Efficiently extracting full parse trees using regular expressions with capture groups

Niko Schwarz[1], **Aaron Karper**[1], **and Oscar Nierstrasz**[1]

[1]**Software Composition Group, University of Bern, Switzerland**

ABSTRACT

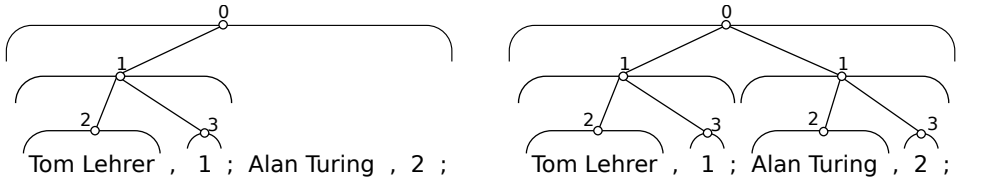
Regular expressions with capture groups offer a concise and natural way to define parse trees over the text that they are parsing, however classical algorithms only return a single match for each capture group, not the full parse tree. We describe an algorithm based on finite-state automata that extracts full parse trees from text in $\Theta(nm)$ time and $\Theta(dn+m)$ space (where n is the size of the text, m the size of the pattern, and d the number of groups in the pattern). It is the first to do so in a single pass with complete control over greediness. This allows the algorithm to process streaming data using all constructs familiar to users of regular expressions.

Keywords: Regular expressions, Parsing, Algorithms.

1 INTRODUCTION

Regular expressions are widely used as a simple and intuitive mechanism to search for patterns in large bodies of text. Standard regexes also allow you to specify and match text fragments of interest by surrounding them in parentheses — these are known as"capture groups". Very efficient algorithms have been developed to match regexes, but these only provide you with the final matching text fragments, not the entire tree of matches.

For example, `((.*?),(\d+);)+` might describe a dataset of ASCII names with their numeric label. Matching the regular expression on "`TomLehrer,1;AlanTuring,2;`" confirms that the list is well-formed, but the match contains only "`TomLehrer`" for the second capture group and "`1`" for the third. That is, the parse tree found by the POSIX is seen in Figure 1a.



(a) Initial parse tree produced by POSIX-compatible matching.

(b) Full parse tree produced by our approach.

Figure 1. Parse trees produced by matching regex `((.*?),(\ d+);)+` against input "`TomLehrer,1;AlanTuring,2;`".

We propose a new algorithm based on finite-state automata that can reconstruct a full parse tree after the matching phase is completed, as seen in Figure 1b. The worst-case run time of our approach is $\Theta(nm)$, the same as the algorithm extracting only single matches. It is the first algorithm to achieve this bound, while extracting parse trees with specified greediness.

In section 2, we review various approaches to regex matching based on non-deterministic finite state automata (NFAs), deterministic automata (DFAs), and "tagged" automata (TNFAs and TDFAs) that track when capture groups start and end. Section 3 presents our approach to efficiently extracting full parse trees for regular expressions with capture groups. Stacks are used to simulate backtracking, coroutines are used to explore different parses pseudo-concurrently, and "histories" log the successful starts and ends of capture groups. Section 4 presents a proof of correctness of the algorithm that adopts the simple

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	Prefer more matched
`*` operator	`a*`	$0-\infty$	
`+` operator	`a+`	$1-\infty$	
`??` operator	`a??`	0 or 1	Prefer less matched
`*?` operator	`a*?`	$0-\infty$	
`+?` 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. Summary of regular expression elements

backtracking algorithm as the baselines for correctness. We show that the new algorithm is faithful to the backtracking one. Section 5 presents the implementation and performance benchmarks, and section 6 briefly concludes.

2 MOTIVATION AND RELATED WORK

Regular expressions originated with Kleene in the 1950s [Sipser (2005)]. They make for scalable and efficient lightweight parsers [Karttunen et al. (1996)]. While there is no shortage of books discussing the usage of regular expressions, the implementation side of regular expression has not been so lucky. As Cox (2007, 2009, 2010) argues, innovations have repeatedly been ignored and later reinvented, not least because the publication medium of source code without an accompanying article was chosen.

The best known algorithms, including the one presented here, run in $O(\min(nm, 2^m + n))$, where n is the length of the string to be matched against and m is the length of the pattern [Sedgewick (1990)]. The input to the algorithm is both the regular expression and the string to match, so let $s = n + m$ be the input size to the algorithm, then the overall run time in s is $O(s^2)$.

A curious aspect of the literature is that many authors perceive regular expression parsing to be a linear problem — linear in the length of the string with a constant for the pattern size. However, there are valid applications to use regular expression matching for large regular expressions.[1] It is not known if there is any algorithm that beats the $O(s^2)$ matching, but in Section 4, we prove a lower bound of $\Theta(s \min(s, |\Sigma|))$, where $|\Sigma|$ is the size of the alphabet.

Regular expressions and capture groups

Table 1 summarizes the key elements of regular expressions. Note that the option (`?`), star (`*`) and plus (`+`) operators are *greedy*, that is, they consume as much input as they can, while their *non-greedy* alternatives (`??`, `*?` and `+?`) attempt to match as few repetitions as possible. In a backtracking implementation, guessing the right path is an important efficiency feature and for all capturing implementation the path taken influences the captured groups.

Capture groups (*i.e.*, patterns enclosed in parentheses) are treated as a single element, thus `(ab)*` captures "`ab`", but not "`aba`". After the match, the capture groups can be extracted: `a(b*)c` will extract "`bbb`" when matched against "`abbbc`", and the empty string when matched against "`ac`".

