UNIVERSITY OF BERN

Efficient regular expressions that produce parse trees

by

Aaron KARPER

A thesis submitted in partial fulfillment for the
degree of Master of Science

in the
Philosophisch-naturwissenschaftliche Fakultät
Institute of Computer Science and Applied Mathematics

supervised by
Prof. Oscar NIERSTRASZ and Niko SCHWARZ

November 23, 2014

UNIVERSITY OF BERN

Abstract

Philosophisch-naturwissenschaftliche Fakultät
Institute of Computer Science and Applied Mathematics

Master of Science

by Aaron KARPER

Regular expressions naturally and intuitively define parse trees that describe the text that they're parsing. We describe a technique for building up the complete parse tree resulting from matching a text against a regular expression.

In standard tagged deterministic finite-state automaton (TDFA) matching, all paths through the non-deterministic finite-state automaton (NFA) are walked simultaneously, in different co-routines, where inside each co-routine, it is fully known when which capture group was entered or left. We extend this model to keep track of not just the last opening and closing of capture groups, but all of them. We do this by storing in every co-routine a history of the all groups using the flyweight pattern. Thus, we log enough information during parsing to build up the complete parse tree after matching in a single pass, making it possible to use our algorithm with strings exceeding the machine's memory. This is achieved in worst case time $\Theta(m\,n)$, providing full parse trees with only constant slowdown compared to matching.

Contents

Chapter 1

Introduction

Regular expressions give us a concise language to describe patterns in strings and can be used for log analysis, natural language processing, and many other tasks involving structured data in strings. Their efficiency make them useful even for large data sets. Standard algorithms run in $\setminus \Updownarrow$, where n is the length of the string to be matched against and m is the length of the pattern[25].

A short-coming of standard regular expression engines is that they can extract only a limited amount of information from the string. A regular expression can easily describe that a text matches a comma separated values file, but it is unable to extract all the values. Instead it only gives a single instance of values:

`((.*?),(\d+);)+` might describe a dataset of ASCII names with their numeric label. Matching the regular expression on ``Tom Lehrer,1;Alan Turing,2;'' confirms that the list is well formed, but the match contains only ``Tom Lehrer'' for the second capture group and ``1'' for the third. That is, the parse tree found by the POSIX is seen in figure 1.1.
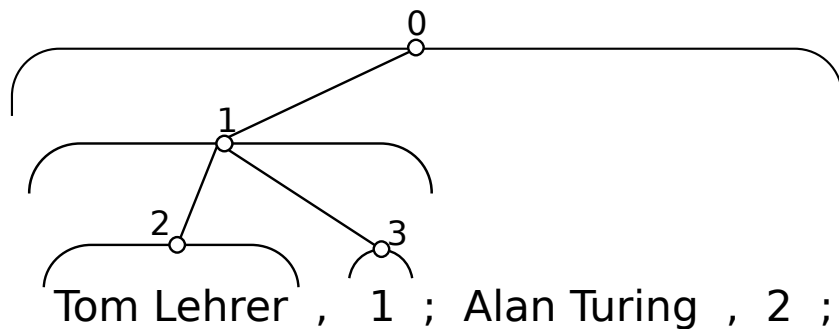

FIGURE 1.1: Parse tree produced by POSIX-compatible matching `((.*?),(\d+);)+` against input "Tom Lehrer,1;Alan Turing,2;".

With our algorithm we are able to reconstruct the full parse tree after the matching phase is done, as seen in figure 1.2.

FIGURE 1.2:   Parse tree produced by our approach matching regular expression
`((.*?),(\d+);)+` against input "Tom Lehrer,1;Alan Turing,2;"

The worst-case run time of our approach is $\Theta(n\,m)$, the same as the algorithm extracting only single matches. It is the first algorithm to achieve this bound, while extracting parse trees.

1.1   More powerful than standard regular expressions

It may at first seem as if all capture groups can always, equivalently, be extracted by splitting the input, and then applying sub-regular expressions on the splits. This is, for example, an entirely valid strategy to extract the parse tree in figure 1.2. However, this can quickly become an exercise of writing an entire parser, using no regular expression engine at all, even if the underlying grammar is entirely regular. The following grammar is hard to parse using a regular expression engine, even though it is regular.

Consider a file of semicolon-terminated records, each record consisting of a comma-separated pair of entries, and each entry can be escaped to contain semicolons, as in the regular expression `((".*?"|[a-z]*),(".*?"|[a-z]*);)+`. Here, expression `.*?` is a non-greedy match which will be discussed in more detail in section 3. This language contains, for example, the string: `"h;i",there;"h;i",Paul;`. It is easy to see that, in order to extract all four capture group matches, it is insufficient to split the input at the semicolon, as that would split the field "h;i" in half. More involved examples, where hand-written parsers become harder to make and more inefficient, are easily constructed. In contrast, our approach yields the entire parse tree, simply from the regular expression in at most $O(n\,m)$ time.

1.2   Motivation

A first step of processing large data sets is often to parse strings in order to obtain a more manageable format. As an example, consider log files. As Jacobs[15] noted, "What makes most big data big is repeated observations over time and/ or space," and thus log

files grow large frequently. At the same time, they provide important insight into the process that they are logging, so their parsing and understanding is important.

Regular expressions make for scalable and efficient lightweight parsers[16].

The parsing abilities of regular expression have provoked Meiners to declare that for intrusion detection, "fast and scalable RE matching is now a core network security issue." [21]

For example, Arasu et al. [1] demonstrate how regular expressions are used in Bing to validate data, by checking whether the names of digital cameras in their database are valid.

Parsers that can return abstract syntax trees are more useful than ones that only give a flat list of matches. Of course only regular grammars can be matched by our approach.

1.3   Regular expressions

Before we dive into the algorithm to match regular expressions, we should first look at the goal – what regular expressions are and what kind of constructs need to be supported by our algorithm.

Regular expressions have some constructs specific to them and literal character matches. They are typically described in a string describing the pattern. They can describe any regular language[26] if one only considers *match* vs *non-match* though they are typically used to extract information.

Name	Example	Repetitions	Description
literal	`a`	1	
character ranges	`[a-z]`	1	any of the characters in the range match
negated character ranges	`[^a-z]`	1	anything except for the characters in the range match
? operator	`a?`	0 or 1	
* operator	`a*`	$0 - \infty$	Prefer more matched
+ operator	`a+`	$1 - \infty$	
?? operator	`a??`	0 or 1	
*? operator	`a*?`	$0 - \infty$	Prefer less matched
+? operator	`a+?`	$1 - \infty$	
alternation operator	`a|b`	1	match one or the other, prefer left
capture groups	`(a)`	1	treat pattern as single element, extract match

TABLE 1.1: Summary of regular expression elements

Literals   The simplest option are literal characters like `a`.

Character ranges   Instead of matching only a single character, a regular expression might match any of a range of characters. This is denoted with brackets. For example `[a-z]` would match any lowercase ASCII letter, `[abc]` would only match the letters a, b, or c, and `[a-gk-t]` would match any letter between a and g, or between k and t. There is also the special character range `.` that matches any character.

Negated character ranges   Ranges of the form `[^...]` negate their content, so `[^a-g]` would match anything except a through g.

Concatenation   Two constructs put after each other have to match the string in order. For example `a[bc]` matches a followed by either b or c.

Option operator   The option operator denoted by `?` allows either zero or one repetitions of the preceding element, preferring to have one repetition. This means `a?` will match the empty string or a, but nothing else.

Star operator   The star operator `*` allows for arbitrary repetition of the preceding element, including zero repetitions, preferring as many repetitions as possible. For example `a*` matches the empty string, "a", "aa", and so on.

Plus operator   The plus operator `+` is similar, but requires at least one repetition. This leads to $a+$ and $aa*$ matching the same strings.

Non-greedy operators   The option, the star, and the plus operators also have corresponding non-greedy operators, which are `??`, `*?`, and `+?` respectively. These operators prefer to match as few repetitions as possible. In back-tracking implementation, guessing the right path is an important efficiency feature and for all capturing implementation the path taken influences the captured groups.

Alternation   Patters separated by the pipe symbol `|` can either match the left part or the right part. This binds weaker than concatenation. Therefore `abc|xyz` would match *abc* or *xyz*, but not *abxyz*.

1.3.1 Capture groups

Patterns enclosed by parens are treated as a single element, thus `(ab)*` captures *ab*, but not *aba*. More importantly for us is that after the match, the capture groups can be extracted: `a(b*)c` when matching *abbbc* can extract *bbb* and the empty string when matching *ac*. Note that the capture groups can be nested and the semantics differ for POSIX and tree regular expressions: In POSIX the regular expression `a((bc+)+)` on the string *abcbccc* gives *bcbccc* for the outer capture group and *bc* for the inner capture group – the leftmost occurrence of outer capture groups is kept and within that substring, the leftmost occurrence of the inner group is kept. In tree regular expressions, all occurrences are kept and returned in a tree structure: The outer capture group contains *bcbccc* and both inner matches *bc* and *bccc*.

The relevance of greedy and non-greedy matches becomes apparent now: The regular expression `a(.*)c?` on the string *abc* captures *bc* in the group, while `a(.*?)c?` captures only *b*. This is because the parsing is ambiguous without specifying the greediness of the match – both *b* and *bc* would be valid.

1.4 (Non-)deterministic finite-state automata

As they are heavily used throughout this paper, let us recall what non-deterministic finite-state automata (NFA) and deterministic finite-state automata (DFA) are. A DFA is a state machine that walks over a finite transition graph, one step for every input character. The choice of transition is limited by the transition's character range. A transition can only be followed if the current input character is inside transition's character range. The possible character ranges are assumed to be disjoint, so that in every step at most one transition can be followed.

NFA differ from DFA in that for some input character and some state, there may be more than one applicable transition and some transitions can be traversed without consuming a character, called ε-transitions. If there is a transition that leads to the accepting state eventually, an NFA finds it, by trying the alternatives in lock-step. Figure 3.1 shows an example of an NFA's transition graph.

An simple way to think about the process of reading input with a NFA is that of *co-routines*. A co-routine is a procedure that has the ability to suspend itself and continue later – it is similar to a thread, but they don't need to be executed in parallel. In order to resume their work later, they contain some kind of memory that is specific to them.

In the context of NFAs, a co-routine contains the current state and has access to the transition graph. When reading a string, the NFA maintains a set of co-routines and advance them in lock-step. This means that each co-routine reads the current character

and spawn new co-routines for all possible transitions. When reading a character, a co-routine must consume it, but might follow ε-transitions afterwards. In order to model that, we propose an additional state kept in the co-routine: it can either be *hungry* or *fed*. If it is hungry, it can only follow transitions that contain the character to be read, but the co-routine becomes fed. If a co-routine is fed, it can only follow ε-transitions.

Furthermore the order in which states are expanded will become relevant. For this, the model of the NFA is expanded for allowing transitions with negative priority. An edge with negative priority will only be expanded when there are no more legal transitions with regular priority. The order for high and low priorities is depth first[1].

---

[1]In an implementation this would imply that a newly seen state is put on a stack.

Chapter 2

Related work

Regular expressions are by no means new and originated with Kleene in the 1950ies[26]. This chapter first introduces some standard procedures for regular expression matching (without extracting any information), such as Backtracking in section 2.1 and various automata based approaches in sections 2.2, 2.3, and 2.4. In section 2.5 we discuss an addition to the automata based approaches called *tags* that allows for the extraction of sub-matches. We show that backtracking based approaches are not the straw man that one could believe them to be in section 2.6. Even though we first believed ourselves to be, we are actually not the first to produce parse trees in competitive time, as seen in section 2.8.

2.1 Backtracking

An intuitive and extensible algorithm for determining whether a string matches a regular expression is backtracking. This algorithm 1 is used as is or in a more optimized form in many languages, such as Java[1], Python[2], or Perl[5]. For all its advantages and ease of implementation the main problem is that it takes $\Theta(2^n\, m)$ time in the worst case:

If we have the pattern `(x*)*y` matching against the string[3] x^n, we see that it cannot match, but it takes exponential time doing so. In each step there are two options, either to collect the x in `x*`, or to step over it and try again. Unfortunately that means that the algorithm branches in each character in the string and never succeeding keeps on trying, so it takes 2^n steps to end in the *no match* case. Backtracking is fast if it guesses correctly, since there is not much overhead, but fails miserably if it guesses wrongly early on.

