TDFA – Fast Submatch Extraction in Regular Expressions

Angelo Borsotti
angelo.borsotti@mail.polimi.it

Ulya Trofimovich
skvadrik@gmail.com

2021

Abstract

In this paper we revisit the lookahead TDFA algorithm for submatch extraction in regular expressions. We provide a detailed description of the algorithm with pseudocode and examples, covering a few important practical optimizations and extensions. We benchmark the algorithm against other submatch extraction algorithms and show that it is very fast in practice. Our research is based on two independent implementations: an open-source lexer generator RE2C and an experimental Java library.

Introduction

This paper does not present a new submatch extraction algorithm — instead, we provide an in-depth description of the algorithm based on *Tagged Deterministic Finite Automata* (TDFA). The paper is primarily targeted at readers who want to implement TDFA, but find the previous papers difficult to follow or lacking important details. Here is a brief history of TDFA development. In 2000 Laurikari published the original paper [Lau00]. In 2007 Kuklewicz implemented the algorithm as a Haskell library [Regex-TDFA] with POSIX longest-match disambiguation. In 2016 Trofimovich came up with the lookahead optimization [Tro17], implemented the algorithm in the open-source lexer generator RE2C [RE2C] and formalized Kuklewicz disambiguation algorithm. In 2017 Borsotti implemented TDFA as a Java library (this wasn't published), and in 2019 Borsotti and Trofimovich described a more efficient POSIX disambiguation algorithm based on the combination of Okui-Suzuki disambiguation and Laurikari TNFA that is also suitable for TDFA construction [BorTro19]. Now that TDFA have been used in real-world software for a while and have shown their practical use, we set forth to provide a better description of the algorithm and share the accumulated insight and experimental data. Before diving into details, we start with a brief discussion of the key concepts.

Regular expressions (RE) are a notation for describing sets of strings known as regular languages, or Type-3 languages in the Chomsky hierarchy [Cho59]. They were first defined by Kleene [Kle51] as string sets constructed from the alphabet symbols and empty word via the application of three basic operations: concatenation, alternative and iteration. Later RE were formalized via the notion of Kleene algebra [Koz94]. The *recognition* problem for RE is to determine if a given string belongs to the language defined by RE. A more complex *parsing* problem additionally requires to construct a derivation of a string in a regular grammar (a parse tree). *Submatch extraction* is similar to parsing, but it does not require a full derivation — only a partial one sufficient to identify parts of the input string that match specific parts of RE. Generic parsing algorithms are too heavyweight for submatch extraction, and a different, faster approach can be used.

The recognition problem for RE can be solved with finite state automata, which come in two flavours: deterministic ones (DFA) and non-determinstic ones (NFA). Both types recognize a string in linear time depending on its length, but DFA are faster in practice, as they follow a single deterministic path, while NFA follow a set of non-deterministic paths simultaneously. NFA can be converted to DFA using the *determinization* procedure, with the caveat that the DFA may be exponentially larger than the NFA. Submatch extraction is straightforward with Thompson's NFA construction which mirrors the structure of RE and allows tracking separate submatch values for different non-deterministic paths. This idea has been described by many authors, including Laurikari [Lau00] and Cox [Cox07], who gives a historic overview. It works well for NFA, but not for DFA where different non-deterministic paths are collapsed into one. To keep track of conflicting submatch values on different paths, DFA are augmented with a fixed amount of memory — *registers*, and operations on transitions that update register values. The most difficult part is determinization, which needs to decide on the number of registers and construct operations. Doing that correctly

and efficiently is the main subject of this paper.

Non-determinism should not be confused with *ambiguity*. Non-determinism is the existense of multiple possibilities during parsing, all of which need to be considered, but some may lead to a deadend. Ambiguity is the existense of multiple different ways to parse the input in principle. Non-determinism is a temporary obstacle that a parsing algorithm has to deal with, and ambiguity is a genuine property of grammar (or sometimes a language). Ambiguity is as much of a problem for NFA as for DFA, as well as any parsing algorithm. To deal with it, one needs a way to choose between ambiguous parses — a *disambiguation policy*. The two most widely known policies are the Perl leftmost-greedy policy and the POSIX longest-match policy. The latter is much harder to implement, although it (arguably) better corresponds to the intuitive notion of the "best match". Some RE engines provide other ways to resolve ambiguity, such as user-defined precedence rules, but these are ad-hoc, error-prone and often difficult to reason about. POSIX disambiguation is out of the scope of this paper; see a separate paper [BorTro19] for an extensive research on the subject. It suffices to say that TDFA can be parameterized over a disambiguation policy, and there is no overhead on disambiguation at runtime — it happens at determinization time.