In POSIX, the regular expression `a((bc+)+)` yields "`bcbccc`" when matched to the string "`abcbccc`" 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. Instead

[1] For example, think of a program that tries to determine the file type of a file. A plausible implementation is to construct one regular expression for each file type. Then, given regular expressions e_k, one for each file type, the regular expression $/(e_1)|(e_2)|...|(e_k)/$ could be used to determine the file type in one pass only.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

we would like all occurrences to be kept and returned in a tree structure: The outer capture group should contain "bcbccc" and the inner matches should yield "bc" and "bccc".

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

Backtracking

Backtracking provides an intuitive and extensible algorithm for determining whether a string matches a regular expression. More importantly, the backtracking algorithm gives an intuitive definition for what the correct submatches are, depending on the greediness operators.

Algorithm 1 is used in some form in many languages, such as Java[2], Python[3], or Perl [Cox (2007)].

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 the greedy repetition a* then
        – x[1:] means removing the first element of the list
        – Greedily try to match inner first
        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 first element of pattern is the non-greedy repetition a*? then
        – Try to match rest first
        if match-bt(string, pattern[1:]) matches then
            return it
        else
            return match-bt(string[1:], pattern)
        end if
    else if ... then
        ...
    end if
end function
```

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 match the pattern `(x*)*y` against the string[4] "x^n", we see that it cannot match, but it takes exponential time doing so.

In this paper, we think of the match returned by the backtracking algorithm as the correct behaviour. It is what most regular expression matchers in the wild do (Java, Perl, ...), even if it does contradict the POSIX definition of a correct match. Its popularity may be due to the advantages of being easy to implement, efficient if back-tracking guesses correctly, and the resulting match being comparatively intuitive, compared to the POSIX-prescribed match.

Memoization

Backtracking makes for easy implementations, but exponential run-time for many patterns. Norvig (1991) 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 $O(\min(nm, 2^m + n))$of the automata-based

[2] `java.util.regex`
[3] The module `re`
[4] x^n means x repeated n times

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

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.[5] It should be noted however that while this approach has been known for some time now and promises exponential speed-up, it is not a standard optimization for backtracking-based regular expression implementations. In the wild, regular expressions are tweaked until they no longer need any backtracking at all, and then memoization leads to a dramatic performance loss.

The backtracking implementation, if it never needs to backtrack, can match input nearly as fast as it can be read from RAM. Practical regex implementations have no room for per-character overhead.

The memoization approach can be extended further to so-called *packrat* parsing [Medeiros et al. (2012)] based on Parsing Expression Grammars (PEG), to obtain $O(nm)$, but the memoization gives a space overhead that is $O(n)$, with a big constant [Ford (2002)] — 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, for example for the grammars of programming languages, but Packrat parsers have a large space overhead that make them infeasible for large inputs such as data analysis.[6]

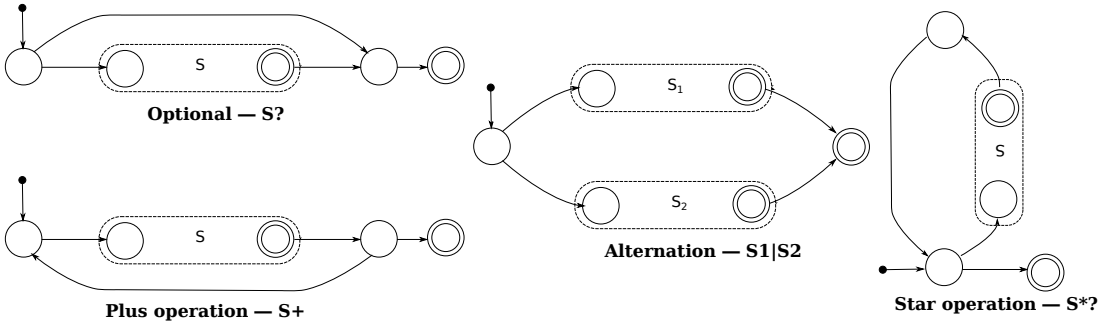


Figure 2. Thompson (1968) construction of the automaton: Descend into the abstract syntax tree of the regular expression and expand the constructs recursively.

Finite-state automata

An alternative with much-improved worst-case bounds to the backtracking approach discussed in Section 2 is to pre-process the regular expression and convert it into an NFA (non-deterministic finite state automaton) using the classical approach by Thompson (1968) shown in Figure 2.

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 $O(\min(nm, 2^m + n))$time.

Dissatisfied with the multiplicative $O(m)$ overhead, we can construct a DFA (deterministic finite state automaton) from the NFA before matching using the power set construction [Sipser (2005)], which has time complexity $\Theta(2^m)$. The idea is to replace all states by the set reachable from it with only ε-transitions — therefore a DFA state represents a set of NFA states. The transitions simulate a step in the original NFA, so they point to another set of states. After compilation, string matching takes $O(n)$ time. This approach is only useful if the regular expression is statically known or small, because constructing the full DFA is exponential in the regular expression size in the best case.

The power set construction simulates every transition possible in the NFA, but that is actually unnecessary: Instead we can intertwine 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 $O(\min(nm, 2^m + n))$. This is the best known result for matching [Cox (2007, 2009, 2010)].

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

[6] Becket and Somogyi (2008): "The Java parser generated by Pappy requires up to 400 bytes of memory for every byte of input."

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Tagged finite state automata

The algorithms we have seen 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 while we traverse the automaton.

NFA interpretation can be understood as being performed by competing coroutines running in lockstep with each other, consuming at each step exactly one character of the input string. This implies that some form of instructions are executed on a transition, so it is possible to add other instructions that allow us to store the capture groups. This is the idea of a *tagged finite state automaton* (TNFA) [Laurikari (2000)], which attaches general *tags* to transitions that modify the coroutine's memory. We can store the position of the start and end of each match in the memory of the coroutine, whenever we encounter a transition that corresponds to the respective start or end of the capture group.

To simplify the algorithm, we will assume that it contains at least one character so that the reading step is executed at least once.[7]

Side effects, such as storing the current location, make coroutines using different routes to the same state differ in meaning. Consider the regular expression `(a)|(.)`. Reading the string "`a`": depending on the path chosen, our capture groups will contain "`a`" in the first or second capture group. Since we consider the match returned by the backtracking algorithm as the only correct one, the correct match stores a in the first capture group, and nothing in the second. This requires us to define a unique order for expanding coroutines on each state, so that we can avoid this ambiguity. This is done by giving a *negative priority* [Laurikari (2000)] to one of the transitions or require one to consume a character, whenever we have an out-degree of two.[8]

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.

Closely related to priorities is greediness control: consider again `((.*?),(\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, matching this regular expression against input "`TomLehrer,1;AlanTuring,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 "`TomLehrer,1;AlanTuring`". This is, of course, not what we expect. Non-greediness, here, ensures that we obtain "`TomLehrer`", then "`AlanTuring`" as the matches of the first capture group.

Implementing this with backtracking is trivial, but in order to keep the coroutines in lockstep, we need to order the NFA states in the DFA state so that the coroutines travelling the left path are always scheduled before the coroutines on the right path so that the scheduling corresponds to trying the left side before the right side in backtracking.

To complicate things further, we want coroutines that have travelled further to have higher priority than the ones that stayed further behind — in backtracking this would be depth-first-search. Take for example `(a??)(a??)` on the string "`a`": without the depth-first-search, we'd capture "`a`" in the first capture group, where it should be in the second group.

Automata-based extraction of parse trees

Memoization is a powerful tool to achieve theoretically fast parsers, but they have a space-overhead in order of the input instead of the parse tree size, which slows down the parser on actual hardware. The other approach to parsing — finite state automata — offers a remedy. These approaches, including the one we present in this paper, use TNFAs to achieve both speed and low memory usage. The approaches differ in: what parse tree is produced, whether greediness control is supported, how the parse tree is stored, and how the NFA can be compiled into a DFA (see table 2).

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 4), but has lower space consumption. The mapping in terms of cells for each group is easy, but costs a factor m space overhead.

Another problem is greediness. Kearns, Dubé, and Nielsen cannot guarantee the greediness of the winning parse. Grathwohl's contribution allows Dubé's algorithm to run with greedy parses. Our priorities

[7] The empty string can be modelled as containing only the '\0' character.

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

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Author	Stores	Automaton	Parse time	Space overhead
Kearns (1991)	Path choices	NFA	$O(nm)$	$O(nm)$
Dubé and Feeley (2000)	Capture groups in linked list	NFA	$O(nm)$	$O(nm^2)$
Nielsen and Henglein (2011)	Bit-coded trees of capture groups		$O(nm\log(m))$	
Grathwohl et al. (2013)				
Laurikari (2000)	Capture group in array	DFA	$O(n+2^m)$	$O(dm)$
This paper	Capture group in array of linked lists	lazy DFA	$O(\min(nm, 2^m+n))$	$O(nd+m)$

Table 2. 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 (2000) does not produce parse trees.

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. This might seem like a negligible improvement, but certain scenarios only open up with this, such as the possibility to efficiently parse a string larger than the memory of a single machine.

3 EFFICIENT REGEX MATCHING WITH CAPTURE GROUPS

Given our basic algorithm 1 for matching regular expressions with backtracking, we will now present an approach that is less wasteful. The algorithm we present is a specific case of the tagged non-deterministic finite state automaton (TNFA) matching algorithm for regular expressions with added logging of the start and end of capture groups. Stacks are used to simulate backtracking, coroutines are used to explore different parses pseudo-concurrently, and "histories" log the successful starts and ends of capture groups.

We first show how to generate a TNFA from the AST of the regular expression by extending Thompson's standard construction. After showing how to simulate backtracking with TNFA interpretation, we present the algorithm for matching capture groups using histories of commit tags. This algorithm is $O(\min(nm, 2^m+n)\,u(m))$, where $u(m)$ describes the amortized cost of logging a single opening or closing of a capture group. A persistent treap allows us to achieve $u(m) = \log m$, and using the data structure described by Driscoll et al. (1989), we can improve this to $u(m) = 1$. This gives us $O(\min(nm, 2^m+n))$run time for the complete algorithm, which is the best known run time for NFA algorithms.

We follow with an example illustrating the executing of our algorithm with the interpretation of commit tags. We also consider practical problems such as caching current results, just-in-time compilation, and compact memory usage.

Conceptually, our approach consists of four stages:

1. Parse the regular expression string into an AST
2. Transform the AST to a TNFA
3. Transform the TNFA to a TDFA
4. Compactify the TDFA

In reality, things are a little more involved, since the transformation to TDFA is lazy, and the compactification only happens after no lazy compilation has occurred in a while. Also compactification can be undone if needed. Since the essence of the algorithm consists in steps 2 and 3, we start with them and we will discuss steps 1 and 4 as part of the implementation in Section 5.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

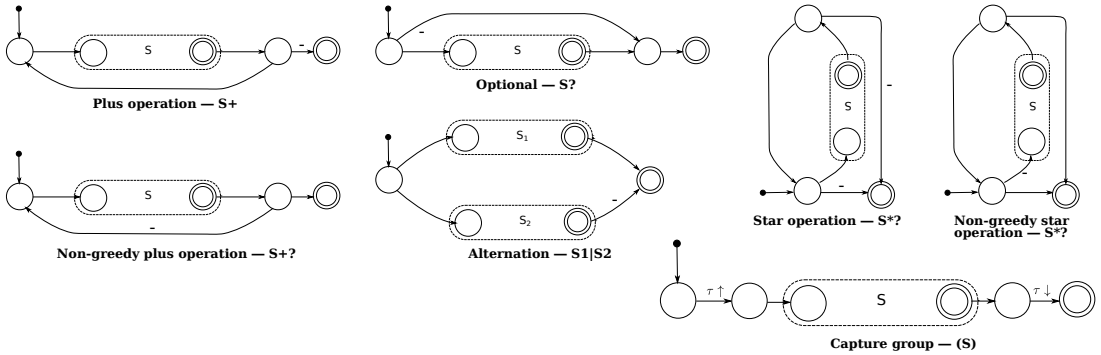


Figure 3. Modified Thompson (1968) construction of the automaton: Descend into the abstract syntax tree of the regular expression and expand the constructs recursively. In comparison to the simple construction in figure 2, the forward transitions from the top state in the star operators should be surprising, but they are necessary if S has a prioritized path that captures the empty string: We cannot return to the start state, because we expanded it already, but we can proceed anyway.

Transforming the AST to a TNFA

We transform the abstract syntax tree (AST) of the regular expression into a TNFA by extending Thompson's NFA construction (figure 3). The additions are needed for greediness control and capture groups. In the diagram, "–" stands for low priority. Tagged transitions mark the beginning or end of capture groups or control the prioritization. $\tau_n\uparrow$ is the opening tag for capture group n, likewise, $\tau_1\downarrow$ is the closing tag for capture group n.

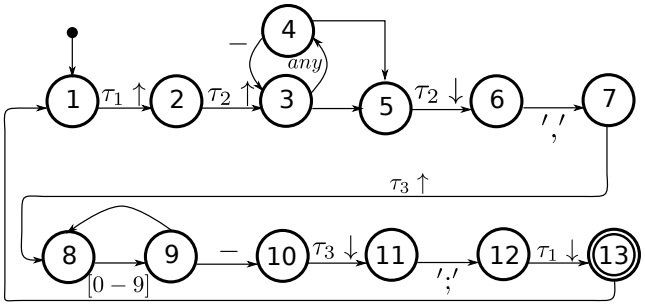


Figure 4. Automaton for `((.*?),(\d+);)+`

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 4 shows how this plays out in our running example (figure 4), which contains the expression `.*?`. An automaton for the expression `.` is constructed. The expression `.` is modeled as just two nodes labeled 3 and 4, and a transition labeled "any" between them.[9]

2. We add prioritized transitions to model repetition. In our example, repetition 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 to leave the repeating expression rather than stay 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.

[9]NB: this is not minimized; a semantically equivalent automaton with just a single node with a *any* transition to itself is smaller.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Algorithm 2 Tagged transition execution.

```
— Returns a list of coroutines that consumed the character
function run_tagged(coroutines, char)
    — coroutines is a list of coroutines in order as returned here.
    — char is a character
    Put all coroutines on the low stack
    Initialize empty buffer stack
    Initialize empty list R                          — the returned list of coroutines
    while the stacks are not both empty do
        if high is not empty then
            Pop c from the high stack
        else
            Pop c from the low stack
            Flush buffer into R                      — thus reversing the order
        end if
        for all transitions that consume char from c to state s do
            push s to buffer                         — remember the state for the next turn
        end for
        for all ε-transitions from c to state s with tag t do
            if no coroutine in state s exists in coroutines then
                copy the coroutine c to c'
                INTERPRET(t, c')
            end if
            if transition is normal priority then
                add a coroutine r in state s with memory m to the high stack
            else
                add a coroutine r in state s with memory m to the low stack
            end if
        end for
    end while
    return R
end function
```

Simulating backtracking for regular expressions

Algorithm 2 illustrates how backtracking can be simulated with TNFA interpretation, which is an original contribution of this paper.

In order to simulate backtracking correctly, we need all paths reachable from the state after the prioritized transition to be processed first, even if they are interrupted by the need to consume another character. This prioritization is achieved by using a buffer that reverses the order of high-priority runs.

Without the buffer, the routines are scheduled in the order in which the states are seen. This gives wrong results, if the state further behind can catch up to one further down, for example in `(a*?)(a*?)`, the second group should contain the match.

Logging capture groups in a TNFA

We now lay out the storage required by the coroutines and the interpretation of the tags that we introduced in algorithm 2. 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, and the transition to leave a capture group is tagged by 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 recording 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.

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, while the *rest*, all other nodes, are immutable. Because the other nodes are immutable, they may be shared between histories. This is an application of the *flyweight* design pattern [Gamma et al. (1995)], 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.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

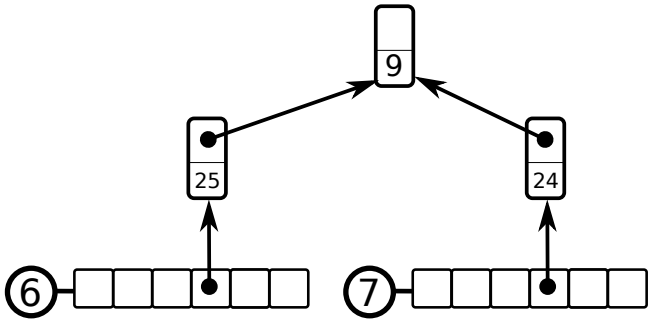


Figure 5. Histories are cells of singly linked lists, where only the first (here bottom-most) cell can be edited. This is a view of the automaton in Figure 4 after the string "`TomLehrer,1;AlanTuring`," has been consumed. Only the cell for the closing of the second capture group is shown.

DFA states are denoted by a capital letter, *e.g.*, Q, and contain multiple coroutines. For example,

$$Q = [(q_1, (([0],[12]),([9,1],[10,2]),([\,],[\,]))),\\ (q_2, (([0],[\,]),([1],[2]),([1],[2])))]$$

means that the current DFA state has one coroutine in NFA state q_1 with histories $(([0],[12]),([9,1],[10,2]))$ and another coroutine in NFA state q_2 with the histories $(([0],[\,]),([1],[2]),([1],[2]))$. Note that histories can be shared across coroutines if they have the same matches. The order of the coroutines is relevant, and a DFA state is thus a list of NFA states.

Histories are linked lists, where each node stores a position in the input text. (See figure 5.) The head is mutable, and 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 have occurred at the positions $x_1, \ldots, x_m$.

Coroutines 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 coroutine 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, and h_2 is the history of the closings of the first capture group, and so on.

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

Take for example the regular expression `(..)+` matching pairs of characters, on the input string "$\overset{0}{\texttt{a}}\overset{1}{\texttt{b}}\overset{2}{\texttt{c}}\overset{3}{\texttt{d}}$". The history array of the finishing coroutine 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, *i.e.*, positions 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.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Algorithm 3 Interpretation of the tags.