---

[1] `java.util.regex`

[2] The module `re`

[3] x^n means x repeated n times

Algorithm 1 Overview of backtracking

---

    function MATCH-BT(string, pattern)
        if string and pattern empty **then**
            return matches
        else if string or pattern empty **then**
            return no match
        else if first element of pattern is $a*$ **then**
            ▷ x[1:] means removing the first element of the list
            if a matches first element of string **then**
                return match-bt(string[1:], pattern)
            else
                return match-bt(string, pattern[1:])
            end if
        else if . . . **then**
            . . .
        end if
    end function

2.2   NFA based matching

A faster alternative to this approach is to pre-process the regular expression and convert it into a NFA by the rules in figure 2.1.

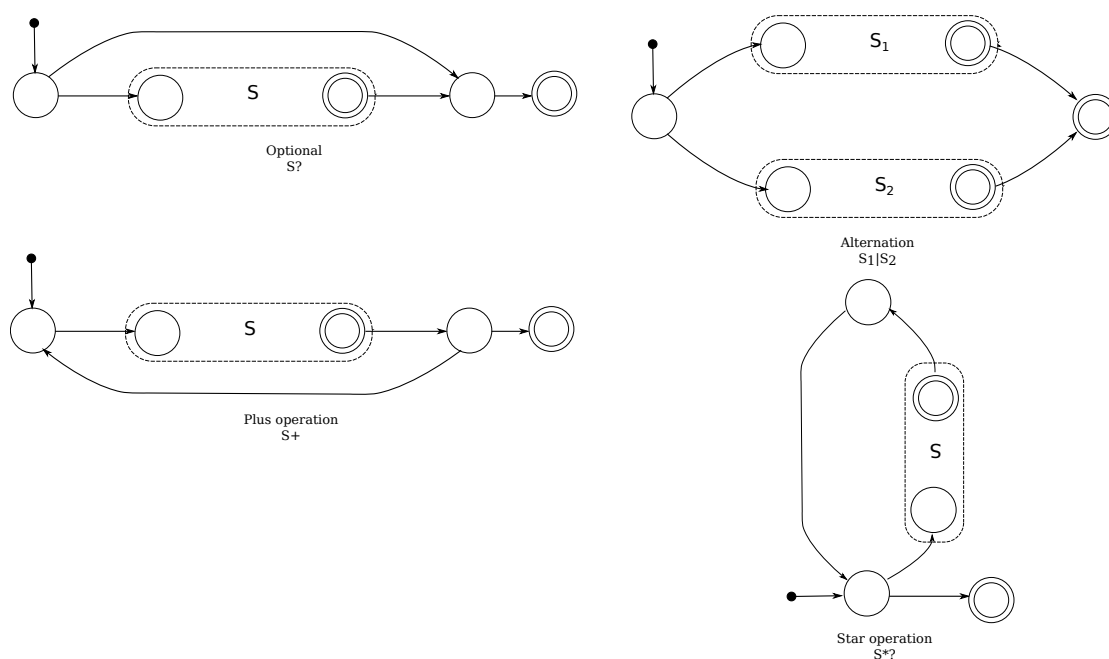FIGURE 2.1: Thompson [28] construction of the automaton: Descend into the abstract syntax tree of the regular expression and expand the constructs recursively.

The NFA thus obtained contains $O(m)$ states and to check if a given string matches the regular expression, we can now simply run the NFA on it. For each character in the input string, we follow all transitions possible from our current states and save the accessible

states as a set. In the next iteration, we consider the transitions from any of these states. This allows us to match in $O(n\,m^2)$ time.

2.3 DFA based matching

Dissatisfied with the $O(m^2)$ overhead, we can construct a DFA from the NFA before matching. This is done by the power set construction[26], which has time complexity $O(2^m)$. The idea is to replace all states by the set reachable from it with only ε-transitions. The transitions simulate a step in the original NFA, so they point to another set of states. After the compilation is done, matching the string is $O(n)$ time.

2.4 Lazy DFA compilation

The DFA based matching takes $O(n + 2^m)$, which is no better than backtracking if m is not fixed. The power set construction simulates every transition possible in the NFA, but that is actually unnecessary: Instead we can intertwining the compilation and the matching to only expand new DFA states that are reached when parsing the string. At most one new DFA state is created after each character read and if necessary the whole DFA is constructed, after which the algorithm is no different from the eager DFA. The time complexity of the match is then $O(\min(n\,m, n + 2^m))$.

This is the best known result for matching[5, 6, 7].

Our algorithm modifies this algorithm by adding instructions to transitions.

2.5 Tagged finite state automata

The algorithms so far did not extract capture groups, because they have no information about where a capture group starts or ends. In order to extract this information, we need to store it in some way, as we traverse the automaton.

Remember that NFA interpretation can be thought as competing co-routines. This model allows trivially to add more instruction as we pass a transition. The solution to keeping track of the capture groups is simply to store the information at what position the capture group started or ended in a way in the co-routine. This is the idea of the tagged finite state automaton[4], which attaches general *tags* to transitions, that modify the co-routine's memory in some way. This way, co-routines can store the complete information, where they matched which capture group in their own memory.

---

[4]First called that by Laurikari[19]

Now that side effects of transitions are introduced, it is necessary to make each step uniquely prioritized, so that we can't get to a situation, where two different paths, that crossed two different lists of tags, lead to the same node and that the choice of the winner is arbitrary. This is avoided by giving a *negative priority* to one of the transitions, whenever we have an out-degree of two[5].

The priorities intuitively mean, that for example in .a|.., we will try to follow the path of .a first, before checking ... Only if we fail on that track, we will consider the second path.

Implementing this in a backtracking implementation is trivial, but in order to keep the co-routines in lock-step, we need to order the NFA states in the DFA state, so that the co-routines travelling the left path are always scheduled before the routines on the right path.

To complicate things further, we want co-routines that travelled further to have higher priority than the ones that stayed further behind – in backtracking this would be depth-first-search.

Algorithm 2 Tagged transition execution

  function run_{tagged}(coroutines)
    Put all coroutines on the stack as negative priority
    while the stack isn't empty **do**
        Pop c from the stack
        for all ε-transitions from c to state s with tag t **do**
            if no co-routine in state s exists in coroutines **then**
                copy own memory into m
                INTERPRET(t, m)
            end if
            if transition is normal priority **then**
                add a co-routine r in state s with memory m to the stack
            else
                add a co-routine r in state s with memory m to the stack beneath all
  high priority co-routines.
            end if
        end for
    end while
  end function

This scheduling can be done by using a buffer, in which states that are seen are stored and that is flushed into the DFA state in LIFO order if there are no more high-priority transitions to follow.

To compile the TNFA to a TDFA we have to capture the modifications that we encounter between reading characters. After doing so, we need to check if we're in a DFA state that we already encountered and that we can create a new connection to. Equality of TDFA

---

[5]Note that in the Thompson construction, we have an out-degree of at most two.

states can't be the same as equality between DFA states – the equality of the contained NFA states doesn't care for the order in which they are visited and further it doesn't respect that two expansions might have different executed instructions. This has been addressed by Laurikari[19] by finding equivalent or *mappable* TDFA states. A mapping is an bijection of two states that needs to be found at compilation time. For the details, the interested reader needs to wait for the detailed description of our algorithm in chapter 3.

The idea of adding other instructions to the co-routines in the automaton that is the finite state machine (be it NFA or DFA) isn't new. The first implementation using this to the authors' knowledge is Pike[24] in his text editor SAM. He used a pure tagged NFA algorithm to find one match for each capture group quite similar to our or Laurikari's approach. This was only published in source code, to a great loss for the academic community.

This close control over greediness was implemented by Kuklewicz[18] for the Haskell implementation of Laurikari's algorithm. This too was only published in source code, to a great loss for the academic community.

2.6   Revisiting backtracking

As seen earlier, backtracking makes for easy implementations, but exponential run-time in the worst case for many patterns. Norvig[23] showed that this can be avoided by using memoization for context free grammars. This allows for $O(n^3 m)$ time parsing. While this is significantly higher than the $O(n\,m)$ of the automata based approaches, it is also more general, because more than just regular grammars can be parsed with this approach. This approach is taken by combinatoric parsers such as the Parsec library[6]. It should be noted however that while this approach has been known for some time now and promises exponential speed-up, it is by no means a standard optimization for backtracking based regular expression implementations.

First let us recapitulate what memoization means: For a function f we can create a memoized function $\hat{f}$ such that

> Initialize hash map *cache*
> **function** $\hat{f}$(*args)
>   **if** args not in *cache* **then**
> $val \leftarrow f(*args)$
> $cache[args] = val$
>   **end if**
>   **return** $cache[args]$
> **end function**

---

[6]The memoization stems from the common subexpression optimization of Haskell

This means that repeatedly calling the same function with the same inputs only costs lookup time after the function has been evaluated once.

The parser presented by Norvig inherently produces parse trees, because it tries to reduce the list of tokens into all possible expansions of the base symbol. For example `a*a*` would be converted to a grammar

$$
\begin{array}{rcc}
\text{A} & \rightarrow & a \\
\text{ASTAR} & \rightarrow & \text{A ASTAR} \\
\text{ASTAR} & \rightarrow & \varepsilon \\
\text{FULL} & \rightarrow & \text{ASTAR ASTAR}
\end{array}
$$

Now we can memoize the result for a non-terminal symbol and the remainder of the string, which avoids exponentially many trial and error parses.

The same arguments that make this efficient for context-free grammars however also make the memoization approach applicable to the backtracking algorithm for regular expressions.

To see the advantage, let's revise the example that produced exponential runtime for regular backtracking: `(x*)*y` on the string x^n. After failing for the first time with greedily consuming the n x characters. This memoizes that x^{n-1} is not parsable, so the recursion stops after only these steps, finding that the string isn't parsable as a whole.

2.7   Packrat parsers

The memoization approach can be extended further to so-called *packrat* parsing[20] based on Parsing Expression Grammars (PEG), to obtain a $O(n\,m)$, but the memoization gives a space overhead that is $O(n)$, with a big constant[11] – or to put it in another way: The original string is stored several times over. This makes them flexible and fast parsers for small input, but cannot be used for big data sets. To understand how they work, let's first look at PEG:

Parsing Expression Grammars are grammars similar to context-free grammars with the difference, that they allow no left-recursion and no ambiguity, each expression has exactly one correct parse tree and consumes at least one character. In this they are similar to regular expressions, but capture more than just regular languages. Such an expression can contain literals, recursive subexpressions, ordered choice , repetitions and non-consuming predicates:

recursive subexpressions allow for recursive repetition of a pattern, for example

$$
\text{S} \rightarrow {'('}\quad \text{S}\quad {')'}\quad /\varepsilon
$$

or

$$S \to {'\text{if}'} \quad P \quad {'\{'} \quad S \quad {'\}'}$$

ordered choice gives two options, but the left path will be checked first and if it matches already, the right path will not even be considered.

repetitions The operators `*`, `+`, and `?` with their meaning identical to their usage in regular expressions.

non-consuming predicates The operator `&` will only match and return the left-hand side, if the right-hand side would also match. The right-hand side doesn't consume any characters however. Similarly the operator `!` only matches the left-hand side, if the right-hand side does not match (without consuming any characters for the right-hand side).

The implementation of these grammars as recursive descent/backtracking parsers is quite simple. The corresponding Packrat parser is the straight forward memoization of this. This is in principle nothing new, the memoization approach to context-free grammars uses the same approach, but the restriction to PEG make the problem solved easier: without ambiguity less references need to be stored to possible parses and since parsers consume at least a character, each character in the input string has at most m possible interpretations.

The downside to this is that Packrat parsers have a large space overhead that make them infeasible for large inputs[2][7].

2.8   Automata based extraction of parse trees

Memoization is a powerful tool to achieve fast parsers, but they have a space-overhead in order of the input instead of the parse tree size. The other approach to parsing – finite state automata – offers a remedy. These approaches, one of which will be presented in this paper, use tagged finite state automata that store the parse tree in some manner, the main differences being the format of the stored tree and the type of automaton running the parse.