The choice of a matching algorithm depends on a particular setting. RE engines can be roughly divided in two categories: libraries and lexer generators. Libraries perform interpretation or just-in-time compilation of RE — they face a tradeoff between the time spent on preprocessing and the time spent on matching. Lexer generators, on the other hand, perform ahead-of-time compilation and do not have such a tradeoff. Consequently, libraries use a variety of algorithms ranging from recursive backtracking to NFA, DFA, string searching or some combination of the above; while lexer generators almost always use DFA and may spend considerable time on optimizations in order to emit better code. TDFA can be used in both settings, but their determinisitc nature and low runtime overhead makes them especailly well-suited for lexer generators. We have used two independent TDFA implementations: an open-source lexer generator RE2C that generates C code and an experimental runtime Java library.

The speed of submatch extraction algorithm depends on the form in which submatch results are constructed. The most generic form is a *parse tree* — it retains the full information about the derivation of a string in the grammar defined by RE. A more concise form is a *list of offsets* per each submatch position in RE. An even more lightweight form is a *single offset* per submatch position (usually the last one). We primarily focus on the single-offset form, as it is useful in practice and it permits various optimizations described in section 3. Optionally the algorithm can extract offset lists (the choice is individual for each submatch position, not global). As for parse trees, there is no standard representation, and the runtime performance of the algorithm depends heavily on a particular representation. We use *tagged strings* which have very low overhead and retain enough information to reconstruct a full parse tree. In section **??** we describe *registerless* TDFA that are better suited to full parsing than TDFA with registers. The benchmarks in section 6 cover different representations and RE with different submatch density.

In this paper we focus on TDFA with *lookahead* described by Trofimovich [Tro17], not the original TDFA described by Laurikari [Lau00]. The two types of automata are called TDFA(1) and TDFA(0) by analogy with LR(1) and LR(0) automata: TDFA(1) utilize the lookahead symbol during determinization, which allows them to reduce non-determinism and use fewer registers and register operations than TDFA(0). We benchmark TDFA(1) against TDFA(0), as well as two other types of automata: sta-DFA [Cho19] and DSST [Gra15]. Sta-DFA are very similar to TDFA, but they have register operations in states rather than on transitions (and do not use lookahead). DSST stands for Deterministic Streaming String Transducers; these are more distant relatives to TDFA, better suited to string rewriting and full parsing. We also benchmark TDFA against Ragel [Ragel], which uses simple DFA with ad-hoc user-defined actions.

The rest of the paper is structured as follows. Section 1 formally defines RE and explains how to construct TNFA. Section 2 defines TDFA and describes the determinization procedure. Section **??** describes important practical optimizations. Section 4 gives a complete example from RE to an optimized TDFA, covering TNFA simulation, determinization and different optimization passes. Section **??** describes registerless TDFA. Section 6 provides benchmarks and comparison with other algorithms. Finally, section 7 has conclusions and directions for future work.

To reduce pseudocode verbosity throughout the paper we assume that function arguments are passed by reference and modifications to them are visible to the calling function.

1 TNFA

In this section we define RE and TNFA, show how to construct TNFA from RE and how to match a string.

Definition 1. *Regular expressions (RE) over finite alphabet Σ are:*
1. *Empty RE ϵ, unit RE $a \in \Sigma$ and tag $t \in \mathbb{N}$.*
2. *Alternative $e_1|e_2$, concatenation e_1e_2 and repetition $e_1^{n,m}$ $(0 \le n \le m \le \infty)$ where e_1 and e_2 are RE over Σ.*

Tags are submatch markers that can be placed anywhere in RE. They may be standalone or correspond to capturing parentheses (the correspondence may be nontrivial, e.g. POSIX capturing groups require insertion of hierarchical tags [BorTro19]). Generalized repetition $e^{n,m}$ can be bounded ($m < \infty$) or unbounded ($m = \infty$). Unbounded repetition $e^{0,\infty}$ is the canonical Kleene iteration, shortened as e^*. Bounded repetition is often desugared via concatenation, but in the presence of tags that could change submatch information in RE.

