A DFA for submatch extraction

Esteban Castro Borsani

January 2020

Abstract

Finite Automata is commonly used to efficiently match a Regular Expression (RE) to a given text input. There are RE engines for submatch extraction based on Non-deterministic Finite Automata (NFA). These algorithms usually return a single match for each submatch, instead of the history of submatches (full parse tree). An NFA can be converted to a Deterministic Finite Automata (DFA) to improve the runtime matching performance. This document describes an algorithm based on DFA that extracts full parse trees from text.

Introduction

Regular Expressions (RE) are used to describe patterns to match over a given text. Most regex engines (such as Perl's PCRE) allow to specify capture groups for submatch extraction, word boundaries (i.e.: `\b`), and character properties (i.e.: `\w`). There are well known algorithms that take a RE and construct a Non-deterministic Finite Automata (NFA). An NFA can be converted to a Deterministic Finite Automata (DFA) to improve performance. The algorithm I describe constructs a ε-NFA, converts the ε-NFA to NFA, and then converts the NFA to DFA. The matching takes O(N*M) time in the length of the input text and the RE, if the RE contains capture groups, otherwise it take O(N) time in the length of the input text. It takes O(N*M) space to construct the full parse tree, however the resulting tree is a suffix-tree of text boundaries, which usually takes little space. The caveat is that the DFA construction may take exponential time, however compilation of the RE into a DFA is done only once. Improvements to the DFA construction runtime and submatch extraction space might be implemented without substantial modifications, since the proposed algorithm is a classical DFA simulation that implicitly simulates the original NFA.

Definitions

ε-NFA : $(Q, \Sigma, \Delta, q0, F)$ where Q is a finite set of states, Σ is a finite set of input symbols called the alphabet, Δ is a transition function $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$, q0 is an initial state $q0 \in Q$, and F is a set of final/accepting states $F \subseteq Q$.

ε-closure : set of states reachable from q by following ε-transitions in the transition function Δ.

https://dx.doi.org/10.6084/m9.figshare.11691723 |

NFA : $(Q, \sum, \Delta, q0, F)$ where Q is a finite set of states, $\sum$ is a finite set of input symbols called the alphabet, Δ is a transition function $\Delta : Q \times \sum \rightarrow P(Q)$, q0 is an initial state $q0 \in Q$, and F is a set of final/accepting states $F \subseteq Q$.

DFA : $(Q, \sum, \delta, q0, F)$ where Q is a finite set of states, $\sum$ is a finite set of input symbols called the alphabet, δ is a transition function $\delta : Q \times \sum \rightarrow Q$, q0 is an initial state $q0 \in Q$, and F is a set of final/accepting states $F \subseteq Q$.

RE to ε-NFA

Conversion from RE to ε-NFA is done using *Thomposon's Construction*[0]. All ε-transitions must be kept, including the capture groups. The construction requires parsing the RE, generating the ε-NFA states, and adding the edges/transitions to them. This can be done as follows: 1. linearize the RE such as each letter in the expression *E* is unique in the expression *E'*, 2. linearize the capture groups as well such as each group is unique, 3. transform *E'* to *E"* in Reverse Polish Notation (RPN) using the *Shunting-yard* algorithm, 4. convert *E"* to ε-NFA using *Thomposon's Construction*. This is well described by Russ Cox's *"Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)"*[1].

ε-NFA to NFA

Conversion from ε-NFA to NFA is done by removing the ε-transitions. The algorithm is similar to traversing the ε-NFA doing a Breadth-First Search, and computing the ε-closure of each state. Each state is processed once, hence the time complexity is linear. The set of NFA (not ε-NFA) transitions *T* are stored along the set of capture groups and other important transitions *Z* (such as word boundaries, text start, text end, etc), while traversing the ε-NFA. *Algorithm 1* shows an implementation of ε-transitions removal.

The algorithm defines *teClosure0(q)* as the set of states that are reachable from *q* by following ε-transitions in the transition function *teClosure,* in addition to capture groups and word-boundaries states between *q* and the next reachable state. The function *n0* denote the (special) start state of the ε-NFA.

NFA to DFA

Conversion from NFA to DFA is done using *Powerset Construction*[2]. This takes exponential time in the number of NFA states. The resulting DFA can be minimized using *Hopcroft's DFA minimization*[3].

The DFA simulation requires checking the ε-closure for each transition, hence the *Powerset Construction* needs to keep track of them. *Algorithm 2* shows the classical *Powerset Construction*.

The function *delta* denote the closure of *q* that accepts the character *c*.

DFA Simulation / Submatches Extraction

The DFA simulation is the classical simulation as long as *Z* is empty. Otherwise, the NFA is implicitly simulated by following the *T* transitions. On each DFA transitions, the current set of NFA transitions is pruned by the current DFA states, and accepted NFA transitions move to the next state. When the simulation completes, we are left with one or more accepted states, the first accepted state is the winner branch and the one we care about (assuming left-to-right PCRE submatching). *Algorithm 5* shows an implementation of the DFA simulation and the implicit NFA simulation.

If an NFA transition contains one or more Z transitions, they are evaluated. Transitions of capture groups are added to the prefix-tree. Transitions of matching types like word-boundary are matched to the current input character, the NFA state moves to the next transition if the character is accepted.

The resulting submatches are constructed by traversing the prefix-tree from the leaf / last-capture within the accepted branch, to the root. The set of submatches contains the full history of submatches, not just the last submatch. *Algorithm 3* shows an implementation of submatches construction.

The subMatch function (*Algorithm 4*) is inspired by *Thompson's NFA simulation*. The asymptotic time complexity is O(N*M) in the length of the text input and the number of NFA states. The space complexity is O(N*M) as well, since the prefix-tree is constructed from all of the paths the NFA took. The correctness of this algorithm lays in the fact that we construct an exhaustive prefix-tree. Thus, the resulting tree must contain the branch that matched the string.

A patological case that takes linear space is **`(?:(.))*`**, this will capture each character of the input string. A patological case that takes O(N * M) space is **`(?:(.))*(?:(.)(.)(.))*`**, this will capture each character 4 times as there are 4 possible branches.

Implementation

There is a full implementation written in the Nim programming language called nregex[4]. It already shows promising results, as the classical DFA is faster than PCRE in several cases, and only a few times slower when the RE contains group captures. However, Nim provides powerful macros that will allow to generate optimized code at compile time and remove most of the current bottlenecks. Hopefully, the code repository will contain benchmarks soon enough.

Algorithm 1: ε-transitions removal

```
input:  ε-NFA result of Thompson's construction
output: NFA, set of transitions T, and set of transitions Z

proc teClosure(result, state, z):
  copy z to z'
  if state is group or word-boundary state:
    add state to z'
  if state is character-property state:
    add state to z'
    add {state, z} to result
    return
  if state is char state:
    add {state, z} to result
    return
  for each next state of state as s:
    teClosure(result, state, z')

proc teClosure0(state):
  initialize result array
  initialize z array
  for each next state of state as s:
    teClosure(result, state, z)
  return result

proc eRemoval(eNFA):
  initialize T transitions
  initialize Z transitions
  copy eNFA to NFA
  q0 = n0({NFA})
  initialize Qw queue
  add q0 to Qw
  initialize Q set
  add q0 to Q
  while Qw is not empty:
    remove first element qa of Qw
    q = teClosure0(qa)
    for each state of q as (qb, z):
      add z to (qa, qb) of Z
      add qb to qa of T
      if qb is not in Q:
        add qb to Q
        add qb to Qw
    replace qa next states by q
```

```
  return NFA, T, Z
```

Algorithm 2: Powerset Construction

input: NFA result of eRemoval
output: table T of DFA transitions, and (special) start state of the NFA

```
proc powersetConstruction(NFA):
  q0 = n0({NFA})
  initialize Qw queue
  add {q0} to Qw
  initialize Q set
  add {q} to Q
  while Qw is not empty:
    remove q from Qw
    for each character of alphabet as c:
      t = delta(q, c)
      T[q, c] = t
      if t is not in Q:
        add t to Q and to Qw
  return T, q0
```

Algorithm 3: Construct Submatches

```
proc constructSubmatches(capture):
  initialize S array
  while capture is not root:
    if S[capture.number] is empty:
      add (-2, -2) to S[capture.number]
    if S[capture.number][last index] [0] is not -2:
      add (-2, -2) to S[capture.number]
    if S[capture.number][last index] [1] is -2:
      S[capture.number][last index] [1] = capture.bound – 1
    else:
      S[capture.number][last index] [0] =  capture.bound
    capture = capture.parent
  for each group of S as g:
    reverse g
  return S
```

Algorithm 4: Submatch

input: array of previous NFA transitions, empty temporary array, transitions T and Z result of eRemoval, DFA closure t, current character index, current character, and previous character
output: input submatchesA contains the current NFA transitions

```
proc subMatch(submatchesA, submatchesB, T, Z, t, charIndex, char, prevChar):
 for each item of submatchesA as (state, capture):
  for each next state of T[state] as stateB:
   if stateB is not in t:
    continue
   if (state, stateB) is not in Z:
    add (stateB, capture) to submatchesB
    continue
   matched = true
   captureX = capture
   for each transition of Z[state, stateB] as z:
    if z is a group:
     captureX = Capture(parent: captureX, bound: charIndex, number: z.number)
    if z is a word-boundary:
     matched = check prevChar and char form a word-boundary
    if matches is false:
     break
   if matched is true:
    add (stateB, captureX) to submatchesB
 swap  submatchesA by submatchesB
 clear submatchesB
```

Algorithm 5: DFA simulation

input: text to match, DFA result of the powerset construction, T and Z transitions result of eRemoval
output: whether the regex has matched, and the set of matches

```
proc match(text, DFA, T, Z):
  T, q0 = DFA
  initialize submatchesA array
  add (q0, root capture) to submatchesA
  initialize submatchesB array
  charIndex = 0
  prevChar = '\0'
  q = {q0}
  for each character of text as c:
    if (q, c) is not in T:
      return false, empty matches
    t = T[q, c]
    if Z is not empty:
      subMatch(submatchesA, submatchesB, T, Z, t, charIndex, c, prevChar)
    q = t
    increment charIndex
    prevChar = c
  if end state is not in q:
    return false, empty matches
  if Z is empty:
    return true, empty matches
  subMatch(submatchesA, submatchesB, T, Z, {end state}, charIndex, '\0', prevChar)
  if  submatchesA is empty:
    return false, empty matches
  state, capture = submatchesA[0]
  return true, constructSubmatches(capture)
```

Conclusion

Regular expression matching and submatches extraction can be achieved using a DFA and implicitly simulating the original NFA. The algorithm described in this document provides a general solution to this problem, there is no need to handle special edge cases. There are no caveats compared to a classical DFA when the RE does not contain capture groups or assertions such as word-boundary. The algorithm supports any feature an NFA executed by *Thompson's simulation* supports, such as generating a full parse tree containing the history of all the extracted submatches.

References

[0]: Ken Thompson (Jun 1968). "Programming Techniques: Regular expression search algorithm". Communications of the ACM. 11 (6): 419–422. doi:10.1145/363347.363387

[1]: Russ Cox *"Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...).* URL: https://swtch.com/~rsc/regexp/regexp1.html

[2]: Rabin, M. O.; Scott, D. (1959). "Finite automata and their decision problems". IBM Journal of Research and Development. 3 (2): 114–125. doi:10.1147/rd.32.0114. ISSN 0018-8646

[3]: Hopcroft, John (1971), "An n log n algorithm for minimizing states in a finite automaton", Theory of machines and computations (Proc. Internat. Sympos., Technion, Haifa, 1971), New York: Academic Press, pp. 189–196, MR 0403320.

[4]: The nregex implementation. URL: https://github.com/nitely/nregex