The rivaling memory layouts are lists of changes and an array with a cell for each group. The former makes it hard to compile the TNFA to a TDFA with aggressive reuse of states via mapping (as described in algorithm 5), but has lower space consumption. The mapping in terms of cells for each group is easy, but costs a factor m space overhead.

---

[7]

[...] the Java parser generated by Pappy requires up to 400 bytes of memory for every byte of input.

Author	Stores	Automaton	Parse time	Space overhead
Kearns [17]	Path choices	NFA	$O(n\,m)$	$O(n\,m)$
Dubé, Feeley [10]	Capture groups in linked list	NFA	$O(n\,m)$	$O(n\,m^2)$
Nielsen, Henglein [13]	Bit-coded trees of capture groups		$O(n\,m\,\log(m))$	
Grathwohl [12]				
Laurikari [19]	Capture group in array	DFA	$O(n + 2^m)$	$O(d\,m)$
This paper	Capture group in array of linked lists	lazy DFA	$O(n\,m)$	$O(n\,d\,m)$

TABLE 2.1: Comparison of automata based approaches to regular expression parsing. n is the length of the string, m is the length of the regular expression, and d is the number of subexpressions. Note that Laurikari [19] does not produce parse trees.

Another problem is that of greediness. Kearns, Dubé, and Nielsen can't guarantee the greediness of the winning parse. Grathwohl's contribution allows Dubé's algorithm to run with greedy parses. Our priorities allow for arbitrary mixes of greedy and non-greedy operators.

Finally when dealing with large n, one might be interested in passing over the string as few times as possible. Kearns, Dubé, and Nielsen do this in three passes to find the beginning and ending of capture groups, whereas Grathwohl only uses two passes. Our algorithm captures the positions of the capture groups in a single pass – thus allowing streaming of the input string, as explained in section 2.10. For access to the content, the original string is still necessary.

2.9   Discussion

While there is no shortage of books discussing the usage of regular expressions, the implementation side of regular expression has not been so lucky. Cox is spot-on when he argues that innovations have repeatedly been ignored and later reinvented [5, 6, 7].

This paper is no exception. The authors of this paper had set out to implement Laurikari's TDFA algorithm [19], only to discover that Laurikari's description of a TDFA is so far from complete that it can rightfully only be called the sketch for an algorithm. Only late in the process did we discover that the blanks had already been filled by Kuklewicz in the course of his implementation of TDFAs in Haskell [18]. Kuklewicz enshrined his added insight into Haskell library, but never published the algorithm as a whole. If the history of regular expressions is evidence of one thing, it is that source code is a terrible medium to convey algorithms.

The situation dramatically improved with Cox's simple and concise explanation of regular expression matching [5]. It seems ironic that this well-versed author published this influential work on his website.

When the practitioners acknowledge each other's work, we can't help but disagree almost universally with the characterizations they produce. Sulzmann and Lu [27] call Kuklewicz's work an "implementation" of Laurikari's algorithm, although Laurikari's algorithm is far too incomplete for that statement to be fair. Laurikari's algorithm is referred to as a POSIX-style automaton. In truth, Laurikari leaves the matching strategy entirely open. It was Kuklewicz that found out how to get POSIX-style matching out of Laurikari's TDFA.

Cox says that Laurikari's TDFA is a reinvention of Pike's algorithm, used in the text editor SAM [24]. While SAM was released as free software, no description of his regular expression matching, besides the source code, was published. Sam uses Thompson's NFA, but adds submatch tracking. Calling Laurikari's TDFA a reinvention of Pike's algorithm seems unfair in that Laurikari's allows for far more aggressive reuse of old states than Thompson, by virtue of introducing the mapping step described in algorithm 5. This should lead to Laurikari's TDFA having fewer states, and therefore better performance, than even Google's RE2[8], which uses Pike's algorithm. This is not confirmed by the benchmarks by Sulzmann and Lu [27], but they offer an explanation: in their profiling, they see that all Haskell implementations spend considerable time decoding the input strings. In other words, the measured performance is more of an artifact of the programming environment used.

Another mistake that permeates the parsing literature is to call regular expression matching linear. As Sedgewick points out correctly [25], Thompson's NFA matching is of complexity $O(mn)$, where m is the size of the input NFA, and n is the size of the input string. To call this linear means to assume that m is fixed, which is not justified. It may well be true that, at present, m tends to be small. But that is a natural consequence of the algorithms for parsing not scaling very well with m. If they did, that would allow for fast feature extracting from text[9]. Therefore, in this paper, we consider the state of the art algorithms to be quadratic, since both m and n are part of the input to a regular expression matcher. There is a lower bound proven in section 5.3 of $\Theta(n \min(m, |\Sigma|))$, but this does not prohibit an asymptotic improvement of this algorithm. It is not known if $O(nm)$ is the best we can hope for.

In comparison, our approach allows full control over greediness, even for subexpressions, thus generalizing over all previous approaches, while achieving the running time guarantees of regular expression matching. It runs in only one pass and supports character ranges, lazy compilation, and compactified DFA representation for practical performance.

---

[8]https://code.google.com/p/re2/

[9]To check if a document contains features $f_1, f_2, \ldots, f_n$, we would match the document against regular expression $(f_1)|(f_2)|\ldots|(f_n)$.

Further since only a single pass is necessary and the memory overhead is small, our algorithm can be applied to a streaming scenario, as discussed in the next section.

Compared to RE2, our algorithm doesn't provide many low-level optimizations, such as limiting the TDFA cache size or an analysis of the pattern for common simplifications such as optimizing for one-state matches[10]. However our algorithm doesn't require a separate pass for match detection and match extraction, which opens different scenarios. RE2 only considers the DFA states equivalent, if the instructions write to the same positions and are ordered the same. Our algorithm adds the mapping phase from Laurikari, which allows us to find DFA states that can be made equivalent by some additional writes.

2.10   Large scale data analysis of a stream of text

Due to its improvements in requiring only a single pass over the input and its memory allocation pattern, our algorithm is especially fitted for a scenario, where the input string is of unknown size or of size to big to fit on a single machine.

Imagine the task of finding a parse tree of a regular grammar for a text that is stored on machines $M_1, \ldots, M_n$. Now in order to parse the string, we set aside a machine P, that will do the parsing. We only need the NFA of the pattern to fit on P and can then receive each of the shards from M_1 to M_n in turn and discard it afterwards. Even if the parse tree (containing the indices of the beginning and end of the groups) is too large to fit on the machine, we can send the cells to other machines. Because the cells are not touched anymore, there will never be a need for P to retrieve old cells.

---

[10]unambiguous NFA can be interpreted as DFA and can be matched more efficiently

Chapter 3

Algorithm

We describe our algorithm in this chapter. Since it is based on previous work and especially the lazy DFA compilation algorithm, you should expect to see previously discussed topics to be reiterated.

3.1 TNFA

The algorithm we present is a extension of the non-deterministic finite state automaton (NFA) matching algorithm for regular expressions with added logging of the start and end of capture groups, as introduced by Laurikari[19]. We first present the algorithm for matching, which is $O(n\,m\,u(m))$, where $u(m)$ describes the amortized cost of logging a single opening or closing of a capture group. We then show a simple data structure that allows us to achieve $u(m) = \log m$ and continue to present a way to improve this to $u(m) = 1$ following Driscoll et al[9]. This gives us $O(n\,m)$ run time for the complete algorithm, which is the best known run time for NFA algorithms. Since this is not a purely theoretical paper, there are also practical considerations such as caching current results, just-in-time compilation, and compact memory usage.

Conceptually, our approach is the following pipeline of four stages:

1. Parse the regular expression string into an AST (section 6.5).
2. Transform the AST to an NFA (section 3.2).
3. Transform the NFA to a DFA (section 3.3).
4. Compactify the DFA (section 6.3).

In reality, things are a little more involved, since the transformation to DFA is lazy, and the compactification only happens after no lazy compilation occurred in a while. Worse, compactification can be undone if needed. Since the essence of the algorithm is step 2 and 3, we start with them and proceed see 1 and 4 as part of the implementation.

3.2   Thompson's construction

We transform the abstract syntax tree (AST) of the regular expression into an NFA, in a modified version of Thompson's NFA construction. To control greediness, or discern capture groups, our approach adds tagged ε transitions to the transition graph. An ε transition has no input range assigned, and can thus always be used. It does not consume an input character. Tagged transitions mark the beginning or end of capture groups or control the prioritization. The additions are needed for greediness control and capture groups. Let's look at both, in turn.

To see the importance of greediness control, consider again the regular expression `((.*?),(\d+);)+`. The question mark sets the `.*` part of the regular expression to *non-greedy*, which means that it matches as little as possible while still producing a valid match, if any. Without provisioning `.*` to be non-greedy, a matching against input ``Tom Lehrer,1;Alan Turing,2;'' would match as much as possible into the first capture group, including the record separator ','. Thus, the first capture group would suddenly contain only one entry, and it would contain more than just names, namely "Tom Lehrer,1;Alan Turing". This is, of course, not what we expect. Non-greediness, here, ensures that we get "Tom Lehrer", then "Alan Turing" as the matches of the first capture group.
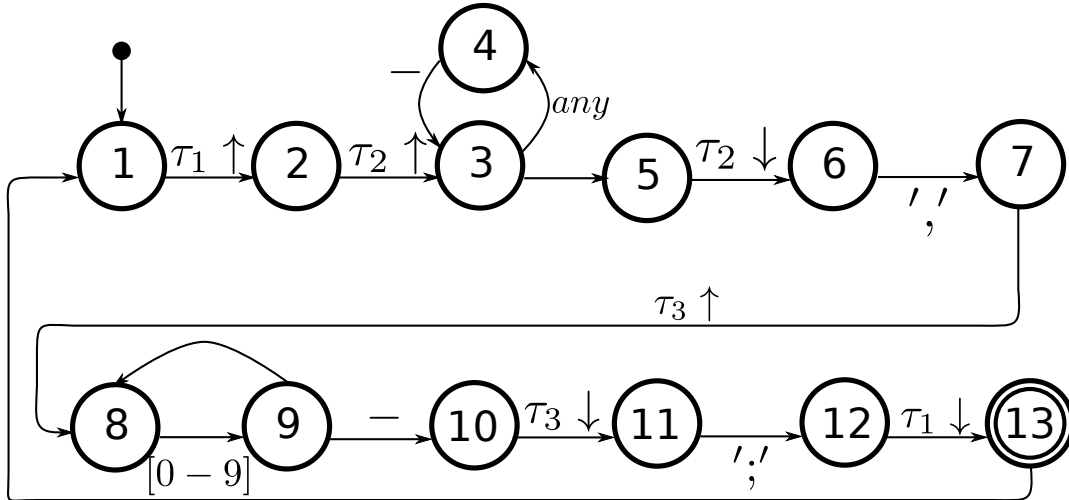

FIGURE 3.1:   Automaton for `((.*?),(\d+);)+` In the diagram, "−" stands for low priority. $\tau_n \uparrow$ is the opening tag for capture group n, likewise, $\tau_1 \downarrow$ is the closing tag for capture group n.

In the NFA, we model greedy repetition or non-greedy repetition of an expression in two steps:

1. We construct an NFA graph for the expression, without any repetition. Figure 3.1 shows how this plays out in our running example, which contains the expression `.*?`. An automaton for expression `.` is constructed. The expression `.` is modeled as just two nodes labeled 3 and 4, and a transition labeled "any" between them.
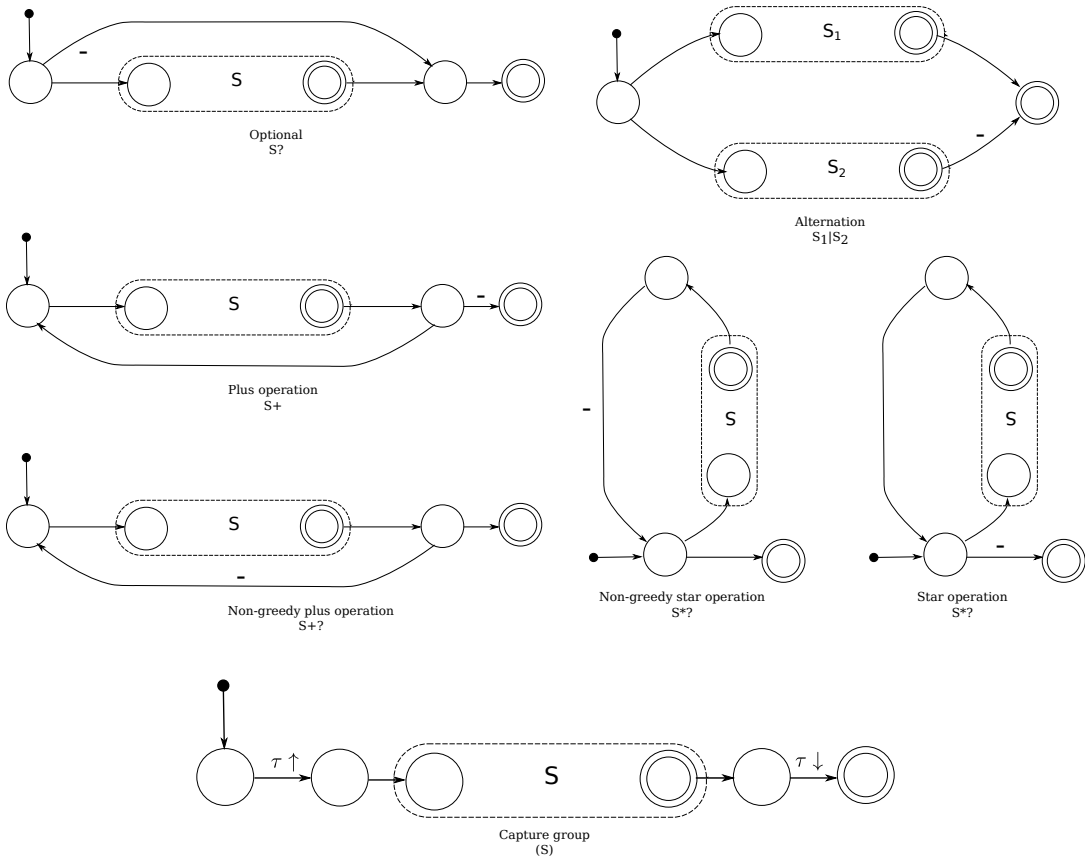
FIGURE 3.2: Modified Thompson [28] construction of the automaton: Descend into the abstract syntax tree of the regular expression and expand the constructs recursively.

2. We add prioritized transitions to model repetition. In our example, repeating is achieved by adding two ε transitions: one from 4 back to 3, to match more than one time any character, and another one from 3 to 5, to enable matching nothing at all. Importantly, the transition from 4 back to 3 is marked as low priority (the "–" sign) while the transition leaving the automaton, from 3 to 5, is unmarked, which means normal priority. This means that the NFA prefers leaving the repeating expression, rather than staying in it. If the expression were greedy, then we would mark the transition from 3 to 5 as low-priority, and the NFA would prefer to match any character repeatedly.

More generally, the NFA prefers to follow transitions of normal priority over those of low priority. Rather than formalize this notion of preference on NFAs, we come back to prioritized transitions when discussing the transformation from NFA states to DFA states.

To model capture groups in the NFA, we add *commit tags* to the transition graph. The transition into a capture group is tagged by a commit, the transition to leave a capture group is tagged by a another commit. We distinguish opening and closing commits. The

NFA keeps track of all times that a transition with an attached commit was used, thus keeping the *history* of each commit. After parsing succeeds, the list of all histories can then be used to reconstruct all matches of all capture groups.
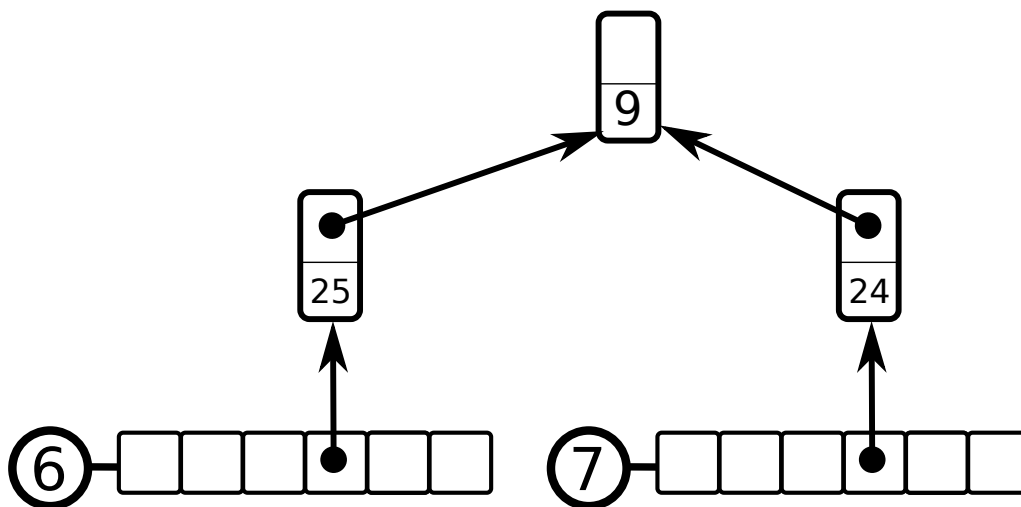

FIGURE 3.3: Histories are cells of singly linked lists, where only the first (here bottom-most) cell can be edited. This is a view of the automaton in figure 3.1 after the string "Tom Lehrer,1;Alan Turing," has been consumed. Only the cell for the closing of the second capture group is shown.

We model histories as singly linked lists, where the payload of each node is a position. Only the payload of the *head*, the first node, is mutable, the *rest*, all other nodes, are immutable. Because the rests are immutable, they may be shared between histories. This is an application of the flyweight pattern, which ensures that all of the following instructions on histories can be performed in constant time. Here, the *position* is the current position of the matcher.

3.3   TDFA

As discussed earlier, a useful metaphor for regular expression matching is that of threads[5] or co-routines (for clarity we use co-routine, because our algorithm is not parallel). Whenever we aren't sure which transition to take, we "fork" a co-routine for every option that we have. This way, when the input is over, there must be at least one co-routine that guessed correctly at all times.

The key insight is that we keep all co-routines in lock-step. To achieve this, we must be very specific about what constitutes the state of a co-routine. Since every co-routine effectively simulates a different NFA run, the state inside of a co-routine contains exactly two items: the NFA state it simulates and the history for every tag. Now, the following would be a correct, although slow, implementation of an NFA interpreter without thaumaturgical component: whenever an input character is read, we can iterate over all

co-routines, kill the ones that have no legal transition for the input character, and fork more co-routines as needed.

Trouble starts when we want to fork a co-routine for an NFA state that is already running. Not only is an explosion of co-routines bad for performance, it would also lead to ambiguity: if the two co-routines after reading the whole string disagree on the histories, which one is correct?

Notation. We use the following vocabulary.

DFA states are denoted by a capital letter, e.g. Q, and contain multiple co-routines.

$$Q = [(q_1, (h_1, h_2, h_3, h_4, h_5, h_6)),$$
$$(q_2, (h_1, h_2, h_3, h_4, h_7, h_8))]$$

for example means that the current DFA state has one co-routine in NFA state q_1 with histories $(h_1, h_2, h_3, h_4, h_5, h_6)$ and another co-routine in NFA state q_2 with the histories $(h_1, h_2, h_3, h_4, h_7, h_8)$. Note that histories can be shared across co-routines if they have the same matches.

Histories are linked list, where each node stores a position in the input text. The head is mutable, the rest is immutable. Therefore, histories can share any node except their heads. We write $h = [x_1, \ldots, x_m]$ to describe that matches occurred at the positions $x_1, \ldots, x_m$.

Co-routines are denoted as pairs (q_i, h), where q_i is some NFA state, and $h = (h_1, \ldots, h_{2n})$ is an array of histories, where n is the number of capture groups. Each co-routine has an array of $2n$ histories. In an array of histories $(h_1, h_2, \ldots h_{2n-1}, h_{2n})$, history h_1 is the history of the openings of the first capture group, $h2$ is the history of the closings of the first capture group, and so on.

Transitions are understood to be between NFA states, $q_1 \to q_2$ means a transition from q_1 to q_2.

Take for example the regular expression `(..)+` matching pairs of characters, on the input string "abcd". The history array of the finishing co-routine is $[h_1 = [0], h_2 = [3], h_3 = [2, 0], h_4 = [3, 1]]$. Histories h_1 and h_2 contain the positions of the entire match: position 0 through 3. Histories h_3 and h_4 contain the positions of all the matches of capture group 1, in reverse. That is: one match from 0 through 1, and another from 2 through 3.

Our engine executes instructions at the end of every interpretation step. There are four kinds of instructions:

$h \leftarrow \mathbf{p}$ Stores the current position into the head of history h.

$h \leftarrow \mathbf{p} + 1$ Stores the position after the current one into the head of history h.

$h' \mapsto h$ Sets head.next of h to be head.next of h'. This effectively copies the (immutable) rest of h to be the rest of h', also.

$c \uparrow (h)$ Prepends history h with a new node that becomes the new head. This effectively *commits* the old head, which is henceforth considered immutable. $c \uparrow (h)$ describes the opening position of the capture group and is therefore called the opening commit.

$c \downarrow (h)$ This is the same as $c \uparrow (h)$ except that it denotes a closing commit marking the end of the capture group. This distinction is necessary, because an opening commit stores the position *after* the current character and the closing commit store the position *at* the current character.

Further, in order to write down our algorithm as a special case of the TNFA execution algorithm 2, we describe the transitions that consume characters as ε-transitions tagged with the character and negatively prioritized transitions as tagged with $-$. The actual implementation deviates from this in favour of speed.

With this, the algorithm is simply implementing the fitting *interpret* function as seen in algorithm 2.

1: **function** $interpret(t, m)$
2:     **if** t is the character c **then**
3:         **if** m is not fed and can current character is consumable by c **then**
4:             become fed
5:         **else**
6:             die
7:         **end if**
8:     **end if**
9:     **if** t is ε **then**
10:         **if** m is not fed **then**
11:             die
12:         **end if**
13:         add new state to *buffer*
14:     **end if**
15:     **if** t is open tag of group i **then**
16:         create new history h as a copy of $m.histories[i]$
17:         write $index + 1$ to history h
18:         create new memory m', where $m'.histories[i] = h$
19:     **end if**
20:     **if** t is close tag of group i **then**
21:         create new history h as a copy of $m.histories[i + 1]$
22:         write $index$ to history h
23:         create new memory m', where $m'.histories[i + 1] = h$
24:         commit $m'.histories[i]$ and $m'.histories[i + 1]$
25:     **end if**
26: **end function**

The states are fed into the algorithm in the order visited, so that the coroutine that got furthest is expanded first when the next character is read. The *buffer* variable is a detail that ensures this.

Note that the ordering of co-routines inside of DFA states is relevant. In figure 3.1, after reading only one comma as an input, state 7 can be reached from two co-routines: either from the co-routine in state 3, via 4, or from the co-routine in state 6. The two co-routines are 'racing' to capture state 7. Since in the starting state, the co-routine of state 6 is listed first, he 'wins the race' for state 7, and 'captures it'. Thus, the new co-routine of state 7 is a fork of the co-routine of state 6, not 3. This matters, since 6 and 3 may disagree about their histories.

Example 3.1. *Execution of algorithm 2 with the interpret as above:*

Consider the automaton in figure 3.1 is in the DFA state[1]

$$Q = [(q_6, H_2 = (h_1, h_2, h_7, h_8, h_5, h_6)),$$
$$(q_3, H_1 = (h_1, h_2, h_3, h_4, h_5, h_6))]$$

This is the case after initialization or before any commas are read.

The algorithm uses a stack lookat *with the possibility to push a low priority transition below all high priority transitions[2].*

We pretend for clarity that instructions are executed directly after they are encountered. The actual algorithm collects them and executes them after the run *call to allow further optimizations and the storage of the instructions.*

Further, in the scope of this algorithm, co-routines have one extra bit of information attached to them: they can be hungry or fed. Hungry co-routines can only follow transitions that consume characters, fed co-routines can only follow ε transitions.