Definition 2. *Tagged Nondeterministic Finite Automaton (TNFA) is a structure $(\Sigma, T, Q, q_0, q_f, \Delta)$, where:*
Σ is a finite set of symbols (alphabet)
$T \subset \mathbb{N}$ is a finite set of tags
Q is a finite set of states with initial state q_0 and final state q_f
Δ is a transition relation that contains transitions of two kinds:
transitions on alphabet symbols (q, a, p) where $q, p \in Q$ and $a \in \Sigma$
optionally tagged ϵ-transitions with priority (q, i, t, p) where $q, p \in Q$, $i \in \mathbb{N}$ and $t \in T \cup \overline{T} \cup \{\epsilon\}$

TNFA is in essence a non-deterministic finite state transducer with input alphabet Σ and output alphabet $\Sigma \cup T \cup \overline{T}$, it rewrites symbolic strings into tagged strings. $\overline{T} = \{-t \mid t \in T\}$ is the set of all negative tags, which represent the absence of match: they appear whenever there is a way to bypass a tagged subexpression in RE, such as alternative or repetition with zero lower bound. Explicit representation of negative match serves a few purposes: it prevents stale submatch values from propagating to subsequent iterations, it spares the need to initialize tags, and it is required by POSIX disambiguation [BorTro19]. Priorities are used for transition ordering during ϵ-closure construction. Algorithm 2 on page 4 shows TNFA construction: it performs top-down structural recursion on RE, passing the final state on recursive descent into subexpressions and using it to connect subautomata. This is similar to Thompson's construction, except that non-essential ϵ-transitions are removed and tagged transitions are added. The resulting automaton mirrors the structure of RE and preserves submatch information and ambiguity in it.

simulation$\big((\Sigma, T, Q, q_0, q_f, \Delta),\ a_1 \dots a_n\big)$

m_0 : vector of offsets of size $|T|$
$C = \{(q_0, m_0)\}$
for $k = \overline{1, n}$ **do**
  $C = epsilon\_closure(C, \Delta, q_f, k)$
  $C = step\_on\_symbol(C, \Delta, a_k)$
  if $C = \emptyset$ **then return** $\varnothing$
$C = epsilon\_closure(C, \Delta, q_f, n)$
if $\exists (q, m)$ in $C \mid q = q_f$ **then return** m
else return $\varnothing$

step_on_symbol$\big(C, \Delta, a\big)$

return $\{(p, m) \mid (q, m)$ in C and $(q, a, p) \in \Delta\}$

epsilon_closure$\big(C, \Delta, q_f, k\big)$

C' : empty sequence of configurations
for (q, m) in C in reverse order **do**
  push (q, m) on stack
while stack is not empty **do**
  pop (q, m) from stack
  append (q, m) to C'
  for each $(q, i, t, p) \in \Delta$ ordered by priority i **do**
    if $t > 0$ **then** $m[t] = k$
    else $m[-t] = \mathbf{n}$
    if configuration with state p is not in C' **then**
      push (p, m) on stack
return $\{(q, m)$ in $C' \mid q = q_f$ or
  $\exists (q, a, \_) \in \Delta$ where $a \in \Sigma\}$

Algorithm 1: TNFA simulation.

Algorithm 1 defines TNFA simulation on a string. It starts with a single configuration (q_0, m_0) consisting of the initial state q_0 and an empty vector of tag values, and loops over the input symbols until all of them are matched or the configuration set becomes empty, indicating match failure. At each step the algorithm constructs ϵ-closure of the current configuration set, updating tag values along the way, and steps on transitions labeled with the current input symbol. Finally, if all symbols have been matched and there is a configuration with the final state q_f, the algorithm terminates successfully and returns the final vector of tag values. Otherwise it returns a failure. The algorithm uses leftmost greedy disambiguation; POSIX disambiguation is more complex and requires a different ϵ-closure algorithm [BorTro19]. Figure 1 in section 4 shows an example of TNFA simulation.

$\underline{tnfa(e, q_f)}$

if $e = \epsilon$ **then**
  return $(\Sigma, \emptyset, \{q_f\}, q_f, q_f, \emptyset)$

else if $e = a \in \Sigma$ **then**
  return $(\Sigma, \emptyset, \{q_0, q_f\}, q_0, q_f, \{(q_0, a, q_f)\})$

else if $e = t \in \mathbb{N}$ **then**
  return $(\Sigma, \{t\}, \{q_0, q_f\}, q_0, q_f, \{(q_0, 1, t, q_f)\})$

else if $e = e_1 \cdot e_2$ **then**
  $(\Sigma, T_2, Q_2, q_2, q_f, \Delta_2) = tnfa(e_2, q_f)$
  $(\Sigma, T_1, Q_1, q_1, q_2, \Delta_1) = tnfa(e_1, q_2)$
  return $(\Sigma, T_1 \cup T_2, Q_1 \cup Q_2, q_1, q_f, \Delta_1 \cup \Delta_2)$

else if $e = e_1 \mid e_2$ **then**
  $(\Sigma, T_2, Q_2, q_2, q_f, \Delta_2) = tnfa(e_2, q_f)$
  $(\Sigma, T_2, Q_2', q_2', q_f, \Delta_2') = ntags(T_2, q_f)$
  $(\Sigma, T_1, Q_1, q_1, q_2', \Delta_1) = tnfa(e_2, q_2')$
  $(\Sigma, T_1, Q_1', q_1', q_2, \Delta_1') = ntags(T_1, q_2)$
  $Q = Q_1 \cup Q_1' \cup Q_2 \cup Q_2' \cup \{q_0\}$
  $\Delta = \Delta_1 \cup \Delta_1' \cup \Delta_2 \cup \Delta_2' \cup \{(q_0, 1, \epsilon, q_1), (q_0, 2, \epsilon, q_1')\}$
  return $(\Sigma, T_1 \cup T_2, Q, q_0, q_f, \Delta)$

else if $e = e_1^{n,m} \mid_{1 < n \leq m \leq \infty}$ **then**
  $(\Sigma, T_1, Q_1, q_2, q_f, \Delta_1) = tnfa(e_1^{n-1,m-1}, q_f)$
  $(\Sigma, T_2, Q_2, q_1, q_2, \Delta_2) = tnfa(e_1, q_2)$
  return $(\Sigma, T_1 \cup T_2, Q_1 \cup Q_2, q_1, q_f, \Delta_1 \cup \Delta_2)$

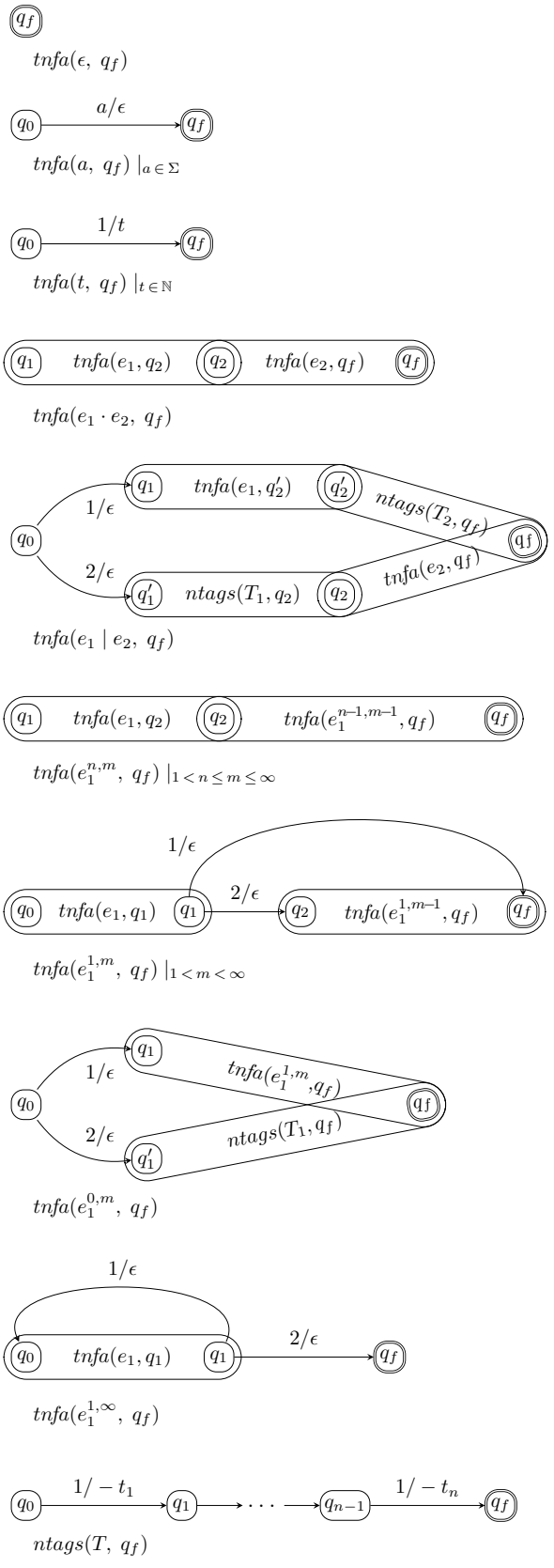
else if $e = e_1^{1,m} \mid_{1 < m < \infty}$ **then**
  if $m = 1$ **then return** $tnfa(e_1, q_f)$
  $(\Sigma, T_1, Q_1, q_1, q_f, \Delta_1) = tnfa(e_1^{1,m-1}, q_f)$
  $(\Sigma, T_2, Q_2, q_0, q_2, \Delta_2) = tnfa(e_1, q_1)$
  $\Delta = \Delta_1 \cup \Delta_2 \cup \{(q_1, 1, \epsilon, q_f), (q_1, 2, \epsilon, q_2)\}$
  return $(\Sigma, T_1 \cup T_2, Q_1 \cup Q_2, q_0, q_f, \Delta)$

