EFFICIENT REGEXP IMPLEMENTATION FOR FINDING PARSE TREES

Abstract. Regular expressions naturally and intuitively define ASTs 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 TDFA matching, all paths through the NFA are walked simultaneously, as if in different threads, where inside each thread, 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 thread, using the fly-weight pattern, a history of the all groups. Thus, we log enough information during parsing to build up the complete AST at the end of parsing.

1. Introduction

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 will only give a single instance of values: `(([a-zA-Z ]+),(\d+);)+` might describe a dataset of ASCII names with their numeric label. Matching the regular expression on `‘‘Tom Lehrer,1;Alan Turing,2;’’` will confirm that the list is well formed, but the match will only contain `‘‘Tom Lehrer’’` for the second capture group and `‘‘1’’` for the third. That is, the full parse tree is:

The amortized run time of our approach is $O(m\,n)$, where m is the length of the regular expression and n is the length of the parsed string. This is asymptotically as fast as the best matchers that don't extract parse trees.

1.1. Motivation.

This is the age of big data, and the first step of processing big data is often to parse strings. As an example, consider log files.. What makes data huge is typically repetition. As Jacobs[?] 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.[?]

Regular expression's parsing abilities have evoked Meiners to declare that for intrusion detection, "fast and scalable RE matching is now a core network security issue." [?]

For example, Arasu et al.[?] 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.

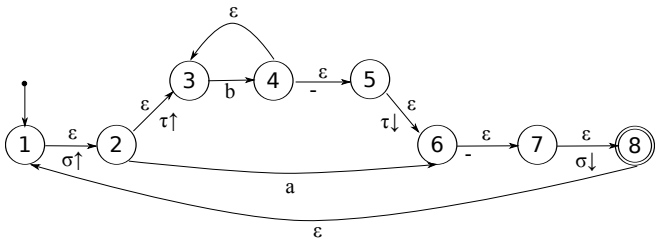

FIGURE 2.1. Automaton for ((b+)|a)+. The ϵ-transition labeled $\sigma\uparrow$ is tagged with an an opening commit of capture group σ. Likewise, $\tau\downarrow$ is tagged with a closing commit of capture group τ. The edge from node 4 to 5 is labeled with a minus, meaning that the edge is of low priority. The start state is 1. The only end state is 8.

2. Algorithm

The first step of our algorithm is to transform the regular expression into the graph of an NFA. Let us recall what NFAs and DFAs are. A DFA is a state machine that will walk over the 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.

NFAs differ from DFAs in that, when several transitions are allowed from one state to another, an NFA will magically guess the correct one. figure **??** shows an example of an NFA's transition graph. For the moment, let us discuss regular expression matching on NFAs. Assuming that the NFA just magicly knows the right transition lets us focus on the important things, greediness and capture groups.

To control greediness, or discern capture groups, our approach adds 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. Let us look at greediness control, and then capture groups, in turn.

When the regular expression ((b+)b?|a)+ is matched against input bb, capture group 2 is expected to match "bb", not "b", because the matching of b+ is greedy. In the NFA, we model this by connecting b+ to its following state via an ϵ-transitions of *low priority*, SEE EXAMPLE. More generally, the NFA will prefer to follow normal edges over those of low priority. Rather than formalize this notion of preference, we come back to priorized edges 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. To keep the transition graphs readable, we will visually distinguish opening and closing commits. However, they work exactly the same way. The NFA keeps a history for every transition with a commit. Each time the transition is walked, the current position is added to its history. It is easy to see that the entire match tree can be reconstructed from the list of histories.

The details of how the NFA is constructed can easily be determined from REFERENCE TO CODE.

2.1. **DFAs.** Our above definition of regular expression assumes a machine that guesses the correct transition through magic. To implement regular expression matching without supernatural intervention, we lazily transform the NFA to a DFA.

A useful metaphor for regular expression matching is that of threads [**?**]. Whenever we aren't sure which transition to take, we "fork" a thread for every option that we have. This way, when the input is over, there must be at least one thread that guessed correctly at all times. We use the word "thread" here to guide intuition only. Our approach is not parallel.

The key insight is that we keep all "threads" in lock-step. To achieve this, we must be very specific about what constitutes the state of a thread. Since every thread effectively simulates a different NFA, the state inside of a thread contains exactly two items: the NFA state it simulates, and the history for every tag. Now, the following would be correct, although slow, implementation of a non-magic NFA interpreter: whenever an input character is read, we can iterate over all threads, kill the ones that have no legal transition for the input character, and fork more threads as needed.