This is the execution of run*(Q):*

1. *Fill the stack with hungry co-routines of all states of Q. Now, lookat $= [(q_6, H_2), (q_3, H_1))]$, where the first element is the head of the stack.*

2. *Initialize* buffer *as an empty stack. The* buffer *stack exists because while following high priority transitions, states are discovered in an order that is reversed with respect to the order in which we would like to output them.*

3. *Initialize $R = [\ ]$, the DFA state under construction.*

4. *co-routine (q_6, H_2) is popped from the stack. It is hungry.*

---

[1]This is the starting state, except for the omission of co-routines that would die immediately after scheduling because there are no consuming transitions attached to their NFA state.

[2]This is easily implemented using two stacks, *high* and *low*.

5. *We iterate all available transitions in the NFA transition graph, and find only $q_6 \to q_7$, which can consume character ",".*

6. *(q_7, H_2) is pushed to the stack as fed, and we continue the main loop.*

7. *(q_7, H_2) is taken from the stack. It is fed.*

8. *(q_7, H_2) is pushed on buffer.*

9. *Since (q_7, H_2) is fed, it follows ε transitions.*

10. *The available transition $q_7 \to q_8$ is evaluated:*

    (a) *This transition has a opening tag for capture group 3 on it, and so we'd like to change h_5, the relevant history (see definition of Q above). However, since we're spawning a new co-routine, we cannot change h_5 itself. Instead, we copy h_5, and change the copy.*

    (b) *A new history h is created.*

    (c) *$h_5 \mapsto h$. Note that this is constant time, no matter how many entries h_5 already has.*

    (d) *$h \leftarrow \mathbf{p}+1$. This is the position after the ",", because the comma was eaten before the capture group starts.*

    (e) *Create $H_3 = h_1, h_2, h_7, h_8, h, h_6)$ as a copy of H_2, with h in the appropriate position.*

    (f) *(q_8, H_3) is pushed on the stack.*

11. *(q_8, H_3) is taken from the stack.*

12. *It is pushed on buffer. buffer$= [(q_8, H_3), (q_7, H_2)]$*

13. *It can follow no further transitions and dies.*

14. *(q_3, H_1) is popped from low. It is hungry.*

15. *We now flush buffer: $R = [(q_8, H_3), (q_7, H_2)]$, buffer$= [\ ]$. Note that now, R contains two co-routines in the reverse order in which they were discovered.*

16. *(q_3, H_1) is one of the two co-routines that constitute the input of this algorithm. Note how the other, (q_6, H_2), got a chance to follow all of its transitions before (q_3, H_1) was first popped off the low stack.*

17. *The only transition that consumes "," is $q_3 \to q_4$:*

    (a) *(q_4, H_1) is pushed to the stack as a fed co-routine.*

18. *(q_4, H_1) is popped from the stack*

19. *It is pushed to the buffer. buffer$= (q_4, H_1)$*

20. *$q_4 \to q_3$ is visited.*

    (a) *co-routine (q_3, H_1) is pushed to the stack after all high priority transitions (but there are none), because $q_4 \to q_3$ has low priority.*

21. *We flush the buffer again: $R = [(q_8, H_3), (q_7, H_2), (q_4, H_1)]$ Note how (q_4, H_1) appears last in R.*

22. (q_3, H_1) *is taken from the stack.*

23. *It is added to* buffer*.*

24. $q_3 \rightarrow q_5$ *is visited:*

    (a) (q_5, H_1) *is pushed to the stack.*

25. (q_5, H_1) *is taken from the stack.*

26. *It is added to* buffer*.*

27. $q_5 \rightarrow q_6$ *is visited and contains the closing commit of the second capture group:*

    (a) *Two histories are created to store the new positions of both the start and the end of the capture group. This ensures that other co-routines will not corrupt the memory.*

    (b) *A new history h is for the opening of the capture group.*

    (c) *A new history h' is created for the closing position.*

    (d) $h_3 \mapsto h$. *See the definition of Q above, to see that $h3$ is the opening capture group position of H_1.*

    (e) $h_4 \mapsto h'$.

    (f) $h' \leftarrow \mathbf{p}$. *This is the position of the ","*.

    (g) *Create a new history array, with h and h' in place.* $H_4 = [h_1, h_2, h, h', h_5, h_6]$

    (h) (q_6, H_4) *is pushed to the stack.*

28. (q_6, H_4) *is taken from the stack.*

29. *It is added to the* buffer*.*

30. *The stack is empty.*

31. *We flush our* buffer*:*

$$R = [(q_8, H_3), (q_7, H_2), (q_4, H_1),$$
$$(q_6, H_4), (q_5, H_1), (q_3, H_1)]$$

32. *R is returned.*

The output contains six co-routines, but three of them, (q_7, H_2), (q_4, H_1), (q_5, H_1), will die as soon as they are scheduled in the next iteration of the algorithm, because there are no outgoing non-ε transitions attached to their NFA states.

The overall run time of algorithm 2 depends heavily on the forking of co-routines being efficient: In the worst case, it takes $\Theta(m\, T_{fork}(m))$ time. A naive solution is a copy-on-write array, for which $T_{fork}(m) = m$ gives $O(m^2)$ for every character read, resulting in a $O(n\,m^2)$ regular expression matching, which is only acceptable if we assume m to be fixed.

Since at most two histories are actually changed, much of the array would not be modified and could be shared across the original co-routine and the forked one. This is easily

achieved replacing the array by a persistent [9] data structure to hold the array. A persistent treap, sorted by array index, has all necessary properties[3] and is further elaborated upon in section 4.1. With $T_{fork} = O(\log m)$, the overall runtime is $O(n\, m\, \log m)$.

---

[3]Clojure [14] features a slightly more complex data structure under the name of 'persistent vectors'. Jean Niklas L'orange offers a good explanation in "Understanding Clojure's Persistent Vectors", http://hypirion.com/musings/understanding-persistent-vector-pt-1.

Chapter 4

data structures

In the previous chapter, we obtained an algorithm with a run-time proportional to the cost of storing updates in a fixed size data structure with one access and one update. This needs to be fully persistent[1] in order to allow for co-routine forking.

This chapter presents some possible data structures, that allow for this.

4.1   Fully persistent data structures

In order to allow modifications of our list of histories, it is necessary to build a data structure that provides the interface an guarantees of a copy-on-write array:

Definition 4.1. An `Arraylike` data structure A of length m must provide the following interface:

Constructor $A(m)$ must return a structure of length m.

Random access An instance of A must provide a method $get(i)$, which gives a history for each $0 \leq i < m$.

Setting elements An instance of A must provide a method $set(i, history)$, which returns a new version of A, so that $get(i) = history$. The original instance must still return the same value for any get and set – it must be logically unmodified.

This is a definition of a fixed-size fully-persistent linear data structure.

Note that multiple versions of the data structure can change, every version that is in a living co-routine. A way to visualize this is as a tree of versions, where each set call forks off a new node, as seen in figure 4.1.

---

[1]This means that old versions of the data structure are still accessible and can be forked so that a new data structure with only that change applied can be accessed.
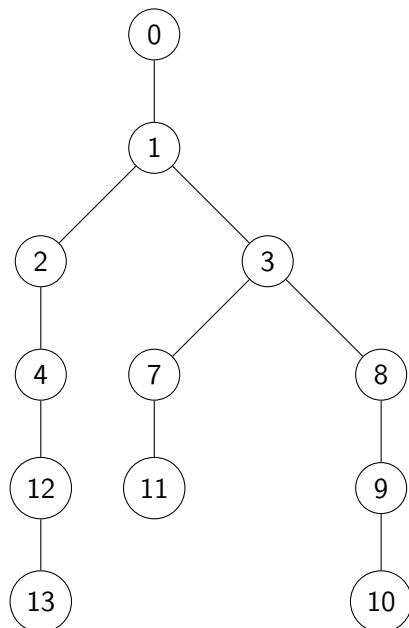
FIGURE 4.1: A tree of versions. Forks in the tree mean that multiple threads forked from the same state in the TNFA. The labels describe be the relative order of creation. It isn't linear.

The rest of this section looks at different possible implementations and their relative performance. Since in the worst case *get* as well as *set* are required in each step of each co-routine, the relevant performance measure is $T = O(T_{get} + T_{set}) = O(max(T_{get}, T_{set}))$ measured in amortized time.

4.2   Copy-on-write array

The arguably simplest class to provide the interface is an array that copies itself before making modifications:

Constructor $A(m)$ allocate an array of histories a of size m.

Random access $get(i)$ gives the ith element of the array in $O(1)$ time.

Setting elements $set(i, history)$ copies a and set the ith element to be *history*. This takes $O(m)$ time.

The performance for *get* and *set* gives us $T = O(1 + m) = O(m)$.

While the asymptotic performance is abysmal, the copy-on-write array is not to be dismissed apriori, since it has great memory locality properties and might outperform more complex structures for small m.

4.3   Treap

The treap data structure allows copy-on-write in $\log m$ time, sharing much of the structure by using the flyweight pattern as can be seen in figure 4.2. The treap is a balanced, left-leaning binary tree with entries in each node, equipped with the operations `get` and `set` as follows.

Algorithm 3 Implementation of the treap methods

1: **function** GET(treap, index)
2:     **if** index $= 0$ **then**
3:         **return** treap.entry
4:     **else if** index $- 1 <$ treap.left.size **then**
5:         **return** GET(treap.left, index $- 1$)
6:     **else**
7:         **return** GET(treap.right, index $-$ treap.left.size $- 1$)
8:     **end if**
9: **end function**

10: **function** SET(treap, index, entry)
11:     **if** index $= 0$ **then**
12:         treap$' \leftarrow copy(treap)$
13:         treap$'$.entry $\leftarrow$ entry
14:         **return** treap'
15:     **else if** index $- 1 <$ treap.left.size **then**
16:         treap$' \leftarrow copy$(treap)
17:         treap$'$.left $\leftarrow$ SET(treap.left, index $- 1$, entry)
18:         **return** treap$'$
19:     **else**
20:         treap$' \leftarrow copy$(treap)
21:         treap$'$.left $\leftarrow$ SET(treap.right, index $-$ treap.left.size $- 1$, entry)
22:         **return** $treap'$
23:     **end if**
24: **end function**

This structure gives us better asymptotic performance at the cost of following $\log m$ pointers, because $T = O(\log m + \log m) = O(\log m)$.

4.4   Storing updates

Driscoll et al[9] describe how any pointer machine data structure with a fixed number of links to each version can be converted to a fully persistent data structure in $O(1)$ amortized time. This section shows the steps involved to equip an array with such an interface.
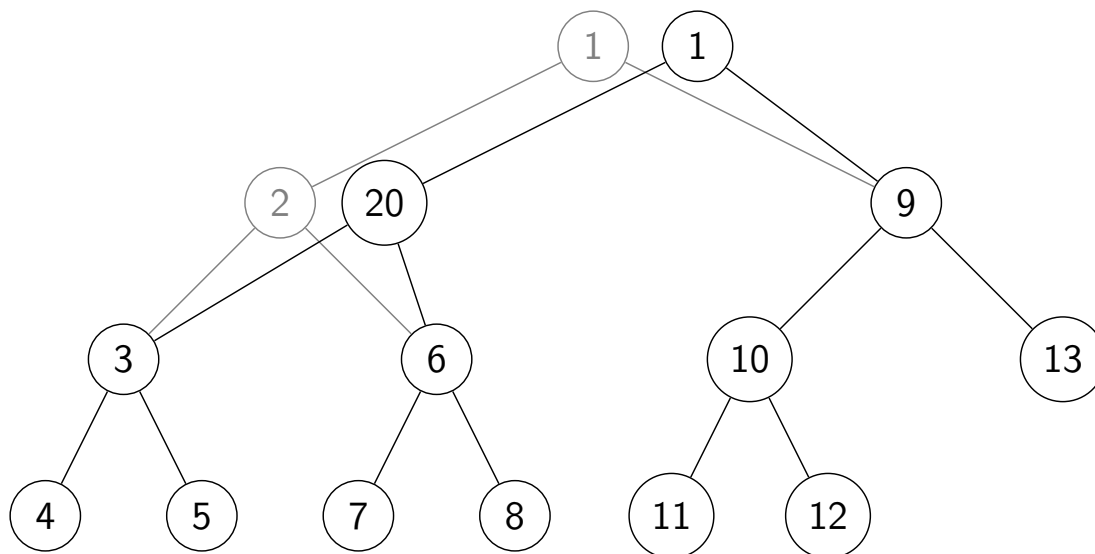
FIGURE 4.2: Writing to the second entry of the treap only requires $O(\log m)$ copies while keeping the old structure intact (persistence).

