scale on both axes.

The time used by GNU regex grows exponentially with the length of the input. At about 25 characters the matching time becomes too long in practice for any sensible use. Both the TNFA implementation and hackerlab handle this test well, with the TNFA implementation beating hackerlab by approximately 40%.

Table 5.3: Matching speeds for test 3

TNFA	GNU regex	hackerlab
3250000 cps	N/A	2330000 cps

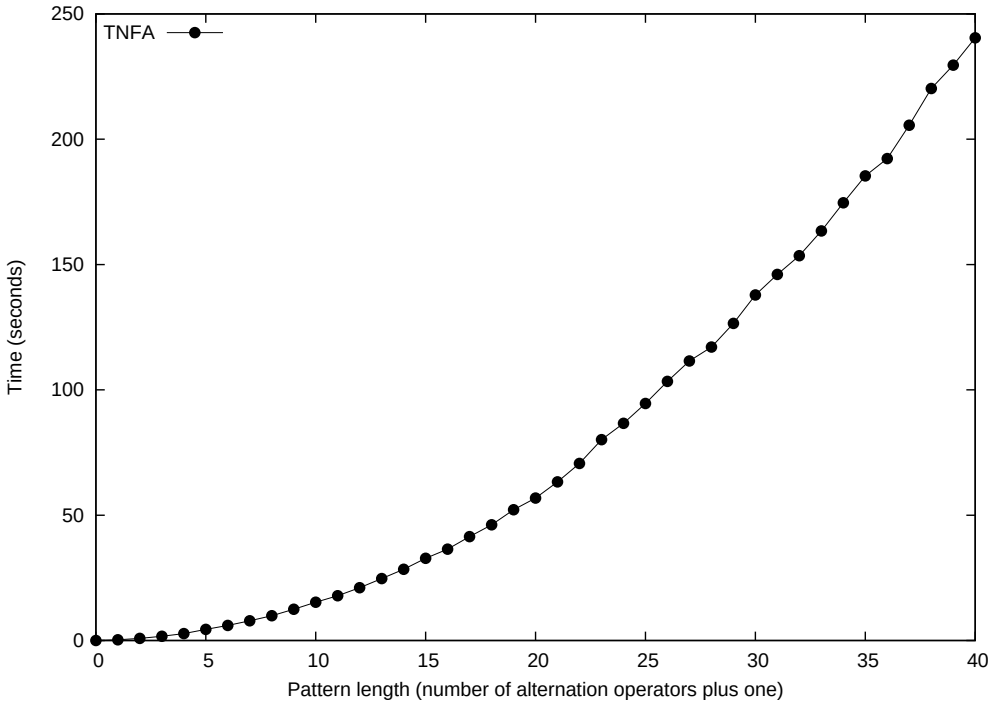


Figure 5.4: Test results for pattern `(a|a|...|a)*` and string `aaaa...` of length 2^{20}.

Figure 5.4 shows the results for pattern `(a|a|...|a)*` and string `aaaa...`. This test shows the worst case behavior of the TNFA matcher. Note that the changing parameter in this test is the pattern, not the input string length as in the other tests. The length of the text in this test was constant 10 megabytes.

In the worst case, the time used by the TNFA implementation grows quadratically with the length of the pattern (see Section 4.1). Neither GNU regex or hackerlab were able to perform this test at all. GNU regex's backtracking algorithm runs out of stack space almost immediately. Hackerlab on the other hand showed nonlinear growth of matching time when the input length (not the pattern length) was rising, and took over two minutes to match a 32 kilobyte string with the regular expression `(a|a)*`. There was no hope of getting results comparable with TNFA, so hackerlab was "disqualified".