Trouble starts when we want to fork a thread for an NFA state that is already running. Not is an explosion of threads bad for performance, it would also lead to ambiguity: if the two threads disagree on the histories, which one is correct?

The following algorithm takes as an input a set of threads, an NFA transition graph, and an input character, and returns the set of threads running after the input character has been read. It makes sure that if there could be two threads with the same NFA state, the one that follows greedy matching will survive.

3. The TDFA powerset construction algorithm

Our algorithm is a modification of Laurikari's algorithm [**?**], which is itself a modified powerset construction algorithm [**?**, p. 55]. However instead of compiling the TNFA to a TDFA before matching, we create the TDFA lazily as we match the string. For clarity, and because the original description contains a few errors, we outline his algorithm here, omitting his proof for correctness and termination. Our description adapts the thread metaphor of [**?**], where each (T)NFA state describes the state of a single thread in a virtual machine and the (T)DFA state is a collection of such threads.

Our algorithm runs two phases for each character read. First it creates a new TDFA state from the current state and tries to map this state and the instructions on the transition to a known state(as in [**?**]), then it executes the instructions found.

The memory model differs from other regular expression engines in that locations are not stored in a fixed number of cells. Instead each cell is the head of a singly linked list that describes the history of the value. This allows us to extract all capture groups matching a subpattern after the matching phase ended. Whenever threads are forked, the histories of the matches sofar are shared, but we ensure that further modifications will be thread-bound.

We need the following instructions:

- $n \leftarrow \mathbf{p}$**:** Stores the current position in the input string into the head of memory location n.
- $n \leftarrow \mathbf{p} + 1$**:** Stores the position after the current one in the input string into the head of memory location n.

$n \leftarrow m$**:** Replaces memory location m with a copy of head and history of memory location n.

$c \uparrow (n)$**,:** $c \downarrow (n)$ Puts the current state of memory location n into the tail of the same memory location. The two instructions don't differ in their effect, but in the order in which they are executed.

Algorithm 1 TDFA from TNFA

```
Name: interpret(input)
Input:  input is a sequence of characters
Output: a tree of matching capture groups for the regex
1. Compute all states after start
start ←
epsilonClosure(startState, ε) and execute its instructions
Initialize TDFA with start as start state.
Q ← start
2. Consume string
for all indices pos and values a of input

    Try to find a mappable state Q' with instructions on the transition in TDFA
    if Q' was found
         execute instructions
         jump back to start of for loop.
    else
         Q', instructions ← epsilonClosure(Q, a)
```

Definition 1. For $I \subset \mathbb{N}$ and $n \in \mathbb{N}$, DFA state$\{(q_i, (a_{ij})_{j=1..n}) : i \in I\}$ is *mappable* to another DFA state $\{(q_i, (b_{ij})_{j=1..n}) : i \in I\}$ iff there is a bijection ν such that $\forall i \in I,\ j = 1..n : (a_{ij}) = (\nu(b_{ij}))$. Bijection ν is called the *mapping*.
For example, $\{(q_0, [4, 3]),\ (q_1, [2, 1]),\ (q_3, [2, 3])\}$ is mappable to $\{(q_0, [0, -2]),\ (q_1, [2, 1]),\ (q_3, [2, -2])\}$ using mapping $1 \mapsto 1,\ 3 \mapsto -2,\ 4 \mapsto 0$.

Let us first consider the caase where N has no ϵ arrows and no tags. Then we can construct the new transition function δ' as follows:

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$

Now we need to consider the ϵ arrows, some of which contain tags. We use the following notation:

For any state R of M we define $E(R) = \bigcup_{(r,t) \in R} e(q, t)$ to be a set of pairs. The first elements of the pairs are the states that can be reached from R by going along only ϵ-arrows, including the members of R themselves. The second element of each pair denotes the memory locations of all tags for their corresponding NFA state.

$$\delta^\star(R, a) = \{q \in Q : q \in E(\delta(r, a)) \text{ for some } r \in R\}$$

Example 2. Computing the next state from the DFA state $(1[2, 5, 3, 4], 2[6, 5, 3, 4], 3[2, 0, 3, 1], 4[2, 0, 3, 1], 5[2, 0, 3,$ which is the state after reading a single "b" input, in figure **??**:

(1) Since only 3 has a b transition and no tags are written in consuming transitions, $high = (4[2, 0, 3, 1])$ after the initializing for loop