The idea behind the conversion is that instead of modifying the array directly, we instead keep a buffer of modifications that is of fixed size. Let's for the moment assume that we have a linear order of modifications and only add the version tree later.

4.4.1   Linear versions

In this part, we only modify the most recent version $t^\star$ and only access the old versions $0 \le t \le t^\star$. In order to store the modifications, we introduce the struct $Set(t, i, history)$, which can be stored in the modification buffer. The implementation of the simple methods can be seen in algorithm 4

The amorized constant time for *get* and *set* arise from the fact that while we iterate through the changes, we only have a constant number of them.

4.4.2   Version tree

In the general case of a version tree, we lose the notion of a total order of versions and remain with a partial order. Version t_1 and version t_2 can now be in three different relations: Either t_1 can happen before t_2, so that the modification in t_1 also needs to be taken into account in t_2, t_2 can happen before t_1, or they can be siblings, so that the modifications shouldn't influence each other.

The problem of generalizing this approach to version trees is two fold: The first problem is, that we need to determine whether a modification applies to a version in constant time, otherwise *get* would become slower. The second problem is that modifying the

---

Algorithm 4 Methods of the *LazyApply* data structure

1: **function** MAKELAZYAPPLY(m)
2:     allocate an array of m histories hs
3:     allocate a buffer b of size p for modifications.
4:     **return** $\{histories : hs, buffer : b\}$
5: **end function**
6: **function** GET(lazyApply, t, index)
7:     $history \leftarrow lazyApply.histories[index]$
8:     **for** each $Set(t', i, history')$ **do**
9:         **if** $t' < t$ and $i = index$ **then**
10:             $history \leftarrow history'$
11:         **end if**
12:     **end for**
13:     **return** $history$
14: **end function**

15: **function** SET(lazyApply, t, index, entry)
16:     **if** $lazyApply.buffer$ has space left **then**
17:         add $Set(t, index, entry)$ to $lazyApply.buffer$
18:         **return** $lazyApply$
19:     **else**
20:         **return** new $LazyApply$ with all changes applied
21:     **end if**
22: **end function**

---

same version reduces to copying the array of histories if said version has a full buffer. The latter is more easily solved and thus will be discussed first.

Emptying a full buffer   Instead of applying all modifications, we can split the modification buffer in two roughly equal parts. In order to do that, we can find a subtree v of modifications of size $\approx \frac{p}{2}$. Instead of applying all modifications to the array of histories, we only apply the ancestors of v. Then we delete all modifications in v from the original node. Now we would be breaking the interface, because some co-routines still have references to the original node, but actually use a version that is in v. This can be avoided by storing back-pointers to the threads and modify their references to the new node if necessary. Since there are at most m threads referencing the versions of each node, we can still update the references in amortized constant time.

Constant time ordering of versions   In order to determine whether to apply a modification on a read, we need to find the relation between the version that is querried and the one that is stored. If $t_{stored} < t_{querried}$, we apply the modification. This requires us however to determine the relative positions of t_{stored} and $t_{querried}$ in the version tree, where the naive implementation would take $\log m$ time. This is the order maintenance

problem and fittingly can be solved using an order maintenance data structure, such as the one proposed by Dietz and Sleator[8] which we discuss in section 4.5.

For the moment let us assume that we have a linear data structure, that

1. allows inserting an element next to a known element in $O(1)$ time.
2. allows us to query the order of two elements a and b in the list in $O(1)$ time. We'll call this operation $query_<(a, b)$.

With this we can flatten the tree to a list by adding a b_t beginning element and a e_t closing element to the list as seen in figure 4.3.

$$(b_1 \ (b_2 \ (b_3 \ e_3) \qquad (b_4 \ e_4) \ e_2) \qquad (b_3 \ b_3) \ e_1)$$
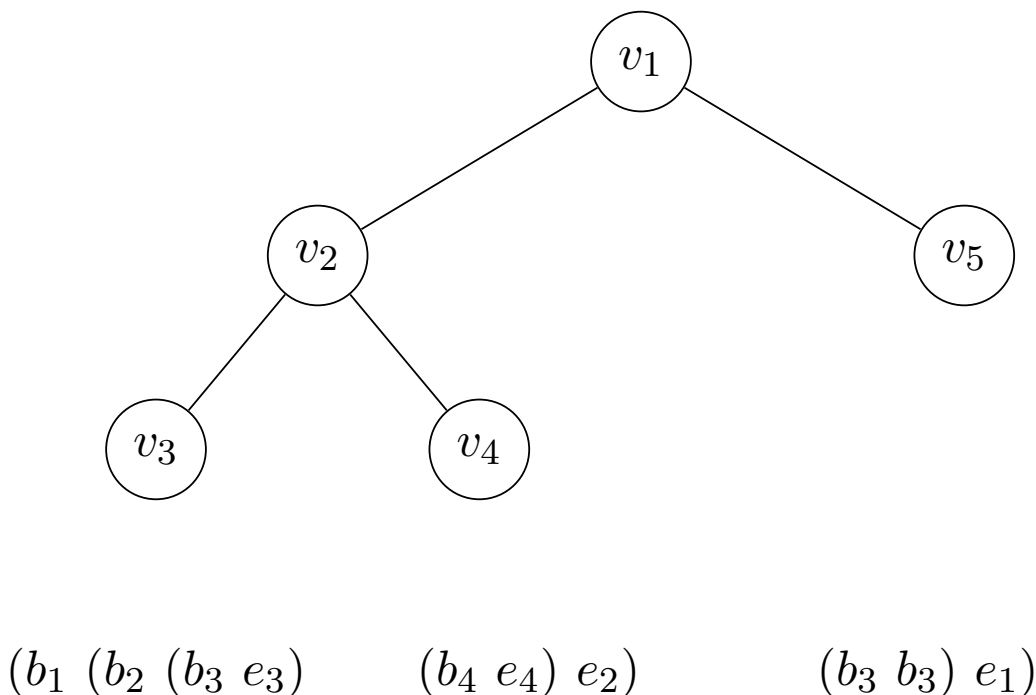
FIGURE 4.3: A flattened version tree

Now we can find if version t_2 depends on version t_1, by calculating

$$query_<(b_{t_1}, b_{t_2}) \ \&\& \ query_<(e_{t_2}, e_{t_1})$$

If t_1 encloses t_2 completely, then t_2 is part of the subtree of t_1 and therefore we'd need to apply the modification at t_1 to find values at t_2.

4.5 Order maintenance

Dietz and Sleator describe in their paper Two Algorithms for Maintaining Order in a List[8][2], a simple one that provides amortized guarantees and a more involved one, which could offer worst-case guarantees. For our purposes the amortized version suffices, as noted by Driscoll et al[9, p. 108].

The data structure keeps so called *tags* $\eta(e)$ for elements, which are integers that describe the order. In order to query the relative position of two elements, their tags are simply compared, which leaves us with the problem of maintaining tags for all list elements such that $\eta(e_1) < \eta(e_2) \Leftarrow index(e_1) < index(e_2)$.

Compare this to the related problem of list-labeling: in order maintenance we need to be able to compute the full label for each element in constant time, whereas in list-labeling, the node has to contain the full label itself. This insight is key to maintaining order in amortized constant time. For the moment however we look at the method we can use for list-labeling.

The key to achieve this is to use a doubly linked list of tags with their corresponding elements. If we need to insert an element after another known element e, we can insert it, but to keep our invariant, we assign $\eta(e_{new}) = \left\lfloor \frac{\eta(e)+\eta(e.next)}{2} \right\rfloor$. This is possible, unless the new tag is actually equal to $\eta(e)$ – if $\eta(e)$ doesn't have any space to $\eta(e.next)$. In this case we need to relabel the existing nodes.

4.5.1 Relabeling

The purpose of the relabeling step is to ensure that there is a gap to fit in the new element and reduce the potential of a next reordering step. An intuitive perspective is shown by Bender et al[3]:

The labels can be thought of a coding of paths in a complete binary tree as visualized in figure 4.4. We can then define the *overflow* in a subtree, which triggers relabeling.

Definition. The *overflow threshold* of a subtree is 1.5^i for any level[3] i, starting counting from the leafs, which are level 0.

Overflow happens, when the number of items in a subtree are bigger than the overflow threshold.

With this definition, we can define the region that we relabel to be the first subtree that isn't in overflow – which means that is is filled sufficiently sparsely to reduce the potential

---

[2]And Bender et al[3] in their revision of this classical paper

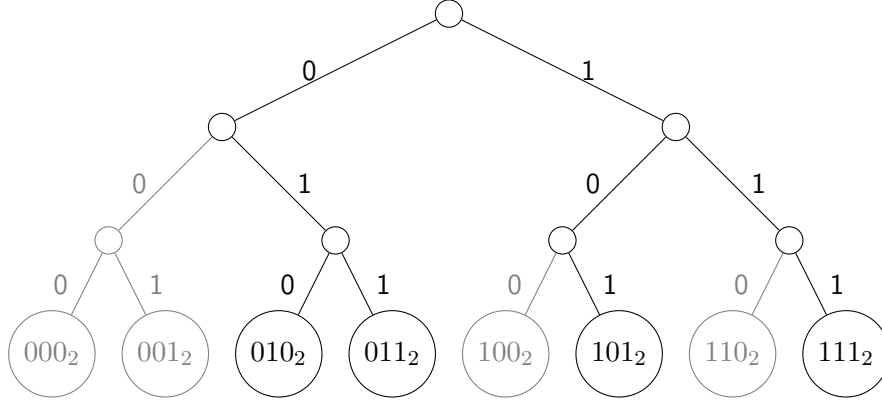[3]$a = 1.5$ is an arbitrary choice for a number $1 < a < 2$.

FIGURE 4.4: Labels are binary numbers describing paths in a tree. Relabeling walks up the tree until it finds a subtree with a density below the threshold. In this range, the labels changed to make them equally spaced.

new relabels. In this subtree we spread the labels equally, which ensures that we will not have to do any relabeling for another $\lfloor 1.5^{-i} \rfloor$ inserts. Note that a much bigger region might be in overflow, but no checks are done if they are not triggered by an overflow at level 0, i.e. placing a node between two adjacent nodes.

This algorithm leaves us with the disappointment of only achieving $O(\log m)$ amortized insert. This can be resolved using a technique known as *indirection*.

4.5.2   Indirection

Indirection is a method introduced by Willard[29] to eliminate annoying log factors under certain conditions such as the one just encountered. With it we leave the realm of list-labeling and cease to store the full label in each node. Instead we now keep a two-level structure as shown in figure 4.5.
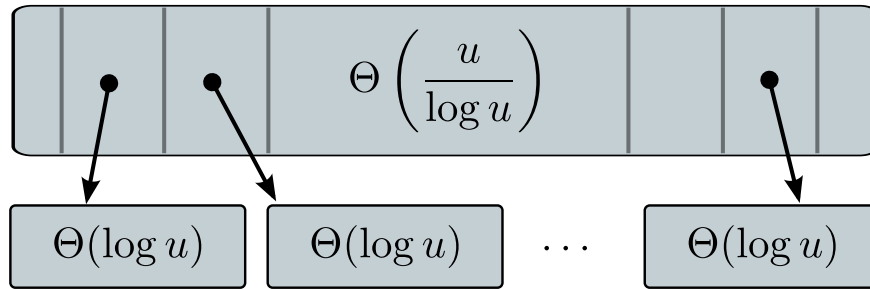
FIGURE 4.5: Indirection structure: use a compound index with the high-order bits being the index for the upper structure and the low-order bits being the index for the lower structure. Since the upper index is stored in the summary structure, $\log u$ items can be relabeled in $O(1)$ time.

We split our universe into blocks of size $\Theta(\log u)$. In these blocks we keep a labeled list as seen in the previous section. Above this, we create a summary structure, which stores a second part of the index. This too is a labeled list, but since each element of

the summary structure represents $\Theta(\log u)$ elements of the universe, we can effectively do relabels faster by a log-factor. This gives us amortized $O(1)$ updates.

It can be said that this seems to set a limit on the size of the match: Once our labels are used up, we cannot do $O(1)$ order queries anymore. This is true, but there is a smaller limit already in place: We can't store pointers to a string that doesn't fit into memory. If we cease to understand positions as fixed in size, we would again get a log-factor. This is typically not considered.

Chapter 5

Proofs

In this chapter we will prove the claimed properties, first and foremost the correctness of the algorithm.

5.1 Correctness

The correctness of the algorithm follows by induction over the construction: If the correct co-routine stops in the end state for all possible constructions of the Thompson construction under the assumption that simpler automata do the same, it follows that no matter how complex the automata get, the algorithm will have the correct output.

To this goal, we will use backtracking as a handy definition of correctness. We will show that our algorithm will prefer the same paths as an backtracking implementation would. It should be noted that the construction is exactly set up so that it matches backtracking and in fact this can be seen as a simple derivation of our algorithm.

First we need a simple formalization of the backtracking procedure:

$$bt(a|b, s) = bt(a, s) \quad | \quad bt(b, s)$$

$$bt(r*, s) = bt(rr * |\varepsilon, s)$$

$$bt(r*?, s) = bt(\varepsilon|rr*, s)$$

$$bt(r?, s) = bt(r|\varepsilon, s)$$

$$bt(r??, s) = bt(\varepsilon|r, s)$$

$$bt(r+, s) = bt(rr*, s)$$

$$bt(r+?, s) = bt(rr*?, s)$$

$$bt(ab, s) = bt(a, s) + bt(b, rest)$$

$$bt(Group(i, r), s) = [WriteOpen(i)] + bt(r, s) + [WriteClose(i)]$$

Second we notice that the algorithm preserves the order of the co-routines after each character read. This means that basically a depth first search is performed, with priorities formalizing what option is to be taken first.

That certain paths are cut off, because the state has already been seen is equivalent to memoization in the backtracking procedure: If a higher priority state already found a path through this part of the parse, the following parse can be pruned.

Now the parses are analogous for our procedure and *bt*:

$bt(a|b, s)$:

1. Check a

2. Check b
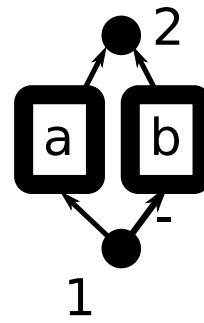
1. Check $1 \to a \to 2$

2. Check $1 \to b \to 2$
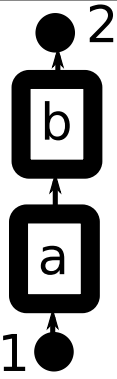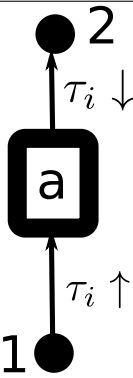
$bt(r*, s)$:

1. Check r

2. Check ε

1. Check $1 \to r \to 2 \to 1 \to 3$

2. Check $1 \to 3$

$\vdots$ $\vdots$

$bt(ab, s)$:

1. Check a
2. Check b on rest
3. Concatenate the updates

1. Run through a consuming some characters
2. Run through b
3. All changes are written

$bt(Group(i, r), s)$:

1. Write current position to changes
2. Check r
3. Write position after matching r to changes

1. Write current position to changes
2. Run through r
3. Write the changed position to changes

5.2 Execution time

The main structure of any NFA based matching algorithm is the nesting of two loops: The outer loop iterating over the n characters of the string, and the inner expanding at most m states. The expansion makes $O(1)$ updates per state expanded, as the construction described in section 2.2 gives a constant out-degree for each state. As described in chapter 4 the update cost of every co-routine is $O(1)$. This gives a total run time of $O(n\,m)$.

5.3   Lower bound for time

There is no known tight[1] lower bound to regular expression matching.

Theorem 1. No algorithm can correctly match regular expressions faster than $\Theta(n \min(m, |\Sigma|))$, where n is the length of the string, m is the length of the pattern, and $|\Sigma|$ is the size of the alphabet.

Proof. Let $S = a^n x_i$ and $R = [\texttt{ax}_1] * | [\texttt{ax}_2] * | \dots | [\texttt{ax}_\texttt{m}] *$. Note that $|S| = \Theta n$ and $|R| = \Theta(\min(m, |\Sigma|))$. Let further $\texttt{match}$ be a valid regular expression matching algorithm, then $\texttt{match}(S, R)$ is equivalent to finding $a^n x_i \overset{?}{\in} \{a^n x_1, \dots, a^n x_m\}$. There is no particular order to $\{a^n x_1, \dots, a^n x_m\}$, so the lower bound for finding this is $\Theta(|S| \, |R|)$. $\square$

---

[1] A lower bound l is tight, if it is the asymptotically largest lower bound

Chapter 6

Implementation

While repeatedly calling algorithm 2 would be sufficient to reach the theoretical time bound we claimed, practical performance can be dramatically improved by avoiding to construct new states. Instead, we build a *transition table* that maps from old DFA states and an input range to a new DFA state, and the instructions to execute when using the transition. We build the transition table, including instructions, as we go. This is what we mean when we say that the DFA is *lazily compiled.*

6.1   DFA transition table

The DFA transition table is different from the NFA transition table, in that the NFA transition table contains ε transitions and may have more than one transition from one state to another, for the same input range. DFA transition tables allow no ambiguity.

Our transition tables, both for NFAs and DFAs, assume a transition to map a consecutive range of characters. If, instead, we used individual characters, the table size would quickly become unwieldy. However, input ranges can quickly become confusing if they are allowed to intersect. To avoid this, and simplify the code dramatically, while keeping the transition table small, we use the following trick. When the regular expression is parsed, we keep track of all input ranges that occur in it. Then, we split them until no two input ranges intersect. After this step, input ranges are never created again. Doing this step early in the pipeline yields the following invariant: it is impossible to ever come across intersecting input ranges.

To give us a chance to ever be in a state that is already in the transition table, we check, after executing algorithm 2, `run`, whether there is a known DFA state that is *mappable* to the output of `run`. If `run` produced a DFA state Q, and there is a DFA state Q' that contains the same NFA states, in the same order then Q and Q' may be mappable. If they are, then there is a set of instructions that move the histories from Q into Q' such that,

afterwards, Q' behaves precisely as Q would have. Algorithm 5 shows how we can find a mappable state, and the needed instructions. The run time of Algorithm 5 is $O(m)$, where m is the size of the input NFA.

Algorithm 5 $findMapping(Q)$: Finding a state that Q is mappable to in order to keep the number of states created bound by the length of the regular expression.

Require: $Q = [(q_i, h_i)]_{i=1\dots n}$ is a DFA state.
Ensure: A state Q' that Q is *mappable* to.
 1: The ordered instructions m that reorder the memory locations of Q to Q' and don't interfere with each other.
 2: **for** Q' that contains the same NFA states as Q, in the same order **do**
 3:      ▷ Invariant: For each history H there is at most one H'
 4: so that $H \leftarrow H'$ is part of the mapping.
 5:     Initialize empty bimap m ▷ A bimap is a bijective map.
 6:     **for** $q_i = q_i'$ with histories H and H' respectively **do**
 7:         **for** $i = 0 \dots length(H) - 1$ **do**
 8:             **if** $H(i)$ is in m as a key already and does not map to $H'(i)$ **then**
 9:                 Fail
10:             **else**
11:                     ▷ Hypothesize that this is part of a valid map
12:                 Add $H(i) \mapsto H'(i)$ to m
13:             **end if**
14:         **end for**
15:     **end for**
16: **end for**
17: ▷ The mapping was found and is in m.
18: sort m in reverse topological order so that no values are overwritten.
        return Q' and m

6.2   DFA execution

With these ingredients in place, the entire matching algorithm is straightforward. In a nutshell, we see if the current input appears in the transition table. Otherwise, we run `run`. If the resulting state is mappable, we map. More formally, we can see this in algorithm 6. Here, algorithm 6 assumes that algorithm 2 does not immediately execute its instructions, but returns them back to the interpreter, both for execution and to feed into the transition table.

6.3   Compactification

The most important implementation detail, which brought a factor 10 improvement in performance, was the use of a compactified representation of DFA transition tables whenever possible. Compactified, here, means to store the transition table as a struct of arrays,

---

Algorithm 6 interpret(input): Interpretation and lazy compilation of the NFA.

Require: *input* is a sequence of characters.
Ensure: A tree of matching capture groups.

 1: ▷ Lazily compiles a DFA while matching.
 2: Set Q to *startState*.
 3: ▷ A co-routine is an NFA state, with an array of histories.
 4: Let Q be all co-routines that are reachable in the NFA transition graph by following ε transitions only.
 5: Execute instructions described in algorithm *run*, when walking ε transitions.
 6: ▷ Create the transition map of the DFA.
 7: Set T to an empty map from state and input to new state and instructions.
 8: ▷ Consume string
 9: **for** position *pos* in *input* **do**
10:     Let a be the character at position *pos* in *input*.
11:     **if** T has an entry for Q and a **then**
12:         ▷ Let the DFA handle a
13:         Read the instructions and new state Q' out of T
14:         execute the instructions
15:         $Q \leftarrow Q'$
16:         jump back to start of for loop.
17:     **else**
18:         ▷ lazily compile another DFA state.
19:         Run run(Q, a) to find new state Q' and instructions
20:         Run findMapping(Q', T) to see if Q' can be mapped to an existing state Q''
21:         **if** Q'' was found **then**
22:             Append the mapping instructions from findMapping to the instructions found by *run*
23:             Execute the instructions.
24:             Add an entry to T, from current state Q and a, to new state Q'' and instructions.
25:             Set Q to Q''
26:         **else**
27:             Execute the instructions found by *run*.
28:             Add an entry to T, from current state Q and a, to new state Q' and instructions.
29:             Set Q to Q'.
30:         **end if**
31:     **end if**
32: **end for**

---

rather than as an array of structs, as recommended by the Intel optimization handbook [4, section 6.5.1]. The transition table is a map from source state and input range to target state and instructions. Following Intel's recommendation, we store it as an object of five arrays: `int[] oldStates, char[] froms, char[] tos, Instruction[][] instructions, int[] newStates`, all of the same length, such that the ith entry in the table maps from oldStates[i], for a character greater than from[i], but smaller than to[i], to newStates[i], by executing instructions[i]. To read a character, the engine now searches in the transition table, using binary search, for the current state and the current input character, executes the instructions it finds, and transitions to the new state.

However, the above structure isn't a great fit with lazy compilation, as new transitions might have to be added into the middle of the table at any time. Another problem is that, above, the state is represented as an integer. However, as described in the algorithm, a DFA state is really a list of co-routines. If we need to lazily compile another DFA state, all of the co-routines need to be examined.

The compromise we found is the following: The canonical representation of the transition table is a red-black tree of transitions, each transition containing source and target DFA state (both as the full list of their NFA states, and histories), an input range, and a list of instructions. This structure allows for quick inserting of new DFA states once they are lazily compiled. At the same time, lookups in a red-black tree are logarithmic. Then, whenever we read a fixed number of input characters without lazily compiling, we transform the transition table to the struct of arrays described above, and switch to using it as our new transition table. If, however, we read a character for which there is no transition, we need to de-optimize, throw away the compactified representation, generate the missing DFA state, and add it to the red-black tree.

The above algorithm chimes well with the observation that usually, regular expression matching needs only a handful of DFA states, and thus, compactifying can be done early, and only seldom need to be undone.

6.4   Intertwining of the pipeline stages

The lazy compilation of the DFA when matching a string enables us to avoid compiling states of it that might never be necessary. This allows us to avoid the full power set construction [26], which has time complexity of $O(2^m)$, where m is the size of the NFA.

6.5   Parsing the regular expression syntax

Parsing the regular expression into an abstract syntax tree is a detail that can easily be missed. Since the algorithm for matching is already very fast, preliminary experiments
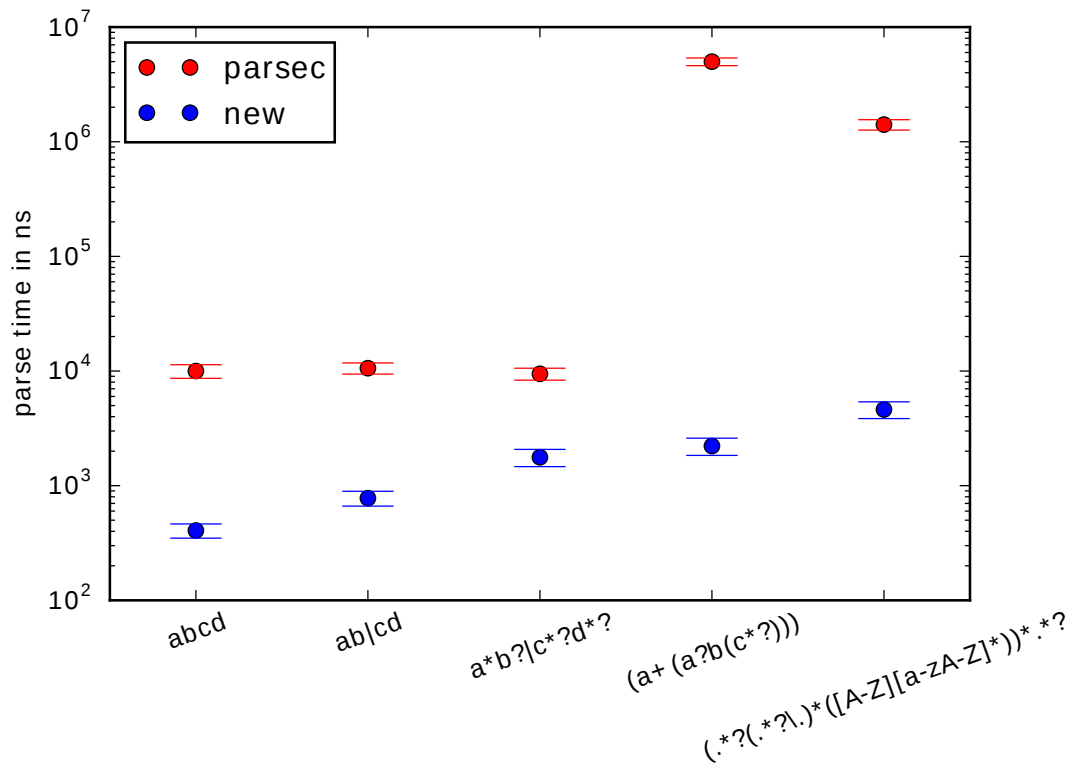
FIGURE 6.1: Comparison of two ways of parsing the regular expression syntax. Since the measurements are very noisy, the median with the MAD (median absolute deviation) are plotted.

showed that the parsing of the regular expression, even in simple regular expressions, this step can take up a major part (25% in our experiment) of the time for running the complete match.

The memory model to parse a regular expression is a stack, since capture groups can be nested. The grammar can be formulated as right recursive and with this formulation it can be implemented with a simple recursive descent parser as opposed to the previous Parsec parser. The resulting parser eliminated the parsing of the regular expression as a bottleneck, as can be seen in figure figure 6.1 (note the log plot).

Chapter 7

Benchmark

All benchmarks were obtained using Google's caliper[1], which takes care of the most obvious benchmarking blunders. It runs a warm-up before measuring, runs all experiments in separate VMs, helps circumvent dead-code detection by accepting the output of dummy variables as input, and fails if compilation occurs during experiment evaluation. The source code of all benchmarks is available, together with the sources of the project, on Github. We ran all benchmarks on a 2.3 GHz, i7 Macbook Pro.

As we saw in section 2, there is a surprising dearth of regular expression engines that can extract nested capture groups – never mind extracting entire parse trees – that do not backtrack. Back-tracking implementations are exponential in their run-time, and so we see in figure 7.1 (note the log plot) how the run-time of "java.util.regex" quickly explodes exponentially, even for tiny input, for a pathological regular expression, while our approach slows down only linearly. The raw data is seen in table 7.1.

n	13	14	15	16	17	18	19	20
java.util.regex	241	484	1003	1874	3555	7381	14561	30116
Ours	225	252	273	32	327	352	400	421

TABLE 7.1: Matching times, in microseconds, for matching $a?^n a^n$ against input a^n.

In the opposite case, in the case of a regular expression that's crafted to prevent any back-tracking, java.util.regex outperforms our approach by more than factor 2, as seen in table 7.2 – but bear in mind that java.util.regex does not extract parse trees, but only the last match of all capture groups. A backtracking implementation that actually does produce complete parse trees is JParsec[2], which, as also seen in table 7.2, performs on par with our approach.

---

[1] https://code.google.com/p/caliper/
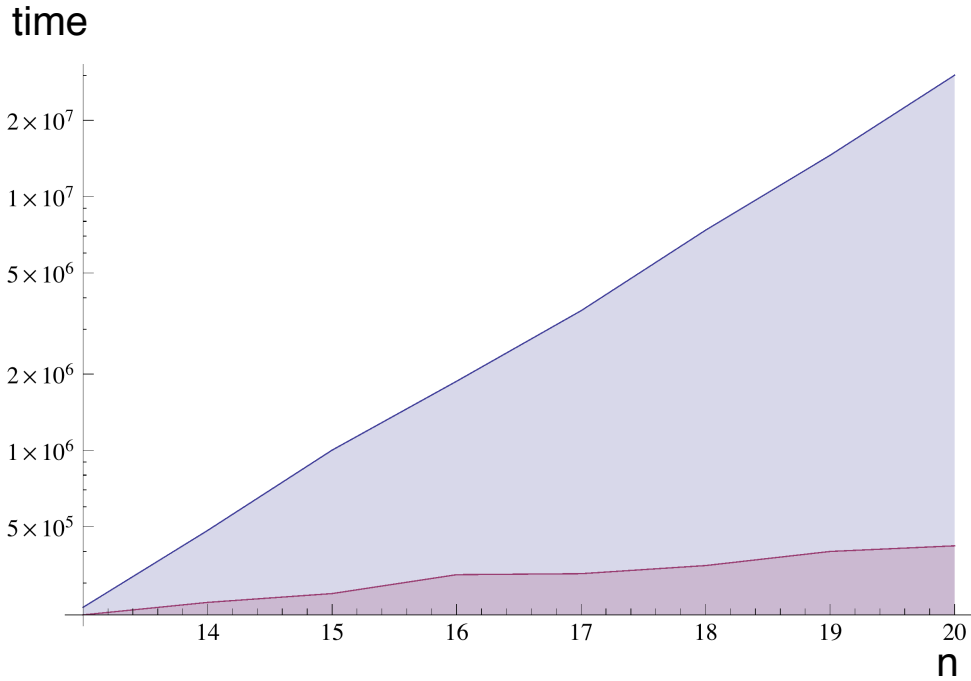[2] http://jparsec.codehaus.org

FIGURE 7.1: Time in nanoseconds for matching $a?^n a^n$ against input a^n. Bottom (purple) line is our approach, top (blue) line is java.util.regex.

Note that because java.util.regex achieves its back-tracking through recursion, we had to set the JVM's stack size to one Gigabyte for it to parse the input. Since default stack size is only a few megabytes, this makes using java.util.regex a security risk, even for unproblematic regular expressions that cannot cause backtracking, since an attacker can potentially force the VM to run out of stack space.

Tool	Time
JParsec	4,498
java.util.regex	1,992
Ours	5,332

TABLE 7.2: Matching regular expression `((a+b)+c)+` against input $(a^{200}bc)^{2000}$, where a^{200} denotes 200 times character 'a'. Time in microseconds.

Finally, as a more realistic example, neither chosen to favor back-tracking nor to avoid it, extracts all class names, with their package names, from the project sources itself. As seen in table 7.3, our approach outperforms java.util.regex by 40%, even though our approach constructs the entire parse tree, and thus all class names, while java.util.regex outputs only the last matched class name. JParsec was not included in this experiment, since it does not allow non-greedy matches. Even though it is possible to build a parser that produces the same AST, it would necessarily look very different (using negation) from the regular expression.

Tool	Time
java.util.regex	11,319
Ours	8,047

TABLE 7.3: Runtimes, in microseconds, for finding all java class names in all .java files in the project itself. The regular expression used is `(.*?([a-z]+\.)*([A-Z][a-zA-Z]*))*.*?`. Runtime in microseconds

Chapter 8

Conclusion

Regular expression make for lightweight parsers and there are many cases where data is extracted this way. If such data is structured instead of flat, a parser that produces trees is superior to a standard regular expression parser. We provide such an algorithm with modern optimizations applied using results from persistent data-structures to avoid unnecessary memory consumption and the slow-down that this would produce.

Our approach can produce entire parse trees from matching regular expressions in a single pass over the string and do so asymptotically no slower than regular expression matching without any extraction. The practical performance is on par with traditional back-tracking solutions if no backtracking ever happens, exponentially outperforms back-tracking approaches for pathological input, and in a realistic scenario outperforms back-tracking by 40%, even though our approach produces the full parse tree, and the back-tracking implementation doesn't. All source code and all benchmarks are available under a free license on Github[22] at https://github.com/nes1983/tree-regex.

Bibliography

[1] Arvind Arasu, Surajit Chaudhuri, Zhimin Chen, Kris Ganjam, Raghav Kaushik, and Vivek R. Narasayya. Experiences with using data cleaning technology for bing services. *IEEE Data Eng. Bull.*, 35(2):14–23, 2012.

[2] Ralph Becket and Zoltan Somogyi. DCGs + Memoing = Packrat parsing, but is it worth it? In *Practical Aspects of Declarative Languages*, volume LNCS 4902, pages 182–196. Springer, January 2008.

[3] Michael A Bender, Richard Cole, Erik D Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Algorithms—ESA 2002*, pages 152–164. Springer, 2002.

[4] Intel Coorporation. Intel 64 and ia-32 architectures optimization reference manual, 2013.

[5] R. Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *URL: http://swtch.com/~rsc/regexp/regexp1.html*, 2007.

[6] R. Cox. Regular expression matching: the virtual machine approach. *URL: http://swtch.com/~rsc/regexp/regexp2.html*, 2009.

[7] R. Cox. Regular expression matching in the wild. *URL: http://swtch.com/~rsc/regexp/regexp3.html*, 2010.

[8] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 365–372. ACM, 1987.

[9] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[10] Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[11] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM.

[12] Niels Bjørn Bugge Grathwohl, Fritz Henglein, Lasse Nielsen, and Ulrik Terp Rasmussen. Two-pass greedy regular expression parsing. In *Implementation and Application of Automata*, pages 60–71. Springer, 2013.

[13] Fritz Henglein and Lasse Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In *ACM SIGPLAN Notices*, volume 46, pages 385–398. ACM, 2011.

[14] Rich Hickey. The Clojure programming language. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–1, New York, NY, USA, 2008. ACM.

[15] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[16] L. Karttunen, J. P. Chanod, G. Grefenstette, A. Schiller, and Received February. Regular expressions for language engineering. In *Natural Language Engineering*, pages 305–328, 1996.

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

[18] Chris Kuklewicz. Regular expressions/bounded space proposal, February 2007.

[19] V. Laurikari. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 181–187. IEEE, 2000.

[20] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 2012.

[21] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, page 8, Berkeley, CA, USA, 2010. USENIX Association.

[22] Aaron Karper Niko Schwarz. O(n m) regular expression parsing library that produces parse trees, 2014.

[23] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[24] R. Pike. The text editor sam. *Software: Practice and Experience*, 17(11):813–845, 1987.

[25] Robert Sedgewick. *Algorithms in C (paperback)*. Addison-Wesley Professional, 1 edition, January 1990.

[26] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2 edition, February 2005.

[27] M. Sulzmann and K.Z.M. Lu. Regular expression sub-matching using partial derivatives. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, pages 79–90. ACM, 2012.

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

[29] Dan E Willard. Maintaining dense sequential files in a dynamic environment. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 114–121. ACM, 1982.

List of Figures

List of Tables

List of Algorithms