Table 5.4: Matching speeds for test 4

TNFA	GNU regex	hackerlab
N/A	no result	no result

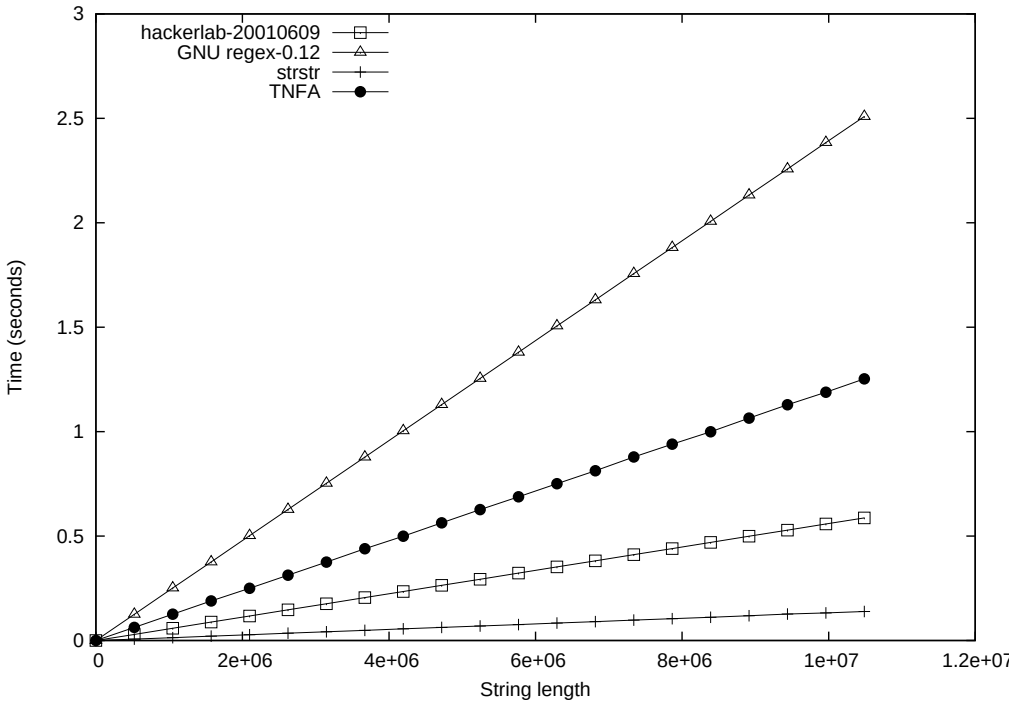


Figure 5.5: Test results for pattern `foobar` and string `aaaa...foobar`

Figure 5.5 shows the results for pattern `foobar` and string `aaaa...foobar`. This test demonstrates the speed of the implementations when given a simple substring searching task.

Hackerlab performs very well. This was anticipated, as hackerlab is based on DFA simulation and submatch addressing is not needed at all for this test. For comparison, Figure 5.5 shows also the timings for the C function `strstr`, from the GNU C library version 2.1.3, which locates a substring from a string.

Table 5.5: Matching speeds for test 5

TNFA	GNU regex	hackerlab	`strstr`
8370000 cps	4180000 cps	17900000 cps	75600000 cps

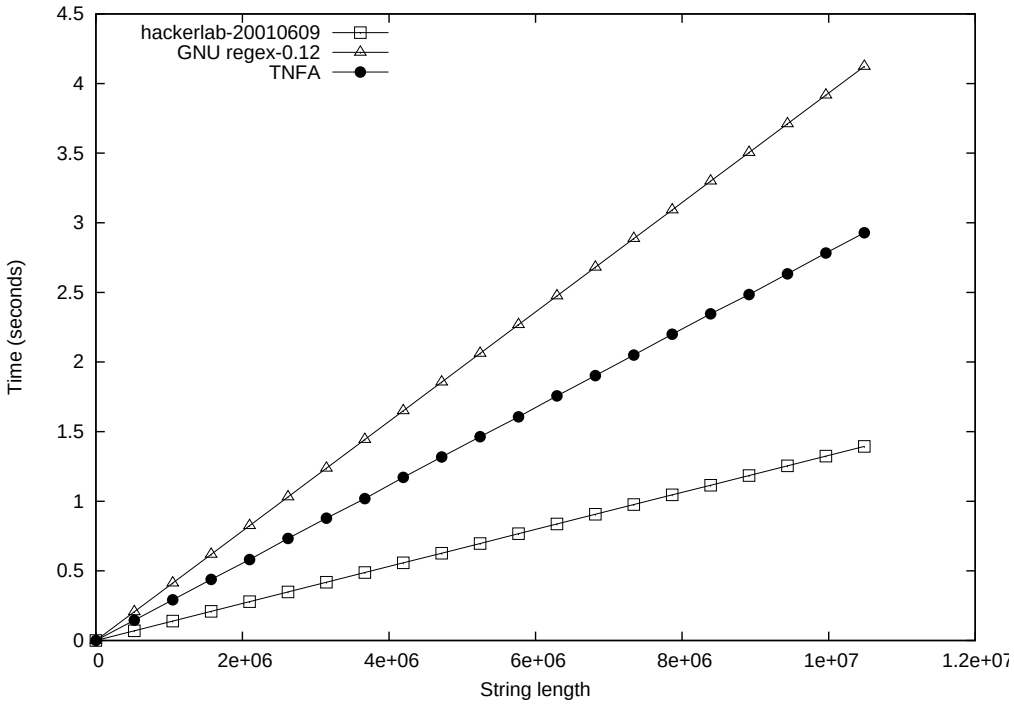


Figure 5.6: Test results for pattern `a*foobar` and string `aaaa...foobar`

Figure 5.6 shows the results for pattern `a*foobar` and string `aaaa...foobar`.

This test is a variation of the previous one. All implementations scan the input slower than in the previous test, with roughly half the speed.

Table 5.6: Matching speeds for test 6

TNFA	GNU regex	hackerlab
3580000 cps	2540000 cps	7520000 cps

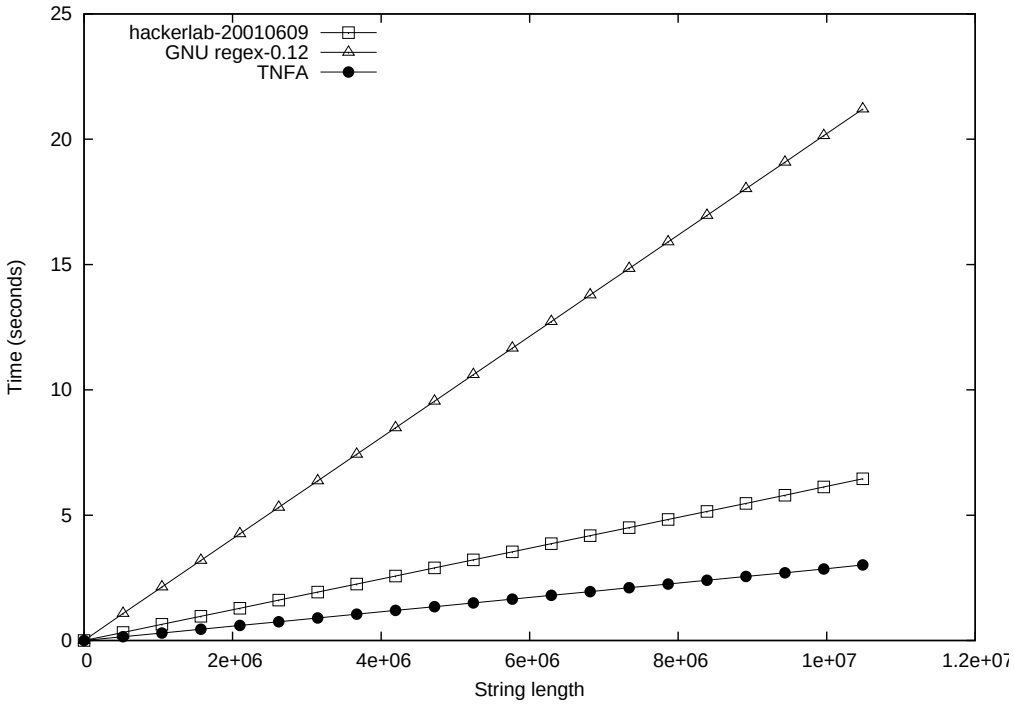


Figure 5.7: Test results for pattern `(a)*foobar` and string `aaaa...foobar`

Figure 5.7 shows the results for pattern `(a)*foobar` and string `aaaa...foobar`.

This is another variation of test number five. Now submatch addressing is brought in by adding the parentheses to the pattern. The TNFA matcher handles this case almost as fast as the previous one. Both hackerlab and GNU regex slow down to about a fifth of their speed in the previous test.

Table 5.7: Matching speeds for test 7

TNFA	GNU regex	hackerlab
3480000 cps	495000 cps	1620000 cps

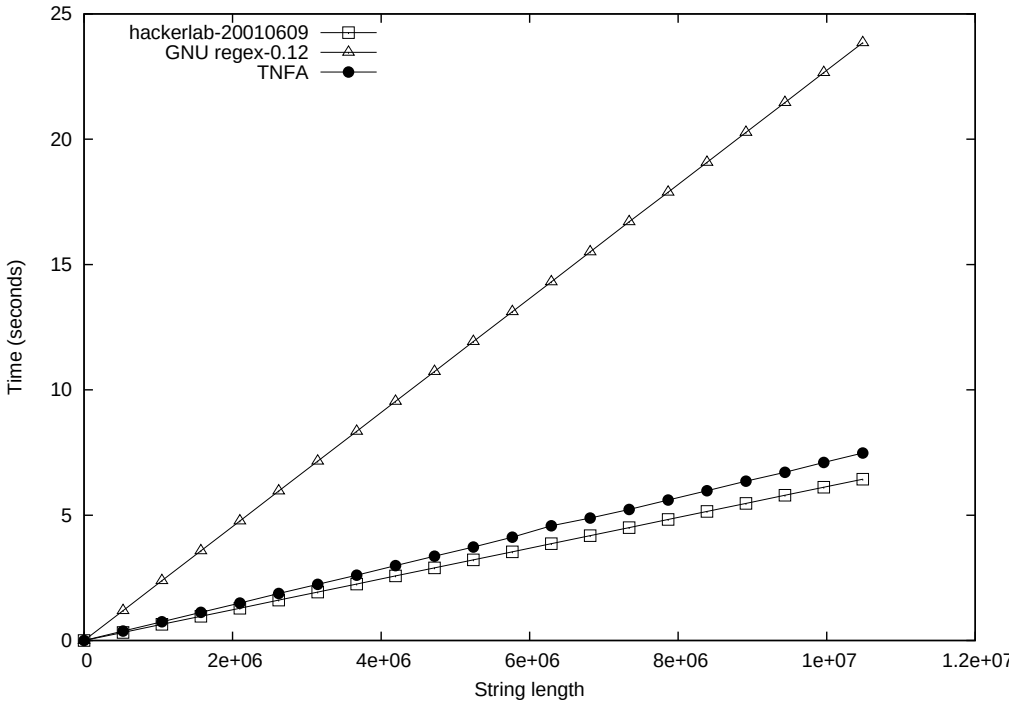


Figure 5.8: Test results for pattern `(a|b)*foobar` and string `abbaba...foobar`

Figure 5.8 shows the results for pattern `(a|b)*foobar` and string `aaaa...foobar`. This is yet another variation of test number five, with more complicated submatch addressing added by introducing the alternation operation and a string of random `a`'s and `b`'s before the suffix `foobar`.

The speed of the TNFA matcher drops down to about 40 percent of the speed in the previous test. GNU regex and hackerlab perform with approximately the same speed as in the previous test.

Table 5.8: Matching speeds for test 8

TNFA	GNU regex	hackerlab
1400000 cps	440000 cps	1630000 cps

5.3 Summary

Table 5.9: Matching speed summary

test number	TNFA	GNU regex	hackerlab
1	3710000 cps	1190000 cps	901 cps
2	3710000 cps	1850000 cps	3130000 cps
3	3250000 cps	N/A	2330000 cps
4	N/A	no result	no result
5	8370000 cps	4180000 cps	17900000 cps
6	3580000 cps	2540000 cps	7520000 cps
7	3480000 cps	495000 cps	1620000 cps
8	1400000 cps	440000 cps	1630000 cps

Table 5.9 shows a summary of the test results. As can be seen from the table, the TNFA implementation seems to perform rather well. While it is certain that these results are not conclusive, and it is not even clear what a set of conclusive tests would consist of (see the beginning of this chapter), it seems that the TNFA implementation has some interesting qualities not present in GNU regex or hackerlab.

Perhaps the most convincing treat of the TNFA matcher is its predictability; the matcher can perform reasonably well with any regular expression and input string. When the input string grows longer, worst-case matching time increases always linearly.

Chapter 6

Future Work

Researching in more detail the nature of consistent TNFAs would be interesting. It is easy to restrict the tag ordering function and the use of tags in a TNFA to ensure consistency, but it would be interesting to know whether tags could be used without restrictions if the tag ordering function is of the form in Equation 3.1.

It is also an open problem whether TNFAs can be converted to TDFAs in full generality while retaining the simplicity required for good performance. An algorithm is outlined in this thesis for doing the conversion and a proof-of-concept TDFA implementation in [31] is referred to, but problems with following the tag ordering function are sidestepped. A C-language [27] implementation of a TDFA matcher would be required to evaluate the performance gain compared to TNFAs in practice.

An implementation of the approximate regular expression matching algorithm outlined in Section 3.3.2 would be welcome. There are a few tools for approximate regular expression matching in the style of `grep`, and it would be interesting to see if the TNFA-based algorithm makes a difference. It would also be sensible to finish the POSIX matcher prototype, so it could be used as a drop-in replacement for other implementations.

Chapter 7

Conclusion

The main objective of this thesis was to find an efficient solution to the submatch addressing problem, suitable to be used in a general purpose regular expression matching library.

I evaluated several existing algorithms and found them problematic, either because of exponential worst-case matching times or linear space consumption where constant space would actually suffice. Some candidates could handle only a subset of all regular expressions, which was not acceptable.

My proposed solution, tagged nondeterministic finite automata (TNFA), is an extension to traditional finite automata where transitions are augmented with operations to keep track of submatch beginning and ending positions while matching. Algorithms for efficiently simulating TNFAs with a single pass over the input string were given.

The TNFA algorithm is capable of finding submatches, decided by tags and the tag ordering function which can be easily changed to accommodate a variety of submatch addressing rules. The algorithm finds the solution in one linear-time pass of the input string for any regular expression and input string. The space consumption during matching is constant, depending only on the regular expression but not the input string. In the author's knowledge, this is a new result.

A POSIX.2 compatible TNFA matcher was implemented as a part of the thesis work. The benchmarking results suggest that the implementation performs favorably against some popular implementations of different algorithms solving the same problem. The TNFA matcher implementation, including the C language source code, is available as free software. It can be downloaded from the WWW at `http://www.iki.fi/vl/libtre/`.

Bibliography

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter 5, pages 255–300. Elsevier Science Publishers B.V., 1990.

[2] A. V. Aho, J. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[3] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison-Wesley, Reading, MA, 1988.

[4] A. V. Aho and D. Lee. Storing a dynamic sparse table. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 55–60, Los Angeles, CA, 1986. IEEE Computer Society Press.

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

[6] American National Standards Institute (ANSI). *Coded Character Set - 7-Bit American Standard Code for Information Interchange. Standard ANSI X3.4-1986*, 1986.

[7] D. Angluin. Finding patterns common to a set of strings (extended abstract). In *Proceedings of the eleventh annual ACM Symposium on Theory of Computing*, pages 130–141, Atlanta, Georgia, 30 Apr. 1979.

[8] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Comput. Sci.*, 155(2):291–319, 11 Mar. 1996.

[9] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Comput. Sci.*, 48(1):117–126, 1986.

[10] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.

[11] C. L. A. Clarke and G. V. Cormack. On the use of regular expressions for searching text. *ACM Trans. Prog. Lang. Syst.*, 19(3):413–426, May 1997.

[12] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, Aug. 1994. `http://epubs.siam.org/sam-bin/dbq/article/19409`.

[13] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 15 Sept. 2000.

[14] B. DuCharme. *XML: The Annotated Specification*. Prentice Hall, Upper Saddle River, NJ, 1999.

[15] M. Erwig. Functional programming with graphs. In *2nd ACM SIGPLAN International Conference on Functional Programming*, pages 52–56, 1997.

[16] D. J. Farber, R. E. Griswold, and I. P. Polonsky. SNOBOL, A string manipulation language. *J. ACM*, 11(1):21–30, Jan. 1964.

[17] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, July 1984.

[18] A. Ginzburg. A procedure for checking equality of regular expressions. *J. ACM*, 14(2):355–362, 1967.

[19] J. Hartmanis. On the succinctness of different representations of languages. *SIAM J. Comput.*, 9(1):114–120, 1980.

[20] B. C. P. Haruo Hosoya, Jérôme Vouillon. Regular expression types for XML. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 11–22, Montreal, Canada, 18–21 Sept. 2000.

[21] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *Proceedings of the symposium on Principles of programming languages*, pages 67–80, London, United Kingdom, 17–19 Jan. 2001.

[22] C.-N. Hsu and M.-T. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Information Systems*, 23(8):521–538, 1998.

[23] Institute of Electrical and Electronics Engineers, Inc. (IEEE), Inst. of EE Engineers, New York. *Portable Operating System Interface (POSIX). IEEE Std 1003.2*, 1992.

[24] S. M. Kearns. TLex v.68 user's manual. Technical Report CUCS-037-90, Columbia University, 1990. `http://www.cs.columbia.edu/~library/TR-repository/reports/reports-1990/cucs-037-90.ps.gz`.

[25] S. M. Kearns. Extending regular expressions with context operators and parse extraction. *Software — Practice and Experience*, 21(8):787–804, Aug. 1991.

[26] S. M. Kearns. TLex. *Software — Practice and Experience*, 21(8):805–821, Aug. 1991.

[27] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[28] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Annals of Mathematics Studies*, volume 34 of *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[29] J. R. Knight and E. W. Myers. Approximate regular expression pattern matching with concave gap penalties. *Algorithmica*, 14:85–121, 1995.

[30] K. S. Larsen. Regular expressions with nested levels of back referencing form a hierarchy. *Inf. Process. Lett.*, 65(4):169–172, 27 Feb. 1998.

[31] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 181–187. IEEE, Sept. 2000.

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

[33] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, Feb. 1966.

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

[35] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[36] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computing*, 9(1):39–47, 1960.

[37] A. R. Meyer and M. J. Fischer. Economy of description by automata, grammars, and formal systems. In *Conference Record 1971 Twelfth Annual Symposium on Switching and Automata Theory*, pages 188–191, East Lansing, Michigan, 13–15 Oct. 1971. IEEE.

[38] P. Mužátko. Approximate regular expression matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '96*, pages 37–41, 1996. Collaborative Report DC–96–10.

[39] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51:5–37, 1989.

[40] E. W. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in LNCS series #1448, pages 91–103, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[41] G. Myers. A four Russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):430–448, Apr. 1992.

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

[43] 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.

[44] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[45] K. Oksanen. *Real-time Garbage Collection of a Functional Persistent Heap*. Licentiates Thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science, 1999.

[46] G. Ott and N. H. Feinstein. Design of sequential machines from their regular expressions. *J. ACM*, 8(4):585–600, 1961.

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

[48] D. Perrin. Finite automata. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 1, pages 1–57. Elsevier Science Publishers B.V., 1990.

[49] E. Roche. Parsing with finite state transducers. In E. Roche and Y. Schabes, editors, *Finite–State Language Processing*, chapter 8. The MIT Press.

[50] E. Roche and Y. Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, 1995.

[51] S. Sippu and E. Soisalon-Soininen. *Languages and Parsing*, volume 1 of *Parsing Theory*. Springer, 1988.

[52] R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, Nov. 1979. See also [17].

[53] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[54] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly and Associates, third edition, July 2000.

Index