Algorithm 2 Compute the follow-up state for DFA state Q

```
Name:   epsilonClosure(q, x)
Input:  DFA state Q, character to consume chr
Output: R is the new DFA state, instructions are the instructions that need to be execu
R ← {}
Initialize empty stack low
Initialize empty stack high
1. Consuming the input
For every transition t from any of q ∈
Q to q' that consumes chr

    if t has high priority, push q' to high, else to low
2. Following ε-transitions
While high and low are not both empty

    pop q' with store [l_1, ..., l_n] from high if possible, else from low
    if q' ∈ R jump to start of while loop
    add q' to R
    for all transitions t from q' to q''
         2.1. continue if q'' has been seen:
         if q'' ∈
         R jump to start of for loop to avoid conflict.
         2.2 Create new NFA state and instructions.
         if t has the opening tag τ_i ↑
             Copy l as newHistories
             Create new history cell h and put it into position_open(i) of newHistories
             put h ← p +
             1 into instructions to store new start of the group.
         else if t has the closing tag τ_i ↓
             Copy l as newHistories
             Create new history cell h and put it into position_open(i) of newHistories
             Create new history cell h' and put it into position_close(i) of newHistories
             put h' ←
             l[position_close(i)] into instructions
             put h' ← p into instructions
             put c ↑ (h) and c ↓
             (h') into instructions to commit them into the immutable history of this thr
             end if
         else
             Copy l as newHistories
         end if
         2.4. Push according to priority of transition:
         if t has low priority, push q'' with newHistories to low, else to high
    end for
end while
return R and instructions
```

(2) The state $4[2,0,3,1]$ is taken from the stack and added to R, which is now $\{4[2,0,3,1]\}$
   (a) The transition $4 \to 5$ might be analysed first, but is added to low, because for greedy consumption in the + operator, the backward way is prefered. The memory is not changed here.
   (b) Then the transition $4 \to 3$ is analysed and 3 is added to $high$ without changes in the memory.
(3) Now the state $3[2,0,3,1]$ is taken from the stack and added to R, but since it has no ϵ exits, no further steps are taken from here
(4) The state $4[2,0,3,1]$ leads to $5[2,0,3,1]$
(5) The state $5[2,0,3,1]$ leads to $6[2,0,3,7]$, because τ is closed here.
(6) The state $6[2,0,3,7]$ leads to $7[2,0,3,7]$.
(7) The state $7[2,0,3,7]$ leads to $8[2,8,3,7]$, closing the σ tag.
(8) The state $8[2,8,3,7]$ leads to $1[2,8,3,7]$
(9) The state $1[2,8,3,7]$ leads to $2[9,8,3,7]$, reopening σ.
(10) The state $2[9,8,3,7]$ would lead to $3[9,8,10,7]$, it is however not added to R, because there is a different 3 state (namely $3[4,1,5,3]$) in R
(11) Now, a mapping is searched by comparing the new DFA state $(1[2,8,3,7], 2[9,8,3,7], 3[2,0,3,1], 4[2,0,3,1$ to any old states, in our case $(1[2,5,3,4], 2[6,5,3,4], 3[2,0,3,1], 4[2,0,3,1], 5[2,0,3,1], 6[2,0,3,4], 7[2,0,3,$
   (a) Now the first NFA state has the memory $[2,8,3,7]$ in the new state and $[2,5,3,4]$ in the old state. This adds the following constraints:
      (i) Memory locations 2 and 3 is mapped to themselves
      (ii) Memory location 8 is mapped to 5
      (iii) Memory location 7 is mapped to 4
   (b) The second NFA state has the memory $[9,8,3,7]$ and $[6,5,3,4]$ respectively.
      (i) Memory location 9 is mapped to 6
      (ii) Memory location 8 is mapped to 5, which conforms to the given constraint from earlier
      (iii) Memory location 7 is mapped to 4
   (c) The third, forth, and fifth NFA state has identical memory, which introduces the constraints for 0 and 1
   (d) The sixth and seventh NFA state conforms to the mapping of 7 to 4
   (e) The eighth NFA state conforms to the mapping of 8 to 5.
   (f) This means that the new state is isomorphic to an existing state and the mapping has been explicitly constructed. The newly introduced locations are assigned the position of character read.

4. TDFA with commits

The overall idea is simple: whenever the TDFA is reading in a run of characters that belong to a capture group, we push the previous one into our history and the work on current one. To implement this, we introduce "commits" into the TDFA. On the level on the TNFA the commits correspond to tags at the end of capture groups. Since capture groups are nested, they form a tree. We will call this three the match tree and will prove that we can reconstruct it. In it, we store all runs that this capture group matches. The subnodes of a hierarchy nodes correspond to the submatches of its capture group.We will shortly discuss how to construct a TDFA that includes instructions that *commit* submatches into the hierarchy nodes..