else if $e = e_1^{0,m}$ **then**
  $(\Sigma, T_1, Q_1, q_1, q_f, \Delta_1) = tnfa(e_1^{1,m}, q_f)$
  $(\Sigma, T_1, Q_1', q_1', q_f, \Delta_1') = ntags(T_1, q_f)$
  $Q = Q_1 \cup Q_1' \cup \{q_0\}$
  $\Delta = \Delta_1 \cup \Delta_1' \cup \{(q_0, 1, \epsilon, q_1), (q_0, 2, \epsilon, q_1')\}$
  return $(\Sigma, T_1, Q, q_0, q_f, \Delta)$

else if $e = e_1^{1,\infty}$ **then**
  $(\Sigma, T_1, Q_1, q_0, q_1, \Delta_1) = tnfa(e_1, q_1)$
  $Q = Q_1 \cup \{q_f\}$
  $\Delta = \Delta_1 \cup \{(q_1, 1, \epsilon, q_0), (q_1, 2, \epsilon, q_f)\}$
  return $(\Sigma, T_1, Q, q_0, q_f, \Delta)$

$\underline{ntags(T, q_f)}$

  $\{t_i\}_{i=1}^n = T$
  $Q = \{q_i\}_{i=0}^n$ where $q_n = q_f$
  $\Delta = \{(q_{i-1}, 1, -t_i, q_i)\}_{i=1}^n$
  return $(\Sigma, T, Q, q_0, q_f, \Delta)$



Algorithm 2: TNFA construction.

2 TDFA

In this section we define TDFA and show how to convert TNFA to TDFA.

Definition 3. *Tagged Deterministic Finite Automaton (TDFA) is a structure* $(\Sigma, T, S, S_f, s_0, R, r_f, \delta, \varphi)$*, where:*
Σ *is a finite set of symbols (alphabet)*
$T \subset \mathbb{N}$ *is a finite set of tags*
S *is a finite set of states with initial state* s_0 *and a subset of final states* $S_f \subseteq S$
$R \subset \mathbb{N}$ *is a finite set of registers with a vector of final registers* r_f *(one per tag)*
$\delta : S \times \Sigma \to S \times \mathbb{O}^*$ *is a transition function*
$\varphi : S_f \to \mathbb{O}^*$ *is a final function*

where $\mathbb{O}$ *is a set of register operations of the following types:*
set register i *to nil or to the current position:* $i \leftarrow v$*, where* $v \in \{\mathbf{n}, \mathbf{p}\}$
copy register j *to register* i*:* $i \leftarrow j$
copy register j *to register* i *and append history:* $i \leftarrow j \cdot h$*, where* h *is a string over* $\{\mathbf{n}, \mathbf{p}\}$

Compared to an ordinary DFA, TDFA is extended with a set of tags T, a set of registers R with one final register per tag, and register operations that are attributed to transitions and final states (the δ and φ functions). $\mathbb{O}^*$ denotes the set of all sequences of operations over $\mathbb{O}$. Operations can be of three types: set, copy, append. Set operations are used for *single-valued* tags (those represented with a single offset), append operations are used for *multi-valued* tags (those represented with an offset list), and copy operations are used for all tags. The decision which tags are single-valued and which ones are multi-valued is arbitrary and individual for each tag. It may be based on whether the tag is under repetition, but not necessarily. Register values are denoted by special symbols **n** and **p**, which mean *nil* and the *current position* (offset from the beginning of the input string).

Recall the canonical determinization algorithm that is based on powerset construction: NFA is simulated on all possible strings, and the subset of NFA states at each step of the simulation forms a new DFA state, which is either mapped to an existing identical state or added to the growing set of DFA states. Since the number of different subsets of NFA states is finite, determinization eventually terminates. The presence of tags complicates things: it is necessary to track tag values, which depend on the offset that increases at every step. This makes the usual powerset construction impossible: DFA states augmented with tag values are different and cannot be mapped. As a result the set of states grows indefinitely and determinization does not terminate. To address this problem, Laurikari used indirection: instead of storing tag values in TDFA states, he stored value locations — *registers*. As long as two TDFA states have the same registers, the actual values in registers do not matter: they change dynamically at runtime (during TDFA execution), but they do not affect TDFA structure. A similar approach was used by Grathwohl [Gra15], who described it as splitting the information contained in a value into static and dynamic parts. The indirection is not free: it comes at the cost of runtime operations that update register values. But it solves the termination problem, as the required number of registers is finite, unlike the number of possible register values.

From the standpoint of determinization TDFA state is a pair. The first component is a set of configurations (q, r, l) where q is a TNFA state, r is a vector of registers (one per tag) and l is a sequence of tags. Unlike TNFA simulation that updates tag values immediately when it encounters a tagged transition, determinization delays the application of tags until the next step. It records tag sequences along TNFA paths in the ϵ-closure, but instead of applying them to the current transition, it stores them in configurations of the new TDFA state and later applies to the outgoing transitions. This allows filtering tags by the lookahead symbol: confgurations that have no TNFA transitions on the lookahead symbol do not contribute any register operations to TDFA transition on that symbol. The use of the lookahead symbol is what distingushes TDFA(1) from TDFA(0) [Tro17]; it considerably reduces the number of operations and registers. During ϵ-closure construction configurations are extended to four components (q, r, h, l) where h is the sequence of tags inherited from the origin TDFA state and l is the new sequence constructed by the ϵ-closure.

The second component of TDFA state is precedence information. It is needed for ambiguity resolution: if some TNFA state in the ϵ-closure can be reached by different paths, one path must be preferred over the others. This affects submatch extraction, as the paths may have different tags. The form of precedence information depends on the disambiguation policy; we keep the details encapsulated in the *precedence* function, so that algorithm 3 can be adapted to different policies without the need to change its structure. In the case of leftmost greedy policy precedence information is a vector of TNFA states that represents an order on configurations: *step_on_symbol* uses it to construct the initial closure, and *epsilon_closure* performs depth-first search following transitions from left to

right. POSIX policy is more complex, and we do not include pseudocode for it in this paper (see another paper [BorTro19] for a detailed explanation).

Algorithm 3 works as follows. The main function *determinization* starts by allocating initial registers r_0 from 1 to $|T|$ and final registers r_f from $|T|+1$ to $2|T|$. It constructs the initial TDFA state s_0 as the ϵ-closure of the initial configuration $(q_0, r_0, \epsilon, \epsilon)$. The initial state s_0 is added to the set of states S and the algorithm loops over states in S, possibly adding new states on each iteration. For each state s the algorithm explores outgoing transitions on all alphabet symbols. Function *step_on_symbol* follows transitions marked with a given symbol, and function *epsilon_closure* constructs ϵ-closure C, recording tag sequences along each fragment of TNFA path. The set of configurations in the ϵ-closure forms a new TDFA state s'. Function *transition_regops* uses the h-components of configurations in C to construct register operations on transition from s to s'. The same register is allocated for all outgoing transitions with identical operation right-hand-sides, but different tags do not share registers, and vacant registers from other TDFA states are not reused (these rules ensure that there are no artificial dependencies between registers, which makes optimizations easier without the need to construct SSA [SSA]). The new state s' is inserted into the set of states S: function *add_state* first tries to find an identical state in S; if that fails, it looks for a state that can be mapped to s'; if that also fails, s' is added to S. If the new state contains the final TNFA state, it is added to S_f, and the *final_regops* function constructs register operations for the final quasi-transition (called so because it does not consume input characters and gets executed only at the end of match).

TDFA states are considered identical if both components (configuration set and precedence) coincide. States that are not identical, but differ only in registers, can be mapped, provided that there is a bijection between registers. Function *map* attempts to construct such a bijection M: for every tag, and for each pair of configurations it adds the corresponding pair of registers to M. If either of the two registers is already mapped to some other register, bijection cannot be constructed. For single-valued tags mapping ignores configurations that have the tag in the lookahead sequence — every transition out of TDFA state overwrites tag value with a set operation, making the current register values obsolete. For multi-valued tags this optimization is not possible, because append operations do not overwrite previous values. If the mapping has been constructed successfully, *map* updates register operations: for each pair of registers in M it adds a copy operation, unless the left-hand-side is already updated by a set or append operation, in which case it replaces left-hand-side with the register it is mapped to. The operations are topologically sorted (*topological_sort* is defined on page **??**); in the presence of copy and append operations this is necessary to ensure that old register values are used before they are updated. Topological sort ignores trivial cycles such as append operation $i \leftarrow i \cdot h$, but if there are nontrivial cycles the mapping is rejected (handling such cycles requires a temporary register, which makes control flow more complex for optimizations).

After determinization is done, the information in TDFA states is erased — it is no longer needed for TDFA execution. States are just atomic values that can be represented with integer numbers. Disambiguation decisions are embedded in TDFA structure; there is no explicit disambiguation at runtime. The only runtime overhead on submatch extraction is the execution of register operations on transitions. TDFA may have more states than an ordinary DFA for the same RE without tags, because states that can be mapped in a DFA cannot always be mapped in a TDFA. Minimization can reduce the number of states, especially if it is applied after register optimizations that can get rid of many operations and make more states compatible. We focus on optimizations in section 3.

3 Optimizations

determinization$(\Sigma, T, Q, q_0, q_f, \Delta)$

S, S_f : empty sets of states
δ : undefined transition function
φ : undefined final function
$r_0 = \{1, \ldots, |T|\},\ r_f = \{|T|+1, \ldots, 2|T|\},\ R = r_0 \cup r_f$
$C = epsilon\_closure(\{(q_0, r_0, \epsilon, \epsilon)\})$
$P = precedence(C)$
$s_0 = add\_state(S, S_f, r_f, \varphi, C, P, \epsilon)$
for each state $s \in S$ **do**
    V : map from tag and operation RHS to register
    for each symbol $a \in \Sigma$ **do**
        $B = step\_on\_symbol(s, a)$
        $C = epsilon\_closure(B)$
        $O = transition\_regops(C, R, V)$
        $P = precedence(C)$
        $s' = add\_state(S, S_f, r_f, \varphi, C, P, O)$
        $\delta(s, a) = (s', O)$
return TDFA $(\Sigma, T, S, S_f, s_0, R, r_f, \delta, \varphi)$

add_state$(S, S_f, r_f, \varphi, C, P, O)$

$X = \{(q, r, l) \mid (q, r, \_, l) \in C\}$
$s = (X, P)$
if $s \in S$ **then**
    return s
else if $\exists s' \in S$ such that $map(s, s', O)$ **then**
    return s'
else
    add s to S
    if $\exists (q, r, l) \in X$ such that $q = q_f$ **then**
        add s to S_f
        $\varphi(s) = final\_regops(r_f, r, l)$
    return s

map$((X, P), (X', P'), O)$

if X and X' have different subsets of TNFA states
  or different lookahead tags for some TNFA state
  or precedence is different: $P \neq P'$ **then**
    return *false*
M, M' : empty maps from register to register
for each pair $(q, r, l) \in X$ and $(q, r', l) \in X'$ **do**
    for each $t \in T$ **do**
        if $history(l, t) = \epsilon$ or t is a multi-tag **then**
            $i = r[t],\ j = r'[t]$
            if both $M[i], M'[j]$ are undefined **then**
                $M[i] = j,\ M'[j] = i$
            else if $M[i] \neq j$ or $M'[j] \neq i$ **then**
                return *false*
for each operation $i \leftarrow \_$ in O **do**
    replace register i with $M[i]$
    remove pair $(i, M[i])$ from M
for each pair $(j, i) \in M$ where $j \neq i$ **do**
    prepend copy operation $i \leftarrow j$ to O
return *topological_sort*(O)

precedence(C)

return vector $\{q \mid (q, \_, \_, \_)$ in $C\}$

step_on_symbol$((X, P), a)$

B : empty sequence of configurations
for $(q, r, l) \in X$ ordered by q in the order of P **do**
    if $\exists (q, a, p) \in \Delta \mid a \in \Sigma$ **then**
        append (p, r, l, ϵ) to B
return B

epsilon_closure(B)

C : empty sequence of configurations
for (q, r, h, ϵ) in B in reverse order **do**
    push (q, r, h, ϵ) on stack
while stack is not empty **do**
    pop (q, r, h, l) from stack
    append (q, r, h, l) to C
    for each $(q, i, t, p) \in \Delta$ ordered by priority i **do**
        if configuration with state p is not in C **then**
            push (p, r, h, lt) on stack
return $\{(q, r, h, l)$ in $C \mid q = q_f$ or
                $\exists (q, a, \_) \in \Delta$ where $a \in \Sigma\}$

transition_regops(C, R, V)

O : empty list of operations
for each $(q, r, h, l) \in C$ **do**
    for each tag $t \in T$ **do**
        if $h_t = history(h, t) \neq \epsilon$ **then**
            $v = regop\_rhs(r, h_t, t)$
            if $V[t][v]$ is undefined **then**
                $i = max\{R\} + 1$
                $R = R \cup \{i\}$
                $V[t][v] = i$
                append operation $i \leftarrow v$ to O
            $r[t] = V[t][v]$
return O

final_regops(r_f, r, l)

O : empty list of operations
for each tag $t \in T$ **do**
    if $l_t = history(l, t) \neq \epsilon$ **then**
        append operation $r_f[t] \leftarrow regop\_rhs(r, l_t, t)$ to O
return O

regop_rhs(r, h_t, t)

if t is a multi-valued tag **then**
    return $r[t] \cdot h_t$
else
    return the last element of h_t

history(h, t)

switch h **do**
    case ϵ **do return** ϵ
    case $t \cdot h'$ **do return** $\mathbf{p} \cdot history(h')$
    case $-t \cdot h'$ **do return** $\mathbf{n} \cdot history(h')$
    case $\_ \cdot h'$ **do return** $history(h')$

Algorithm 3: Determinization of TNFA $(\Sigma, T, Q, q_0, q_f, \Delta)$.

4 Example

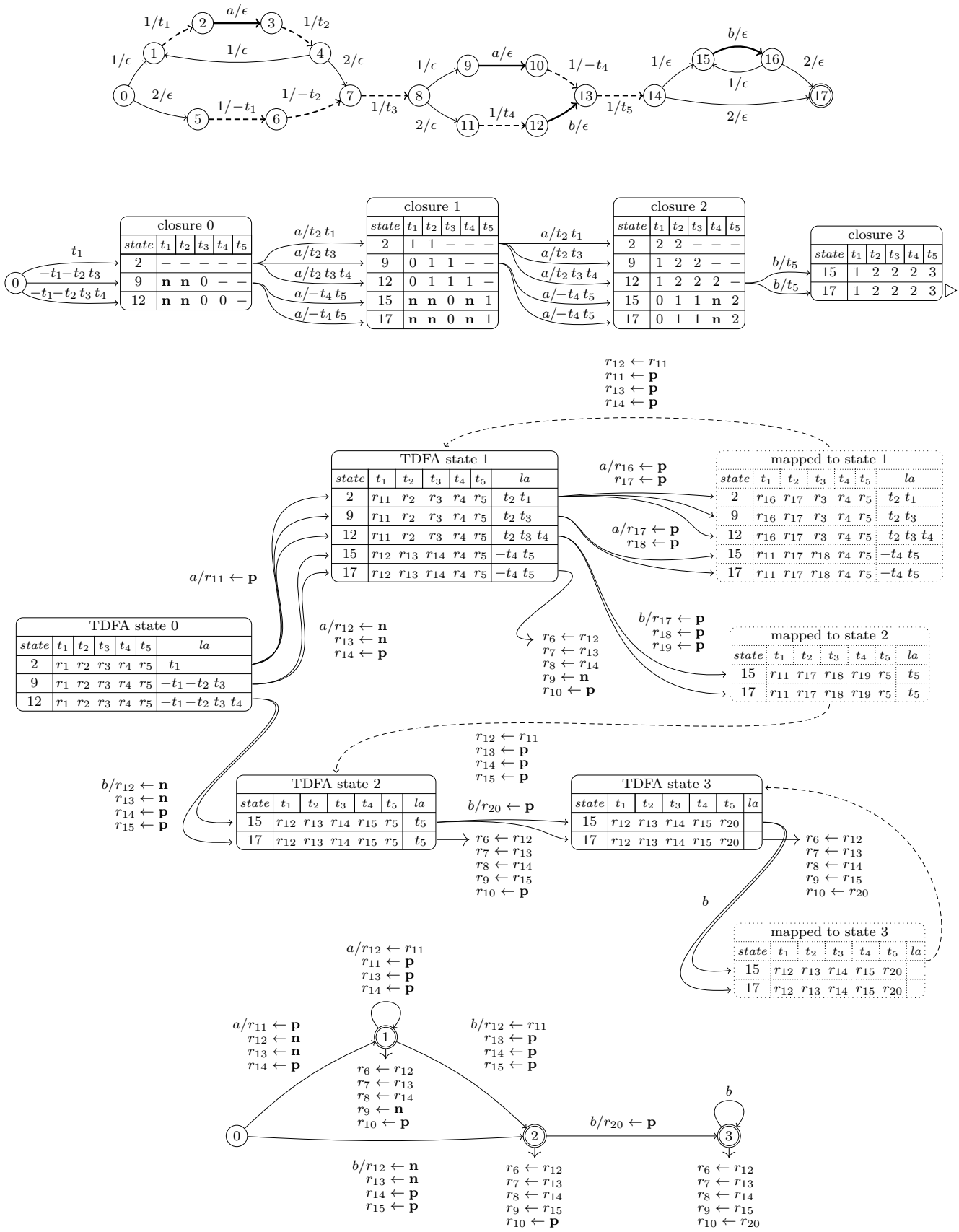


Figure 1: Example for RE $(1a2)^*3(a|4b)5b^*$: TNFA, simulation on string aab, determinization, TDFA.

5 Registerless TDFA

6 Evaluation

7 Conclusions