```
− Update the coroutine, interpreting the tag
function interpret(t, c)
    − t is a tag, c is a coroutine
    if t is open tag of group i then
        − Do not commit, in case we pass edge again
        set(index + 1, c.histories[i].left)
    end if
    if t is close tag of group i then
        set(index, c.histories[i].right)
        commit c.histories[i].left and c.histories[i + 1].right
    end if
end function
```

Here we provide the *interpret* function seen in algorithm 2.

The states are given to 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 to ensure that the correct order of coroutines is produced. If our procedure is consistently used, the prioritization will lead to a correct match.

Note that the ordering of coroutines inside of DFA states is relevant. In Figure 4, after reading only one comma as an input, state 7 can be reached from two coroutines: either from the coroutine in state 3, via 4, or from the coroutine in state 6.

The two coroutines are 'racing' to capture state 7. Since in the starting state, the coroutine of state 6 is listed first, it 'wins the race' for state 7, and 'captures it'. Thus, the new coroutine of state 7 is a fork of the coroutine of state 6, not 3. This matters, since 6 and 3 may disagree about their histories.

The overall run time of algorithm 2 depends heavily on the forking of coroutines being efficient: In the worst case, it takes $\Theta(mT_{fork}(m))$ 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 $O(\min(nm, 2^m + n)m)$ 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 coroutine and the forked one. This is easily achieved replacing the array by a persistent data structure [Driscoll et al. (1989)] to hold the array. A persistent treap, sorted by array index, has all necessary properties.[10] With $T_{fork} = O(\log m)$, the overall runtime is $O(\min(nm, 2^m + n) \log m)$. With the persistent data structure described by Driscoll et al. (1989) we obtain an amortized $O(1)$ update cost for the claimed $O(\min(nm, 2^m + n))$overall runtime.

Example

We now demonstrate an example of the execution of algorithm 2 with the function *interpret* as defined in algorithm 3.

Consider the automaton in figure 4 is in the DFA starting state

$$Q = [(q_1, (([\,], [\,]), ([\,], [\,]), ([\,], [\,])))]$$

This is the case after initialization.

The algorithm uses a *high* stack and a *low* stack, corresponding to the two priorities.

We pretend for clarity that instructions are executed directly after they are encountered. In practice, the algorithm collects them and executes them after the *run* call to enable further optimizations and the storage of the instructions.

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

1. Fill the *low* stack with the coroutine in Q. Now, $low = [(q_1, (([\,], [\,]), ([\,], [\,]), ([\,], [\,])))]$, where the first element is the head of the stack. *high* is empty.

[10] Clojure [Hickey (2008)] 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`. See chapter 3 of Karper (2014) for a more extensive discussion of suitable data structures.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

2. Initialize *buffer* to an empty stack. This stack is used to reverse the order of states discovered while following high priority transitions.
3. Initialize the DFA state under construction: $R = [\,]$,
4. Coroutine $(q_1, (([\,],[\,]),([\,],[\,]),([\,],[\,])))$ is popped from the *high* stack.
5. We iterate over all available transitions in the NFA transition graph, and find only $q_1 \to q_2$, which contains the tag $\tau_1 \uparrow$.
   (a) We need to change the opening tag of the first capture group, so we call set$(1, histories[0].left)$.
   (b) We push q_2 with the new memory to the *high* stack.
6. Coroutine $(q_2, (([1],[\,]),([\,],[\,]),([\,],[\,])))$ is popped from the *high* stack.
7. We see $q_2 \to q_3$, which contains the tag $\tau_2 \uparrow$.
   (a) We need to change the opening tag of the first capture group, so we call set$(1, histories[1].left)$.
   (b) We push q_2 with the new memory to the *high* stack.
8. Coroutine $(q_3, (([1],[\,]),([1],[\,]),([\,],[\,])))$ is popped from the stack.
9. We see $q_3 \to q_4$ with negative priority, we push q_4 on the *low* stack.
10. We see $q_3 \to q_5$ and push q_5 on the *high* stack.
11. Coroutine $(q_5, (([1],[\,]),([1],[\,]),([\,],[\,])))$ is popped from the *high* stack. It contains $\tau_2 \downarrow$
    (a) We need to change the opening tag of the first capture group, so we call set$(0, histories[1].right)$.
    (b) We push q_6 with the new memory to the *high* stack.
12. Coroutine $(q_6, (([1],[\,]),([1],[0]),([\,],[\,])))$ is popped from the *high* stack.
13. We see $q_6 \to q_7$ consuming ",". We do not push anything on the *high* or *low* stack, but put $(q_7, (([1],[\,]),([1],[0]),([\,],[\,])))$ in the *buffer*.
14. Our *high* stack is empty.
    (a) We flush the *buffer* into the DFA state R: $R = [(q_7, (([1],[\,]),([1],[0]),([\,],[\,])))]$, $buffer = [\,]$
15. Coroutine $(q_4, (([1],[\,]),([1],[\,]),([\,],[\,])))$ is popped from the *low* stack.
16. We see $q_4 \to q_3$ consuming any character. We put $(q_3, (([1],[\,]),([1],[\,]),([\,],[\,])))$ on the *buffer* stack.
17. No transitions remain.
    (a) We flush the *buffer*: $R = [(q_7, (([1],[\,]),([1],[0]),([\,],[\,]))), (q_3, (([1],[\,]),([1],[\,]),([\,],[\,])))]$, $buffer = [\,]$
18. R is returned.

Some of the histories contain pairs of the kind $([1],[0])$, which would be a group that starts after it began. This means that no character was matched, as can easily be checked by comparing it to $/((.*?),(\texttt{nd}+))+/$ on the string ",".

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Conversion to tagged DFA

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 are in a DFA state that we have already encountered and that we can create a new connection to. Equality of TDFA states cannot be the same as equality between DFA states — the equality of the contained NFA states does not care about the order in which they are visited and furthermore it does not respect that two expansions might have different executed instructions. This has been addressed by Laurikari (2000) by finding equivalent or *mappable* TDFA states. A mapping is a bijection of two states that needs to be found at compilation time.

The idea of adding other instructions to the coroutines in the automaton that is the finite state machine (be it NFA or DFA) is not new. The first implementation using this to the authors' knowledge is Pike (1987) 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.

The correct handling of greediness (not of non-greediness) was implemented by Kuklewicz (2007) for the Haskell implementation[11] of Laurikari's algorithm. This too was only published in source code, to a great loss for the academic community.

Cox calls Laurikari's TDFA a reinvention of Pike's algorithm, but while that is in part true, Laurikari introduces the mapping step described in algorithm 4. This leads Laurikari's algorithm to contain fewer states and one would hope that this would lead to a better run-time than Google's RE2[12], which is based on Pike's algorithm.

This is not confirmed by the benchmarks by Sulzmann and Lu (2012), 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.

Compared to RE2, our algorithm does not 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[13]. Further its algorithm to simulate the backtracking is simpler. However our algorithm does not require a separate pass for match detection and match extraction, which opens different scenarios — the reason we can avoid this is that we are able to collect the instructions and incorporate them into the lazy DFA state compilation. Our algorithm adds the mapping phase from Laurikari, which allows us to find DFA states that can be made equivalent by some additional writes.

4 PROOFS

We now sketch proofs of the claimed properties, first and foremost the correctness of algorithm 2 under the interpretation of algorithm 3.

Correctness

The correctness of the algorithm follows by induction over the construction: If the correct coroutine 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 become, 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 a 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, $bt(e,s)$, where regex e is applied to input string s:

[11] Or free interpretation, since Laurikari leaves the matching strategy open.

[12] `https://code.google.com/p/re2/`

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

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

$$
\begin{aligned}
bt(a|b,s) &= bt(a,s) \quad | \quad bt(b,s) \\
bt(r*,s) &= bt(rr*|\varepsilon,s) \\
bt(r*?,s) &= bt(\varepsilon|rr*,s) \\
bt(r?,s) &= bt(r|\varepsilon,s) \\
bt(r??,s) &= bt(\varepsilon|r,s) \\
bt(r+,s) &= bt(rr*,s) \\
bt(r+?,s) &= bt(rr*?,s) \\
bt(ab,s) &= bt(a,s) + bt(b,rest) \\
bt(Group(i,r),s) &= [WriteOpen(i)] + bt(r,s) + [WriteClose(i)] \\
bt(ab,ss') &= bt(a,s) + bt(b,s')
\end{aligned}
$$

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

The correct parse is found if and only if after reading the whole string,

1. the coroutine in the end state has consumed all characters of the string (and only those) in order, and
2. there is such no coroutine that has taken "later" low-priority edges. This corresponds to the depth-first search of backtracking.

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.

There is the possibilities of cycles, so that the depth-first solution would loop. This can be seen for example in the regular expression `(a*?)*`, where the preferred route in the graph is actually to capture an empty repetition of `a`. We tweak the Thompson construction for this scenario, by giving a path to the logically following state after the automaton with the same priority as from the start node for the star-operator, because it is the only automaton where the start state competes with a complete run through the pattern.

Now the parses are analogous for our procedure and *bt* (see Figure 6).

Execution time

The main structure of any NFA-based matching algorithm is the nesting of two loops: The outer loop iterates over the n characters of the string, and the inner loop expands at most m states. The expansion produces $O(1)$ updates per state expanded, as Thompson's construction gives a constant out-degree for each state. The update cost of every coroutine is $O(1)$. This gives a total run time of $O(\min(nm, 2^m + n))$.

Lower bound for time

There is no known tight[14] 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 $R = [\mathtt{ax}_1]*|[\mathtt{ax}_2]*|\ldots|[\mathtt{ax}_m]*$. Note that $|S| = \Theta n$ and $|R| = \Theta(\min(m, |\Sigma|))$. Let further `match` be a valid regular expression matching algorithm, then $\mathtt{match}(S,R)$ is equivalent to finding $a^n x_i \overset{?}{\in} \{a^n x_1, \ldots, a^n x_m\}$. There is no particular order to $\{a^n x_1, \ldots, a^n x_m\}$, so the lower bound for finding this is $\Theta(|S|\,|R|)$. □

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

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

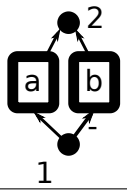
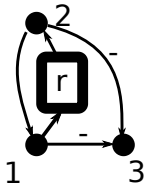
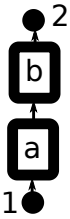
$bt(a\|b,s)$: 1. Check a 2. Check b	 1. Check $1 \to a \to 2$ 2. Check $1 \to b \to 2$
$bt(r*,s)$: 1. Check r 2. Check ε	 1. Check $1 \to r \to 2 \to 1 \to 3$ 2. Check $1 \to 3$
⋮	⋮
$bt(ab,s)$: 1. Check a 2. Check b on rest 3. Concatenate the updates	 1. Run through a consuming some characters 2. Run 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

Figure 6. Backtracking compared to the generated TNFA.

5 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.

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 keep track of all input ranges that occur in the regular expression when is parsed. We then split the ranges until no two of them intersect. After this step, input ranges are never created again. By performing this step early in the pipeline we establish the invariant that it is impossible to ever come across intersecting input ranges.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

To give us a chance to ever reach a state that is already in the transition table, we check, after executing algorithm 2, whether there is a known DFA state that is *mappable* to the output of algorithm 2. If algorithm 2 has 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 4 shows how we can find a mappable state, and the needed instructions. The run time of algorithm 4 is $O(m)$, where m is the size of the input NFA.

Algorithm 4 $findMapping(Q)$: Find a state that Q is mappable to, in order to keep the number of states created bounded by the length of the regular expression.

```
1:  function FINDMAPPING(Q)
Require: Q = [(q_i, h_i)]_{i=1...n} is a DFA state.
Ensure: A state Q' that Q is mappable to.
2:  The ordered instructions m that reorder the memory locations of Q to Q' and do not interfere with
    each other.
3:      for Q' that contains the same NFA states as Q, in the same order do
4:          − Invariant: For each history H there is at most one H' so that H ← 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...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) ↦ 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
19: end function
```

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 algorithm 2. If the resulting state is mappable, we map. More formally, we can see this in algorithm 5. Here, algorithm 5 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.

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, rather than as an array of structs, as recommended by the Intel optimization handbook (Intel Corporation, 2013, 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 *i*th 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 is not 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

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

Algorithm 5 interpret(input): Interpretation and lazy compilation of the NFA.

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


PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

as an integer. However, as described in the algorithm, a DFA state is really a list of coroutines. If we need to lazily compile another DFA state, all of the coroutines need to be examined.

We adopted the following compromise: 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 insertion 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 regular expression matching usually needs only a handful of DFA states, and thus, compactification can be performed early, and only seldom needs to be undone.

Intertwining of the pipeline stages

Lazily compiling the DFA when matching a string allows us to avoid compiling states that might never be needed. This allows us to avoid the full power set construction [Sipser (2005)], which has time complexity of $O(2^m)$, where m is the size of the NFA.

Parsing the regular expression syntax

Parsing the regular expression into an abstract syntax tree is a detail that can easily be overlooked. Since the algorithm for matching is already very fast, preliminary experiments showed that parsing the regular expression, even simple ones, can take up a major portion (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 7 (note the log plot).

Benchmark

All benchmarks were obtained using Google's caliper[15], 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. Backtracking implementations are exponential in their run-time, and so we see in Figure 8 (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 3.

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

Table 3. 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 has been crafted to prevent any backtracking, java.util.regex outperforms our approach by more than factor 2, as seen in Table 4 — 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[16], which, as also seen in Table 4, performs on par with our approach.

[15] `https://code.google.com/p/caliper/`
[16] `http://jparsec.codehaus.org`

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

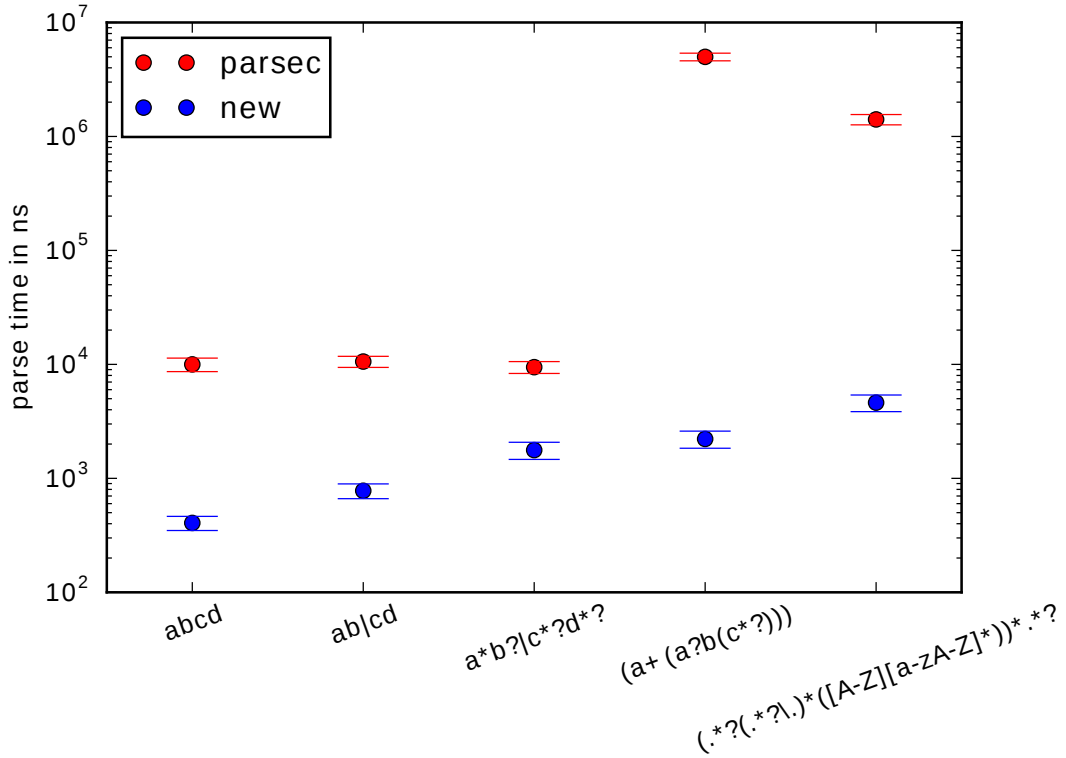


Figure 7. Comparison of parsec and our hand-written top-down parser to parse the regular expression syntax. Since the measurements are very noisy, the median with the MAD (median absolute deviation) are plotted.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

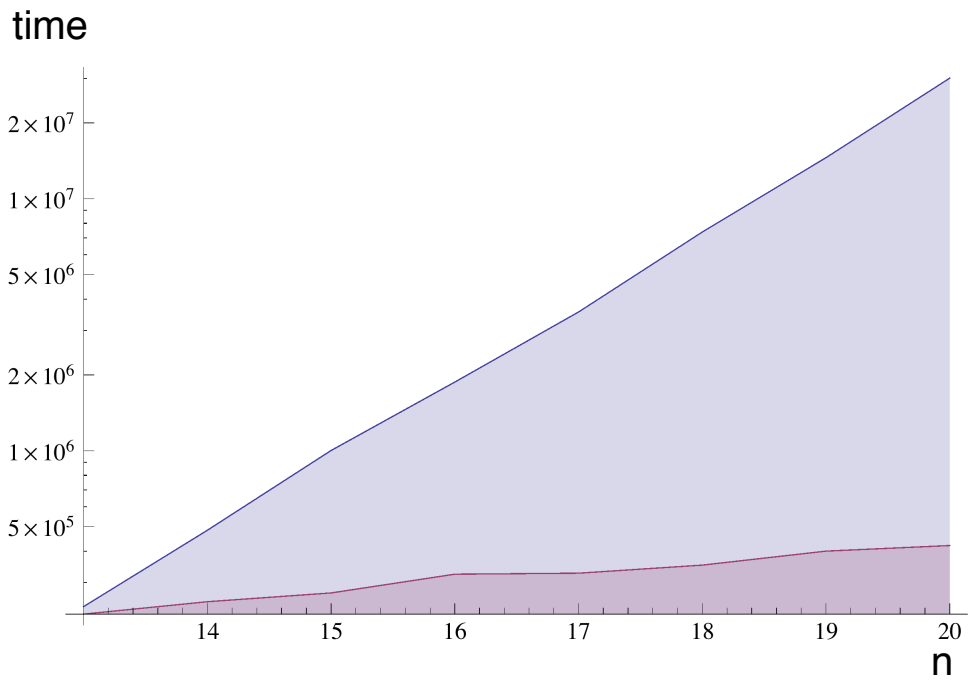


Figure 8. Time in nanoseconds for matching $a?^n a^n$ against input a^n. Bottom (purple) line is our approach, top (blue) line is java.util.regex.

Note that because java.util.regex achieves its backtracking 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 4. Matching regular expression `((a+b)+c)+` against input $(a^{200}bc)^{2000}$, where a^{200} denotes 200 times character 'a'. Time in microseconds.

Finally, a more realistic example, neither chosen to favor backtracking nor to avoid it, extracts all class names, with their package names, from the project sources itself. As seen in Table 5, 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.

All Java source code and benchmarks are available under a free license on Github [Schwarz and Karper (2014)].[17] In his dissertation, Schwarz (2014) reports on a potential application of this implementation to large scale clone detection. In his MSc thesis Karper (2014) also presents an implementation in Python.

[17] `https://github.com/nes1983/tree-regex`

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015

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

Table 5. Runtimes, in microseconds, for finding all java class names in all .java files in the project itself. The regular expression used is $/(.*?([\texttt{a}-\texttt{z}]+\texttt{n}.)*([\texttt{A}-\texttt{Z}][\texttt{a}-\texttt{zA}-\texttt{Z}]*))*.*?/$.

6 CONCLUSION

Regular expressions 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. This algorithm is able to provide the same semantics as backtracking, but without an exponential worst case.

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 backtracking solutions if no backtracking ever happens, exponentially outperforms backtracking approaches for pathological input, and in a realistic scenario outperforms backtracking by 40%, even though our approach produces the full parse tree, and the backtracking implementation does not.

7 ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank Jan Kurš for his thorough review of this paper.

REFERENCES

Becket, R. and Somogyi, Z. (2008). DCGs + Memoing = Packrat parsing, but is it worth it? In *Practical Aspects of Declarative Languages*, volume LNCS 4902, pages 182–196. Springer.

Cox, R. (2007). Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). `http://swtch.com/~rsc/regexp/regexp1.html`.

Cox, R. (2009). Regular expression matching: the virtual machine approach. `http://swtch.com/~rsc/regexp/regexp2.html`.

Cox, R. (2010). Regular expression matching in the wild. `http://swtch.com/~rsc/regexp/regexp3.html`.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. (1989). Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124.

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

Ford, B. (2002). 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. ACM.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass.

Grathwohl, N. B. B., Henglein, F., Nielsen, L., and Rasmussen, U. T. (2013). Two-Pass greedy regular expression parsing. In Konstantinidis, S., editor, *Implementation and Application of Automata*, volume 7982 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg.

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

Intel Corporation (2013). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, 248966-028 edition.

Karper, A. (2014). Efficient regular expressions that produce parse trees. Masters thesis, University of Bern.

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

Kearns, S. M. (1991). Extending regular expressions with context operators and parse extraction. *Softw: Pract. Exper.*, 21(8):787–804.

Kuklewicz, C. (2007). Regular expressions/bounded space proposal. `https://wiki.haskell.org/Regular_expressions/Bounded_space_proposal`.

Laurikari, V. (2000). 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.

Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. (2012). From regexes to parsing expression grammars. *Science of Computer Programming*.

Nielsen, L. and Henglein, F. (2011). Bit-coded regular expression parsing. In Dediu, A.-H., Inenaga, S., and Martín-Vide, C., editors, *Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 402–413. Springer Berlin Heidelberg.

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

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

Schwarz, N. (2014). *Scaleable Code Clone Detection*. PhD thesis, University of Bern.

Schwarz, N. and Karper, A. (2014). O(n m) regular expression parsing library that produces parse trees. `http://dx.doi.org/10.5281/zenodo.10861`.

Sedgewick, R. (1990). *Algorithms in C (paperback)*. Addison-Wesley Professional, 1 edition.

Sipser, M. (2005). *Introduction to the Theory of Computation*. Course Technology, 2 edition.

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

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

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.1248v1 | CC-BY 4.0 Open Access | rec: 20 Jul 2015, publ: 20 Jul 2015