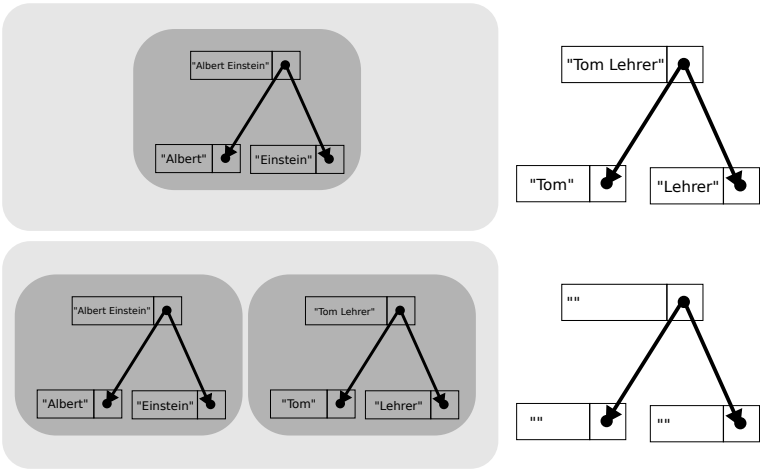

FIGURE 4.1. Matching `(\w+) (\w+)`

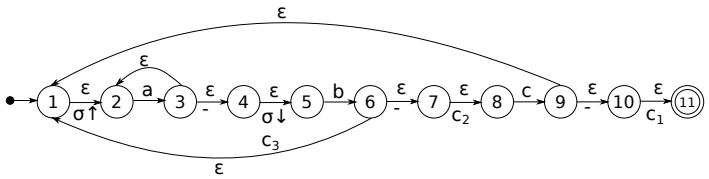

FIGURE 4.2. NFA about to be transformed.

Once we have the instructions, during interpretation of the TDFA, whenever we encounter an input character that closes a capture group, the TDFA will *commit* that capture group. To commit a substring into a hierarchy node has the following semantics.

A hierarchy node, as a tree node, has a corresponding subtree. The data in this subtree represents precisely the subtree of the AST that is currently being matched. Upon commit of the hierarchy node, it constructs this AST subtree, and stores it as a previous match. Iteratively, this produces all matches of all capture groups.

Example 3. In the following, the regex `(((a+)b)+c)+` will be converted to a DFA lazily while matching the string "abaababc", the resulting DFA can be seen in **??**:

(1) The starting state is $(1[-1,-2], 2[0,-2])$ and the memory location 0 is initialized to hold the index 0.
(2) Now the letter a is read:
    (a) $2[0,-2]$ leads to $3[0,-2]$
    (b) $3[0,-2]$ leads to $2[0,-2]$
    (c) $3[0,-2]$ leads to $4[0,-2]$
    (d) $4[0,-2]$ leads to $5[0,1]$, with the instruction to write the index to 1 and commit the memory locations $(0,1)$ on level 3 afterwards. The hierarchical memory is now $c_1 = nil, c_2 = nil, c_3 = (0,1)$, where $(0,1)$ are the indeces, not the memory locations.
(3) Next the letter b is read:

   (a) $5[0,1]$ leads to $6[0,1]$
   (b) $6[0,1]$ leads to $1[0,1]$
   (c) $1[0,1]$ leads to $2[2,1]$, with the instruction to write the current index to 2.
   (d) $6[0,1]$ leads to $7[0,1]$
   (e) $7[0,1]$ leads to $8[0,1]$, with the instruction to commit to the second level. The hierarchical memory is now $c_1 = nil, c_2 = (0,1), c_3 = (0,1)$.
4. Next the letter a is read:
   (a) $2[2,1]$ leads to $3[2,1]$
   (b) $3[2,1]$ leads to $2[2,1]$
   (c) $3[2,1]$ leads to $4[2,1]$
   (d) $4[2,1]$ leads to $5[2,3]$, with the instruction to write the current index to memory location 3 and commit the third level: $c_1 = nil, c_2 = (0,1), c_3 = (2,3)$
   (e) We find, that this state is mappable to the previous state $(2[0,-2], 3[0,-2], 4[0,-2], 5[0,1])$ with the mapping $2 \to 0, 1 \to -2$.
5. Next the letter a is read:
   (a) $2[0,-2]$ leads to $3[0,-2]$
   (b) $3[0,-2]$ leads to $2[0,-2]$
   (c) $3[0,-2]$ leads to $4[0,-2]$
   (d) $4[0,-2]$ leads to $5[0,1]$, with the instruction to write the current index to memory location 1 and commit the third level: $c_1 = nil, c_2 = (0,1), c_3 = (2,4)$.
6. Next the letter b is read, but the transition is already known:
   (a) We write the current index into memory location 2 and commit the second level. Now the match $(2,4)$ is compared against the previous match $(0,1)$, but the length of the new match is bigger, therefore the new match overwrites the old one. $c_1 = nil, c_2 = (2,4), c_3 = (2,4)$.
7. Next the letter a is read, but the transition is already known:
   (a) We write the current index into memory location -2 (because of the mapping) and commit $(2,-2)$ to the third level. $c_1 = nil, c_2 = (2,4), c_3 = (5,6)$.
8. Next the letter b is read, but the transition is already known:
   (a) We write the current index into memory location 2 and commit to the second level. The match $(5,6)$ is compared against the previous match $(2,4)$, but the length of the old match is bigger, therefore the old match is kept. $c_1 = nil, c_2 = (2,4), c_3 = (5,6)$.
9. Next the letter c is read:
   (a) $8[0,1]$ leads to $9[0,1]$
   (b) $9[0,1]$ leads to $1[0,1]$
   (c) $1[0,1]$ leads to $2[2,1]$, with the instruction to write the current index to memory location 2.
   (d) $9[0,1]$ leads to $10[0,1]$)
   (e) $10[0,1]$ leads to $11[0,1]$, with the instruction to commit to the first level. $c_1 = (2,4), c_2 = (2,4), c_3 = (5,6)$.
10. Finally the end of the string is read.
   (a) We are in a finishing state $11[0,1]$, therefore the match succeeds. The current commit of the top layer is returned: $c_1 = (2,4)$.

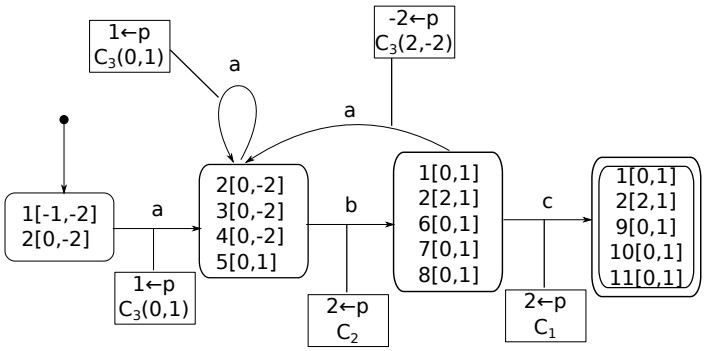

FIGURE 4.3. The DFA of `(?:(?:(a+)b)+c)+` matching "abaababc"

5. Benchmark

6. Related work

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 [**?**].

This paper is no exception. The authors of this paper had set out to implement Laurikari's TDFA algorithm [**?**], 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 [**?**]. 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 [**?**]. It seems ironic that this well-versed author published his influential work on his website. Although the joke may be on Academia's side.

When the taciturn practioners acknowledge each other's work, we can't help but disagree almost universally with the characterizations they produce. Sulzmann and Lu [**?**] 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 Pike's published only in code algorithm [**?**], which is Thompson's NFA with submatch tracking. This seems unfair in that Laurikari's allows for far more aggressive reusing of old states than what Thompson allows. This should lead to Laurikari's TDFA having fewer states, and therefore better performance, than even Google's RE2, which uses Pike's algorithm. This is not confirmed by the benchmarks by Sulzmann and Lu [**?**], 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 scarce literature is to call regular expression matching linear. As Sedgewick points out correctly [**?**], 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 requires to assume m to be fix, which we cannot bring ourselves to justify. It may well be true that, at present, m tends to be small. But that is a natural consequence of the algorithms not scaling very well with m. If they did, that would allow for fast feature extracting from text. 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. We cannot rule out that a linear algorithm exists, in fact, we hope for it. To insist that regular expression matching is done in linear time is to insist that the optimal algorithm has already been found; that is probably not true.

Sulzmann and Lu add to the table a new matching strategy that yields good practical performance, although the theoretical bounds are considerably worse than the state of the art, at $O(n^2m)$ [**?**].

NOTES:

Pseudocode
Beispiel(e) '(?:(?:(a+)b)+c)+'
Algorithm:
Definition: